Elec 327 Final Project Report
5/1/2019

Life in Three Dimensions

For our final project we set out to create a hardware system that can perform the famous simulation known as Conway's Game of Life, but with an added twist that creates a unique layer of depth to the classic model. Rather than the typical flat grid of units which most Game of Life simulators implement, we chose to develop a three-dimensional cube of LEDs that enables the simulation to be able to wrap around the edges. The corners also introduce an added twist to the game, as the cells surrounding each of the eight corners have one less neighbor than the cells on the flat surfaces of the cube. The final product consists of six printed circuit boards with a six by six grid of LEDs as each face, leading to 216 total LEDs distributed across the surface of the cube. The large amount of LEDs, each of which represents a single cell in the simulation, paired with the capability for the simulation to wrap around the surface of the cube in every direction, leads to fascinating and unique simulations.

When turned on, the cube powers up and displays a single green LED in the center of one of the faces. Using the buttons on that face, the user is able to move the LED up, down, right, and left to its neighbors. By clicking the middle button, the user is able to select that LED to be on in the initial state of the game. Upon moving away from the selected LED, it will glow blue to indicate that it has been selected. After navigating around as much of the cube as desired, and selecting all of the LEDs to be on in the initial round of the simulation, the user simply needs to pause a few seconds for the cube to begin the Game of Life.

The rules of the Game of Life are simple. If a cell is off but three of its neighbors are on in a given round, then the cell will become alive in the next round. If a cell is on and it has either two or three neighbors that are also on, then it will stay on in the next round. If a cell is on and it has less than two neighbors on then it "dies" due to isolation, and if it has more than three neighbors it "dies" as if by overpopulation. This cube is able to fully simulate the Game of Life around its entire surface, following the rules described above. The cube will iterate through rounds of the simulation as long as it receives power, whether the simulation continues forever or eventually dies out. If the user wishes to restart and select a new initial state, they simply need to restart the cube with the power switch and the process begins anew.

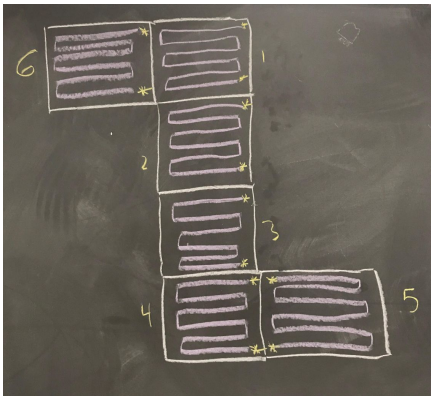

Figure 1: LED array

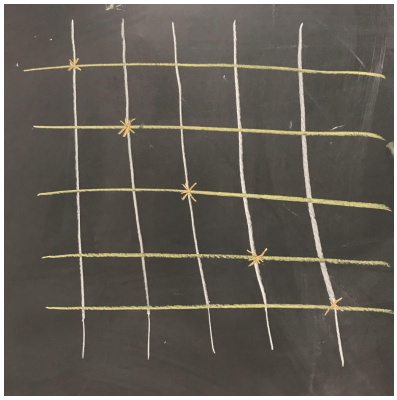

Figure 2: Button Grid

The cube itself is composed of six printed circuit boards, each of which has 36 LEDs soldered on in an even 6x6 grid. The layout of the LED array on each board is shown above in Figure 1. Each board also contains five buttons, four of which are used for navigation and the middle button for selecting LEDs to be in the initial state. The printed circuit boards utilize a button matrix for organizing the buttons on each face. This consisted of parallel lines running in both directions across each PCB that connect in the middle to the button, and this enables another face to be able to connect a button in parallel by connecting to the line on any side of the PCB. Utilizing this button matrix we were able to connect each corresponding button on every face in parallel so that they form one input to the MSP430. This button matrix layout is shown below in figure 2.

