Lab 14

Tic-Tac-Toe Game

In this project, we will create a tic-tac-toe game on the LCD pixel display located at the BOOSTXL-EDUMKII BoosterPack attached to the MSP-EXP430FR6989 LaunchPad. The game will be controlled with the joystick on the BoosterPack and two buttons on the LaunchPad. The game will have 2 modes and 2 features: human vs. human mode, human vs. computer mode, game board resize feature, and game board reposition feature.

14.1 Logic of the Game

In this part, we will represent the tic-tac-toe game board with an array, create a variable to keep track of whose turn it is (X/O), and write a function that will check for a win.

Logical Game Board

Here is how we will represent the game board in code:

```
char gameBoard[9] =
{
 '0', '0', '0',
 '0', '0', '0',
 '0', '0', '0'
}
```

The 9 cells in the array represent the 9 fields on the tic-tac-toe board. In the game, a field can have 3 states and we will represent these 3 states in this way:

```
1) Field is not occupied    -> '0' (zero)
2) Field is occupied with X -> 'X'
3) Field is occupied with O -> 'O'
```

Whose Turn is it?

To represent whose turn it is, we will use an integer variable named *state*. If *state* is currently 0, this means it is X's turn. If *state* is currently 1, this means it is O's turn. When the game begins, *state* will be 0 (X always starts first) and all the cells of the gameBoard array will be in state 1 since the game board is empty. When a player makes a move, we will update our logical game board so that the player's move is saved and update *state* to indicate whose turn it is.

Checking for Game Result

Now that we established our logical game board's design and maintenance, it is time to write a function that checks for the result.

In tic-tac-toe, it is considered a win for one side to have 3 consecutive Xs or Os horizontally, vertically, or diagonally. In the game board, there are 3 rows, 3 columns, and 2 diagonals that are 3 fields long. Therefore, in the function we create, we will check whether one of the 3 rows, 3 columns, or 2 diagonals are occupied with only Xs or Os. In the near future when the game

is set up and being played, this function will be called each time after a move is made and *state* has been updated. Thus, rather than checking for both X and O, we will only check if the player who played last won the game.

The function is shown below. If the game has a result, the function returns 1. If the game is still going on, it returns 0. Observe the way that the checking process is being done and complete the missing parts (labeled with "...") of the function. Above the function is a tic-tac-toe board with gameBoard array's indices in their corresponding positions for visual purposes.

0	1	2
3	4	5
6	7	8

```
int checkForResult(void)
{
    volatile int i;

    // It is O's turn. That means X just played; check if X won.
    if (state == 1)
    {
        for (i = 0; i < 3; i++)
        {
            // Row checks:
            if (gameBoard[3*i] == 'X' && gameBoard[3*i + 1] == 'X' &&
               gameBoard[3*i + 2] == 'X')
            {
                ... // Turn on red LED
                return 1;
            }

            // Column checks:
            if (...)
            {
                ... // Turn on red LED
                return 1;
            }
        }

        // Diagonal checks:
```

```
    if (gameBoard[0] == 'X' && gameBoard[4] == 'X' && gameBoard[8]
        == 'X')
    {
        ... // Turn on red LED
        return 1;
    }

    if (...)
    {
        ... // Turn on red LED
        return 1;
    }
}

// It is X's turn. That means O just played; check if O won.
else
{
    for (i = 0; i < 3; i++)
    {
        // Row checks:
        if (gameBoard[3*i] == 'O' && gameBoard[3*i + 1] == 'O' &&
           gameBoard[3*i + 2] == 'O')
        {
            ... // Turn on green LED
            return 1;
        }

        // Column checks:
        if (...)
        {
            ... // Turn on green LED
            return 1;
        }
    }

    // Diagonal checks:
    if (gameBoard[0] == 'O' && gameBoard[4] == 'O' && gameBoard[8]
        == 'O')
    {
```

```
            ... // Turn on green LED
            return 1;
        }

        if (...)
        {
            ... // Turn on green LED
            return 1;
        }
    }

    // In the case where nobody won, check if the game is drawn or
       still going on.
    for (i = 0; i < 9; i++)
    {
        // If a single field is available, the game is still going on.
        if (gameBoard[i] == ...)
        {
            return 0; // Game is still going on
        }
    }

    // If all spaces are busy and nobody won, game is drawn.
    ... // Turn red LED on
    ... // Turn green LED on
    return 1;
}
```

Note that the function does not take the gameBoard array and *state* as inputs. This means that they will be declared globally (outside of any function). It would be a better programming practice to have these things inside a struct; but to make things less complicated, we will not do that. If you want to take on a challenge, you could store all the global variables we declare from now on inside a struct and work with that for the rest of the project.

Once you complete all the missing parts in the function, create a few test cases to check if the function works as expected. Set red and green LEDs as outputs, turn them off, change the gameBoard array so that a game is won by a side, drawn, or still going on, adjust *state* accordingly, and call the *checkForResult* function in your main function. If X wins, the red LED will turn on. If O wins, the green LED will turn on. If it is a draw, both LEDs will turn on.

Lastly, if game is still going on, no LEDs will turn on. Here is one test case example where X wins the game (red LED should turn on):

```
// '0': Field is empty, 'X': Field is occupied by X,
// 'O': field is occupied by O
char gameBoard[9] =
{
 '0', 'O', '0',
 'X', 'X', 'X',
 '0', 'O', '0'
}

// Represents the turns of players: 0 -> X-turn, 1 -> O-turn
volatile unsigned int state = 1;
```

14.2 Drawing the Game Board

In this part, we will draw the game board on the LCD Pixel Display located in the BoosterPack. Make sure to do the following before you start this section (these were all done in lab 11, you can refer back to there):

```
1) Include all the header files included in lab 11 along with
   stdio.h and stdint.h (these two header files will be in
   between angle brackets, not quotation marks)
2) In your current project directory, have all the files and
   folders that you had back in lab 11 (except the main.c file
   from lab 11, you should only have one main.c function
   inside your directory which is your current one)
3) Declare your graphics context globally (in this lab, it
   will be named g_sContext)
4) Configure the Serial Peripheral Interface (SPI) between the
   MCU and the LCD Pixel Display if not done already. You can
   simply copy paste the main.c file Dr. Abichar provided in
   lab 11 to your current main.c.
5) Open MSP Graphics Library 3.21.00.00 version User's Guide
   provided in lab 11 for information on functions
   specifically made for graphics purposes
```

Configuring SMCLK to 16 MHz

SMCLK will be used as the SPI clock as we know from lab 11. But in lab 11, SMCLK was configured to 8 MHz. In this project, we will configure it to its maximum clock rate in order for the LCD display to draw the pixels twice as fast as before. Although most of the code will be provided to you, it is important to understand the concept behind the code.

SMCLK is configured to DCOCLK by default. DCOCLK is 8 MHz by default but SMCLK has a default divider by 8 which is why SMCLK is 1 MHz by default (8 MHz / 8 = 1 MHz). DCOCLK can be configured to various clock rates including 16 MHz, which is what we will do in order to configure SMCLK to 16 MHz. We will also configure SMCLK's divider to 1 (already done in Dr. Abichar's code). Assuming you copied Dr. Abichar's main.c function from lab 11, find the following code in there:

```
// Configure SMCLK to 8 MHz (used as SPI clock)
CSCTL0 = CSKEY;                 // Unlock CS registers
CSCTL3 &= ~(BIT4|BIT5|BIT6);    // DIVS=0
CSCTL0_H = 0;                   // Relock the CS registers
```

Change this block of code with the following and complete the missing part by looking at CSCTL1 register information in Family User's Guide (page 104 for me but could be different for someone else):

```
// ***** Configure SMCLK to 16 MHz *****
CSCTL0 = CSKEY; // Unlock CS registers
CSCTL1 |= DCORSEL; // Set DCORSEL for high frequency selection
CSCTL1 &= ~DCOFSEL_7; // Reset DCOFSEL bitfield (not 0 by default)
CSCTL1 |= ... // Set DCO to 16 MHz
// SMCLK is already configured to DCOCLK by default

// Set SMCLK divider to 0 (could also be done by: CSCTL3 &= ~DIVS_7)
CSCTL3 &= ~(BIT4|BIT5|BIT6);
CSCTL0_H = 0; // Relock the CS registers
```

Note that when the code is copied to the editor, you should retype the characters ~(inverse) and ^(xor) from your keyboard. Keep this in mind for the rest of the project.

Visual Game Board Coordinates

The game board will have 4 coordinate points which we can adjust to decide its position and size. Since it resembles a big square, its four coordinates will be its corners. We will declare them globally just like the following:

```
volatile int W_max = 128; // max Width
volatile int W_min = 0; // min Width
volatile int H_max = 128; // max Height
volatile int H_min = 0; // min Height
```

Max width and height are 128 since the LCD Pixel Display has 128x128 pixel resolution. This will be our initial visual board configuration which represents the board at its maximum size on the display.

Drawing the Game Board on the LCD Pixel Display

The tic-tac-toe board consists of two vertical and horizontal lines. These lines divide the game board up to 3 fields horizontally and vertically. We want to write a function that automatically draws the game board on the LCD Pixel Display depending on its coordinates. This way, resizing/repositioning the game board will be as simple as adjusting its coordinates and calling the function we wrote. Part of the function definition is given below. Observe the code and try to understand it. To be able to complete the missing parts (indicated with '?') in the function calls made, browse the library's API documentation file to find out about the specifics of those functions.

```
void drawBoard()
{
    // First horizontal line's y-coordinate
    int32_t H1 = (H_max - H_min)/3 + H_min;
    // Second horizontal line's y-coordinate
    int32_t H2 = ((H_max - H_min)/3)*2 + H_min;
    // First vertical line's x-coordinate
    int32_t W1 = (W_max - W_min)/3 + W_min;
    // Second vertical line's x-coordinate
    int32_t W2 = ((W_max - W_min)/3)*2 + W_min;

    // Set up the 3-by-3 board.
    Graphics_drawLineH(&g_sContext, ?, ?, ?);
```

```
    Graphics_drawLineH(&g_sContext, ?, ?, ?);
    Graphics_drawLineV(&g_sContext, ?, ?, ?);
    Graphics_drawLineV(&g_sContext, ?, ?, ?);
}
```

When you figure out the missing parts, call the *drawBoard* function inside the main function to check it is drawn on the LCD Pixel Display correctly. Once that's done, adjust the board coordinates and draw the board again to see it drawn in different sizes and at different positions. For the board to remain its proper shape, please make sure to adjust its coordinates so that they represent a square.

14.3 Creating Cursors for Picking a Move

In this part, we will create cursors for each field inside the tic-tac-toe board. When playing the game, the players will choose their moves through moving the cursor among the fields and picking one field to place X/O.

