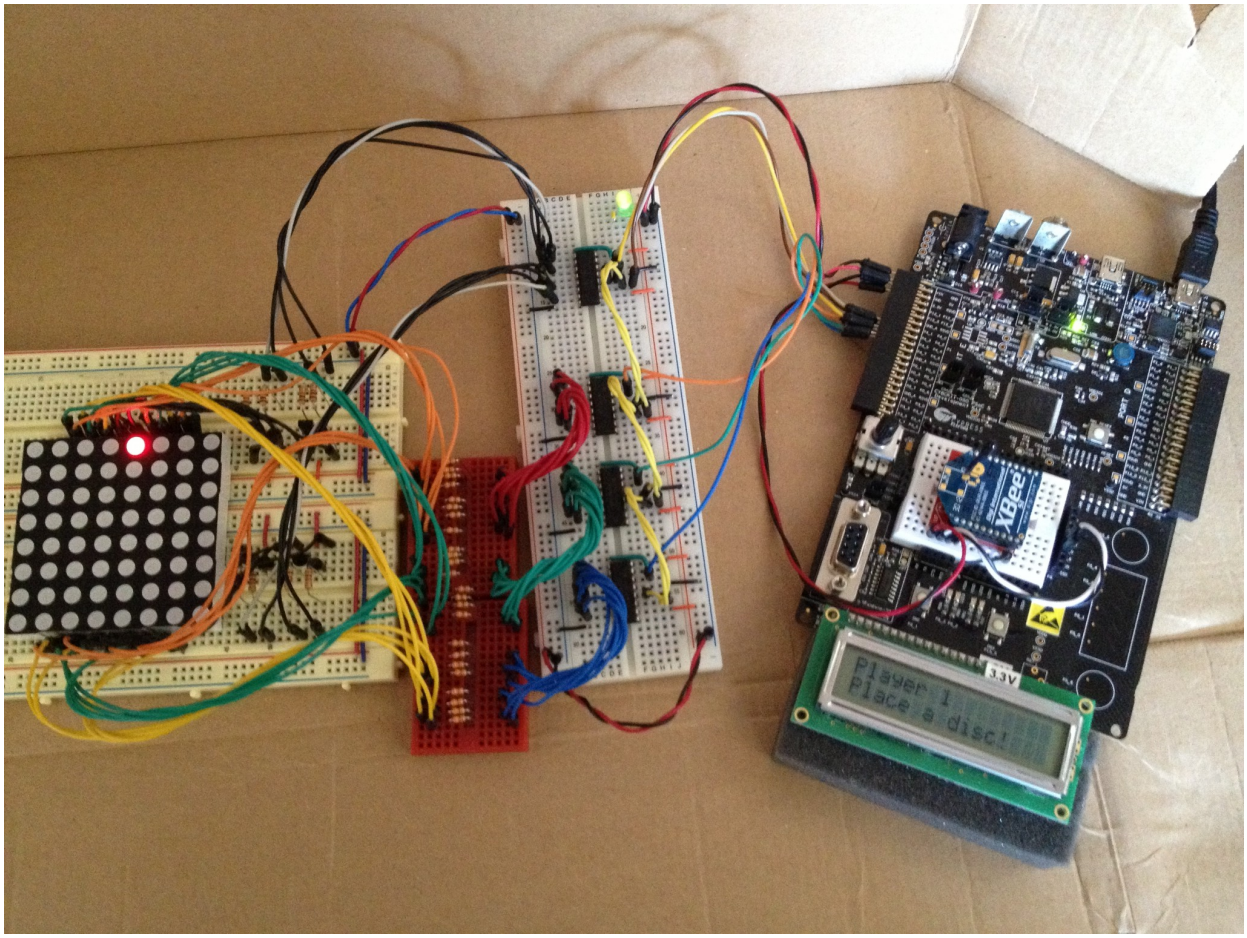
Connect 4 on a Microcontroller

Microprocessing System Design



Description

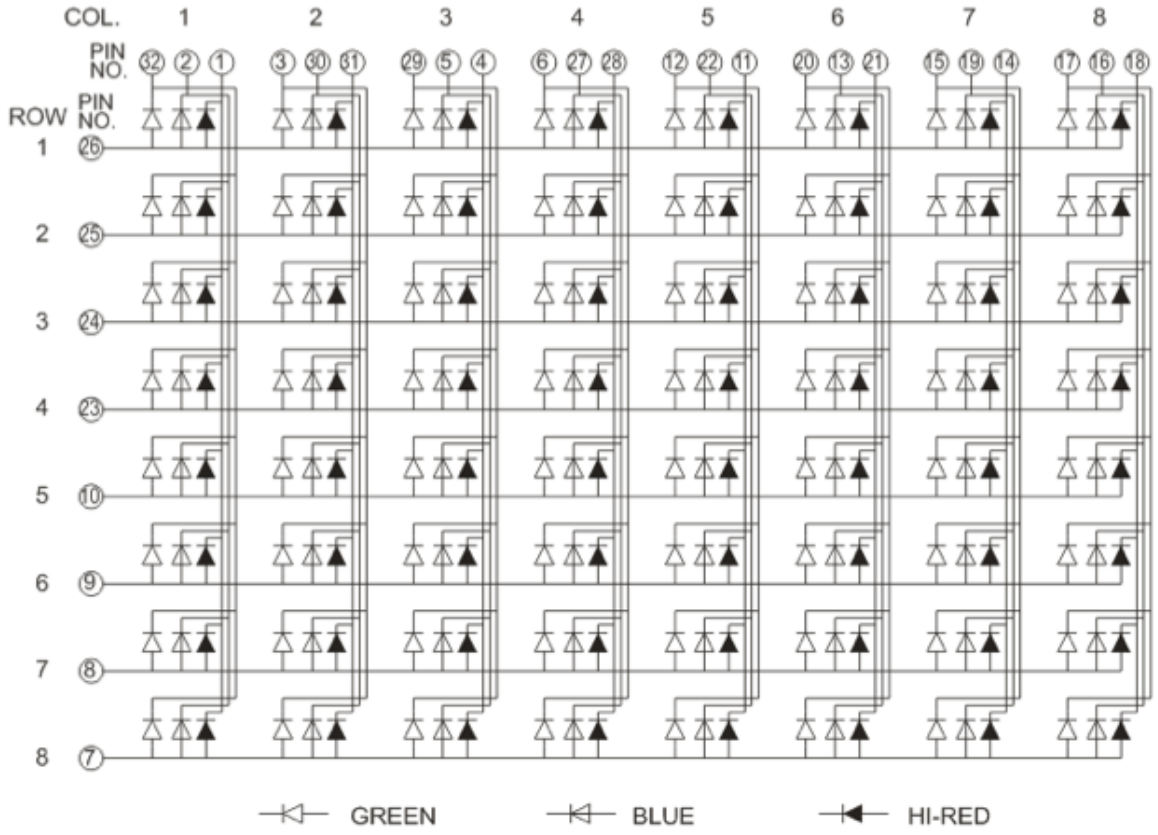
An implementation of the classic Milton Bradley game Connect Four using a Cypress PSoC-5LP microcontroller, an 8x8 RGB LED matrix, and an XBee wireless radio