Given the amount of cells in the simulation and the limited on-board storage capabilities that the MSP430 supports, we ran into multiple challenges in this project related to memory utilization. There is not enough RAM on the MSP430 for there to be an *int* value for every LED that can be both read from and written to. Yet since each LED is only in one of two states, on or off, they each only need two bits of storage: one for the current state and one for the next state. Sadly C does not provide any way to store individual bits in a bit field, so we had to develop our own. Since there are 216 LEDs that each need 1 bit of storage, we kept their current and next states stored in two size 14 arrays of *unsigned int*. Each index in the array contained a 16 bit long *unsigned int* value, which contained the single-bit states for 16 of the LEDs. Since the LEDs are communicated to in a single chain 216 LEDs long, we identified each LED by its index in that chain. Therefore to access the state of any individual LED, we would divide the index of that LED by 16 and use that value to get the corresponding 16 bit unsigned int in the state array that would hold the bit for that specific LED. Then we would perform the modular division of the index of the LED with 16 to get the specific bit in that 16 bit unsigned integer. By masking the 16 bit unsigned integer with the result of the modular division we would get the single state value of that LED. We performed the division by 16 using bit shifting and the modular operation using bitwise AND to increase performance. By using this storage system we were drastically able to reduce the amount of RAM memory required to store the state of every LED.

However, this was not the only value we needed for every LED. In order to perform the simulation, we also needed to be able to keep track of the 8 neighbors of every LED on the cube. Of course there was no way that we could store eight integer values for every LED on the cube in RAM. Yet RAM storage on the MSP430 is only necessary for values that need both read and write access. The neighbors of each LED are values that never change, and thus only need to be read. This meant we were able to hardcode a 216x8 nested array of the neighbors for every LED and by storing the array as a static const, the compiler/linker stored it in Flash memory on the MSP430 rather than RAM. Figuring out the solutions to these storage problems was difficult, but as a result of going through these trials we now have a much better understanding of the capabilities and limitations of embedded system programming. We also have a much more thorough understanding of how data is stored and accessed in such system and how to utilize the given memory hierarchy to perform a task.

The control logic of the cube is relatively straightforward. We use interrupts from the button grid to set boolean values for moving the selecting bit up, down, right or left, as well as for selecting a bit to be on in the initial state. There is a while loop that checks these boolean

values and moves the selecting LED appropriately, as well as setting the initial state based on the LEDs set by the user. After performing those checks, the while loop sends a signal to the LED chain using SPI communication to turn on the appropriate LEDs for the setup phase. After sending this signal, the loop sends the MSP430 into low power mode 0 and waits for an interrupt from any of the buttons to awaken the CPU and start the while loop over. The program leaves this control loop using the timer module. Every time that one of the button interrupts is called, the interrupt sets a timer value, which decrements every time the timer A interrupt is called. If the timer ever gets below a certain threshold, then the timer interrupt sets a boolean value which causes the control while-loop to end. Since the timer gets reset every time an interrupt from one of the buttons is received, the user is able to continue to set the initial state of the simulation as long as they continue to click buttons every few seconds. Then to end the setup phase the user simply needs to wait and after a short amount of time the timer will have decremented low enough that the setup phase ends and the simulation begins.

The simulation itself consists of iterating over every LED and counting how many of the LED's neighbors are on, and then using this value to decide the next state of the LED. There is an array for the current state of the LED and a separate array for the next state of the LED. There is a for loop that iterates over every index in the LED chain, computes the next state of that LED based on its neighbors, and sends a signal to the LED based on its current state. After the loop is performed the current state array is updated to the next state array which was calculated in the iteration. The program then delays a short amount of time so that the user can see the current state of the simulation, and then performs the next round. This cycle is repeated indefinitely, allowing the simulation to run continuously, until the MSP430 is powered down.

In this project we were able to utilize many of the important concepts that we have learned from Elec 327 about embedded system programming including interrupts, SPI communication, low power modes, timer modules, and overall system architecture. In this way, this project was a true accumulation of all that we have learned this semester, as well as a project that exposed us to new challenges and ways of overcoming them. Our final project is functional and something that we are both proud of, and while it took quite a lot of work we both agree was worth it in the end.