Cursor Coordinates

To be able to create the cursors, we need to know the cursor coordinates that corresponds to the initial size (maximum size which covers the entire display) of the visual game board. Note that the cursors should be centered within each field for visual purposes. Figure 14.1 shows these coordinates along with the initial coordinates of the tic-tac-toe game board:

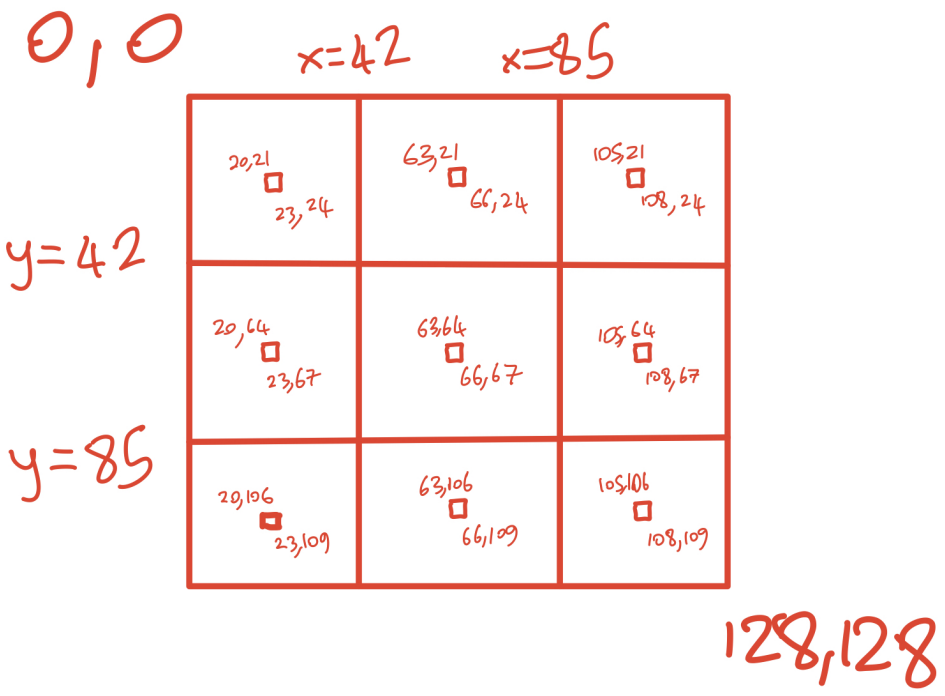

Figure 14.1: Initial Cursor Coordinates

As it could be seen from Figure 14.1, each cursor is represented with a small square; we will do the same in code. Inside your project folder, find the file named *grlib.h* and look for a struct named *Graphics_Rectangle* and learn about what it contains. This struct is provided to us to create rectangles. We will use this struct to represent each cursor. To store all the cursor coordinates in one place, we will use a global array that holds *Graphics_Rectangle* structs inside it. Its indices will match the indices of the *gameBoard* array so that when we offset to get the coordinates of a certain cursor, that index will also be where we will place the cursor on the game board. Lastly, we will use a global integer to keep track of which field the cursor is currently located at. In the given code, observe how the mapping of the first cursor's coordinates is done, and complete the rest of the array according to Figure 14.1.

```
// Initial coordinates of the cursors when the board size is
// at its maximum.
Graphics_Rectangle Cursor[9] = { {20, 21, 23, 24}, ... };

volatile unsigned int cursorIndex = 0; // Cursor[] index: 0 to 8
```

Drawing the Cursor

To draw the cursors, we need a function from the MSP Graphics Library that could draw a filled rectangle/square. Browse the library to find a function for this purpose. Make sure that this function also takes in the *Graphics_Rectangle* struct as an input. To keep track of whether the cursor is currently on the board or not, create a global variable named *cursor*. If this variable is 1, this will indicate that there is currently a cursor on some field of the game board. If the variable is 0, it will indicate otherwise. Given all this information, write the *drawCursor* function by completing the missing parts below. Note that the *cursorIndex* variable indicates which field the cursor currently is at.

```
void drawCursor(void)
{
    // Draw the cursor (cursorIndex indicates which field the
    // cursor should be drawn to) with the function found from
    // the MSP Graphics Library
    ...
    // Update the cursor variable to indicate that the cursor
    // is currently shown on the display
    ...
}
```

Erasing the Cursor

We will use a neat trick for erasing the cursors. Rather than actually erasing a certain range of pixels on the display, we will simply draw over the cursor with the background color. For example, if the background color is white and foreground color is blue (meaning the drawn cursor is blue), we will first set the foreground color same as the background color. Then we will simply draw the same cursor again. This will make the cursor mix into the background. Given this information, complete the missing parts of the *eraseCursor* function below.

```
void eraseCursor(void)
{
    // Set the foreground color same as the background color
    ...
    // Draw the cursor to erase the previously drawn cursor
    ...
    // Update the cursor variable to indicate that there are
    // no cursors present on the board
    ...
    // Configure back to original foreground color for future
    // drawings
    ...
}
```

Note that both *drawCursor* and *eraseCursor* functions know which cursor to erase because of the *cursorIndex* variable and the fixed *Cursor[]* array. In order to test both the *Cursor[]* array and functions written, you could create a small animation where the cursor blinks in the first field of the visual game board (with the help of a delay loop and the functions you wrote) for a little while and then goes to the second field, does the same thing and so on. Make sure that the cursors are centered within each field. If they seem very off from the center, make sure to check the *Cursor[]* array again and correct any mistake.

14.4 Blinking the Cursor & Moving it with the Joystick

In this part, we will use concurrent event handling in order to work with two different interrupts thrown by a single timer. You can refer back to Lab 6 for concurrent event handling examples. Our goal is to blink the cursor every 0.5 seconds along with performing ADC conversions to check the joystick movement every 0.1 seconds in order to move the cursor around the game board. We will have Timer0_A run at 32 KHz (ACLK configured to the crystal) in continuous mode. Here is a visual of the events:

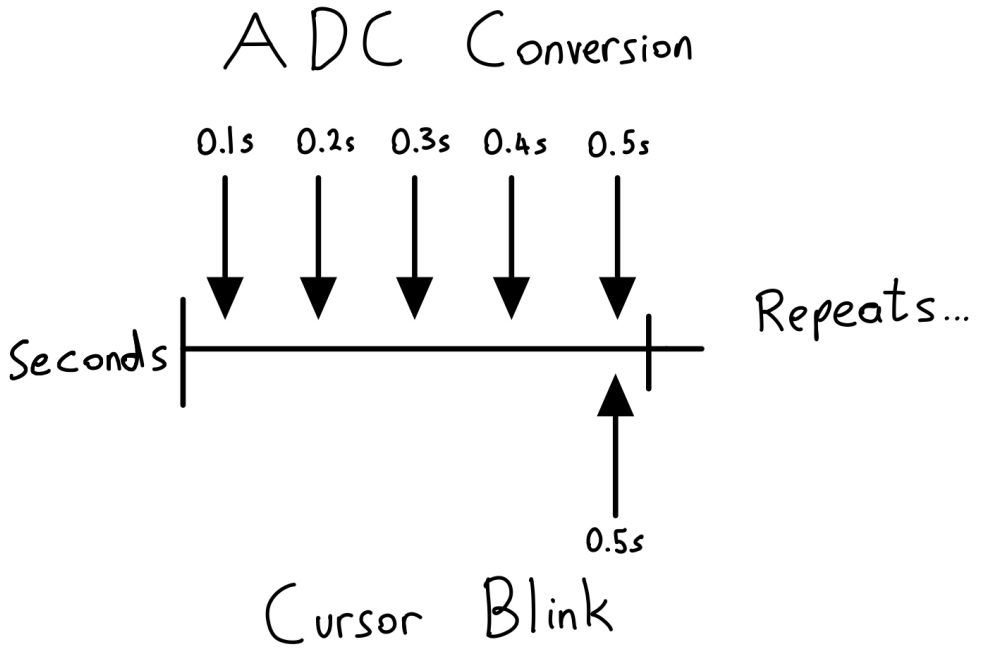

Figure 14.2: Concurrent Events

Before starting this part, make sure to include the *Initialize_ADC* function from lab 11 in your code and call it in your main function.

Blinking the Cursor

We will blink the cursor to further indicate where the user is at inside the visual game board. This will also ensure the user that the display is not frozen and the game is still going on. We will set up the interrupt for blinking the cursor through Channel 1. Before attempting to get the blinking functionality to work, make sure to configure ACLK to the 32 KHz crystal using the function provided from the previous labs and set up Timer0_A in continuous mode with a division of 1. After getting the initial steps done, set up the Channel 1 interrupt by calculating the clock cycles that correspond to 0.5 seconds. Lastly, write the ISR function for Channel 1 where the cursor blinking will occur. One idea is to utilize the *cursor* variable to help decide

whether a cursor is already placed on the board. If a cursor is currently drawn on the board, then erase the cursor. Else, draw the cursor. Do not forget to clear the flag and update the TA0CCR1 before the ISR is over. Make sure to update TA0CCR1 by adding (TA0R + # of cycles you found) to it. Since this is a big project, adding TA0R to it ensures that it will be updated sooner or later even if TA0CCR1 falls behind TA0R due to some code taking too long.

Moving the Cursor

Now that the cursor blinks every 0.5 seconds, it is time to add the functionality of moving it around the game board. We will set up the interrupt for ADC conversions through Channel 0. Before attempting to get this functionality to work, [1] we need to set certain thresholds for the joystick which indicate the movement intention of the user and [2] we also need to plan out how we will move the cursor depending on the circumstances. Let's figure out [1] first.

From the analog to digital conversion, we get a number between 0-4095 inclusive for both x and y directions. As we move the joystick left and/or down, x and/or y values should be getting smaller. As we move the joystick right and/or up, x and/or y values should be getting bigger. When the joystick isn't being moved, both x and y values should be around the middle point of 0-4095 (somewhere around 2047). For this project, we will assume that when the user moves the joystick to certain direction by 80% or more, the user's intention is to move the cursor to that direction. Therefore, we need to calculate 20% and 80% of the maximum value we can get from the ADC which is 4095. These values are 819 and 3276 respectively. When checking for whether joystick is moved left or down, we will use 819 along with the correct ADC channel. When checking for whether joystick is moved right or up, we will use 3276 along with the correct ADC channel. This will make more sense once the code is given to you to complete the missing parts. Now, let's figure out [2].

When considering where to move the cursor in certain situations, what should we base our decisions on? For example, if the calculated *cursorIndex* is less than 0 or greater than 8, what will we do? Or if the joystick is moved downwards but there already is an X or O in the spot right below, where should we place the cursor? These decisions should be made based on providing the most intuitive visuals to the user. Take a look at the following cases and figures which explain the cursor placement rules for this project. The solid arrows next to the cursor (drawn as a blue filled circle but will be squares in our game) represent the joystick movement. The dashed arrows represent where the cursor will land.

Edge Cases

Edge cases are when the user tries going off the board with the cursor. In these cases, we will not move the cursor which is indicated by the keyword ”No”. Note that moving left when the cursor is at index 3 or 6 and moving right when cursor is at 2 or 5 aren’t edge cases.

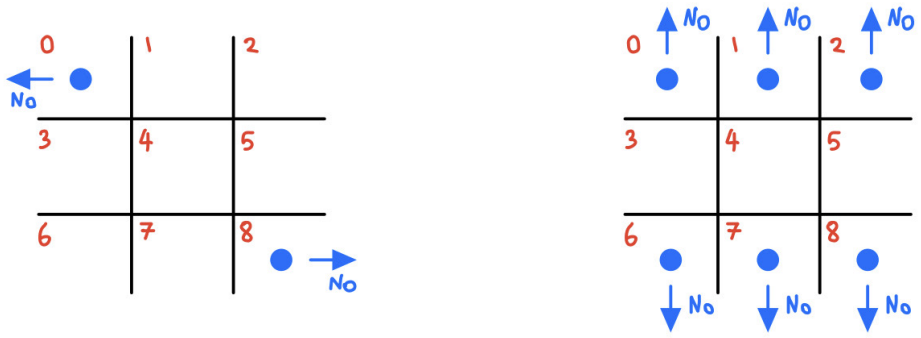

Figure 14.3: Edge Cases

Moving Left

When moving left, we will check the availability of all the indices from the left of the cursor all the way up to index 0 in order. We will place the cursor at the first available spot out of the spots we checked. If no spots are available for this movement intention, we will not move the cursor.