Design

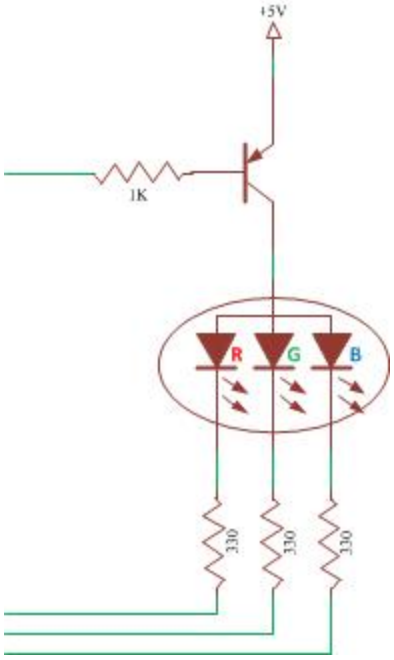
LED Matrix

Hardware

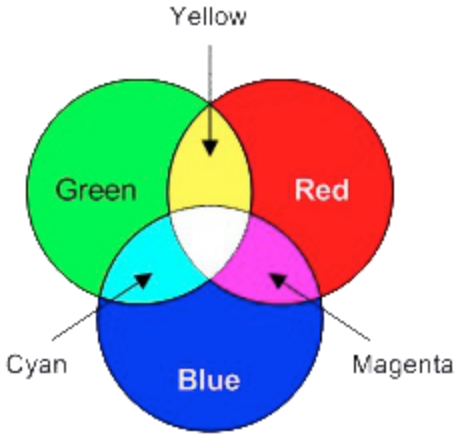
I began by wiring the LED matrix using a parallel controlling scheme (the PSoC controlled each row and column). Using the schematic below, I could tell that the current that would power the LEDs would be sourced from the rows (anodes) and sinked by the desired column (cathodes). Since there are only 8 anodes (left) as opposed to 24 cathodes (top), I added 8 current-limiting, 330Ω resistors between the microcontroller and the row pins to avoid blowing out the LEDs.



Later, to make the LEDs brighter by supplying more current to them, I added PNP transistors to the anodes and moved the 330Ω resistors to the cathodes. Each cell was then driven in the following configuration:

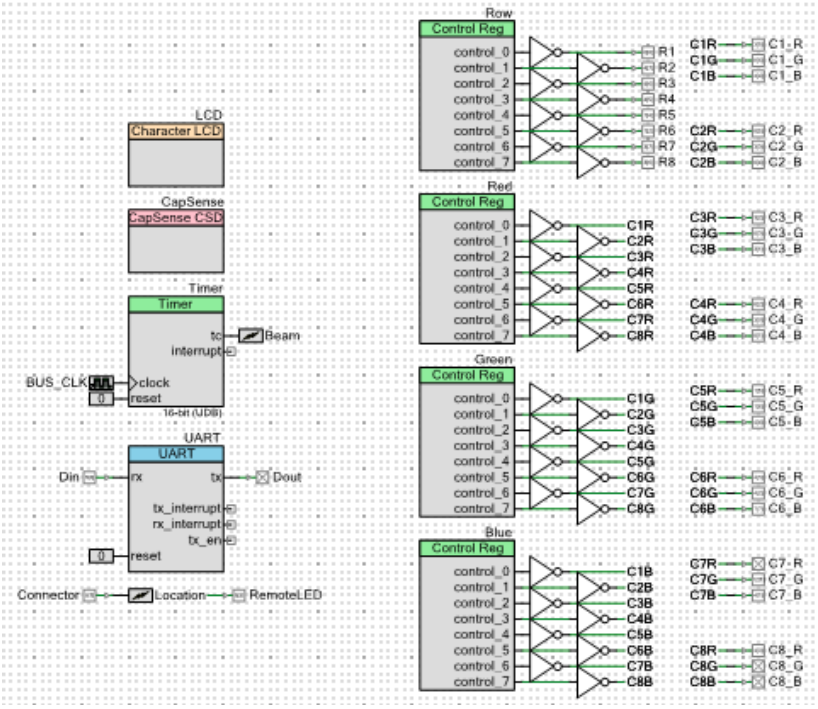


This required a significant rewire, but it paid off nicely as it also allowed me to display more colors in the visible spectrum instead of only red, green, and blue as shown below.

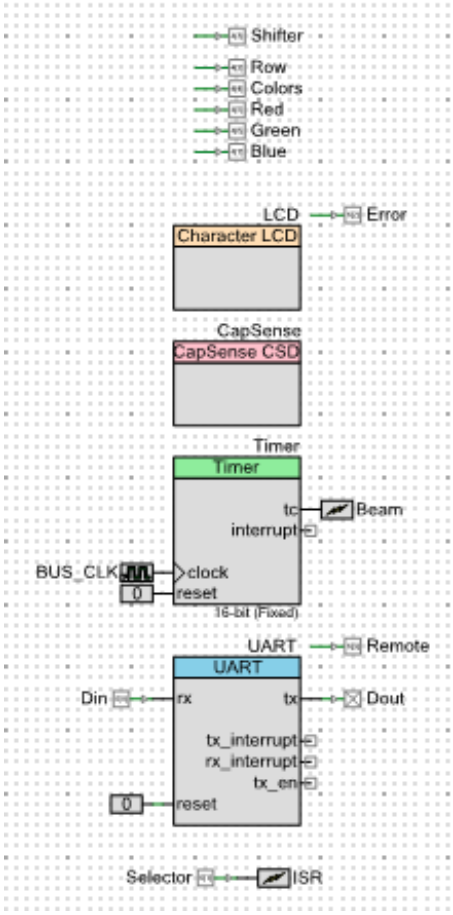


Finally, after implementing the game logic (see below), I was able to reduce the number of controlling pins used by the PSoC by adding shift registers. These registers operate in a serial in, parallel out (SIPO) manner which meant that controlling 8 columns of a given color only required 3 pins on the microcontroller (2 of which could be shared among each register). Effectively, this massively simplified my design as shown on the following page.

Parallel



Serial



Software

To interface the matrix and game logic, I wrote a library capable of illuminating, testing (to ensure proper wiring), and turning off the LED array. Since I implemented the game logic before switching to a serial output design, this library can drive either configuration with the switch of a flag. See `matrix.c` and `matrix.h` in the Appendix for C code.

Game Logic

Next, I implemented the rules of the game by using functions to create a state machine. The processor may be in any of the states below at the appropriate time. See `main.c` in the Appendix for C code.

