INTRODUCTION:

The purpose of this document is to explain the design of the game Tetris on an embedded system for the EE319k final project. EE319k is a course where students are introduced to the basics of embedded systems, and the purpose of the final project is to create an embedded system from scratch, and program it to run a video game. The project is constrained by a list of required hardware components, as well as several required software components. The game Tetris is a complex game that utilizes several inputs to control the game; these inputs can be easily and intuitively adapted to fit within the constraints of the project. Therefore Tetris was chosen as the design solution.

THE PROBLEM

The objective of the project is to first design, test, and debug a large program written in C. Secondly, by requiring the use of several hardware inputs, students can practice I/O interfacing techniques taught in class and previous labs. Additionally, the overall purpose is to design an embedded system that performs a useful task.

The project specifies a list of hardware components that students are required to implement; these components should serve as game inputs and outputs that work together to create a cohesive embedded system video game. The inputs include at least two buttons, at least one slide-pot, a speaker, and an LCD screen. A button is an integral part of any embedded system, and both buttons must affect gameplay; additionally these buttons can be implemented using edge-triggered interrupts, which satisfies a software requirement mentioned later. The slide potentiometer is a prime opportunity for students to practice periodic input sampling, and perform calculations using the Nyquist Theorem. The outputs include a speaker and an LCD display. The speaker should be controlled through a periodic interrupt software routine, which is a routine for any embedded system, and also allows students to practice DAC calculations and sound generation. The LCD screen is perhaps the most important hardware component, as it outputs all visual aspects of the game.

Additionally, the entire system is designed around the TM4C Launchpad, an evaluation board created by Texas Instruments. The Launchpad contains everything a developer needs to quickly

create an embedded system, such as memory, buttons, an ADC, etc. Most importantly, the Launchpad contains a microprocessor, which can easily interface with all the hardware components previously mentioned, and much more; this is the main tool used through the course, and similar boards are a staple within the embedded systems industry.

The project also specifies some required software components. This includes at least three sprites or images that must be dynamically generated on the LCD screen through user inputs, at least two sounds, at least two interrupt service routines, and two languages for the system user interface. All of these software requirements allow students to practice important and common software concepts that are integral parts of any embedded systems engineer's repertoire, and also for any ECE professional in general.

DESIGN SOLUTION

Tetris is a game where players fill rows on a grid by rotating differently shaped pieces that constantly descend onto the play field. Each completely filled row grants the player some points, then disappear, allowing the vacant space to be filled again. The game ends when the uncompleted rows reach the top of the play field, preventing more pieces from descending. The essential gameplay features of Tetris perfectly match the required inputs for the project: the slide potentiometer lets the player move the piece horizontally, one button can rotate the piece, and the other button accelerates the piece quickly to the bottom of the play field. Each input is connected directly to a GPIO port on the TM4C, with the potentiometer going through an analog to digital signal converter. After the TM4C receives these inputs, it performs the necessary calculations and updates the LCD display. There is also constant background music being outputted to the speaker.

The Playing Field

The playing field for this particular iteration of Tetris consists of a grid measuring 20 blocks wide and 35 blocks high, equating to an area of 80x140 pixels. This means that each block measures 4x4 pixels. Within the software, this playing field is represented by a 20 by 35 2D array, with each array element directly representing a square on the field. The value of each array

element also represents the current color of the square, with the default color black being a value of 9. Figure [1] [2] give clear side by side comparisons of the display and the array inside memory. This type of data structure is intuitive and algorithms can be easily written to perform essential tasks in Tetris such as checking filled rows for points, movement of pieces, and current status of the field. There is a variable within the software that acts as an origin point on the grid, dynamically moving around through inputs by the user. This origin point is used to display game pieces.

The Game Pieces

Tetris features 7 pieces in total, each with a different shape and color. There are four structs that contain data about the pieces. Each struct is associated with a particular rotation of the pieces (0, 90, 180, 270 degrees). Each struct features 7 elements, one for each of the unique game pieces; each element contains data on which piece it is, the color of the piece, and a set of offset values that can be fed into a helper function to correctly print out the piece onto the LCD display. These offset values are also used to calculate collision between individual pieces, namely to stop the current piece from moving when it hits something.

Helper Functions

There are two helper functions, *printPixel( )* and *printPiece( )*. The *printPixel( )* function has three inputs, an x value, a y value, and a color code. Despite its name, *printPixel( )* actually prints 16 pixels onto the LCD screen in the shape of a square. Recall from a previous section that each block in the playing field is 4x4 pixels, *printPixel( )* effectively prints one block on the field based on the x and y coordinate in the appropriate color, based on the inputs passed in. The *printPiece( )* function is a more complicated version of *printPixel( )*, as it calls *printPixel( )* several times to print a game piece. Recall from a previous section that each game piece is associated with a set of offset values, as well as the dynamic origin point. These offset values are used in conjunction with the origin point to print a game piece with the correct orientation at a specific point on the playing field. Figure [3] shows this process.

