Experiment 2 - Lab Notes

Mathieu Perreault (260158758)
Logan Smyth (260179735)

Functional Specifications

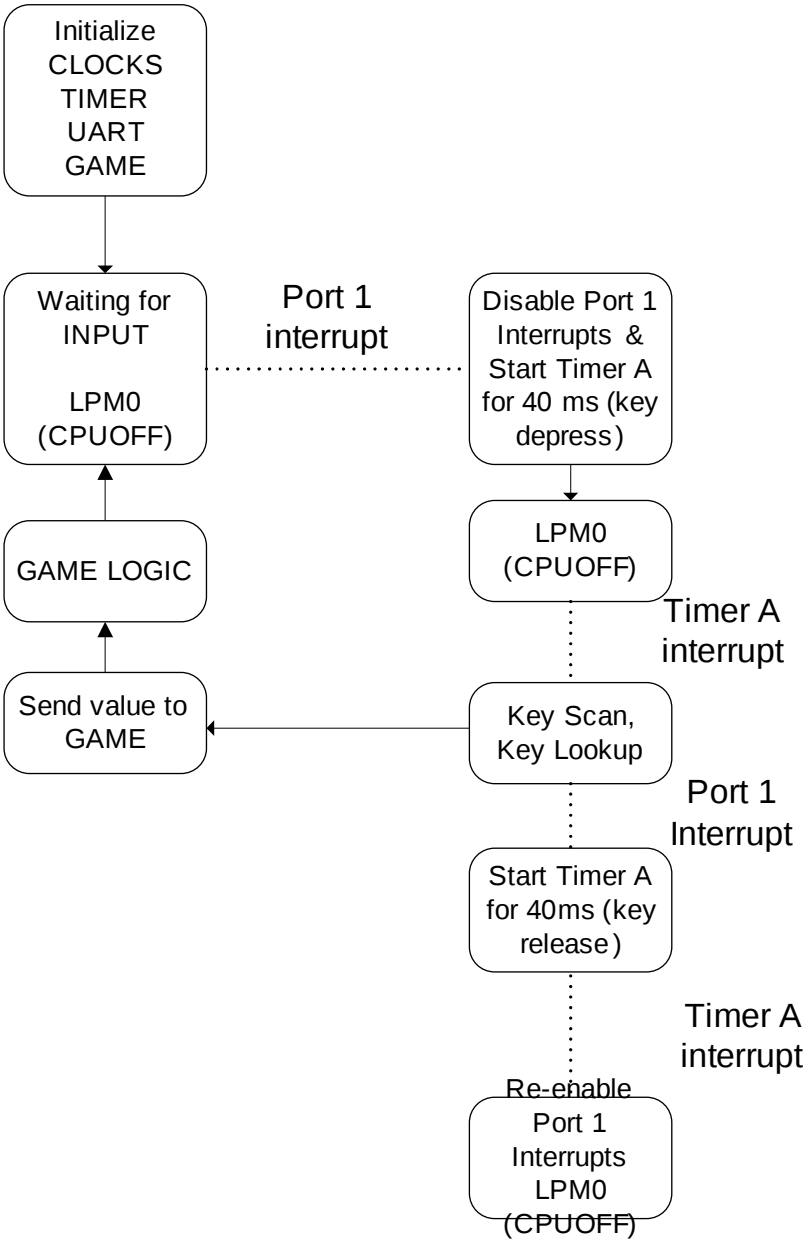
According to Experiment 2 specifications, our system should be able to run on the MSP430 evaluation board, commonly called the McGill McGumps board.
The system should implement a simple memory game based on a grid and cells. A grid of 8 cells by 8 cells in which an ASCII character (from a pair of two in the grid) will be placed for a total of 64 characters and 32 character pairs. The output of our program will be using the serial communication link to a terminal via the UART interface. The input of the system will be an external keypad with keys 0-9#*. Once the user enters a choice to try and reveal an identical pair, the game might reveal the identical pair if it is the case, or show the incorrect pair for 0.5s. The player will then continue to uncover all the right pairs in the grid. The game finished when the player has found all the 32 pairs.

