"Connect 4" on PSoC 5

Introduction

This design document will demonstrate an implementation of a simple "Connect 4" game. The game supports single player and multiplayer through a serial communication link. Additionally, the inclusion of a micro SD card will allow players to save a previously played game.

Hardware Components Used

Components	
8 X 8 RGB LED Matrix	SD Card
PSoC 5 Microcontroller	Breadboard
Wires	PCB boards

Design

Hardware Design

The design uses two colors red and blue which occupies two columns of the LED matrix. Instead of using transistors and resistors to satisfy the current requirement to drive different colors of the LEDs on, two synchronized software interrupts are used for alternating between displaying the two different colors. This has a minor drawback of not sinking as much current for displaying multiple same color LEDs on the same row. However, this design minimized wiring which makes debugging easy.

Pins

LED Matrix

Total number of pins required for the LED matrix is **24**: 8 for rows plus 18 for columns of each color used. The hardware assumes one row and one color to be on at a time. Since the current requirement is satisfied through software means, the LED pins can be routed directly to the GPIO pins of the PSoC 5 micro processor (see fig. 1 and 2 for details)

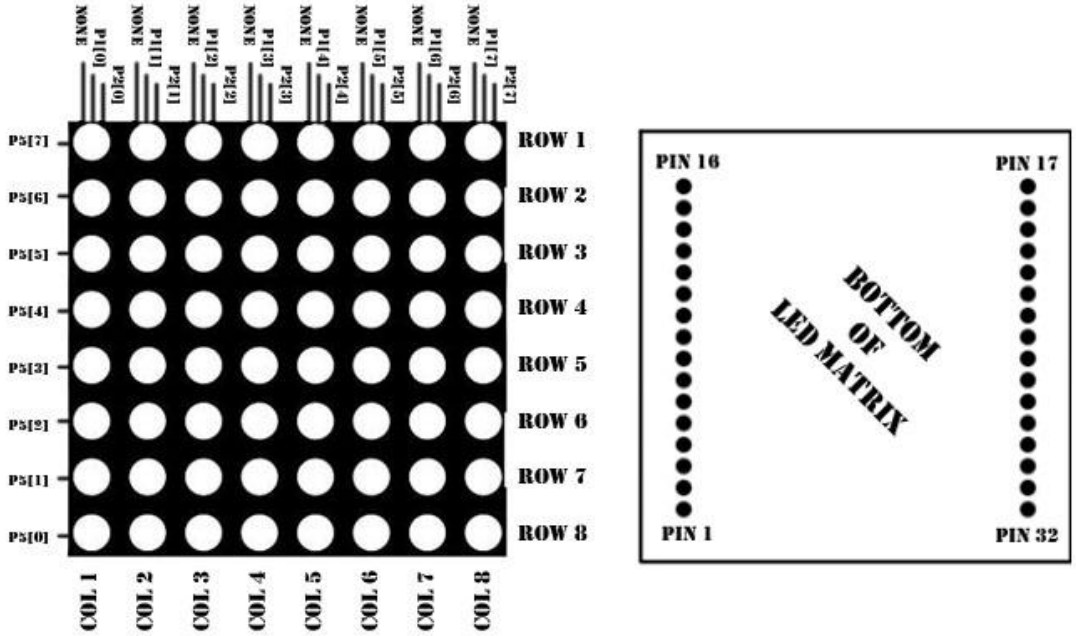


Fig 1. (LEFT) Top view and (RIGHT) bottom view of LED matrix