Initializing

The initialization routine sets up an components necessary to play a single game. These components include: the LCD (for user interfacing), the board (location of dropped discs), the matrix (including a test of all colors), interrupts (for refreshing and more), the timer (for refreshing), the CapSense apparatus (for getting local input), the location of the opponent (local or remote), and, if necessary, the UART (for interfacing wireless), and the XBee (for remote coordination). Finally, matrix refreshing is started. These are done in an optimal order.[1]

Making a Move

Moves are retrieved from the CapSense slider, previewed on the top row of the matrix, and confirmed using a CapSense button. This is where logic is implemented to avoid illegal moves such as placing a disc in a full column or detecting a lack of available moves (full board).

Getting a Move

If a local game is being played, the logic is identical as the “Making a Move” state; however, if a move is being received, it is necessary to verify received information (i.e. check for channel noise, duplicates, invalid moves, etc.) and retransmit the previous local move in the event that the last transmission was lost or distorted in transit.

Animating

Aside from providing a nice simulation of an actual move on the original Connect Four board, this state provides useful functionality by determining the lowest row a disc may fall to without going off the bottom of the board or landing in an occupied position.[2]

Finding a Winner

The game transitions to this state after any move is placed to check the board for a valid winning combination. Since 4 discs are required to create a win state, only a subset of the board is check during each verification. For instance, only the first 5 columns are checked for the start

[1] For example, if refreshing begins before the opponent is located, a notable delay is added to the countdown that times how long the user has to select local or remote.
[2] The game logic essentially discards remotely received row information and determines the appropriate final position while making the disc appear to be “falling”.

of a lateral win, and lateral wins are always found checking left to right. This works since to win laterally in, say, column 6, discs would have to be located in columns 7 through 9. Since there are only 8 columns, checking there only wastes time. This method is implemented for vertical, upward diagonal, and downward diagonal wins as well.

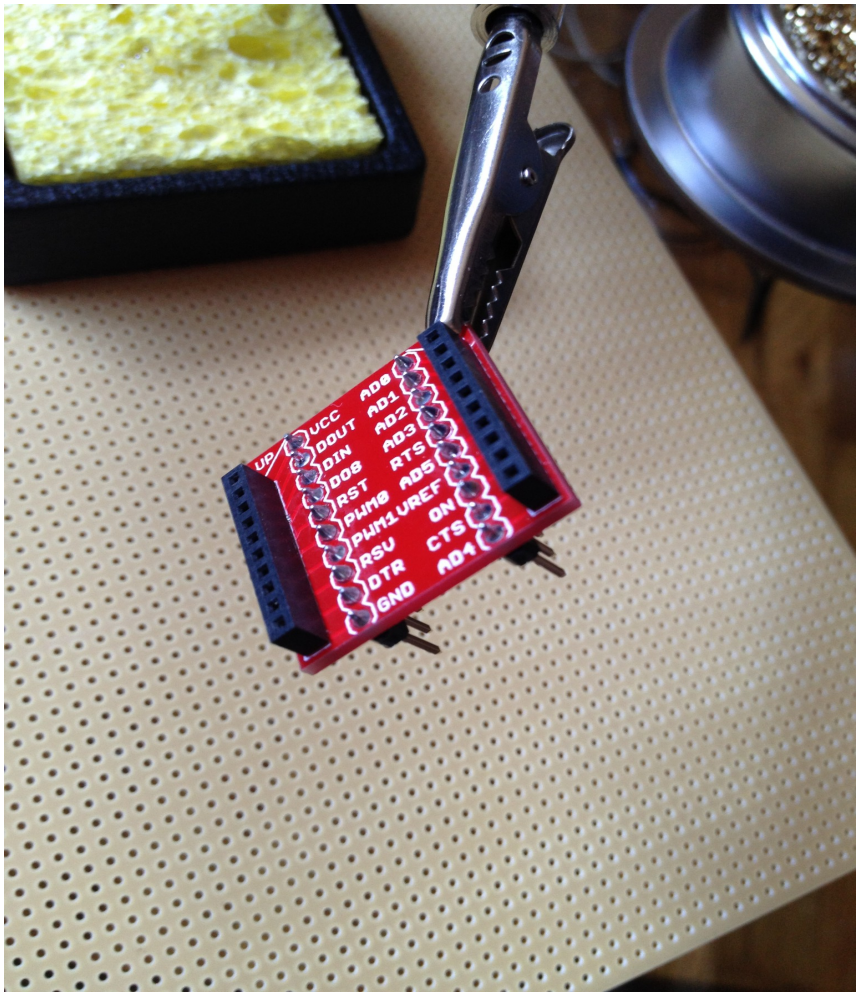
Cleaning Up

Although this state is implemented as a dead state, it exists to represent the graceful shutdown of the microcontroller. If this implementation were to go into production, this state would be entered by the toggling of an on/off switch (when switching to the “off” position).

XBee

Hardware

After getting local games to work, I began working with a partner to implement remote games using an XBee S1 wireless radio. The radios are just small enough to require an adaptive “breakout” board to fit my prototyping boards. I began by assembling this adapter using a soldering iron and appropriate header pins. The result is shown in detail below.



Software

The software generation proved to be much more difficult than the hardware assembly. The XBee is capable of running in different modes and on different channels, both of which needed to be altered for the purposes of this game. This configuration is executed in the “Initializing” state where a specific character sequence must be transmitted and a subsequent delay observed. The associated datasheets specify responses that may be expected; however, despite successful configuration (since I can communicate to other implementations), these responses have not typically been observed.

Despite interfacing issues, the radio works quite well. A UART is used to interface with the radio, and while waiting to receive an opponent’s move, it’s transmission buffer is never allowed to be empty. As soon as it transfers the previously made local move (and before receiving anything valid), the buffer is refilled so as to retransmit until deemed unnecessary.

Methodology

I generally followed the following steps when adding a new component:

1. Successfully use the component in isolation.
2. Implement regular logic that allows the microcontroller to do 1.
3. Use the regular logic in a game logic state.
4. Connect and debug.

This worked well for me, and I used it to add the necessary components in the following order:

1. LED Matrix (parallel, unamplified)
2. Refreshing
3. Game Logic
4. LED Matrix (parallel, amplified)
5. LED Matrix (serial, amplified)
6. XBee Communication
7. Remote Gameplay
8. General Optimization

Observations

Refreshing

This implementation never has more than 1 LED on at any given moment, and it refreshes at 3kHz. When running in parallel, this was not a problem since output gets written to the appropriate pins, and the processor returns from the refreshing interrupt to game logic; however, implementing shift registers for serial communication caused some issues since time must be taken to move output into each of the 4 registers 1 bit at a time. This extra overhead slowed down initialization, but was ultimately remedied by starting the refresh routine last. I have since seen some interesting ways of refreshing with the help of components the operate independent of the processor that may be implemented in the future.[3]

Backups

Just after implementing game logic for the first time (See 3 in Methodology above), most of my software was lost due to a computer malfunction. This set me back significantly in my implementation timetable and showed me that backing up should be a priority when working on any project of significance.