Details:

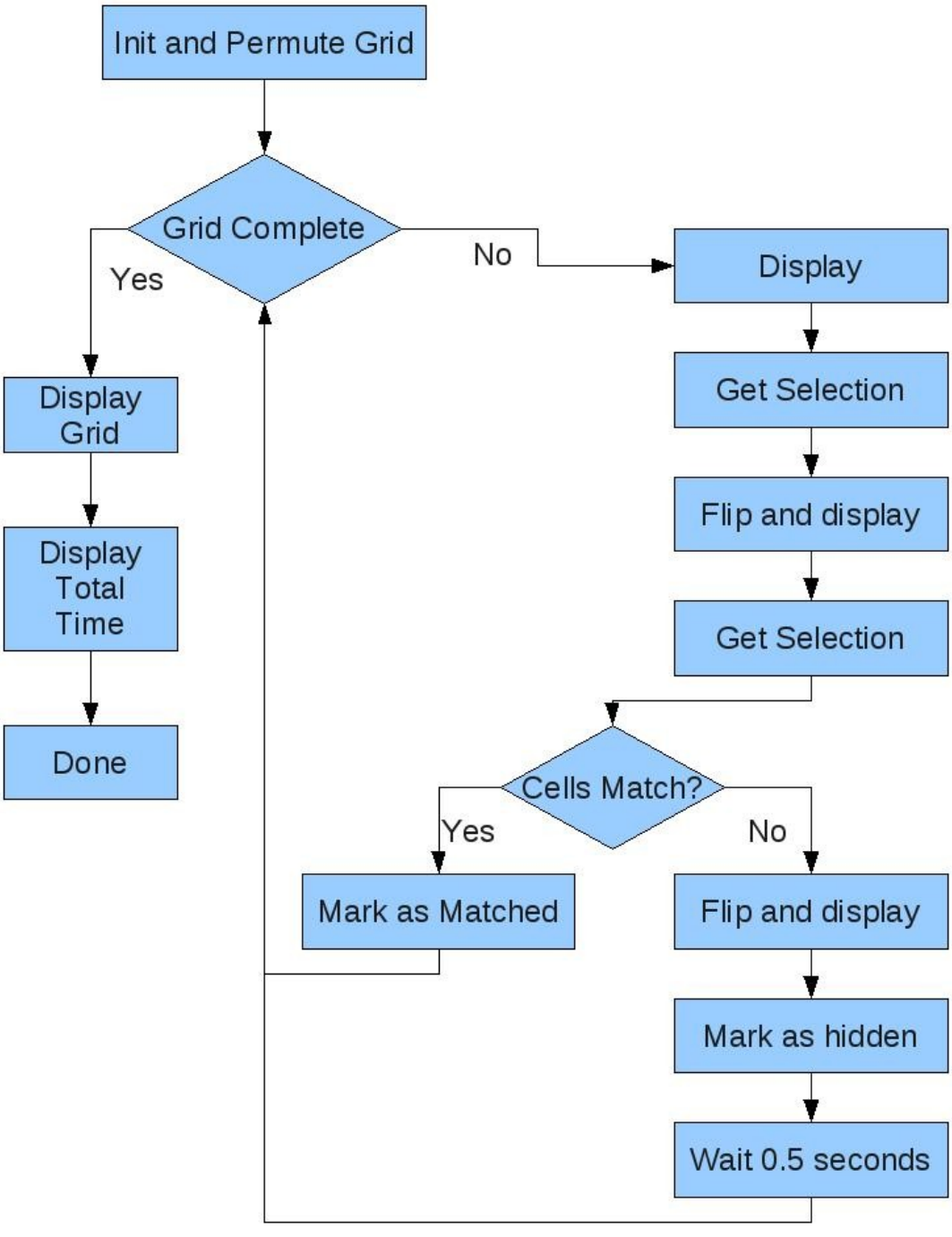
- The user interface should be intuitive.
- The UART parameters will be a 57600 baud rate, no parity bit, 8 data bits and 1 stop bit.
- A timer should be implemented to keep track of the game duration. When the player wins, the program will display the number of seconds the game lasted.
- The coordinates will be entered using the keypad (as mentioned) in the following manner: the user will hit ROW, COLUMN and # to confirm their choice. For example, hitting 0,2,# will uncover the value at row 0 under column 2. If the location is already visible to the player, the game will react accordingly and new coordinates will be allowed.
- The keypad is implemented using the interrupt functionality of McGumps. The board should be autonomous (not connected to a programmer).