User Inputs

Through the duration of the game, the user can use the buttons and slide-pot that are interfaced to the system to control the game pieces moving on the board. Both types of inputs are sampled through interrupts, meaning the program will stop running and service the assigned subroutines before returning to the main program. In the case of the buttons, there are two functions: piece rotation and acceleration. One button can be pressed to rotate the pieces, this simply changes which struct the software access, which corresponds with different offset values for the game pieces. The other button, when pressed, will prevent all other inputs and immediately move the current piece down to the bottom of the playing field (or until there is a collision). For the slide-pot, its value is measured and reduced to a number between 0 and 19. This number is then used to move then used to move the game piece horizontally on the board. This is achieved by changing the y value of the dynamic origin point to correspond with the converted ADC value. Additionally, the buttons also serve as menu navigators, with one to cycle through different menu options, and another to confirm the selection.

The Main Software Loop

A main menu is first displayed by the software, allowing the user to select a language between English and Spanish, and starting the game. Once the user starts the game, the playing field is displayed, and pieces begin falling from the top. Each piece is randomly generated by a random number generator, and loaded into a three value array acting as a buffer to simulate the preview feature displayed on the right side of the playing field.

The top of the buffer is chosen as the current active piece. As the piece is falling from the top, the software first uses the appropriate offsets to check the blocks below the piece to detect any collisions. If no collisions exist, the piece may safely descend down one block. The descending motion of the game piece is simulated by deleting the current game piece, moving the origin point down by one, and reprinting the game piece at the new origin point. This process loops continuously until the piece collides with either another piece of the bottom of the playing field, at which point the loop ends, and the piece stops moving. Then the proper array value is updated; the buffer is updated by deleting the first array element, pushing the two remaining values forward, and refilling the last empty element with a new randomly generated piece; and the next piece is chosen, and the piece descends again. This process can be observed in Figure [2].

Additionally, the program periodically checks the entire playing field for filled rows, and any piece that has reached top. If a row is filled, then that row is cleared, points are added, and all values in the array are shifted down by one row. If the program detects that a piece has reached the top, then the game ends, and a new screen is displayed, showing the score and prompting the user to restart.

The Speaker

The speaker is interfaced through a 6 bit DAC directly to the TM4C. The software has two arrays containing values that generate a sound. The first one is a background music that plays throughout the entire game, and the second is a sound that plays when a row is filled and points are earned. These sounds are generated and outputted by a periodic interrupt.

CONCLUSION

While this project seems somewhat simplistic, with a strict pre-made design for the system, it actually allows students to explore a real-world example of creating an embedded system, and challenges that go along with that. This project highlights how important it is to understand each component of a system deeply, how they work and how to program them. The project also helps students understand just how complex a seemingly simple system really is, and how tedious the process for creating one can be without proper design. Additionally, several improvements can be made to the game's software and hardware design. The collision detection algorithm and helper functions can likely be optimized for better performance, and the entire software architecture can likely be redesigned to be more efficient. Hardware components can also be improved, such as a more sophisticated DAC and Amp circuit to drive the speaker for improved sound quality, as well as an enclosure for the system to improve aesthetic appearance.