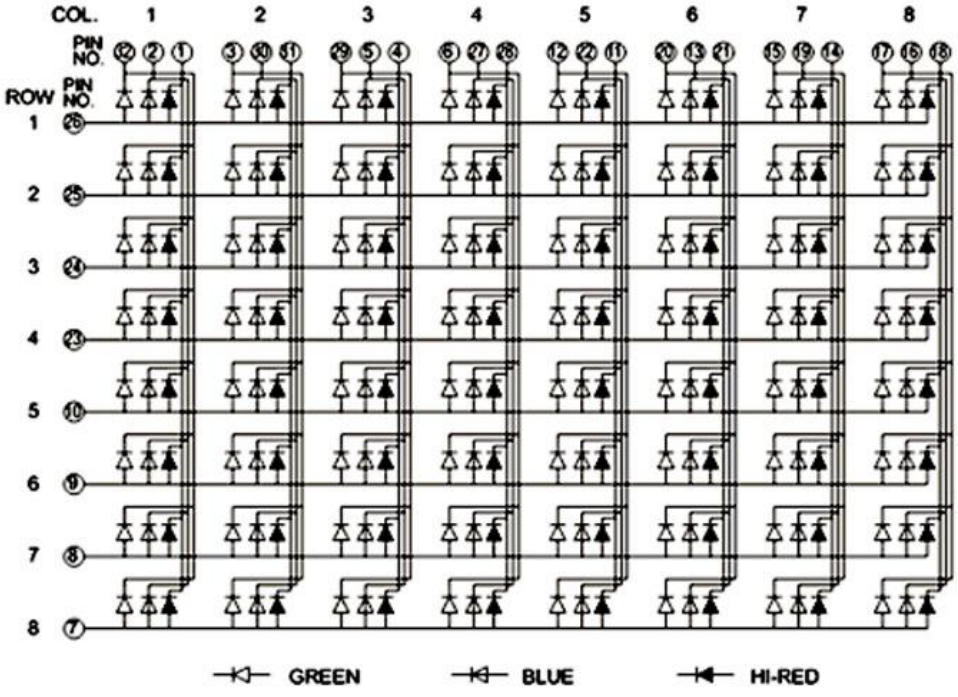
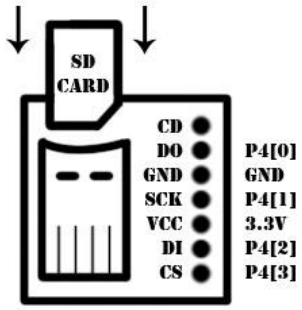


Fig 2. LED matrix circuit (analogous to left diagram of fig. 1) and pin number routing

SD Card



The pins on the SD card adapter must be soldered to male or female ports to ensure that the connections are properly made between the microcontroller and the adapter. When doing so, do not leave the micro SD card inserted as it may damage the card on accident. Also, note that the SD card requires a V_{CC} of 3.3V. Otherwise, it will not function properly.

Software Design

LED Refresh

The refresh is achieved by using a custom timer (counter) which has two interrupt points (see schematic in project files). At the half way point in the counter, an ISR is invoked and it turns off one color and turns on the other by writing values into the control register for each color. The same goes for when the counter resets, but it also increases a counter which keeps track of which row is activated. This will provide mutual exclusion and displays only one color at a time (again, this is a solution to the current requirement issue). Upon realization, the ISR can be triggered on positive and negative edges of a clock respectively which further simplifies the implementation.

Game State Machine (Event-driven)

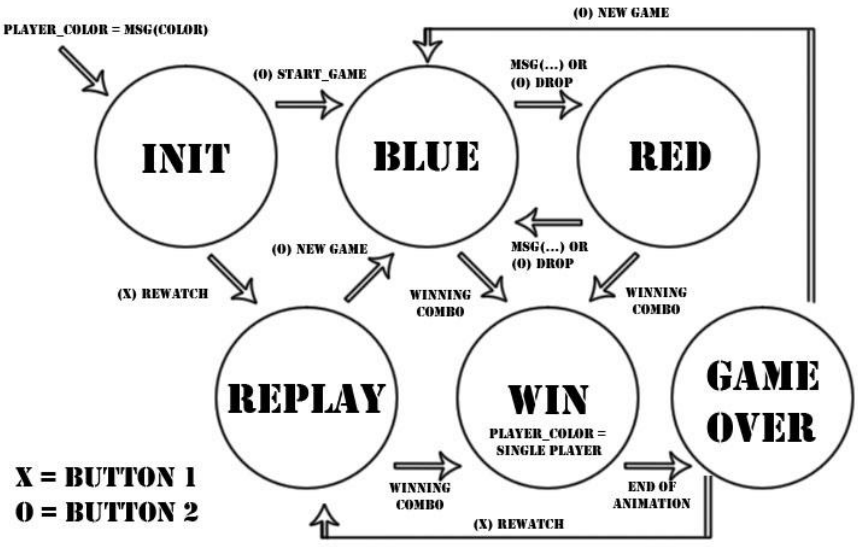


Fig 3. A general model of the program which describes all the possibilities of the game play