Flow Diagram

Input flow



Game Logic


Init and Permute Grid
Grid Complete
Yes
No
Display Grid
Display Total Time
Done
Display
Get Selection
Flip and display
Get Selection
Cells Match?
Yes
No
Mark as Matched
Flip and display
Mark as hidden
Wait 0.5 seconds

Implementation Notes

The UART has been implemented using the UART framework in the MSP430F149. We did not implement it using the interrupt vectors at our disposal. We connected the UART clock to SMCLK, which was in turn connected in turn to the main XT2 oscillator at 8MHz via the `BCSCTL1 and BCSCTL2 registers`. We used a website[1] to calculate the different divider and modulation values for the 8MHz to have a baud rate as close as possible to 57600 bps. We set the correct direction and use for P3.4 (the UART TX pin in the system). To send a character, the system would wait until the UART interrupt flag was not enabled and then put the character on the TX buffer.

Both Timers A and B were used in this program for different purposes. Timer A was used both for debouncing the keypresses and for waiting 0.5 seconds when incorrect values were uncovered in the game. To initialize the timer A (or B), we set TACTL to accept the ACLK (at 32.768KHz) and enabled interrupts on the timer. We started Timer B immediately to record the game time and let Timer A off until it was needed.

Since we enabled the interrupts in the system, we wired the columns of the keypad to pins 1,2,3 of Port 1 and set P1DIR to IN. When a key is needed, the CPU is put to sleep, where is waits to be woken up after a keypress is processed.When a key is pressed, the system automatically enters the Port 1 interrupt service routine and checks the value of P1.1, P1.2, P1.3. In the case that the event is a key depress, the timer is started for a period of 40ms to debounce the key. When the timer A ISR kicks in, it checks the value of the key again to see if the key is still pressed. While the key is still depressed it turns all the rows off (connected on port 3) and turns them back on again one at a time to see which row affects the column currently getting checked by the ISR. This way, the program determines which key has been depressed. When the finger releases the key, we do the debounce again of 40ms with the same Timer A and then wake the CPU from it's sleep state.
To clear the screen we send character 0xc (form feed) to the terminal.

For the game implementation, there were seveal options for overall architecture. Instead of a state machine, which provides greater scalability for larger systems, we opted to use a simple loop to keep things simple. Our primary data structure is an array of “character” structs. This struct contains a char and a flag to track it's status (hidden, flipped or matched). The struct is filled with pairs of letters two cells at a time, and after that, it is randomly permuted. To generate a random seed, we ask the user to “press any key” and then base the seed on the time that it took them to press it.

We then have a method for displaying the grid via UART or debug_putchar depending on a flag and a method for retreiving a coordinate set that is both in range, and not already flipped or matched. With these functions, we contruct the simple loop which displays, retrieves one set of coordinates, flips the cell, dispays again, retrieves again, and then will either flip and wait 0.5 seconds, or make both cells permanently visible and start the next term. At the start of the turn, it checks if less that 2 cells remain unselected and exits of this is the case.

We have also implemented logic to only allow valid input and alerts the user if invalid input is entered. These cases are covered in the next section, so I will simply list our coverage here. Error messages are presented in the case of only entering one digit, entering an out of range value, a previously visible cell or the cell previously selected in the current turn.