Auto-Drop

While reading the CapSense buttons to determine when a player has selected a column to drop a disc into, I experienced many “auto-drops” where a column (usually the first, default column) was automatically selected for the player. This was particularly agitated by remote play. The remedy I discovered was to not only detect when the button had been pressed, but also to wait until the button had been released. I have had experience doing this using edge detectors in hardware, but the software implementation I ultimately employed only required 3 lines of code!

[3] It should be noted that using a DMA while in serial mode would likely be much more difficult than when controlling the matrix using parallel signals out of the PSoC.

Q&A

How do key modules of the software interact with each other?
The PSoC acts as a hub for all signals so that the processor may interact with components at appropriate times. The matrix acts as an output-only device. The CapSense provides input only, and the UART/XBee does both. Once every 3000th of a second, the processor stops to interact with the matrix. When it reaches the appropriate state, it ignores the CapSense and wait for the XBee and vice versa. The processor coordinates everything, all other hardware is peripheral.

How did you make sure that an event such as the arrival of a packet or a display refresh event is serviced within an acceptable time limit?
Mostly, I accomplished this by trial and error. By setting the refresh rate too high, I observed an unacceptable slowing of game logic processing. When the refresh rate is too low, the matrix flickers. Likewise, if too many retransmissions are created before an opponent's move is received, the UART's transmission buffer overflows and gibberish is sent which results in no move being received! If it is not sent often enough, the opponent may be waiting for you while you wait for him which effectively introduces a dead state.

What factors drove your decisions on the hardware and software design for interfacing the 8x8 LED display?
See the design section above. First, I wanted to get it working, then I wanted brighter LEDs. Finally I wanted fewer pins so that disconnecting the matrix and XBee and reattaching later would be simpler (e.g. when they are all soldered to a single PCB). With fewer pins, the PSoC does not have to be dedicated to the connect 4 as a sort of embedded system, it may be used for other purposes and reconnected later.

How did you determine the refresh rate of the display?
See the 2nd question above, and the Refreshing subsection in Observations.

Conclusion

This has definitely been one of the most interesting and complex hardware projects I've built. I can't say I'd have done too much different If I were to do it again; however, I would have liked more time to make a more robust and dynamic system. For example, at the moment only a single game may be played before having to reset the PSoC or unplug it. It cannot gracefully turn off and on while plugged in, and it cannot support multiplayer. It has the potential to utilize the home row as a valid row to play in, and it could handle channel changing dynamically based on automatic detection of congestion. An undo feature could allow players to reverse accidental disc drops. Unfortunately, all of these neat features took a backseat to the necessary core features that were required in the timetable given.