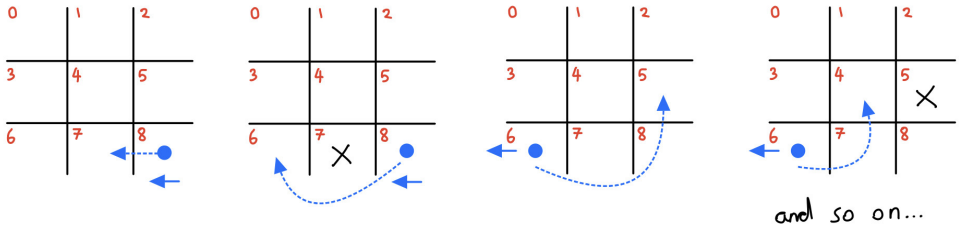

Figure 14.4: Left Movement Cases

Moving Right

When moving right, we will check the availability of all the indices from the right of the cursor all the way up to index 8 in order. We will place the cursor to the first available spot out of the spots we checked. If no spots are available for this movement intention, we will not move the cursor.

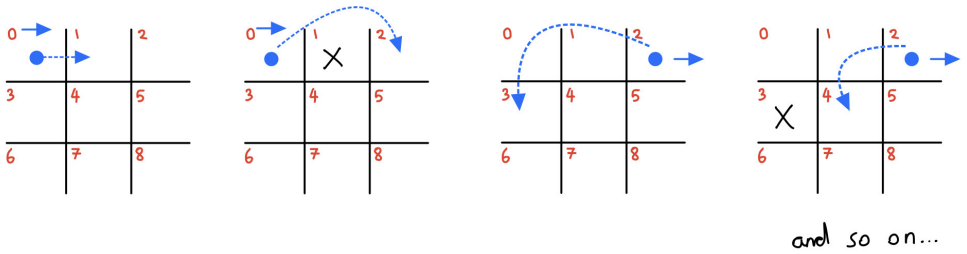

Figure 14.5: Right Movement Cases

Moving Down

Due to the index placement of the game board, it was easier to make moving left and right intuitive for the user. It will take a bit more work to make moving down and up intuitive. When moving down, we will first check if the cursor is at the bottom row. In that case, we will not move the cursor (this is shown in Figure 14.3 as an edge case). If that is not the case, then we will give the priority to the spot right below the cursor. If that spot is busy, then we will figure out which row the cursor is located at. The second priority will be given to the field that's two spots below the cursor if the cursor is at the top row. If no intuitive solution is available, then we will check the availability of all the indices from the beginning of the next row (the row right below the row that the cursor is at) all the way up to index 8 in order. We will place the cursor at the first available spot out of the spots we checked. If no spots are available for this movement intention, we will not move the cursor.

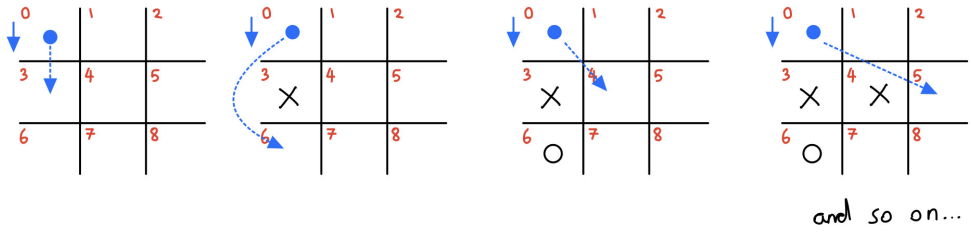

Figure 14.6: Down Movement Cases

Moving Up

When moving up, we will first check if the cursor is at the top row. In that case, we will not move the cursor (this is shown in Figure 14.3 as an edge case). If that is not the case, then we will give the priority to the spot right above the cursor. If that spot is busy, then we will figure out which row the cursor is located at. The second priority will be given to the field that's two spots above the cursor if the cursor is at the bottom row. If no intuitive solution is available, then we will check the availability of all the indices from the beginning of the next row (the row right above the row that the cursor is at) all the way down to index 0 in order. We will place the cursor at the first available spot out of the spots we checked. If no spots are available for this movement intention, we will not move the cursor.

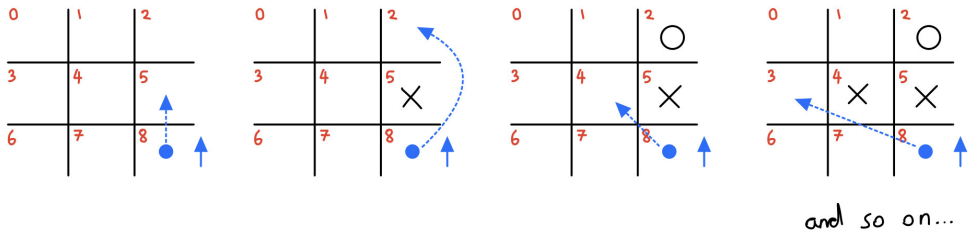

Figure 14.7: Up Movement Cases

Now that we decided the joystick thresholds and established a set of rules for cursor movement, we can start transferring all the information into code. Set up the Channel 0 interrupt by calculating the clock cycles that correspond to 0.1 seconds. Then create a global integer named *JS_Cursor* and set it to 1 inside the main function. This variable will act as an indicator of whether we are currently using the joystick to move the cursor. If so, it will be 1. Else, it will be 0. The joystick will have other functionalities as we will see later on. After doing all the steps mentioned, read through the ISR function given below to understand how all the information provided is translated to code. Then complete the missing parts of the given ISR function and include it in your code.

```
// ISR of Channel 0 -> ADC conversion event
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR()
{
    volatile int32_t i; // int 32-bit type

    // Start analog to digital conversion to read the joystick.
    ADC12CTL0 |= ...
```

```
// wait until the conversion operation is over
while ( (ADC12CTL1 & ADC12BUSY) != 0) {}

 // If currently playing the game (moving the cursor around)...
if (JS_Cursor)
{
    // Joystick direction: Left and if cursor isn't at top-left
       corner...
    if (ADC12MEM0 <= 819 && cursorIndex != 0)
    {
        // Check available spots to the left and above in order.
        for (i = (cursorIndex - 1); i >= 0; i--)
        {
            // If spot is available...
            if (gameBoard[i] == ...)
            {
                eraseCursor();
                cursorIndex = i; // Update cursor
                drawCursor();
                for (i = 8000; i > 0; i--) {} // Delay loop for
                   smooth transition
                break;
            }
        }
    }

    // Joystick direction: Right and if cursor isn't at bottom-
       right corner...
    if (ADC12MEM0 >= 3276 && cursorIndex != ...)
    {
        // Check available spots to the right and below in order.
        for (i = (cursorIndex + 1); i <= 8; i++)
        {
            // If spot is available...
            if (gameBoard[i] == ...)
            {
                eraseCursor();
                cursorIndex = i; // Update cursor
                drawCursor();
```

```
            for (i = 8000; i > 0; i--) {} // Delay loop for
                smooth transition
            break;
        }
    }
}

// Joystick direction: Up and if cursor isn't at top row...
if (ADC12MEM1 >= ... && cursorIndex > 2)
{
    // Give priority to the spot right above (this makes
       visuals intuitive).
    if (gameBoard[cursorIndex-3] == '0')
    {
        eraseCursor();
        cursorIndex -= 3; // Update cursor
        drawCursor();
        for (i = 8000; i > 0; i--) {} // Delay loop for smooth
            transition
    }

    // If cursor is at the bottom row...
    else if (cursorIndex > 5)
    {
        // Give priority to the spot two fields above (
           intuitive visuals).
        if (gameBoard[cursorIndex-6] == '0')
        {
            eraseCursor();
            cursorIndex -= 6; // Update cursor
            drawCursor();
            for (i = 8000; i > 0; i--) {} // Delay loop for
                smooth transition
        }

        // Choose the first available spot from middle row all
            the way down to index 0
        else
        {
```

```
            for (i = 5; i >= 0; i--)
            {
                // If spot is available...
                if (gameBoard[i] == '0')
                {
                    eraseCursor();
                    cursorIndex = i; // Update cursor
                    drawCursor();
                    for (i = 8000; i > 0; i--) {} // Delay
                       loop for smooth transition
                    break;
                }
            }
        }
    }

    // If cursor is at the middle row...
    else
    {
        // Choose the first available spot from top row
        for (i = 2; i >= 0; i--)
        {
            // If spot is available...
            if (gameBoard[i] == ...)
            {
                eraseCursor();
                cursorIndex = i; // Update cursor
                drawCursor();
                for (i = 8000; i > 0; i--) {} // Delay loop
                   for smooth transition
                break;
            }
        }
    }
}

// Joystick direction: Bottom and if cursor isn't at bottom
   row...
if (ADC12MEM1 <= ... && cursorIndex < 6)
```

```
{
    // Give priority to the spot right below (this makes
       visuals intuitive).
    if (gameBoard[cursorIndex+3] == '0')
    {
        eraseCursor();
        cursorIndex += 3; // Update cursor
        drawCursor();
        for (i = 8000; i > 0; i--) {} // Delay loop for smooth
            transition
    }

    // If cursor is at the top row...
    else if (cursorIndex < 3)
    {
        // Give priority to the spot two fields below (
           intuitive visuals).
        if (gameBoard[cursorIndex+6] == '0')
        {
            eraseCursor();
            cursorIndex += 6; // Update cursor
            drawCursor();
            for (i = 8000; i > 0; i--) {} // Delay loop for
               smooth transition
        }

        // Choose the first available spot from middle row all
            the way up to index 8
        else
        {
            for (i = 3; i <= 8; i++)
            {
                // If spot is available...
                if (gameBoard[i] == ...)
                {
                    eraseCursor();
                    cursorIndex = i; // Update cursor
                    drawCursor();
                    for (i = 8000; i > 0; i--) {} // Delay
```

```
                            loop for smooth transition
                        break;
                    }
                }
            }

            // If cursor is at the middle row...
            else
            {
                // Choose the first available spot from bottom row
                for (i = 6; i <= 8; i++)
                {
                    // If spot is available...
                    if (gameBoard[i] == ...)
                    {
                        eraseCursor();
                        cursorIndex = i; // Update cursor
                        drawCursor();
                        for (i = 8000; i > 0; i--) {} // Delay loop
                           for smooth transition
                        break;
                    }
                }
            }
        }
    }

    TA0CCR0 = TA0R + ... // Update for next interrupt

    // Hardware clears the flag (CCIFG in TA0CCTL0) due to Channel 0
}
```

To make sure everything is working correctly, test the code by moving the joystick around and see how the cursor moves. Make sure to have all the spots in the logical game board available. If you want to check how the cursor will move in the cases where there is an X or O on the board, you could fill the logical game board up with some Xs and Os. Note that they won't be shown on the visual game board yet since we didn't implement that functionality yet. But the joystick will

still move to the right spots since the code uses the logical game board to decide the movement of the cursor.

14.5 Board Resizing & Repositioning Features

In this part, we will add board resizing and repositioning features to our project. Adding these features will allow the program to be scalable in terms of board size. In other words, the program will support playing the game regardless of the board's size or position. We want to implement these features in real-time. This means that while the user is playing the game, they will be able to just resize and/or reposition the game board and continue on with their game. These features will use the 2 push buttons located at the bottom of the LaunchPad [1], the joystick on the BoosterPack, and the joystick button. Note that if you push the joystick down, you will hear a clicking sound; that's the joystick button. Let's say the user wants to resize the board. How this will work is while playing the game, the user will press button 1. When this happens, the blinking of the cursor will stop to indicate that the game is paused for resizing the game board. Then if the user moves the joystick to the left, the game board will be getting smaller. Vice versa for when the user moves the joystick to the right. Once the user decides the size, he/she will simply push the joystick button and the new board size will be set. As soon as this is done, the cursor will start blinking in the field that it was located at before indicating that the game is still going on. This means that along with updating the board's coordinates, we will also update the cursors' coordinates accordingly. As it could be seen, the joystick has multiple tasks depending on the game stage we are at. This is exactly why we created the *JS_Cursor* global indicator.