[1] `http://www.daycounter.com/Calculators/MSP430-Uart-Calculator.phtml`

Performance Testing

Input Sequence	Expected Parsed Input	Real Parsed Input	Effect on game
00#	00#	00#	Reveal cell at row 0 - column 0
99#	99#	99#	Print “99#” is out of range
00# (duplicate)	00#	00#	Print”Already Flipped”
1234#	34#	34#	Reveal cell at row 3- column 4
2#	2#	2#	Print “Enter 2 digits”
00#	00#	00#	Print “Already Matched” if 00# and 34# were same before

Table 1 – Some test cases to test behaviour of our system.
Note: these assume they were entered in this order

Appendix – Code

```
#include <msp430x14x.h>
#include <stdlib.h>
#include <cross_studio_io.h>


// Use debug_getchar for testing
//#define DEBUG_INPUT

// Use debug_putchar for testing
//#define DEBUG_OUTPUT

#define TRUE        1
#define FALSE       0

#define GRID_HEIGHT 3
#define GRID_WIDTH  3
#define GRID_SIZE   (GRID_WIDTH*GRID_HEIGHT)
#define co(w,h)     (h*GRID_WIDTH + w)

#define STATUS_HIDDEN     0
#define STATUS_FLIPPED    1
#define STATUS_MATCHED    2

//Struct for grid cells
typedef struct {
  char c;
  char stat;
} character;

//Struct for game cell selection
typedef struct {
  int row;
  int col;
} selection;

char keyValue = '0';    //Global for currently pressed key from ISR
int timeSeconds = 0;    //Global for current game time
int timerCounter = 0;    //Global for ISR counter for total time

void permuteGrid(character *grid, int seed);
void initGrid(character *grid);
selection getSelection(void);
selection getHiddenSelection(character *grid);
int checkComplete(character *grid);
char keyLookup(int row, int col);
char getChar(void);
void putChar(char a);
void putString(char* string);
void initUART(void);
void clearScreen(void);
void keyScan(void);
void initTIMER(void);
```

```
void initBOARD(void);
void wait(int ticks);
void makeFaces(character *grid);

void main(void) {
  int i, seed=24;
  int done = FALSE;
  character grid[GRID_SIZE];
  character *c1, *c2;
  selection s1, s2;
  char timeString[6]; // if you play for more that 99999 seconds, you need to take
a break anyway

  // Stop watchdog.
  WDTCTL = WDTPW + WDTHOLD;
  _EINT();

  //init UART stuff
  initBOARD();
  initUART();
  initTIMER();

  //Direction
  P1DIR &= ~(BIT1|BIT2|BIT3);
  P1SEL &= ~(BIT1|BIT2|BIT3);

  P3DIR |= BIT0|BIT1|BIT2|BIT3; //OUT
  P3SEL &= 0xF0;
  //Set all P3 high
  P3OUT |= 0x0F;             //BIT0|BIT1|BIT2|BIT3;

  P1IFG = 0x0;
  P1IE = BIT1|BIT2|BIT3;
  P1IES &= ~(BIT1|BIT2|BIT3);

  clearScreen();

  putString("Welcome to a new game, enter row and column and press # to confirm!\r\
n");
  putString("Press any key to start the game now\r\n");
  getChar();

  seed = timerCounter;
  timerCounter = 0;
  timeSeconds = 0;

  initGrid(grid);
  permuteGrid(grid, seed);

  while ( checkComplete(grid) == FALSE) {

    displayGrid(grid);

    s1 = getHiddenSelection(grid);
    c1 = &grid[co(s1.col, s1.row)];
```

```
    c1->stat = STATUS_FLIPPED;

    displayGrid(grid);

    s2 = getHiddenSelection(grid);
    c2 = &grid[co(s2.col, s2.row)];
    c2->stat = STATUS_FLIPPED;


    if (c1->c == c2->c) {
      c1->stat = STATUS_MATCHED;
      c2->stat = STATUS_MATCHED;
    }
    else {
      displayGrid(grid);
      c1->stat = STATUS_HIDDEN;
      c2->stat = STATUS_HIDDEN;