Appendix

Code

main.c

```
#include <device.h>
#include <stdbool.h>
#include <display.h>
#include <matrix.h>

#define BUS_CLK_FREQ    24000000 // Hz
#define REFRESH_RATE    3000     // [370,BUS_CLK_FREQ] Hz
#define XBEE_CONFIGURED false
#define XBEE_VERIFY     true
#define XBEE_GUARD_TIME 1000     // ms
#define XBEE_CHANNEL    0x0C     // [0x0B,0x1A]
#define XBEE_REMEMBER   false
#define PLAYER1_ID      0xDA     // [0x00,0xFF]
#define PLAYER2_ID      0x85     // ID > max(MATRIX_ROWS,MATRIX_COLS)
recommended
#define PACKET_TYPE     0x00     // version

typedef struct play {
    uint8  row;
    uint8  col;
} play_t;

bool  player1_turn  = true;
bool  player2_local = true;
uint8 player1_color = RED;
uint8 player2_color = BLUE;

bool  player1[MATRIX_ROWS][MATRIX_COLS];
bool  player2[MATRIX_ROWS][MATRIX_COLS];
play_t last_move;

void  initialize     ( );
void  clear_board    ( );
void  configure_XBee ( );
uint8 get_move       ( );
uint8 receive_move   ( );
void  receive_packet ( play_t * move );
void  send_packet    ( play_t * move );
void  send_move      ( uint8 row , uint8 col );
uint8 animate        ( uint8 c );
void  winner_check   ( );
void  cleanup        ( );

/* These functions implement the CE121 API when XBee is in transparent mode */
```

```
//void  receive_packet ( packet_t * move );
//void  show_packet    ( packet_t * move );
//void  send_packet    ( packet_t * move );
//void  send_move      ( uint8 r , uint8 c );



bool locating = false;//, channel_change = false;
CY_ISR (toggle) { if (locating) player2_local = !player2_local; }



uint8 rPos = -1, cPos = -1;
CY_ISR (refresh) {
    uint16 cells = MATRIX_ROWS*MATRIX_COLS;
    while (cells-- > 0) {
        if (++cPos > MATRIX_COLS) {
            cPos = 0;
            if (++rPos > MATRIX_ROWS) rPos = 0;
        }
        if (player1[rPos][cPos]) {
            //matrix_deluminate();
            matrix_illuminate(rPos,cPos,player1_color);
            break;
        }
        if (player2[rPos][cPos]) {
            matrix_deluminate();
            matrix_illuminate(rPos,cPos,player2_color);
            break;
        }
    }
}



void main () {
    initialize();
    LCD_UpdateStatus("Connect 4");
    while (true) {
        uint8 row, col;
        if (player1_turn || player2_local) {
            LCD_UpdateStatus((player1_turn) ? "Player 1" : "Player 2");
            col = get_move();
            row = animate(col);
            if (!player2_local) send_move(MATRIX_ROWS-row,col+1);
        }
        else {
```

```
            LCD_UpdateStatus("Player 2");
            col = receive_move();
            row = animate(col);
        }
        winner_check();
        player1_turn = !player1_turn;
    }
    cleanup();
}



void initialize () {
    LCD_Start();
    LCD_UpdateStatus("initializing...");

    LCD_UpdateMessage("board");
    clear_board();

    LCD_UpdateMessage("matrix");
    matrix_test(true,false);
    matrix_deluminate();

    LCD_UpdateMessage("interrupts");
    CyGlobalIntEnable;

    LCD_UpdateMessage("timer");
    Timer_Start();
    Timer_WritePeriod(BUS_CLK_FREQ/REFRESH_RATE);

    LCD_UpdateMessage("CapSense");
    CapSense_Start();
    CapSense_InitializeAllBaselines();

    LCD_UpdateStatus("locate opponent");
    ISR_StartEx(toggle);
    locating = true;
    uint8 t;
    for (t=5; t > 0 ;--t) {
        LCD_UpdateMessage((player2_local) ? "local" : "remote");
        CyDelay(500);
        LCD_Position(1,15);
        LCD_PrintNumber(t);
        CyDelay(500);
    }
    locating = false;
    ISR_Stop();
    if (!player2_local) {
        LCD_UpdateStatus("initializing...");
```

```
        LCD_UpdateMessage("UART");
        UART_Start();
        UART_ClearTxBuffer();
        UART_ClearRxBuffer();
        LCD_UpdateMessage("XBee");
        if (!XBEE_CONFIGURED) configure_XBee();
    }

    LCD_UpdateMessage("display");
    Beam_StartEx(refresh);
}




void clear_board () {
    uint8 row, col;
    for (row=0; row < MATRIX_ROWS ;++row) {
        for (col=0; col < MATRIX_COLS ;++col) {
            player1[row][col] = false;
            player2[row][col] = false;
        }
    }
}




void configure_XBee () {
    LCD_Position(1,5);
    LCD_PrintString("Config");
    CyDelay(XBEE_GUARD_TIME);
    UART_PutString("+++");
    CyDelay(XBEE_GUARD_TIME*3);
    if (XBEE_VERIFY) {
        while (UART_GetRxBufferSize() < 3);
        if (UART_ReadRxData() != 'O'
        ||  UART_ReadRxData() != 'K'
        ||  UART_ReadRxData() != '\r') Error_Write(0x01);
    }

    LCD_Position(1,5);
    LCD_PrintString("API Mode");
    UART_PutString("ATAP1\r");
    if (XBEE_VERIFY) {
        while (UART_GetRxBufferSize() < 3);
        if (UART_ReadRxData() != 'O'
        ||  UART_ReadRxData() != 'K'
        ||  UART_ReadRxData() != '\r') Error_Write(0x01);
    }
```

```
    LCD_Position(1,5);
    LCD_PrintString("Channel ");
    UART_PutString("ATCH");
    UART_WriteTxData(XBEE_CHANNEL);
    UART_PutChar('\r');
    if (XBEE_VERIFY) {
        while (UART_GetRxBufferSize() < 3);
        if (UART_ReadRxData() != 'O'
        ||  UART_ReadRxData() != 'K'
        ||  UART_ReadRxData() != '\r') Error_Write(0x01);
    }

    if (XBEE_REMEMBER) {
        LCD_Position(1,5);
        LCD_PrintString("Write   ");
        UART_PutString("ATWR\r");
        if (XBEE_VERIFY) {
            while (UART_GetRxBufferSize() < 3);
        if (UART_ReadRxData() != 'O'
        ||  UART_ReadRxData() != 'K'
        ||  UART_ReadRxData() != '\r') Error_Write(0x01);
        }
    }

    LCD_Position(1,5);
    LCD_PrintString("Ready   ");
    UART_PutString("ATCN\r");
    if (XBEE_VERIFY) {
        while (UART_GetRxBufferSize() < 3);
        if (UART_ReadRxData() != 'O'
        ||  UART_ReadRxData() != 'K'
        ||  UART_ReadRxData() != '\r') Error_Write(0x01);
    }

    Remote_Write(0x01);
}




uint8 get_move () {
    LCD_UpdateMessage("Place a disc!");

    //guide
    uint8 disc = 0;
    while (player1[1][disc] || player2[1][disc]) {
        if (++disc >= MATRIX_COLS) {
            LCD_UpdateStatus("Game Over");
            LCD_UpdateMessage("Board Full");
```

```
            while (true);
        }
    }

    bool selected = false, dropped = false;
    do {
        CapSense_UpdateEnabledBaselines();
        CapSense_ScanEnabledWidgets();
        while (CapSense_IsBusy());
        if (CapSense_CheckIsWidgetActive(CapSense_BUTTON0__BTN)) {
            if (player1[1][disc] || player2[1][disc])
                LCD_UpdateMessage("Column Full");
            else selected = true;
        }
        else {
            if (selected) dropped = true;
            else {
                player1[0][disc] = player2[0][disc] = false;
                uint16 reading =
(CapSense_GetCentroidPos(CapSense_LINEARSLIDER0__LS)*(MATRIX_COLS-1))/97;
                if (reading < 8) disc = reading;

                //guide
                int dir = 1;
                while (player1[1][disc] || player2[1][disc]) {
                    if (disc >= MATRIX_COLS-1)  dir = -1;
                    if (disc == 0 && dir == -1) dir =  1; // redundant
                    disc += dir;
                }
                if (player1_turn) player1[0][disc] = true;
                else              player2[0][disc] = true;
            }
        }
    } while (!dropped);
    player1[0][disc] = player2[0][disc] = false;
    return disc;
}




uint8 receive_move () {
    play_t move;
    while (true) {
        receive_packet(&move);
        move.row = MATRIX_ROWS-move.row;
        --move.col;
        if (player2[move.row][move.col]) send_packet(&last_move);