Now that there are two additional tasks being added to the joystick, we need to represent these tasks with indicators in the code. Go ahead and declare 2 global integers named *JS_BoardResize* and *JS_BoardReposition*. These will work just like *JS_Cursor* where if they are 0, this means those tasks are currently inactive. Else, they are active and the joystick will performing them instead. Reset the newly declared indicators while *JS_Cursor* is set to 1. This is how the game should always start since the other 2 features are only activated during the game.

[1] Throughout the project, "button 1" will represent the left push button and "button 2" will represent the right one.

Overview of Events

To be able to implement these 2 features, we will use the concept of concurrency via interrupts (recall lab 7). Here is an overview of events in order to keep track of everything that's going on from the beginning of when board resizing/repositioning feature is activated to the end when we are setting everything up to return back to the game:

1. While playing the game, user pushed button 1 or 2 in order to resize/reposition the game board which automatically directs the PC to enter Port 1 ISR (PORT1_VECTOR). Here is what will happen in the Port 1 ISR:
    - If there is a cursor on the board, erase it.
    - Clear and set indicators to change the joystick's functionality based on whether button 1 or 2 was pushed.
    - Disable button 1 and 2 interrupts so that the user cannot select a different task before finishing the task at hand.
    - Disable the cursor blinking interrupt event so that the cursor doesn't keep blinking while resizing/repositioning the game board.
    - Implement a delay loop to wait for bouncing to end.
2. At this point, joystick functionality is set and user control is limited for preventing bugs. ADC conversion interrupt event still occurs every 0.1 seconds. This allows the user to resize or reposition the game board using the joystick. Note that each time the user changes the board size/position, cursors' coordinates get updated inside ADC conversion interrupt event's ISR.
3. User resized/repositioned the game board and pushed the joystick button. This automatically sends the PC to the ISR function of the joystick button that will be created later on in this part of the project. Here is what will happen in the ISR of the joystick button:
    - Clear and set indicators for converting joystick's functionality back to moving the cursor around.
    - Turn the green LED on for a little while to both indicate that the game board resize/reposition event is finalized and to wait for bouncing to end.
    - Schedule the next interrupt for the cursor blinking event, enable its interrupt, and clear its flag.
    - Enable button 1 and 2 interrupts and clear their flags.

- Clear joystick button's interrupt flag.

4. The game is now playable again with the new edited board! The cursor should be blinking now.

We will first implement the #1 and #3 in order to fully grasp the control flow. Then we will go back to #2 which is the actual implementation of the game board resize/reposition feature.

Setting Up Port 1 ISR

Define button 1 and 2 globally to their corresponding bits for code readability. In this project, "button1" and "button2" will be used in the code. Set the two push buttons up in the main just like you did in previous labs. Then complete the missing parts in the Port 1 ISR below and include it in your code.

```
/*ISR of Port 1 (PORT1 Vector) -> Game board resizing/repositioning (
   button 1/button 2)*/
#pragma vector = PORT1_VECTOR
__interrupt void Port1_ISR()
{
    volatile uint32_t i;

    // If cursor is on, erase it.
    if (cursor != 0)
        eraseCursor();

    ... // Clear the JS_Cursor indicator

    // Button 1: Resize the game board
    if ( (P1IFG & button1) != 0 )
    {
        ... // Set the resize indicator
    }

    // Button 2: Reposition the game board
    else if ( (P1IFG & button2) != 0 )
    {
        ... // Set the reposition indicator
    }
```

```
    /*Disable button 1 and 2 interrupts so the current task cannot be
       overwritten until it is over.*/
    ...

    ... // Disable cursor blinking interrupt.

    // Delay loop to wait for bouncing to end.
    for (i = 10000; i > 0; i--){}

    /*No need to clear the flags, interrupts are already disabled. The
        flags will be cleared in the joystick button's ISR later on.
       */
}
```

Setting Up Joystick Button ISR

To find out which pin is connected to the joystick button on the LaunchPad, take a look at the BoosterPack. Under the joystick, there should be a line that says "SEL: J1.5" which indicates the pin that the joystick button corresponds to on the BoosterPack. J1 is the left column of pins. The first pin is J1.1 which is the top pin of the column. From this information, we could locate where J1.5 is. The way we connect the BoosterPack to the LaunchPad allows J1.5 to share its contents with a pin on the LaunchPad. We could find this pin by simply locating which pin on the LaunchPad J1.5 connects to: P3.2. Define the word "JSbutton" as BIT2 so that when we interact with the joystick button, it is obvious in code. Now that we know which pin the joystick button is represented by, let's set it up. Set up the joystick button just like how you set button 1 and 2 inside the main function. Then complete the missing parts in the given Port 3 ISR and include it in your code.

```
// ISR of Port3 (PORT3 Vector) -> Joystick button
#pragma vector = PORT3_VECTOR
__interrupt void Port3_ISR()
{
    volatile uint32_t i; // unsigned int 32-bit type

    // If joystick button is pressed...
    if ( (P3IFG & JSbutton) != 0)
    {
```

```
        /*If just got done with resizing or repositioning the game
           board...*/
        if (JS_BoardResize || JS_BoardReposition)
        {
            /* Enable the JS_Cursor indicator to change the
               functionality of the joystick and disable the other
               indicators.*/
            JS_BoardResize = ...
            JS_BoardReposition = ...
            JS_Cursor = ...

            /*Turn green LED on for a short period of time to indicate
                that the board size/position is set.
            This waiting time also allows for bouncing to be over for
               the joystick button.*/
            P9OUT |= ...
            for (i = 8000; i > 0; i--) {}
            P9OUT &= ...

            /*Channel 1 interrupt has been off for awhile. So update
               TA0CCR1 to schedule next interrupt, enable its
               interrupt, and clear its flag.*/
            TA0CCR1 = TA0R + ...
            TA0CCTL1 |= ...
            TA0CCTL1 &= ...

            /*Enable button 1 and 2 interrupts to allow the user to
               resize/reposition the game board again if they wish to
                later on.*/
            ... // Enable interrupts
            ... // Clear flags
        }

        ... // Clear the JSbutton flag
    }
}
```

Closer Look at Game Board Resize/Reposition Features

When resizing/repositioning the game board in real-time, 4 actions will be performed in order over and over again until the joystick button is pushed:

1. Erase the game board
2. Update game board coordinates accordingly
3. Update cursor coordinates based on the current size of the board
4. Draw the game board

We already have the function for #4 which is the *drawBoard* function. We will take care of #2 at the end while taking a closer look at the each individual feature. So let's work on getting #1 and #3 ready.

Erasing the Game Board

Erasing the game board will have a very similar process with drawing the game board. We will simply make the foreground color same as the background color, draw the game board with that color, erase the cursor if it is on the board, and then configure the foreground color back to its original color for future drawings. Note that the reason why we do not just clear the entire display each time is because board resize/reposition will be done in real-time and we need to do things at a fast paste. Complete the missing parts of the given function and include it in your code.

```
void eraseBoard(void)
{
// First horizontal line's y-coordinate
    int32_t H1 = (H_max - H_min)/3 + H_min;
    // Second horizontal line's y-coordinate
    int32_t H2 = ((H_max - H_min)/3)*2 + H_min;
    // First vertical line's x-coordinate
    int32_t W1 = (W_max - W_min)/3 + W_min;
    // Second vertical line's x-coordinate
    int32_t W2 = ((W_max - W_min)/3)*2 + W_min;

    /*Configure foreground color to same as the background color in
       order to erase.*/
    ...
```

```
    // Erase the 3-by-3 board
    Graphics_drawLineH(&g_sContext, ?, ?, ?);
    Graphics_drawLineH(&g_sContext, ?, ?, ?);
    Graphics_drawLineV(&g_sContext, ?, ?, ?);
    Graphics_drawLineV(&g_sContext, ?, ?, ?);

    // Erase cursor if it is drawn.
    if (cursor == 1)
        Graphics_fillRectangle(&g_sContext, &Cursor[cursorIndex]);

    // Configure foreground color back to original.
    ...
}
```

Updating the Cursor Coordinates

When updating the cursor coordinates, we need to take board length into account and divide the board up to certain sections in order to be able to center each cursor inside the fields. When we find the center point of the fields, we need to offset by a certain amount from the center point in order to create the cursor (a square). We could also change this offset amount depending on the board size so that when the board size becomes very small, the cursor will also get smaller to look proportional with the board. Here is a visual of how the board is divided up to 6 sections to find the center point of each field and how offsetting from the center point allows us to decide the new coordinates of the cursors:

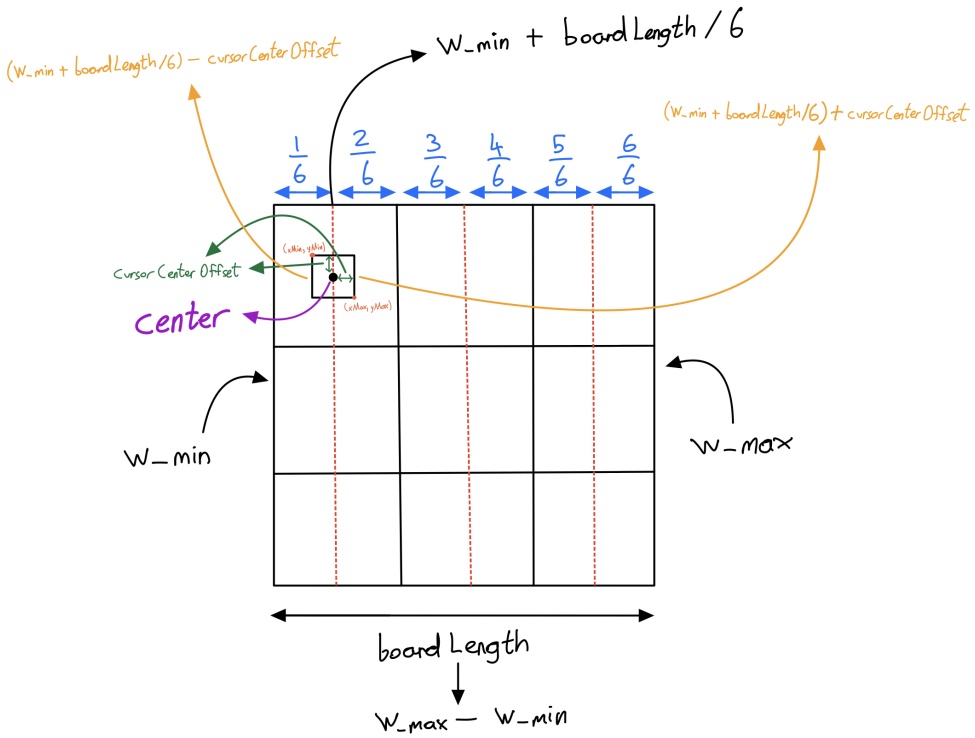

Figure 14.8: Cursor Coordinate Update

This figure specifically explains how *Cursor[0]* is found by dividing the board up to 6 sections and offsetting from the center point of the first field. Please zoom into the page in order to see the details inside the cursor (square) inside the first field. Note that H_min and H_max could also have been used in this case. But since the game board is a square, board length and height are the same regardless. Include the declaration of the global variable *cursorCenterOffset* in your code. Then read the comment in order to understand its functionality better. Its functionality is also shown visually in Figure 14.8.

```
volatile unsigned int cursorCenterOffset = 2; /* Cursors are
   represented by a square. This variable represents the offset from
   the center of the square in order to draw a cursor. Here is an
   example to further explain the use of this variable: If this
   variable is 1, then from the coordinate we are at, we will offset
   to left, right, top, and bottom by 1 to create a square that is 3
   pixels of length. cursorCenterOffset changes depending on the game
    board's size compared to the display size. If game board's
   current size is less than the third of display's size, then
   cursorCenterOffset will be 1. Else 2. This variable is updated
   inside updateCursorCoordinates(). */
```