      putString("Incorrect\r\n");
      wait(16384);   //Wait for 0.5 seconds
    }
  }
  makeFaces(grid);
  displayGrid(grid);

  putString("You beat this game in ");
  itoa(timeSeconds, timeString, 10);
  putString(timeString);
  putString(" seconds!\r\nPress a key to start a new game");

  getChar();    //wait for a keypress so you can actually SEE the final results :)
  return;
}


/**
 * Initialize the grid with pairs of characters
 */
void initGrid(character *grid) {
  //72 characters, enabling a memory game of up to 12 by 12 (144 cells)
  char* chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789@#$
%&\"?!<>";
  char ch;
  int w,h;
  for (h = 0; h < GRID_HEIGHT; h++) {
    for (w = 0; w < GRID_WIDTH; w++) {
      grid[co(w,h)].stat = STATUS_HIDDEN;
      grid[co(w,h)].c = chars[(co(w,h))/2]; //copy letters into grid, each letter
twice
    }
  }
}
```

```c
/**
 * Permutes a given grid randomly
 *
 * To accomplish this, we pick two random slots and swap them, as mentioned in the
lab
 */
void permuteGrid(character *grid, int seed) {
  int i;
  int rand1, rand2;
  char temp;
  srand(seed);

  for (i = 0; i < GRID_SIZE/2; i++) {
    rand1 = rand() % GRID_SIZE;
    rand2 = rand() % GRID_SIZE;

    temp = grid[rand1].c;
    grid[rand1].c = grid[rand2].c;
    grid[rand2].c = temp;
  }
}

/**
 * Convert all hidden elements to smiley faces
 *
 * The most important part
 */
void makeFaces(character *grid) {
  int i;

  for (i = 0; i < GRID_SIZE; i++) {
    if (grid[i].stat == STATUS_HIDDEN) {
      grid[i].c = 0x2;
      grid[i].stat = STATUS_MATCHED;
    }
  }
}

/**
 * Loop over grid, and display with column labels
 */
void displayGrid(character *grid) {
  int w,h;
  char ch;
  character curChar;

  clearScreen();
  putChar(' ');
  for (w = 0; w < GRID_WIDTH; w++) {
    putChar('0'+w);
    putChar(' ');
  }
  putChar('\r');
  putChar('\n');
```

```
  for (h = 0; h < GRID_HEIGHT; h++) {

    putChar('0'+h);

    for (w = 0; w < GRID_WIDTH; w++) {
      curChar = grid[co(w,h)];
      if (curChar.stat == STATUS_HIDDEN) {
        ch = '*';
      }
      else {
        ch = curChar.c;
      }
      putChar(ch);
      putChar(' ');
    }
    putChar('\r');
    putChar('\n');
  }
}

/**
 * Get a valid selection
 *
 * Get a selection that is within range, then check the cell's status
 */
selection getHiddenSelection(character *grid) {
  int done = FALSE;
  selection sel;
  while(!done) {

    sel = getSelection();

    if (grid[co(sel.col, sel.row)].stat == STATUS_MATCHED) {
      putString("Already Matched\r\n");
    }
    else if (grid[co(sel.col, sel.row)].stat == STATUS_FLIPPED) {
      putString("Already Flipped\r\n");
    }
    else {
      done = TRUE;
    }
  }

  return sel;
}
```

```
/**
 * Get a selection that is in range of the grid
 */
selection getSelection(void) {
  selection sel;
  char c;
  sel.row = -1;
  sel.col = -1;

  while (TRUE) {
    c = getChar();
    if((c < '0' || c > '9') && c != '#') continue;

    // If pound selected, check range of input
    if(c == '#') {
      if(sel.row >= 0 && sel.row < GRID_HEIGHT && sel.col >= 0 && sel.col <
GRID_WIDTH) {
        break;
      }
      else {
        //Output information about input
        if(sel.row == -1 || sel.col == -1) {