INIT

When the program starts up, it is set to this state. It will wait for a message from an external player which signals multiplayer mode and who goes first by setting a global variable. If the message is not available, the game mode remains single player. Before the player decides to begin the game, new messages can overwrite this variable. The player can begin the game by pressing a button or re-watch the previously saved game by pressing the other button

all_initialized (flag)

Buttons generate pulses when the board is started, which invokes the ISR causing an event to occur. The flag is set after a delay of 200ms to avoid any unwanted pulses.

RED and BLUE

It simply indicates whose turn it is. Depending whether the player went first or second, the two flags below keep track of how messages are passed around. Although the flags are mutually exclusive, they are exhaustive so it requires two flags.

msg_received

This flag is set when the first message has been successfully decoded and any further messages will be ignored. The flag resets when the player drops and begins sending a message.

keep_sending

When the player drops, this flag is then set and the periodic interrupt will be allowed to continuously send new messages to the opponent. The flag is reset when the player first successfully decode the message.

RED_WIN / BLUE_WIN

These are simply transition states for animation. Everything will be disabled and the entire game should have been saved at the beginning of this state.

GAMEOVER

The screen displays a serene ocean wave animation and waits for the player to decide whether he/she wants to re-watch the game or start a new game. It is identical to INIT except for the animation, i.e. it can receive multiplayer request or begin a new game in single player mode or re-watch the game.

REPLAY

This is the state for re-watching a saved game. The user can press a button to either walk through each step in the game or quit and begin a new game by pressing another. Once the entire game is replayed, victory conditions will send the player to GAMEOVER mode again.

Testing

Testing is incremental. Progress is saved in each state of development: 1) refreshing LED screen 2) test the movement and dropping module, 3) implement victory conditions, 4) coordinate the game with another person through serial link, and 5) recording the movements of the game play.

refresh.c

This simply tests whether or not the screen refreshes correctly without bleeding, dimness issues, or other hardware related bugs that may arise. One notable thing is that the LED screen was not able to display 2 different colors simultaneously. One color would dominate the other if it is activated on the same row. As discussed before, this is solved by toggling between display the two colors.

move_drop.c

At this stage, buttons are implemented because it is simple and quick to ensure time is not wasted on getting CapSense to work. Many cases of CapSense interfering with the hardware in an unexpected way have led me to believe my decision of not using it was correct. Beside, the controls are inappropriate and inconvenient even after extensive use.

victory.c

Due to the way color bitmaps stored in the design (as a 1D array of uint8), unintuitive bit-wise operations were required for victory conditions checking. To make things a bit simpler, the program brute force checks every time a player makes a move. In spite of a long function call in an ISR, the results were snappy and swift.

uart.c

This part of the project is the most time consuming because many students have artifacts from previous parts of the assignment and simply left it to rot. This has caused many bugs creating a problem which is not

simply a protocol implementation. Either way, once I identified someone with healthy programming habits, it was simply coordinating macros and message formats.

sd.c or (main.c)

sd.c is a copy of main.c which is the final version of the code. In this implementation, player moves are written onto the SD card and read back for re-watching a game as mentioned before. I discovered that the return values of FS library functions mean nothing because the operations worked even though the return values are always incorrect.

Conclusions

The instructor's words were right: "if you don't have to use hardware, do it software". This is particularly true of my early stages in development because all I did was hook the LED straight to the micro-controller, and used ISRs for everything. It resulted in being a very consistent event-drive state machine. Although I am beginning to see micro-delay issues with read and writes during an interrupt cycle, it was minor and did not cause any bugs.

Apparently, people have cheated off me in some of my labs and I had no idea. It is regrettable to hear how this is such a commonplace in college (also in other classes). I am not looking to point fingers at anybody because I have no evidence and my only source is hearing it from other students in lab.

Reference

I didn't use any help!