Now include the *updateCursorCoordinates* function shown below in your code. Since Figure 14.8 shows how *Cursor[0]* Graphics_Rectangle struct is found, compare the visual to the code where *Cursor[0]*'s fields are calculated. Once you understand it, look at the rest of the calculations in the code and use the visual as a reference to understand them. If you have any questions in your mind about a certain part of this function, draw everything out to provide yourself with a visual. Visual thinking will help a lot in understanding this function.

```
// Updates cursor locations when coordinates of the game board change.
void updateCursorCoordinates(void)
{
    /*displayWidth should be 128 since the context covers the entire
       display.*/
    volatile int displayWidth = Graphics_getDisplayWidth(&g_sContext);
    volatile int boardLength = W_max - W_min;
    /*NOTE: W_max - W_min = H_max - H_min, so boardLength represents
       both height and width of the game board.*/

    /*If boardLength is less than third of the displayWidth, then let
```

```
   the cursorCenterOffset be 1. Else 2.*/
if ( boardLength < (displayWidth/3) )
    cursorCenterOffset = 1;
else
    cursorCenterOffset = 2;

Cursor[0].xMin= (W_min + boardLength/6) - cursorCenterOffset;
Cursor[0].yMin= (H_min + boardLength/6) - cursorCenterOffset;
Cursor[0].xMax= (W_min + boardLength/6) + cursorCenterOffset;
Cursor[0].yMax= (H_min + boardLength/6) + cursorCenterOffset;

Cursor[1].xMin= (W_min + boardLength*0.5) - cursorCenterOffset;
Cursor[1].yMin= (H_min + boardLength/6) - cursorCenterOffset;
Cursor[1].xMax= (W_min + boardLength*0.5) + cursorCenterOffset;
Cursor[1].yMax= (H_min + boardLength/6) + cursorCenterOffset;

Cursor[2].xMin= (W_min + (boardLength/6)*5 ) - cursorCenterOffset;
Cursor[2].yMin= (H_min + boardLength/6) - cursorCenterOffset;
Cursor[2].xMax= (W_min + (boardLength/6)*5 ) + cursorCenterOffset;
Cursor[2].yMax= (H_min + boardLength/6) + cursorCenterOffset;

Cursor[3].xMin= (W_min + boardLength/6) - cursorCenterOffset;
Cursor[3].yMin= (H_min + boardLength*0.5) - cursorCenterOffset;
Cursor[3].xMax= (W_min + boardLength/6) + cursorCenterOffset;
Cursor[3].yMax= (H_min + boardLength*0.5) + cursorCenterOffset;

Cursor[4].xMin= (W_min + boardLength*0.5) - cursorCenterOffset;
Cursor[4].yMin= (H_min + boardLength*0.5) - cursorCenterOffset;
Cursor[4].xMax= (W_min + boardLength*0.5) + cursorCenterOffset;
Cursor[4].yMax= (H_min + boardLength*0.5) + cursorCenterOffset;

Cursor[5].xMin= (W_min + (boardLength/6)*5 ) - cursorCenterOffset;
Cursor[5].yMin= (H_min + boardLength*0.5) - cursorCenterOffset;
Cursor[5].xMax= (W_min + (boardLength/6)*5 ) + cursorCenterOffset;
Cursor[5].yMax= (H_min + boardLength*0.5) + cursorCenterOffset;

Cursor[6].xMin= (W_min + boardLength/6) - cursorCenterOffset;
Cursor[6].yMin= (H_min + (boardLength/6)*5 ) - cursorCenterOffset;
Cursor[6].xMax= (W_min + boardLength/6) + cursorCenterOffset;
```

```
    Cursor[6].yMax= (H_min + (boardLength/6)*5 ) + cursorCenterOffset;

    Cursor[7].xMin= (W_min + boardLength*0.5) - cursorCenterOffset;
    Cursor[7].yMin= (H_min + (boardLength/6)*5 ) - cursorCenterOffset;
    Cursor[7].xMax= (W_min + boardLength*0.5) + cursorCenterOffset;
    Cursor[7].yMax= (H_min + (boardLength/6)*5 ) + cursorCenterOffset;

    Cursor[8].xMin= (W_min + (boardLength/6)*5 ) - cursorCenterOffset;
    Cursor[8].yMin= (H_min + (boardLength/6)*5 ) - cursorCenterOffset;
    Cursor[8].xMax= (W_min + (boardLength/6)*5 ) + cursorCenterOffset;
    Cursor[8].yMax= (H_min + (boardLength/6)*5 ) + cursorCenterOffset;
}
```

Now that we are done with #1, #3, and #4 in the list of actions taken when resizing/repositioning the game board, it's finally time to implement #2 for both resizing and repositioning the game board!

Board Resize Implementation

We will resize the board by 10 pixels in width and height from its maximum point (right bottom corner point of the game board) inside Channel 0 ISR each time the user wants to change the board size. As mentioned before, if the user moves the joystick to the left by 80% or more, the game board will get smaller and if the user moves the joystick to the right by 80% or more, the game board will get larger. Observe the figure below to see how this will work visually. Left side is the previous state of the game board. When the joystick is moved to the left by more than 80%, the right side shows what happens as it is the current state of the game board.

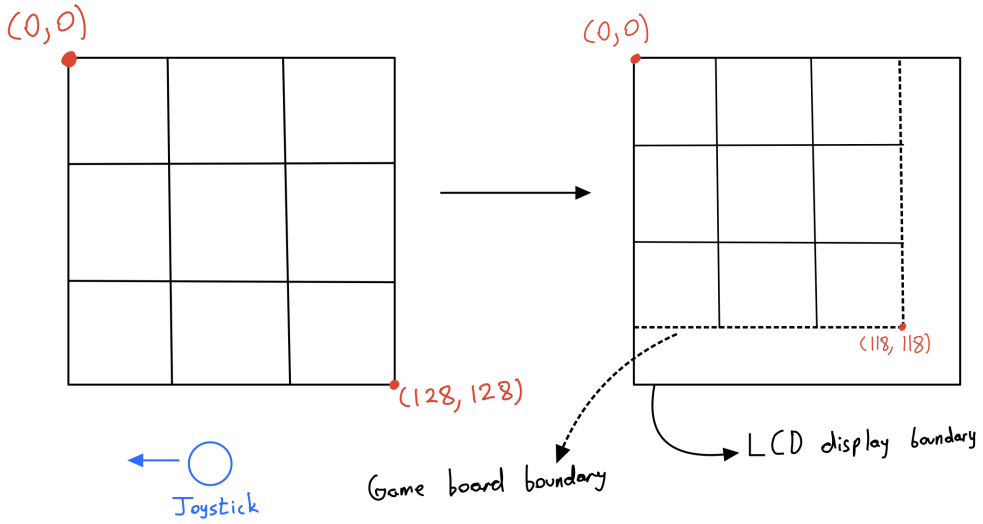

Figure 14.9: Board Resize

The LCD display has 128x128 pixel resolution, so the possible board sizes (in pixels) in this case are 18x18, 28x28, 38x38, 48x48, 58x58, 68x68, 78x78, 88x88, 98x98, 108x108, 118x118, and 128x128. Note that we do not have 8x8 as a choice since that would be way too small. Before we start implementing the board resize feature, let's take a look at the edge cases:

1. Trying to get the game board to be larger when its right-bottom corner is already at the bottom-right corner of the LCD display
2. Trying to make the game board smaller when it is 18x18

Note that first point takes care of the possibility of making the game board larger than 128x128 and also doesn't allow the user to make the game board larger when the board is resized and repositioned to the bottom-right corner of the display. Observe the figure below to further understand these two edge cases. Direction of the joystick is shown and in both cases, nothing will change.

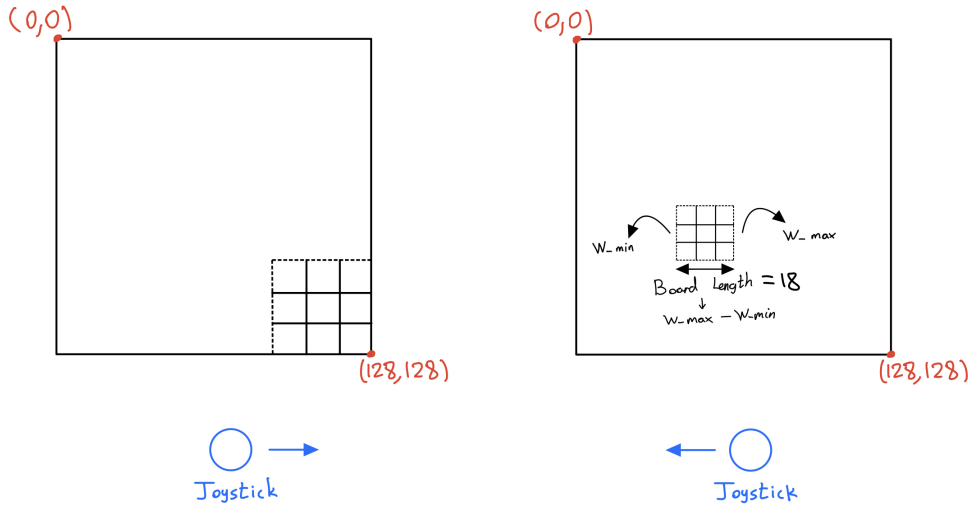

Figure 14.10: Board Resize Edge Cases

Now that we know how the board resize feature will work and its edge cases, let's implement it in code! In Channel 0, we will check for which indicator is currently on to activate the correct joystick functionality. Currently, your Channel 0 ISR includes only the *JS_Cursor* indicator and the joystick implementation of it. Add an *else if* statement that checks whether *JS_BoardResize* is set. By having an *else if* statement rather than an *if* statement, we ensure further ensure that the joystick will not have two functionalities at once. Inside the brackets of the statement, include the following code which implements the board resize feature and complete the missing parts.

```
/*If joystick is moved to the left by 20 percent of 4095 or more (
   value is 819 or less) and the game board is not at its minimum
   reasonable size, make the board smaller.*/
if (... && (W_max - W_min) > 18)
{
    ... // erase the board
    ... // decrease W_max by 10
    ... // decrease H_max by 10
    ... // update cursor coordinates
    ... // draw the board again
}

/*If joystick is moved to the right by 80 percent of 4095 or more (
   value is 3276 or more) and the game board is not going to pass
   display boundaries, make the board bigger.*/
```

```
if (... && ((W_max + 10) <= 128) && ((H_max + 10) <= 128))
{
    ... // erase the board
    ... // increase W_max by 10
    ... // increase H_max by 10
    ... // update cursor coordinates
    ... // draw the board again
}
```

Note that we could have just updated the cursor coordinates once the board size is set (joystick button is pushed) which would increase the performance. The reason why we update it each time the board size changes in size is because when drawing Xs and Os, we will use cursor coordinates to center them inside the fields of the game board. Therefore if there are Xs and Os on the game board currently, when resizing the game board, they will also be updated and shown on the display in real-time.

Board Reposition Implementation

For this feature, we will support both straight and diagonal movement of the board. It will be very similar to resizing the board except the way we change the board's coordinates will be different. A single unit for this feature will be 4 pixels. In other words, in each instance of repositioning the game board the game board, we will move the game board 4 pixels towards the desired direction. For example, when the user wants to move the game board one unit down, we will erase the board, add 4 to both *H_min* and *H_max*, update cursor coordinates, and draw the board. Here is a visual to help you further understand how adding 4 to both *H_min* and *H_max* works in this case. Left side shows the previous state of the game board and the right side shows the current state of the game board.