          putString("Must enter at least two digits\r\n");
        }
        else {
          putChar('0'+sel.row);
          putChar('0'+sel.col);
          putString("# is out of range\r\n");
        }
      }
    }
    else {
      sel.row = sel.col;
      sel.col = c-'0';
    }
  }
#ifdef DEBUG_INPUT
  getChar();  //Clear new line after #, if reading from PC debug
#endif

  return sel;
}

/**
 * Check if the game has been completed by looping over grid
 *
 * If there is more than one empty spot, game is incomplete.
 */
int checkComplete(character *grid) {
  int i, count = 0;

  for (i = 0; i < GRID_SIZE; i++) {
    if (grid[i].stat == STATUS_HIDDEN) {
      count++;
      if(count > 1) return FALSE;
```

```
    }
  }

  return TRUE;
}


/**
 * Interrupt handler for PORT1
 *
 * When a key is pressed/released, start the timer and disable interrupts on port
one
 */
void keyPressed (void) __interrupt [PORT1_VECTOR] {
  P1IE &= ~(BIT1|BIT2|BIT3);        //turning off interrupts
  P1IFG = 0x0;                      //clearing the interrupt flag
  TACCR0 = 1311;                     //should be 0.040 s * (1/32768)
  TACTL |= MC1;                     //start the Timer in the up mode
}

/**
 * Iterrupt handler for TimerA
 *
 * Handles events for keypress debounce and wait timer
 * If wait timer, it simply start the CPU again, as the time has passed
 * If keypad, check keypad status and if needed, swap interrupt polarity
 */
void timerA_wakeup (void) __interrupt[TIMERA0_VECTOR] {
  //TACTL &= ~(TAIE | TAIFG);       //disabling timer A interrupt and Timer A
Interrupt Flag
  //debug_printf("%d\n",TACCR0);
  if (TACCR0 == 1311){
    if((P1IN & (BIT1|BIT2|BIT3)) && !(P1IES & 0x0E)) { //p1.1, p1.2, p1.3 pressed
in, low to high interrupts
      keyScan();
      if(P1IN & (BIT1|BIT2|BIT3)) {
        P1IES |= BIT1|BIT2|BIT3;          //set high to low interrupts
      }
    }
    if(!(P1IN & (BIT1|BIT2|BIT3)) && (P1IES & 0x0E)) {
      P1IES &= ~(BIT1|BIT2|BIT3);          //set low to high interrupts
      _BIC_SR_IRQ(CPUOFF);  //Wake up
    }

    P1IFG = 0x0;
    P1IE |= BIT1|BIT2|BIT3;
  }
  else {
    _BIC_SR_IRQ(CPUOFF);
  }

  TACTL &= ~(MC1 | MC0);  // Stopping the counter
  TACTL |= TACLR;          //clearing the counter
}
```

```
/**
 * Increment second counter to keep track of game status
 *
 * Fires ever 256/32768 seconds, and adds to timer.
 * Also imcrements timerCounter for use as a random seed
 */
void timerB_wakeup (void) __interrupt[TIMERB0_VECTOR] {
  timerCounter++;
  if(timerCounter & 0x80) { // every (128*256/32768) = 1 second
    timeSeconds++;
    timerCounter = 0;
  }
  TBCTL |= TBCLR;
}

/**
 * Wait a given number of clock ticks by using TimerA
 *
 */
void wait (int ticks){
  TACCR0 = ticks;
  TACTL |= MC1;
  _BIS_SR(CPUOFF);
}


/**
 * Scan keypad to determine which key is pressed
 */
void keyScan(void){
  int row, col;

  P3OUT &= ~(BIT0|BIT1|BIT2|BIT3);

  for (row = 1; row <= 8; row <<= 1) {
    P3OUT &= 0xF0;
    P3OUT |= row;
    for (col = 2; col <= 8; col <<= 1) {
      if (P1IN & col) {
        keyValue = keyLookup(row, col);
        P3OUT |= (BIT0|BIT1|BIT2|BIT3);
        return;
      }
    }
    P3OUT |= (BIT0|BIT1|BIT2|BIT3);
  }
}
```

```
/**
 * Look up a given key value based on a *bitwise* port value