Figure 1. The "I" Piece Stored In Memory

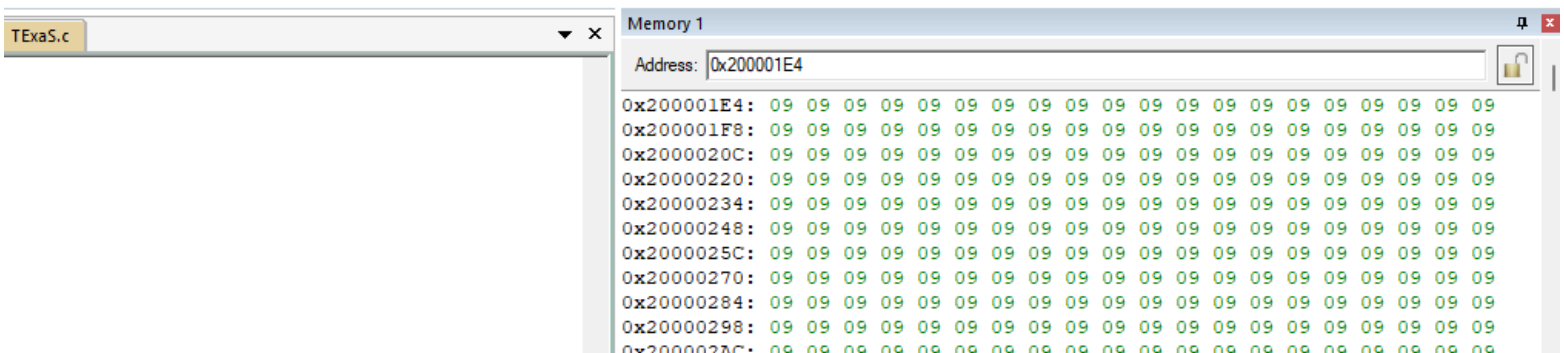

Figure 2. The “L” Piece Colliding With The “I” Piece

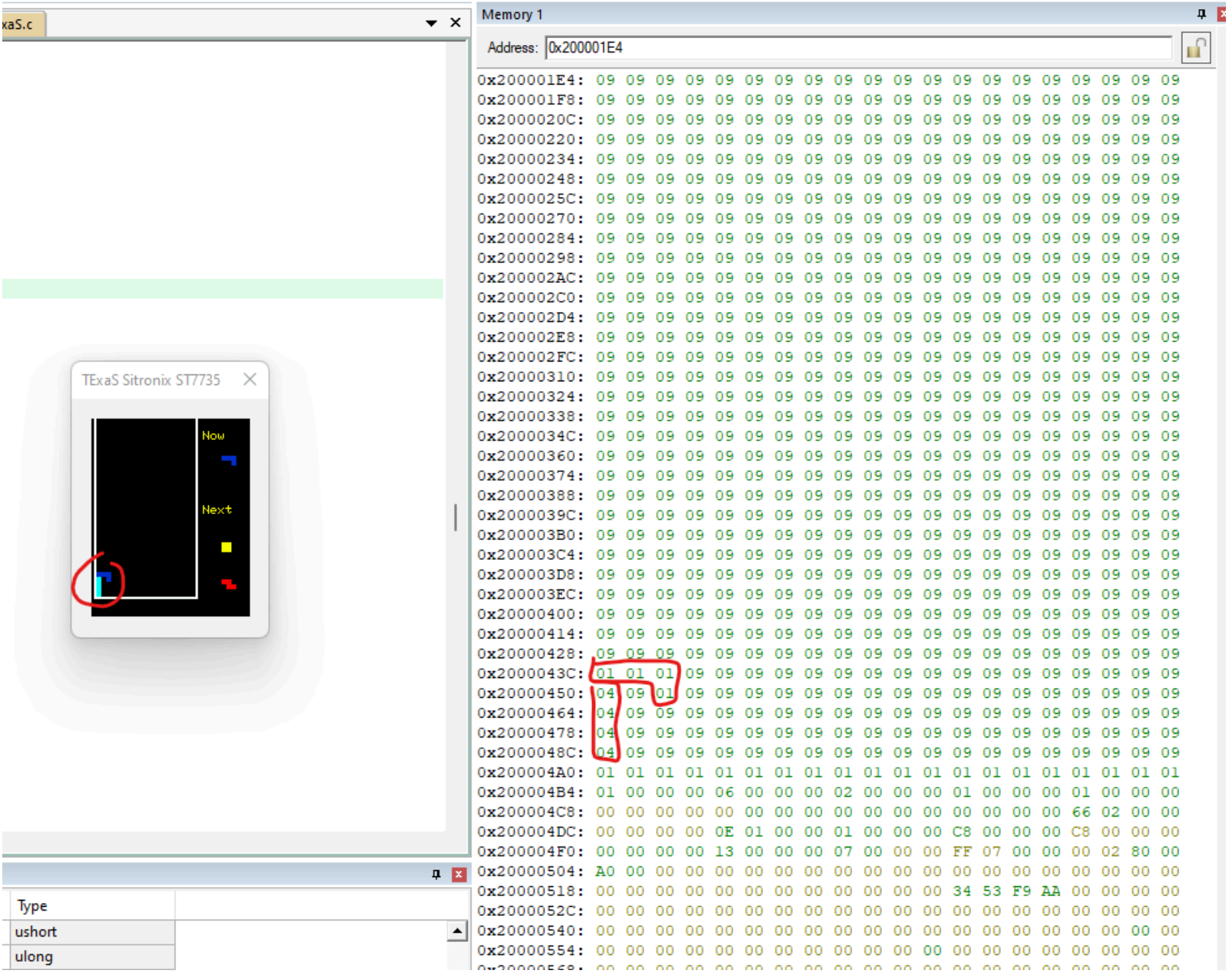

Figure 3. ***printPiece( )*** **Inner Workings**