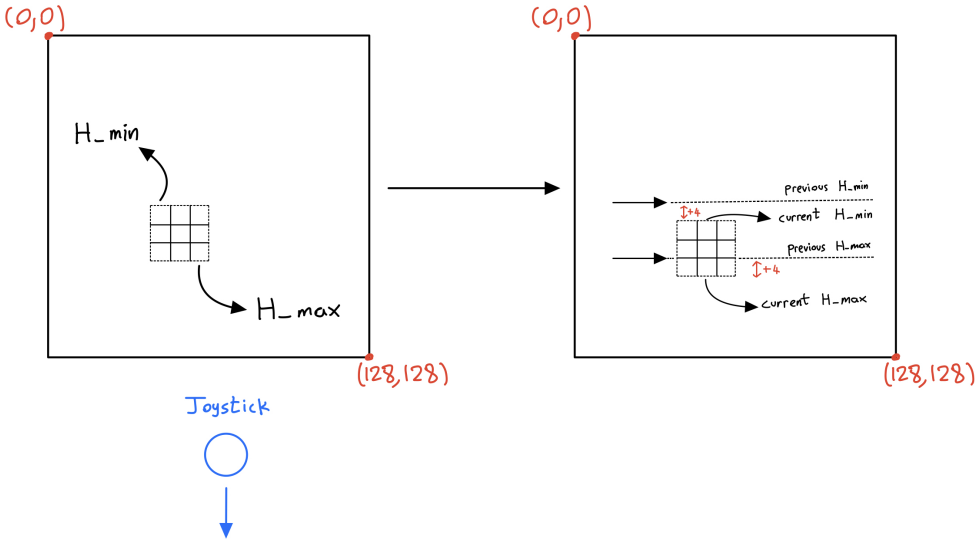

Figure 14.11: Board Reposition

Since both diagonal and straight movements are supported, we will have 8 movement cases to check for: bottom-right, top-right, bottom-left, top-left, left, right, bottom, and top. When checking which direction the user wants to move the game board, we need to give priority to diagonal moves since they include both x and y directions. For example, if we give priority to moving left by checking it first but the user wanted to move to bottom-left, it will move to left since the threshold for moving left was satisfied. We also need to make sure to avoid moving the game board out of the boundaries of the LCD display. For example, if the user wants to move the game board one unit across bottom-right, we want to check whether *W_max* and *H_max* are less than or equal to 128. If so, then we will move the game board accordingly. This will apply to all the other cases with different things to check for.

In Channel 0 ISR, add another *else if* statement that checks whether *JS_BoardReposition* is

set. Inside the brackets of the statement, include the following code which implements the board reposition feature and complete the missing parts.

```
// Move across to bottom-right.
if (ADC12MEM0 >= 3276 && ((W_max + 4) <= 128) && ADC12MEM1 <= 819 &&
    ((H_max + 4) <= 128))
{
     eraseBoard();
     W_max += 4;
     W_min += 4;
     H_max += 4;
     H_min += 4;
     updateCursorCoordinates();
     drawBoard();
}

// Move across to top-right.
else if (ADC12MEM0 >= 3276 && ((W_max + 4) <= 128) && ADC12MEM1 >=
    3276 && ((H_min - 4) >= 0))
{
     eraseBoard();
     W_max += 4;
     W_min += 4;
     H_max -= 4;
     H_min -= 4;
     updateCursorCoordinates();
     drawBoard();
}

// Move across to bottom-left.
else if (ADC12MEM0 <= ... && ((W_min - 4) >= ...) && ADC12MEM1 <= ...
    && ((H_max + 4) <= ...))
{
     eraseBoard();
     W_max -= 4;
     W_min -= 4;
     H_max += 4;
     H_min += 4;
     updateCursorCoordinates();
```

```
    drawBoard();
}

// Move across to top-left.
else if (ADC12MEM0 <= 819 && ((W_min - 4) >= 0) && ADC12MEM1 >= 3276
   && ((H_min - 4) >= 0))
{
    eraseBoard();
    W_max ...
    W_min ...
    H_max ...
    H_min ...
    updateCursorCoordinates();
    drawBoard();
}

// Move one unit towards left.
else if (ADC12MEM0 <= 819 && ((W_min - 4) >= 0))
{
    eraseBoard();
    W_max -= 4;
    W_min -= 4;
    updateCursorCoordinates();
    drawBoard();
}

// Move one unit towards right.
else if (... && ((W_max + 4) <= ...))
{
    eraseBoard();
    W_max += 4;
    W_min += 4;
    updateCursorCoordinates();
    drawBoard();
}

// Move one unit towards bottom.
else if (ADC12MEM1 <= 819 && ((H_max + 4) <= 128))
{
```

```
    eraseBoard();
    H_max ...
    H_min ...
    updateCursorCoordinates();
    drawBoard();
}

// Move one unit towards top.
else if (... && ...)
{
    eraseBoard();
    H_max ...
    H_min ...
    updateCursorCoordinates();
    drawBoard();
}
```

With the board reposition functionality added, we finally are done with implementing board resize/reposition features! Now that you have all the code needed, test these functionalities yourself. Make sure to initialize everything correctly to begin with, draw the board, enable global interrupts, put the board into an appropriate low power mode (or just an infinite loop at the end of the main function), and start testing the functionalities. Here are some of the things you should test to make sure everything works correctly:

1. Move the cursor around and make sure it works as expected
2. Push button 1 and check if the cursor is erased. Then resize the game board to be as small as possible and try to make it smaller to make sure nothing happens after a certain point. Then push the joystick button to set the game board size.
3. At this point, you should see the cursor blinking where you left it last. Push button 2 and check if the cursor is erased. Then reposition the game in every possible direction to make sure it moves correctly and in sync with the joystick's movements. Make sure to check all the possible cases where there could be certain bugs (like trying to go beyond the LCD display's boundaries). After that, position the game board close to the bottom-right corner of the LCD display. Then click the joystick button set the board position.
4. The cursor should start blinking again. Now push button 1 and try increasing the board size just like the left visual shown in Figure 14.10. The game board shouldn't bypass the

LCD display's boundaries at any condition. Push the joystick button to set the board size and check if the cursor starts blinking again.

You should play around with these features yourself to further test for bugs as well. After following all these steps, if no bugs were found, you are good to continue on with the next part. If there were some bugs that you encountered, then first double check the missing parts you wrote the code for. Note that y direction's polarity is reversed compared to the standard (going downwards is +y while going upwards is -y). If you are sure that you completed the missing parts with no errors, then check the control flow and make sure you are doing everything precisely (like clearing the interrupt flags at the correct time).

14.6 Making a Move

In this part, we will implement the functionality for the user to be able to make a move by simply pushing the joystick button. Along with that, we will implement drawing/erasing Xs/Os using cursor coordinates along with *cursorCenterOffset*, update game board resize/reposition features, and figure out a spot to place the cursor after a move is made.

Playing X

When the user plays X, we will draw X on the visual game board and update the logical game board. The letter 'X' consists of two diagonal lines. We will use the MSP Graphics Library function named *Graphics_drawLine* to draw an X in the desired spot. Take a look at the details of this function to understand its parameters. Since the cursors are already in the middle of the fields, we will use them to draw the diagonal lines. We know that the cursor is a small square. The middle of the square is the center of the field. We know the distance between the center and the square's edges: *cursorCenterOffset*. The idea is to have an algorithm that first finds the center of the field through the cursor's coordinates, and then offsets by some estimated amount to define the points of X without exceeding the field's boundaries. Observe the visual shown at the beginning of the following page.

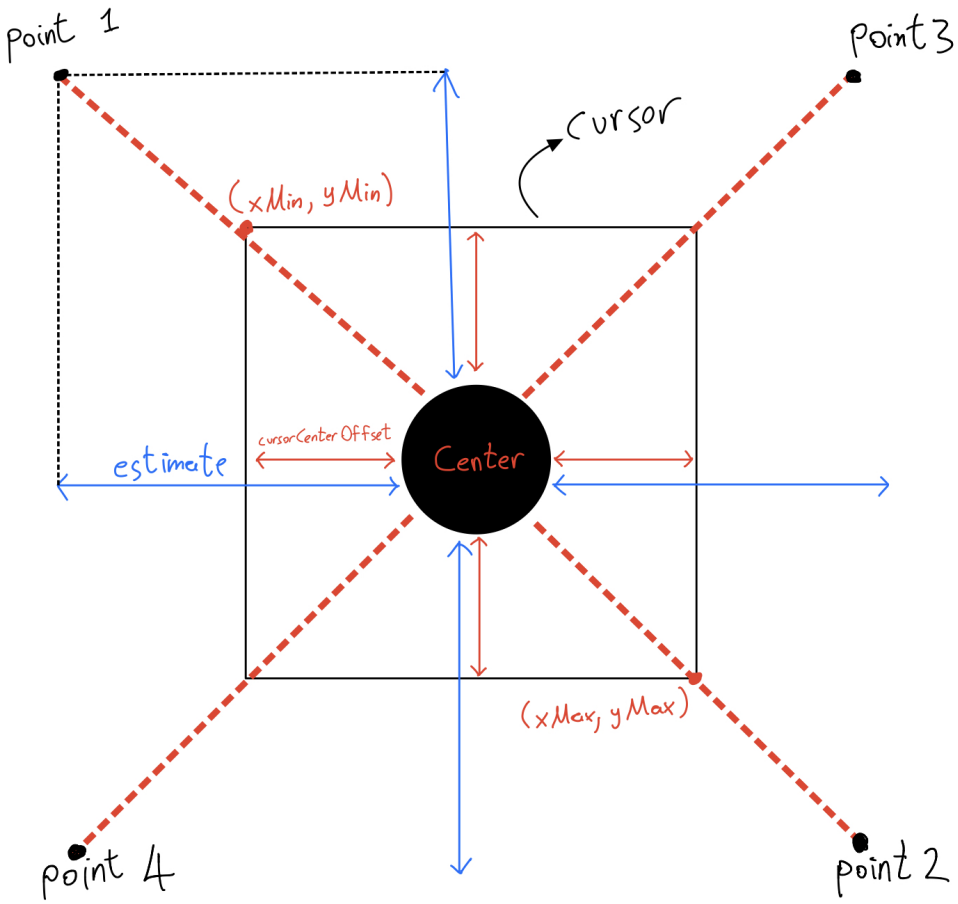

Figure 14.12: Algorithm for Finding Points of X

Take a look at how the x and y-coordinates of point 1 could be found in the figure. If you understand how the coordinates of one point is found, you will understand the process for others as well. Here is how to find the x-coordinate of point 1:

1. Add *cusorCenterOffset* to *xMin* coordinate of the cursor in order to get to the center
2. Generate an estimate value based on game board size and offset to the left by that amount

The same process applies for all the other points except we start from a different coordinate and the estimated offset is towards other directions. We will make our estimate by simply dividing the game board length by 10. You could choose a different value to divide the board length by. But in my experience, this value worked well for me. Note that we could find the board length using either *_max* and *W_min* or *H_max* and *H_min* since the board is a square. After drawing the X on the desired spot, we will update the logical game board (the *gameBoard* array) by assigning

the corresponding cell with the letter 'X'. The following is the function to play X. Observe the function and compare it to Figure 14.12 to further understand it. Then complete the missing parts and include it in your code.

```
/*Draws X in the specified field on the visual board and updates the
   logical board*/
void play_X(int index)
{
    // Estimate the offset from the center.
    volatile int estimate = (W_max - W_min) / 10;

    /*Below, the addition or subtraction of cursorCenterOffset is for
       centering purposes. Cursors' max and min coordinates are
       cursorCenterOffset away from the center*/
    Graphics_drawLine(&g_sContext,
    Cursor[index].xMin + cursorCenterOffset - estimate,
    Cursor[index].yMin + cursorCenterOffset - estimate,
    Cursor[index].xMax - cursorCenterOffset + estimate,
    Cursor[index].yMax - cursorCenterOffset + estimate);

    Graphics_drawLine(&g_sContext,
    Cursor[index].xMax - cursorCenterOffset + estimate,
    Cursor[index].yMin + cursorCenterOffset - estimate,
    Cursor[index].xMin ...,
    Cursor[index].yMax ...);
    gameBoard[index] = 'X'; // Update the logical game board
}
```