 */
char keyLookup(int row, int col) {
  switch(row) {
    case 0x1: //P3.0
      switch(col) {
        case 0x2: //P1.1
          return '*';
        case 0x4: //P1.2
          return '0';
        case 0x8: //P1.3
          return '#';
      }
    case 0x2: //P3.1
      switch(col) {
        case 0x2: //P1.1
          return '7';
        case 0x4: //P1.2
          return '8';
        case 0x8: //P1.3
          return '9';
      }
    case 0x4: //P3.2
      switch(col) {
        case 0x2: //P1.1
          return '4';
        case 0x4: //P1.2
          return '5';
        case 0x8: //P1.3
          return '6';
      }
    case 0x8: //P3.3
      switch(col) {
        case 0x2: //P1.1
          return '1';
        case 0x4: //P1.2
          return '2';
        case 0x8: //P1.3
          return '3';
      }
  }
  return '0';
}

/**
 * Get a character from the keypad or debug input
 */
char getChar(void) {
#ifdef DEBUG_INPUT
  return debug_getchar();
#else
  _BIS_SR(CPUOFF);
  return keyValue;
#endif
```

```
}

/**
 * Output a string using putChar
 */
void putString(char* string){
  for(;*string != '\0'; string++) {
      putChar(*string);
  }
}

/**
 * Output a character using either the UART or debug output
 */
void putChar(char a){
#ifdef DEBUG_OUTPUT
  debug_putchar(a);
#else
  while ((IFG1 & UTXIFG0) == 0);
  TXBUF0 = a;
#endif
}

/**
 * Clear the screen using a form-feed character
 */
void clearScreen(void){
    putChar(0xc);
    //putString("\x1B[2J");
}


/**
 * Initialize the UART connection
 *
 * Wait for faults to clear
 * Select the SMCLK for use by UART
 * Set transfer of 8bits, 57600baud, no parity
 * Enable the module
 * Set pin directions
 */
void initUART(void){
  unsigned int i;
  do
    {
    IFG1 &= ~OFIFG;                         // Clear OSCFault flag
    for (i = 0xFF; i > 0; i--);             // Time for flag to set
    }
  while ((IFG1 & OFIFG) != 0);          // OSCFault flag still set?

  //Select SMCLK clk for UART (page 252 of family guide)
  U0TCTL = SSEL1|SSEL0;

  // 8-bit characters
  UCTL0 = CHAR;
```

```
  // Set baud rate 67.5 kbps @ 8MHz
  // Calculated here: http://www.daycounter.com/Calculators/MSP430-Uart-
Calculator.phtml
  /* uart0 8000000Hz 57595bps =~ 8000000/(0x8A.0x00) */
  UBR00=0x8A; //setting the baudrate divider
  UBR10=0x00;
  UMCTL0=0xEF;

  //Enabling the UART0 TX Module
  ME1 |= UTXE0;

  P3SEL |= 0x10;  // P3.4 = USART0 TXD
  P3DIR |= 0x10;  // P3.4 output direction
}

/**
 * Initialize the timers
 *
 * Select ACLK(32.768KHz) for use by timers
 * Enable interrupts
 * Stop timerA
 * Initialize TimerB to interrupt every 256 ticks for use by the timer tracker
 */
void initTIMER(void){
  TACTL = TASSEL_1 + TACLR;              // ACLK (32.768KHz), clear TAR
  TACCTL0 = CCIE;                       // CCR0 interrupt enabled
  TACTL &= ~(MC1 | MC0);                // Stopping the counter

  TBCTL = TBSSEL_1 + TBCLR;       //Use ACLK (32.768KHz), clear TAR
  TBCCTL0 = CCIE;
  TBCCR0 = 256;
  TBCTL |= MC1;
}

/**
 * Initialize XT2 clk
 *
 * Enable XT2CLK
 * Direct SMCLK = MCLK = XT2CLK
 */
void initBOARD(void){
  // Turning 8MHZ Oscillator ON
  BCSCTL1 &= ~XT2OFF;
  // Route XT2 into SMCLK and MCLK
  BCSCTL2 |= SELM1|SELS;

}
```