        else break;
    }
```

```
    UART_ClearRxBuffer();
    return move.col;
}



void receive_packet ( play_t * move ) {
    UART_ClearRxBuffer();
    while (true) {
        LCD_UpdateMessage("waiting");
        LCD_Position(1,7);
        LCD_PrintString("         ");
        LCD_Position(1,7);
        uint8 vol;
        for (vol=(UART_GetRxBufferSize()*8)/128; vol > 0 ;--vol)
            LCD_PutChar('.');

        do if (UART_GetTxBufferSize() == 0) {
            send_packet(&last_move);
            CyDelay(50);
        }
        while (UART_GetRxBufferSize() < 4);
        if (UART_ReadRxData() != PLAYER2_ID) continue;

        while (UART_GetRxBufferSize() == 0);
        if (UART_ReadRxData() != PACKET_TYPE); // DEBUG newer?

        while (UART_GetRxBufferSize() == 0);
        move->row = UART_ReadRxData();
        if (move->row < 1 || move->row > MATRIX_ROWS-1) continue;

        while (UART_GetRxBufferSize() == 0);
        move->col = UART_ReadRxData();
        if (move->col < 1 || move->row > MATRIX_COLS) continue;

        break;
    }
}



void send_packet ( play_t * move ) {

    /*************************/
    char   delimiter   = '~';
    uint8  api_id      = 0x01;
    uint8  frame_id    = 0x00;
    uint16 destination = 0xFFFF; //broadcast
```

```
uint8  disable_ack = 0x01;
/*************************/

uint16 length = sizeof(destination) +
                sizeof(api_id) +
                sizeof(frame_id) +
                sizeof(disable_ack) +
                4; //Payload: Player ID, Packet Type, Row, Column

uint8 checksum = 0xFF - (uint8) (api_id +
                                 frame_id +
                                ((uint8) (destination >> 8)) +
                                ((uint8)  destination) +
                                 disable_ack +
                                 PLAYER1_ID +
                                 PACKET_TYPE +
                                 move->row +
                                 move->col);


/* show *
uint16 delay = 100;

LCD_UpdateStatus("Delimiter");
LCD_ClearLine(1);
LCD_PutChar(delimiter);
LCD_Position(1,2);
LCD_PrintNumber(delimiter);
CyDelay(delay);

LCD_UpdateStatus("Length");
LCD_ClearLine(1);
LCD_PrintNumber(length);
CyDelay(delay);

LCD_UpdateStatus("API ID");
LCD_ClearLine(1);
LCD_PrintString("0x");
LCD_PrintInt8(api_id);
CyDelay(delay);

LCD_UpdateStatus("Frame ID");
LCD_ClearLine(1);
LCD_PrintString("0x");
LCD_PrintInt8(frame_id);
CyDelay(delay);

LCD_UpdateStatus("Destination");
LCD_ClearLine(1);
LCD_PrintString("0x");
```

```
LCD_PrintInt16(destination);
CyDelay(delay);

LCD_UpdateStatus("Disable ACK");
LCD_ClearLine(1);
LCD_PrintString("0x");
LCD_PrintInt8(disable_ack);
CyDelay(delay);

LCD_UpdateStatus("Player ID");
LCD_ClearLine(1);
LCD_PrintString("0x");
LCD_PrintInt8(PLAYER1_ID);
CyDelay(delay);

LCD_UpdateStatus("Packet Type");
LCD_ClearLine(1);
LCD_PrintString("0x");
LCD_PrintInt8(PACKET_TYPE);
CyDelay(delay);

LCD_UpdateStatus("Row");
LCD_ClearLine(1);
LCD_PrintNumber(move->row);
CyDelay(delay);

LCD_UpdateStatus("Column");
LCD_ClearLine(1);
LCD_PrintNumber(move->col);
CyDelay(delay);

LCD_UpdateStatus("Checksum");
LCD_ClearLine(1);
LCD_PrintString("0x");
LCD_PrintInt8(checksum);
CyDelay(delay);
/**/


UART_ClearTxBuffer();
UART_WriteTxData(delimiter);
UART_WriteTxData((uint8) (length >> 8));
UART_WriteTxData((uint8)  length);
UART_WriteTxData(api_id);
UART_WriteTxData(frame_id);
UART_WriteTxData((uint8) (destination >> 8));
UART_WriteTxData((uint8)  destination);
UART_WriteTxData(disable_ack);
UART_WriteTxData(PLAYER1_ID);
UART_WriteTxData(PACKET_TYPE);
```

```
    UART_WriteTxData(move->row);
    UART_WriteTxData(move->col);
    UART_WriteTxData(checksum);
}



void send_move ( uint8 row , uint8 col )  {
    last_move.row = row;
    last_move.col = col;
    send_packet(&last_move);
}



uint8 animate ( uint8 col ) {
    uint8 row;
    for (row=1; row < MATRIX_ROWS ;++row) {
        if (player1_turn) player1[row][col] = true;
        else              player2[row][col] = true;
        if (player1[row+1][col] || player2[row+1][col] || row+1 ==
MATRIX_ROWS) break;
        else CyDelay(20);
        if (player1_turn) player1[row][col] = false;
        else              player2[row][col] = false;
    }
    return row;
}



void winner_check () {
    uint8 row, col;
    uint16 strobe = 100;
    for (row=1; row < MATRIX_ROWS ;++row) {
        for (col=0; col < MATRIX_COLS ;++col) {
            int offset;

            //vertical
            if (row < MATRIX_ROWS-3) {
                bool win = true;
                for (offset=0; win && offset < 4 ;++offset) {
                    if (player1_turn) {
                        if (!player1[row+offset][col]) win = false;
                    }
                    else {
                        if (!player2[row+offset][col]) win = false;
```

```
            }
        }
        if (win) {
            LCD_UpdateMessage("You win!");
            bool blink = false;
            while (true) {
                for (offset=0; win && offset < 4 ;++offset) {
                    if (player1_turn)
                        player1[row+offset][col] = blink;
                    else
                        player2[row+offset][col] = blink;
                }
                CyDelay(strobe);
                blink = !blink;
            }
        }
    }

    //lateral
    if (col < MATRIX_COLS-3) {
        bool win = true;
        for (offset=0; win && offset < 4 ;++offset) {
            if (player1_turn) {
                if (!player1[row][col+offset]) win = false;
            }
            else {
                if (!player2[row][col+offset]) win = false;
            }
        }
        if (win) {
            LCD_UpdateMessage("You win!");
            bool blink = false;
            while (true) {
                for (offset=0; win && offset < 4 ;++offset) {
                    if (player1_turn)
                        player1[row][col+offset] = blink;
                    else
                        player2[row][col+offset] = blink;
                }
                CyDelay(strobe);
                blink = !blink;
            }
        }
    }

    //down-diagonal
    if (row < MATRIX_ROWS-3 && col < MATRIX_COLS-3) {
        bool win = true;
        for (offset=0; win && offset < 4 ;++offset) {
            if (player1_turn) {
```

```
                    if (!player1[row+offset][col+offset]) win = false;
                }
                else {
                    if (!player2[row+offset][col+offset]) win = false;
                }
            }
            if (win) {
                LCD_UpdateMessage("You win!");
                bool blink = false;
                while (true) {
                    for (offset=0; win && offset < 4 ;++offset) {
                        if (player1_turn)
                            player1[row+offset][col+offset] = blink;
                        else
                            player2[row+offset][col+offset] = blink;
                    }
                    CyDelay(strobe);
                    blink = !blink;
                }
            }
        }

        //up-diagonal
        if (row > 3 && col < MATRIX_COLS-3) {
            bool win = true;
            for (offset=0; win && offset < 4 ;++offset) {
                if (player1_turn) {
                    if (!player1[row-offset][col+offset]) win = false;
                }
                else {
                    if (!player2[row-offset][col+offset]) win = false;
                }
            }