Playing O

The process of playing O is very similar to playing X with the only difference being that the function we will call is different: *Graphics_drawCircle*. As O is simply a circle, we will draw have a circle to represent it. The first 2 parameters of this function is the center point of the circle. In this case, the center of the circle will be the same as the center of the cursor. The third parameter is the radius which is found the exact same way estimate was found in *play_X* function. Given the information above, complete the missing parts in the following function and include it in your code. Make sure to use Figure 14.12 as a reference as it still will help with providing a visual.

```
/*Draws O in the specified field on the visual board and updates the
   logical board*/
void play_O(int index)
{
    volatile int radius = (W_max - W_min) / 10;

    Graphics_drawCircle(...);
    ... // Update the logical game board
}
```

Erasing X/O

We will write functions to erase Xs and Os for the purpose of erasing the game board faster while resizing/repositioning the board. At this point, you learned how to erase the cursors and the game board. The idea is to change the foreground color to background color, draw over whatever you want to erase, and then configure foreground color back to original. Erasing X/O functions therefore will be very similar to *play_X* and *play_O*. Here are the headers for the two erasing functions:

```
void erase_X(int index);
void erase_O(int index);
```

Write these two functions. Note that you shouldn't update the logical game board when erasing the Xs and Os. In other words, do not assign '0' to the cell after erasing an X or an O. These two functions will be used mainly for the board resize/reposition features. We want to be able to erase the entire board without losing the information about where the Xs and Os are. That way, we can draw the board accurately right after erasing it.

Updating Certain Functions

Currently, the board resize/reposition features do not support drawing and erasing Xs and Os due to the fact that *eraseBoard* and *drawBoard* functions do not have the necessary code. So, let's update these two functions!

drawBoard Function Update

We need an algorithm that goes through the logical game board and as soon as it sees an X or an O, it draws it on the visual game board. Take a look at the following code to understand how this algorithm is implemented in code. Then include it in an appropriate spot inside the *drawBoard* function.

```
// Draw Xs and Os if there are any.
unsigned volatile int i;
for (i = 0; i < 9; i++)
{
    if (gameBoard[i] == 'X')
        play_X(i);
    else if (gameBoard[i] == 'O')
        play_O(i);
}
```

Note that even though *play_X* function updates the logical game board by assigning a letter to a cell of the *gameBoard* array, it just overwrites the existing letter with the same letter.

eraseBoard Function Update

We will use the same algorithm we used for *drawBoard* for this function. The only difference will be that rather than drawing the Xs and Os, we will erase them. Given the algorithm, figure out how to erase the Xs and Os rather than drawing them. Then update the *eraseBoard* function accordingly.

Where to Locate the Cursor After a Move?

We will locate the cursor to the first available spot on the visual game board after each move. We need to write a simple algorithm that goes through each field of the game board from index 0 to 8 and places the cursor in the first available spot assuming that the game is still going on. Since we will check whether the game has a result after each move, it would be a good idea to locate the cursor to the first available spot inside *checkForResult* function. Take a look at the end of the *checkForResult* function where we are checking whether the game is still going on. We already go through each field of the game board in there with the help of a for loop. The if statement is true when the **first** available spot is found and the game is still going on. This meets all the requirements for when we should locate the cursor to first available spot. Therefore, inside that if statement, simply update the *cursorIndex* to the first available spot and draw the cursor before returning a 0. Do not worry about erasing the previous cursor before drawing the new one since we will take care of that somewhere else in the code.

Making a Move with the Joystick Button

As soon as the user pushes the joystick button to make a move, the PC will automatically go to Port 3 ISR. When the user makes a move, *JS_Cursor* indicator will be active since they move the cursor around before making a move. Just like the joystick, the joystick button also has multiple functionalities based on the active indicator. Therefore, we will check whether *JS_Cursor* is active in Port 3 ISR just like how we did for the other indicators. If it is active and the other indicators aren't, then we know that the user just made a move. Therefore, add an *else if* statement in the Port 3 ISR that checks whether *JS_Cursor* is on or not. Inside the brackets of the statement, we will do the following:

1. Erase the current cursor if it is on the game board
2. Update both the visual and logical game board with the move user just made (*play_X/play_O* functions do this for us)
3. Update the state of the game. In other words, update whose turn it is (*state* variable represents turns)
4. Check whether the game has a result (simply calling the *checkForResult* function should be fine for now)
5. Add a delay loop to wait for bouncing of the joystick button to end

Given the steps to follow, write the code. For #2, note that using the *state* variable could help you figure out whether an X or an O was played.

Once you are done with writing all the required code for this part, test the program and play the game using all the available features. When you press the joystick button, depending on *state*, X or O should appear in the desired spot and the cursor should start blinking in the first available field of the game board. When using the game board resize/reposition features, Xs and Os should be updating along with the game board itself. Note that we will write the code for what happens after the game is over in the next part; so do not test your program for what happens after the game is over. Just make sure that based on the result of the game, the correct LED(s) lights up.

Lastly, it is important to mention that when the game board length is 28 (second to last smallest size of the game board), the cursors at the last row and column are not centered precisely. This causes the drawn Xs and Os to be off-centered as well. Check it yourself as well to see what I am exactly talking about. I unfortunately didn't have enough time to fix this bug. My

first guess is that this is caused by integer division and the fact that as the board gets smaller, we need more precision since we have less space to work with. This suggests that there might also be a problem with the initial cursor coordinates and/or with the algorithm we chose to update the cursor coordinates. If you want to take on the challenge, try to fix this bug as it would be a great exercise. Make sure to test your idea thoroughly. Once you are sure that your solution fixes this bug, contact me at uygarubaran@gmail.com. I will greatly appreciate your input!

14.7 Playing the Game Continually

Currently, we can only play the game once until we have to reset the board and start a new game. We also aren't keeping a track of the scores in the game. We can only play one round and once the round is over, the LEDs indicate the result of the game. In this part, we will update the game so that after a round is over, the scores are shown on the LDC display (rather than through the LEDs) and a new game is started once the user presses the joystick button. In other words, in this part, we will be finalizing the human vs. human mode. When the game is over, here is what we will do in order:

1. Disable user control for certain functionalities
2. Erase the board
3. Update the score, show the result of the last game played, and show the overall scores of X and O
4. Wait for user to push the joystick button
5. Once the joystick button is pushed, prepare for the next game by resetting the game board and setting/resetting certain variables
6. Start the game by drawing the board and enabling user control

We already have the function to perform #2. Therefore, let's work on figuring out how to perform the other steps.

Disable/Enable User Control

The reason we want to disable user control when the game is over is to prevent users from trying to play the game when the game isn't even going. For example, if the user control is not disabled when the game is over, then the user could just place an X/O on the board when that's not supposed to happen. We will disable user control by simply disabling all the interrupt events.

While disabling global interrupt bit could work, to understand the control flow of the game, we will disable each interrupt ourselves. When the game is over and the scores are shown, the user will push the joystick button to start the next game (polling the flag). Complete the missing parts in the given function and include it in your code.

```
void disableUserControl(void)
{
    // Disable resize and reposition features (button1 and button2
       interrupts)
    ...

    ... // Disable ADC conversion interrupt
    ... // Disable cursor blinking interrupt

    ... // Disable joystick button interrupt
}
```

When enabling user control, we want to clear the interrupts flags, schedule the next interrupts, and enable the interrupt events. If we do not clear the flags, then the user's button push from a while ago (when interrupts were disabled) could cause an interrupt. Complete the missing parts in the given function and include it in your code.

```
void enableUserControl(void)
{
    ... // Clear button 1 and 2 interrupt flags
    ... // Enable button 1 and 2 interrupts

    ... // Clear JSbutton interrupt flag
    ... // Enable JSbutton interrupt

    ... // Clear Channel 0 flag
    ... // Clear Channel 1 flag

    ... // Schedule the next interrupt for ADC conversion event
    ... // Schedule the next interrupt for cursor blinking event

    ... // Enable ADC conversion interrupt
    ... // Enable cursor blinking interrupt
```

```
}
```

Resetting the Game

Before starting a new game, we will clear the entire display to erase the information about scores, set certain variables to their default values, and make all the spots in the logical game board available to use. Complete the missing parts in the given function and include it in your code.

```
// Prepares the program to start a new game.
void resetGame(void)
{
    volatile int i;
    ... Clear the display

    cursorIndex = 0; // Cursor location starts at 0
    cursor = 0; // cursor is off
    state = 0; // X's turn, X always starts first
    ... // Reset resize indicator
    ... // Reset reposition indicator
    ... // Set cursor indicator

    // Make busy spots available to players.
    for (i = 0; i < ???; i++)
    {
        gameBoard[i] = ...
    }
}
```

Starting the Game

To start the game, all we have to do for now is to draw the board and enable user control. Given the function header below, write the function and include it in your code.

```
void startGame(void);
```

Game Over Screen

We need a screen that shows who won the last game played (or shows that the game is drawn if that's the case), the overall scores, and a note stating that the user should push the joystick button to start the next game. When saving the overall scores, you can either use static variables inside the function we will update the scores in or simply have global variables for the scores. Let's say that X's score is currently 5, O's is 2, and O wins the next game. In this case, the game over screen should look something like this:

```
O won!

X: 5      O: 3

Press JS button to start the next game
```

The first line would say "X won!" if X won the game and "Draw!" if the game was drawn. If the result is a draw, the scores should be kept the same as before. Note that your design could look different than the design shown above. This is just a visual for you to get the idea. We will include all the steps listed in the beginning of this part (14.7) in a single function. This function will take a character as an input. If the character is 'X', it means X won. Else if the character is 'O', it means O won. Otherwise, the game is simply a draw. Complete the function below and include it in your code.

```
void gameOver(char ch)
{
    volatile unsigned int i;

    /*Declare static variables for scores of X and O only if you don't
        want to have global variables for them*/
    ...

    ... // Disable user control

    ... // Erase the game board

    /*Increment the score of the winning side and draw the game over
       screen*/
    ...
```

```
    // Clear the JSbutton flag right before polling the flag
    ...

    // Wait while the JSbutton isn't pushed
    while(...) {}

    // Delay loop to wait for bouncing to be over
    for (i = 20000; i > 0; i--) {}

    ... // Clear the JSbutton flag again

    ... // Reset the game (clears the display as well)
    ... // Start the game
}
```

Finalizing Human VS. Human Mode

Now that we have the a single compact function that allows us to play the game repeatedly, we need to figure out where to call it. Currently, we are indicating who won the game with the LEDs on the LaunchPad inside the *checkForResult* function. Since we are able to show the results on the LCD display now, the LEDs are no longer needed. Therefore, go ahead and make the correct adjustments to the *checkForResult* function to finalize the human vs. human mode. The idea is to call the *gameOver* function with a correct parameter in the places where we turn the LEDs on. If the game is drawn, you could pass in any character that doesn't indicate X or O winning the game. Your *gameOver* function should take care of this case. Lastly, in your main function, you could initially start the game by calling the *resetGame* and *startGame* functions before going into low-power mode/infinite loop. Once all the required code is written, test all the cases in the game (X wins, O wins, draw). Play at least 10 rounds to make sure that there are no bugs.