            if (win) {
                LCD_UpdateMessage("You win!");
                bool blink = false;
                while (true) {
                    for (offset=0; win && offset < 4 ;++offset) {
                        if (player1_turn)
                            player1[row-offset][col+offset] = blink;
                        else
                            player2[row-offset][col+offset] = blink;
                    }
                    CyDelay(strobe);
                    blink = !blink;
                }
            }
        }
    }
}
```

```
}



void cleanup () {
    LCD_UpdateStatus("cleaning...");

    LCD_UpdateMessage("display");
    Beam_Stop();

    if (!player2_local) {
        LCD_UpdateMessage("UART");
        UART_Stop();
    }

    LCD_UpdateMessage("CapSense");
    CapSense_Stop();

    LCD_UpdateMessage("timer");
    Timer_Stop();

    LCD_UpdateMessage("matrix");
    matrix_deluminate();

    LCD_UpdateMessage("board");
    clear_board();

    LCD_Stop();
}









/*
typedef struct packet {
    char   delimiter;
    uint16 length;
    uint8  api_id;
    uint8  frame_id;
    uint16 destination;
    uint8  disable_ack;
    uint8  player_id;
    uint8  packet_type;
    uint8  row;
```

```
    uint8  col;
    uint8  checksum;
} packet_t;



packet_t last_move;



void receive_move () {
    packet_t move;
    while (true) {
        receive_packet(&move);
        if (player2[MATRIX_ROWS-move.row][move.col-1])
send_packet(&last_move);
        else break;
    }
    player2[MATRIX_ROWS-move.row][move.col-1] = true;
    UART_ClearRxBuffer();
}



void receive_packet ( packet_t * move ) {
    UART_ClearRxBuffer();
    while (true) {
        LCD_UpdateMessage("waiting"); // DEBUG

        while (UART_GetRxBufferSize() == 0);
        do move->delimiter = UART_ReadRxData();
        while (move->delimiter != '~' && UART_GetRxBufferSize() > 0);


        /* DEBUG *
        LCD_UpdateMessage("received");
        LCD_Position(1,12);
        uint16 tst = UART_GetRxBufferSize();
        LCD_PrintNumber(tst);
        CyDelay(1000);
        /**


        if (move->delimiter != '~') continue;

        while (UART_GetRxBufferSize() == 0);
        move->length   = UART_ReadRxData();
```

```
        move->length <<= 8;
        while (UART_GetRxBufferSize() == 0);
        move->length  |= UART_ReadRxData();

        while (UART_GetRxBufferSize() == 0);
        move->api_id = UART_ReadRxData();
        //if (move->api_id != 0x01) continue; //do something?

        while (UART_GetRxBufferSize() == 0);
        move->frame_id = UART_ReadRxData();
        //if (move->frame_id != 0x00) continue; //maybe this needs to be a
sequence #?

        while (UART_GetRxBufferSize() == 0);
        move->destination   = UART_ReadRxData();
        move->destination <<= 8;
        while (UART_GetRxBufferSize() == 0);
        move->destination  |= UART_ReadRxData();
        if (move->destination != 0xFFFF) continue; //compare to xbee src addr

        while (UART_GetRxBufferSize() == 0);
        move->disable_ack = UART_ReadRxData();
        //if (move->disable_ack != 0x01) continue; //ack?

        while (UART_GetRxBufferSize() == 0);
        move->player_id = UART_ReadRxData();
        if (move->player_id != PLAYER2_ID) continue;

        while (UART_GetRxBufferSize() == 0);
        move->packet_type = UART_ReadRxData();
        //if (move->packet_type != 0x00); //a later version is trying to play
with this game

        while (UART_GetRxBufferSize() == 0);
        move->row = UART_ReadRxData();
        if (move->row < 1 || move->row > MATRIX_ROWS-1) continue;

        while (UART_GetRxBufferSize() == 0);
        move->col = UART_ReadRxData();
        if (move->col < 1 || move->row > MATRIX_COLS) continue;

        while (UART_GetRxBufferSize() == 0);
        move->checksum = UART_ReadRxData();
        /*
        if ((move->api_id +
             move->frame_id +
             move->destination +
             move->disable_ack +
             move->player_id +
             move->packet_type +
```

```
            move->row +
            move->col +
            move->checksum) != 0xFF) continue;
        *

        break;
    }
}



void show_packet ( packet_t * move ) {
    uint16 delay = 100;

    LCD_UpdateStatus("Delimiter");
    LCD_ClearLine(1);
    LCD_PutChar(move->delimiter);
    LCD_Position(1,2);
    LCD_PrintNumber(move->delimiter);
    CyDelay(delay);

    LCD_UpdateStatus("Length");
    LCD_ClearLine(1);
    LCD_PrintNumber(move->length);
    CyDelay(delay);

    LCD_UpdateStatus("API ID");
    LCD_ClearLine(1);
    LCD_PrintString("0x");
    LCD_PrintInt8(move->api_id);
    CyDelay(delay);

    LCD_UpdateStatus("Frame ID");
    LCD_ClearLine(1);
    LCD_PrintString("0x");
    LCD_PrintInt8(move->frame_id);
    CyDelay(delay);

    LCD_UpdateStatus("Destination");
    LCD_ClearLine(1);
    LCD_PrintString("0x");
    LCD_PrintInt16(move->destination);
    CyDelay(delay);

    LCD_UpdateStatus("Disable ACK");
    LCD_ClearLine(1);
    LCD_PrintString("0x");
    LCD_PrintInt8(move->disable_ack);
    CyDelay(delay);
```

```
    LCD_UpdateStatus("Player ID");
    LCD_ClearLine(1);
    LCD_PrintString("0x");
    LCD_PrintInt8(move->player_id);
    CyDelay(delay);

    LCD_UpdateStatus("Packet Type");
    LCD_ClearLine(1);
    LCD_PrintString("0x");
    LCD_PrintInt8(move->packet_type);
    CyDelay(delay);

    LCD_UpdateStatus("Row");
    LCD_ClearLine(1);
    LCD_PrintNumber(move->row);
    CyDelay(delay);

    LCD_UpdateStatus("Column");
    LCD_ClearLine(1);
    LCD_PrintNumber(move->col);
    CyDelay(delay);

    LCD_UpdateStatus("Checksum");
    LCD_ClearLine(1);
    LCD_PrintString("0x");
    LCD_PrintInt8(move->checksum);
    CyDelay(delay);
}



void send_packet ( packet_t * move ) {
    UART_ClearTxBuffer();
    UART_WriteTxData(move->delimiter);
    UART_WriteTxData((uint8) (move->length >> 8));
    UART_WriteTxData((uint8) move->length);
    UART_WriteTxData(move->api_id);
    UART_WriteTxData(move->frame_id);
    UART_WriteTxData((uint8) (move->destination >> 8));
    UART_WriteTxData((uint8) (move->destination));
    UART_WriteTxData(move->disable_ack);
    UART_WriteTxData(move->player_id);
    UART_WriteTxData(move->packet_type);
    UART_WriteTxData(move->row);
    UART_WriteTxData(move->col);
    UART_WriteTxData(move->checksum);
}
```

```
void send_move ( uint8 r , uint8 c )  {
    last_move.delimiter   = '~';
    last_move.length      = 13 - (sizeof(last_move.delimiter) +
                                  sizeof(last_move.length) +
                                  sizeof(last_move.checksum));
    last_move.api_id      = 0x01;
    last_move.frame_id    = 0x00;
    last_move.destination = 0xFFFF;
    last_move.disable_ack = 0x01;
    last_move.player_id   = PLAYER1_ID;
    last_move.packet_type = 0x00;
    last_move.row         = r;
    last_move.col         = c;
    last_move.checksum    = 0xFF - (uint8) (last_move.api_id +
                                            last_move.frame_id +
                                           (last_move.destination >> 8) +
                                           ((uint8) last_move.destination) +
                                            last_move.disable_ack +
                                            last_move.player_id +
                                            last_move.packet_type +
                                            last_move.row +
                                            last_move.col);
    send_packet(&last_move);
}
*/
```