14.8 Human VS. Computer Mode

As the title suggests, in this part we will add the new mode where the user can play against the computer. For us to add this new mode into the game, we need to figure out an algorithm for the computer to use when playing against a human and then figure out how to add this feature into the control flow. Let's start with the algorithm.

Computer Algorithm

For each move the computer makes, it will simply choose a random available spot on the board. To implement this algorithm, here is what we need to do in order:

1. Go through the logical game board and save the indices of available spots in a different array.
2. Choose a random spot from the available spots but do not let the computer play just yet.
3. Create a small delay to give the impression that the computer thinks like a human.
4. Let the computer make its move.

Before we write the function that implements these steps, we need to create a variable to indicate whether the computer is X or O and have a random number generator. Create a global variable named *computer_x_or_o*. If this variable is 0, then the computer is X. If it is 1, then the computer is O. Note that this matches the cases of *state* variable. For the random number generator, first include *time.h* and *stdlib.h* header files in your code (they should be inside angle brackets). After that, include the following line of code somewhere towards the beginning of your main function:

```
srand(time(NULL));
```

If you haven't done this before and don't know about what this is, visit https://mathbits.com/MathBits/CompSci/LibraryFunc/rand.htm to learn about it. With the random number generator set up, all we have to do is to call *rand* function in order to generate a random number. Now that we have everything ready to follow the steps mentioned above, observe and understand the function below. Then complete the missing parts and include it in your code.

```
/*Lets the computer pick one spot to play through randomly selecting
   from the available choices.*/
void computerPlay(void)
```

```
{
    // Used for holding available spots in the game board.
    volatile int availableSpots[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};

    volatile int32_t i; // Game board index
    volatile int counter = 0; /*Counter of available spots and index
       of availableSpots array*/
    volatile int computerChoice; // Computer's spot choice

    // Check how many available spots there are.
    for (i = 0; i < ???; i++)
    {
        // If spot is available, save it in the availableSpots array
           and update counter.
        if (gameBoard[i] == ???)
        {
            availableSpots[counter] = i;
            counter++;
        }
    }

    /*If no spots are available, return. This function won't be called
        in the case where no spots are available, but it is good to
       be safe in case something goes wrong.*/
    if (counter == ???)
        return;

    // Pick one available spot randomly.
    computerChoice = availableSpots[ rand() % counter ];

    /*Create a small delay to give the impression that the computer
       thinks like a human.*/
    for (i = 100000; i > 0; i--) {}

    // If computer is X
    if (computer_x_or_o == ???)
        play_X(...);
    // If computer is O
    else
```

```
        play_O(...);
}
```

Addition to the Control Flow

Now that we have our function that contains the computer algorithm, we need to figure out how to integrate the new human vs. computer mode into the game. Currently, the program always waits for the user to make a move since the only mode available is human vs. human mode. We need a way to distinguish between the two different modes; we could do this with an indicator. Create a global variable named *computerMode*. If it is 0, then this means the computer mode is off (human vs. human mode is on). Vice versa for when *computerMode* is 1.

In both modes, we know that the human makes a move. Therefore, we will check whether the computer should make a move right after the human's move. In order for the computer to make a move, the computer mode should be on **and** it should be the computer's turn to make a move. If one of these conditions fail, then it will be the human's turn to play regardless of what mode we are in. With the help of these conditions, we will be able to integrate this new feature into the control flow. For example, if we are in human vs. human mode, the computer will never be able to play since *computerMode* variable is 0. If we are in human vs. computer mode, then the computer will only be able to play when it's the computer's turn with the help of *computer_x_or_o* variable. Once the computer makes a move, if the game is not over after the move, it will be the human's turn. Same thing goes for when the human makes a move.

In the Port 3 ISR (joystick button ISR), go to the if statement that checks whether the *JS_Cursor* indicator is on. If you followed the steps under the subtitle "Making a Move with the Joystick Button" in part 14.6 correctly, the last 2 things being done inside that if statement is calling the *checkForResult* function and a delay loop for debouncing purposes. We know that the *checkForResult* function returns 0 or 1 based on whether the game has a result. Therefore, rather than just calling the function, we will call it inside an if condition that checks whether the game is still going on. Inside the same if condition, we will also check whether the computer mode is on and if it's the computer's turn. If all these conditions are met, then we will do the following:

1. Disable the user control so that the user cannot interrupt the game while the computer is preparing to make a move.
2. Let the computer make its move.
3. Update the state of the game to indicate who will play next (*state* variable)
4. Check for the result of the game. If the game is still going, then enable the user control in

order for the human to make the next move.

Following these steps, complete the given code below. Then include it right under the line where you update the state of the game.

```
/*If the game is still going on, computer mode is on, and it is
   computer's turn to play...*/
if (checkForResult() == 0 && ... && computer_x_or_o == state)
{
    ... // Disable the user control

    ... // Let the computer make its move

    ... // Update state to indicate who plays next

    // If the game still is going on, enable the user control.
    if (checkForResult() == 0)
        ...
}
```

And just like that, the human vs. computer mode is finalized! Or is it? What if the computer is X (starts first)? Currently, we check if it's the computer's turn after the user makes a move. This means that in the current code, the computer cannot get started by itself; the user has to start it by making a move. Therefore, we need to write some code to make this tiny, dependent computer a grown, mature, and an independent individual who doesn't rely on humans! Since this problem only occurs when the computer has to start first, we need to solve this problem right at the beginning of the game after drawing the game board. This means we will do this inside the *startGame* function. So, right after drawing the game board inside the *startGame* function, we will do the following:

1. Check if computer mode is on and if it's the computer's turn to play
2. If so, let the computer play
3. Update the state of the game to indicate it's the human's turn
4. Place the cursor to the first available spot

Note that in the steps above, we didn't check for the result of the game since we know that this

is the first move of the game. Complete the following code based on the steps above and include it right after drawing the game board inside *startGame* function.

```
/*If computer mode is on and it's the computer's turn, then let the
   computer make the first move.*/
if (...)
{
    ... // Let the computer play

    ... // Update the state of the game
}

/*Place the cursor to the first available spot after computer makes
   its first move.*/
if (gameBoard[0] == 'O')
    cursorIndex = 0;
else
    cursorIndex = 1;
```

And just like that, the human vs. computer mode is now **ACTUALLY** finalized! If you do not believe me due to tricking you last time, try it yourself! Activate the computer mode, select whether the computer is X or O, and play against it! Make sure that you test the program trying out all the cases. Play against the computer when it's X multiple times, then do the same for when it's O. Lastly, play in human vs. human mode to make sure that the computer never interrupts the game.

14.9 Main Menu

We implemented all the modes and features in the game. The only thing left to do is to add a main menu into the game for when the user first starts the game. Along with giving a good introduction to the game, this will also solve the issue of having to set the game mode manually in the code. This part will be mostly on you; use your imagination and coding skills to create a main menu for the user! For example, you could start the game with a cool image using Image Reformer Tool that converts an image file to a C file. There are so many ideas, do whatever you wish! The following are the 2 things you should do in the main menu:

1. Ask the user which mode they want to play in and indicate what they have to do in order to select it. Suppose that human vs. human mode is activated by pressing button1 and human

vs. computer mode is activated by pressing button2. In the main screen, you would show something like ”Play against: human - left button, computer - right button”. You would simply poll the flags of the two buttons and then based on which one is pressed, set the correct game mode on (*computerMode* variable).

2. If the game mode chosen is human vs. human, after turning computer mode off, simply starting the game should be enough. If the human vs. computer mode is selected, however, we need another screen to ask the user whether they want to play as X or O against the computer. Based on the choice, you would assign the *computer_x_or_o* variable accordingly. Then the game would be ready to get started. Note that deciding whether to show certain screens could simply be arranged with the help of if statements.

For any button flag polling, make sure to clear their flags right before and after. If not cleared before, what could happen is that during the preparation process for displaying the next screen, the user could randomly press the buttons for no reason whatsoever. Then when the next screen is shown telling the user to push one of the buttons to select the desired outcome, their older press could be counted for the current selection. The reason to clear the flags right after the polling is self-explanatory at this point. Lastly, make sure that you clear the display when transitioning from one screen to another.

Now that you have an idea of the must-dos inside the main menu, you could go ahead and design your own main menu utilizing the MSP Graphics Library and possibly the different peripherals on the LaunchPad and the BoosterPack! Here is the function header:

```
void mainMenu(void);
```

If your code for the main menu design is pretty long, you could create helper functions that will be called inside this function. The idea is that the entire main menu will be embedded inside this function so that when we call it, main menu will be displayed on the LCD display. Lastly, check the main function example below to make sure that your main function does everything required for this project. Note that you do not need to change your main function to look **exactly** like this if yours already does what it’s supposed to do.

```
void main(void)
{
    volatile unsigned int i;

    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
```

```
PM5CTL0 &= ~LOCKLPM5; /*Disable GPIO power-on default high-
   impedance mode*/

srand(time(NULL)); // This is for random number generation.

// Configure the green and red LEDs.
...

/*Configure all the buttons used in this project (Do not enable
   their interrupts, startGame function does that already)*/
...

config_ACLK_to_32KHz_crystal();

// Configure SMCLK to 16 MHz
...

// ********************************************************

Crystalfontz128x128_Init(); // Initialize the display

// Set the screen orientation
Crystalfontz128x128_SetOrientation(0);

// Initialize the context
Graphics_initContext(&g_sContext, &g_sCrystalfontz128x128);

// Set background and foreground colors
Graphics_setBackgroundColor(...);
Graphics_setForegroundColor(...);

// Set the default font for strings
GrContextFontSet(...);

// Clear display before starting
Graphics_clearDisplay(&g_sContext);

// ********************************************************
```

```
    Initialize_ADC(); // Initialize ADC to be able to use the joystick

    // Start Timer_A0: ACLK, div by 1, continuous mode, clear TAR
    ...

    resetGame(); // Reset the game

    /*Display the main menu for the user to select which game mode to
       play in*/
    mainMenu();

    startGame(); // Start the game

    // CPU off (enables global interrupt as well)
    _low_power_mode_0();
}
```

Once everything is completed, make sure to test the finalized game thoroughly. Note that when you compile and run the code, while it works in debug mode, the program might initially freeze when run in normal mode. At least that's what happened in my experience. If this happens to you as well, simply reset the board by pushing the top-left button on the LaunchPad (sometimes you need to hold it for a while). It should work in normal mode after the reset.

IMPROVEMENT IDEAS

If you have come this far, congratulations on finishing this project! Just like any other "finalized" project, this one still has a lot to work on if you are looking to improve it. Here are some ideas to improve this project:

1. Dividing up the main.c file into multiple files. This is one of the standard coding practices. Normally, main.c file should only contain the main function and the inclusion of the required header files. For this project, logical and visual functions could be put into different files. Then a header file could be created for those files and included in the main.c file right before the main function.
2. Additional game modes such as playing up to 3 or 5. Currently, the game goes on forever.
3. Users being able to switch sides after each round. Currently, if a user chose to be X, he/she can only play as X until the game is restarted.
4. Showing the scores on the LaunchPad's LCD display.
5. Adding difficulty levels to the human vs. computer mode. The current computer algorithm is considered easy since the computer simply picks a random available spot. There could be a hard mode where the computer picks the most optimal spot to make its move. Look into Minimax algorithm for this.
6. Playing music and sound effects through the Piezo buzzer located on the BoosterPack.
7. Controlling the game through speech recognition using an appropriate library and the microphone located on the BoosterPack.

There are many more ideas one can think of. Use your imagination and come up with new ideas yourself if you hope to improve this project! If you happen to implement any of the listed ideas/an idea of your own and want to showcase it to me, you could always contact me at uygarubaran@gmail.com. I am looking forward to seeing your upgrades on this project!