display.h

```
#ifndef DISPLAY_H
#define DISPLAY_H
void LCD_ClearLine     (int line);
void LCD_UpdateStatus  (const char string[]);
void LCD_UpdateMessage (const char string[]);
void LCD_Spin          (int row, int col, int speed, int count);
#endif
```

display.c

```
#include <device.h>



void LCD_ClearLine (int line) {
    LCD_Position(line,0);
    LCD_PrintString("                ");
    LCD_Position(line,0);
```

```
}



void LCD_UpdateStatus (const char string[]) {
    LCD_ClearDisplay();
    LCD_Position(0,0);
    LCD_PrintString(string);
}



void LCD_UpdateMessage (const char string[]) {
    LCD_ClearLine(1);
    LCD_Position(1,0);
    LCD_PrintString(string);
}



void LCD_Spin (int row, int col, int speed, int count) {
    while (count-- > 0) {
        LCD_Position(row,col);
        LCD_PutChar('|');
        CyDelay(speed/4);
        LCD_Position(row,col);
        LCD_PutChar('/');
        CyDelay(speed/4);
        LCD_Position(row,col);
        LCD_PutChar('-');
        CyDelay(speed/4);
        LCD_Position(row,col);
        LCD_PutChar('\\');
        CyDelay(speed/4);
    }
}
```

matrix.h

```
#include <stdbool.h>

#ifndef MATRIX_H
#define MATRIX_H

#define SERIAL true

#define MATRIX_ROWS 8
```

```
#define MATRIX_COLS 8

#define RED     0x01
#define GREEN   0x02
#define YELLOW  0x03
#define BLUE    0x04
#define MAGENTA 0x05
#define CYAN    0x06
#define WHITE   0x07

void matrix_illuminate  ( int row , int col , uint8 color );
void matrix_deluminate  ();
void matrix_test        ( bool extended , bool slow );
#endif
```

matrix.c

```
#include <device.h>
#include <stdbool.h>
#include <display.h>
#include <matrix.h>



void matrix_illuminate ( int row , int col , uint8 color ) {
    uint8 off = 0x00, on = 0x01;
    if (SERIAL) {
        uint8 rows = 0b00000001 << row;
        uint8 reds = 0b00000001;
        uint8 grns = 0b00000001;
        uint8 blus = 0b00000001;
        switch (color) {
            case RED:     reds <<= col;
                          grns   = off;
                          blus   = off; break;
            case GREEN:   reds   = off;
                          grns <<= col;
                          blus   = off; break;
            case YELLOW:  reds <<= col;
                          grns <<= col;
                          blus   = off; break;
            case BLUE:    reds   = off;
                          grns   = off;
                          blus <<= col; break;
            case MAGENTA: reds <<= col;
                          grns   = off;
                          blus <<= col; break;
            case CYAN:    reds   = off;
                          grns <<= col;
```

```
                    blus <<= col; break;
        case WHITE:   reds <<= col;
                    grns <<= col;
                    blus <<= col; break;

        default: LCD_UpdateStatus("error");
                LCD_UpdateMessage("color mismatch");
    }

    rows = ~rows;
    reds = ~reds;
    grns = ~grns;
    blus = ~blus;

    Colors_Write(off);
    int i;
    for (i=0; i < 8 ;++i) {
        Row_Write(  (rows & (0b10000000 >> i)) ? on : off);
        Red_Write(  (reds & (0b10000000 >> i)) ? on : off);
        Green_Write((grns & (0b10000000 >> i)) ? on : off);
        Blue_Write( (blus & (0b10000000 >> i)) ? on : off);

        //generate a rising edge
        Shifter_Write(on);
        Shifter_Write(off);
    }
    Colors_Write(on);
}
else {
    uint8 r = on, c = on;
    while (row-- > 0) r *= 2;
    Row_Write(r);
    while (col-- > 0) c *= 2;
    switch (color) {
        case RED:       Red_Write(c); break;
        case GREEN:   Green_Write(c); break;
        case YELLOW:    Red_Write(c);
                    Green_Write(c); break;
        case BLUE:     Blue_Write(c); break;
        case MAGENTA:   Red_Write(c);
                     Blue_Write(c); break;
        case CYAN:    Green_Write(c);
                     Blue_Write(c); break;
        case WHITE:     Red_Write(c);
                    Green_Write(c);
                     Blue_Write(c); break;

        default: LCD_UpdateStatus("error");
                LCD_UpdateMessage("color mismatch");
                while(true);
```

```
                    break;
            }
        }
    }
```

```
void matrix_deluminate () {
    uint8 off = 0x00, on = 0x01;
    if (SERIAL) {
        Colors_Write(off);
        int i;
        for (i=0; i < 8 ;++i) {
            Row_Write(0xFF);
            Red_Write(0xFF);
            Green_Write(0xFF);
            Blue_Write(0xFF);

            //generate a rising edge
            Shifter_Write(on);
            Shifter_Write(off);
        }
        Colors_Write(on);
    }
    else {
        Row_Write(off);
        Red_Write(off);
        Green_Write(off);
        Blue_Write(off);
    }
}
```

```
void matrix_test ( bool extended , bool slow ) {
    matrix_deluminate();
    uint8 color = 0x01;
    while (color < 0x08) {
        uint8 i, j;
        for (i=0; i < MATRIX_ROWS ;++i) {
            for (j=0; j < MATRIX_COLS ;++j) {
                LCD_ClearLine(1);
                LCD_Position(1,0);
                LCD_PrintNumber(i);
                LCD_Position(1,1);
                LCD_PutChar(',');
                LCD_Position(1,2);
                LCD_PrintNumber(j);
```

```
                LCD_Position(1,4);
                LCD_PrintString("       ");
                LCD_Position(1,4);
                switch (color) {
                    case RED:     LCD_PrintString("Red");     break;
                    case GREEN:   LCD_PrintString("Green");   break;
                    case YELLOW:  LCD_PrintString("Yellow");  break;
                    case BLUE:    LCD_PrintString("Blue");    break;
                    case MAGENTA: LCD_PrintString("Magenta"); break;
                    case CYAN:    LCD_PrintString("Cyan");    break;
                    case WHITE:   LCD_PrintString("White");   break;

                    default: LCD_UpdateStatus("error");
                             LCD_UpdateMessage("color mismatch");
                             while(true);
                             break;
                }
                matrix_illuminate(i,j,color);
                CyDelay((slow) ? 1000/MATRIX_COLS : 5);
                matrix_deluminate();
            }
        }
        color = (extended) ? color+1 : color*2;
    }
}
```