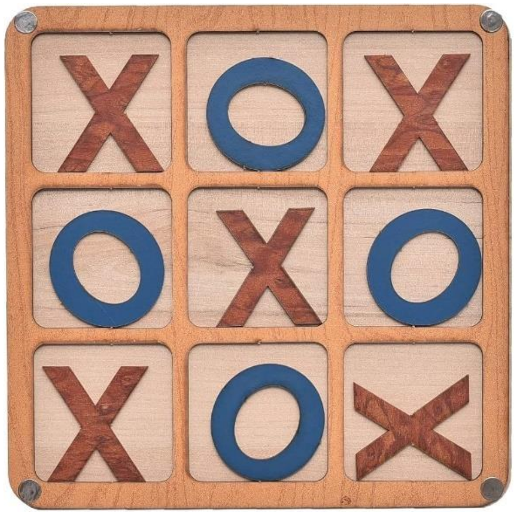
كلية الهندسة بشبرا
FACULTY OF ENGINEERING AT SHOUBRA

XO Game Using TM4C123GH6PM Microcontroller

El-Osood El-Mefrhda | Embedded Systems | December 22, 2024

Supervised By:

Dr. Lamiaa, Eng. Mahmoud Nawar



<< Team Members >>

1. عبد الرحمن أسامة عادل سعد	(13)
2. محمد علاء أحمد رفاعي	(30)
3. محمد حمدي عبداللاه حماد	(27)

Introduction

The XO game, also known as Tic-Tac-Toe, is a simple and popular two-player game played on a 3x3 grid. The objective is to align three marks (either 'X' or 'O') in a row, column, or diagonal before the opponent does. This project demonstrates the implementation of the XO game on the TM4C1234GH6PM microcontroller, incorporating various hardware modules for an interactive and engaging user experience.

The game features dual control methods: one using a potentiometer, interfaced with the microcontroller's ADC module to select grid positions, and the other via a keyboard through the UART module. A switch is provided to confirm the selection, and visual feedback is provided through a Nokia 5110 LCD display. Additionally, LEDs are used to enhance the game experience, and the DAC module controls these LEDs to display various states during the game.

This system demonstrates the integration of various peripherals on the TM4C1234GH6PM microcontroller, showcasing how analog and digital interfaces can work together to create a functional game environment.

System Overview

❖ GPIO MODULES:

1. General Purpose Input/Output (GPIO) Overview

The GPIO ports on the TM4C1234GH6PM microcontroller serve as the primary interface for connecting external devices and peripherals. They are versatile and can be configured as input or output pins depending on the application's requirements. In this project, GPIO modules played a critical role in interfacing with switches and external LEDs, enabling user input and visual feedback.

2. GPIO Ports Used in the XO Game

2.1. PORTF: Switches for Position Selection

- **Purpose**: PORTF was utilized for handling user inputs via switches. These switches allowed the user to select the position on the XO grid where they wanted to draw their marker. And the Position changing is controlled via either UART communication with the Computer the MCU is connected to, or by the Potentiometer connected to the MCU.
- **Configuration**:
  - **Pins used:** PF0 and PF4.
  - **Mode:** Configured as input pins.
  - **Pull-up Resistors**: Internal pull-up resistors were enabled to ensure stable high logic when the switches are not pressed.
  - **Interrupts**: Configured to trigger interrupts on falling edges (switch press) for precise and real-time user input handling.
- **Functionality**:
  - **PF0** and **PF4** were mapped to specific actions in the game logic. For instance:
    - **PF0**: Selects or confirms the position.
    - **Both of Them:** They are use in some places throughout the game so the user can choose such as, when a game ends the user chooses to continue or not.
  - **Debouncing**: Software debouncing was implemented to avoid multiple detections from a single press. This one was critical in the Edge Interrupt to avoid undesired actions.

2.2. PORTD: External LEDs for Visual Feedback

- **Purpose**: PORTD was employed to control external LEDs, which provided visual cues during gameplay. These LEDs were designed to operate in a structured manner, enhancing the user experience by visually indicating the game's state.
- **Configuration**:
  - Pins used: Pins 1, 2, 3 of Port-D (`PD1-PD3`).
  - Mode: Configured as output pins.
  - Driving strength: Configured to provide sufficient current for external LEDs.
- **Functionality**:
  - LEDs acted in a **predefined pattern**, not random, to indicate specific game states:
    - **X's Turn**: A specific LED or set of LEDs illuminated when it was Player X's turn.
    - **O's Turn**: A different LED or set of LEDs illuminated when it was Player O's turn.
    - **Draw Case**: A unique pattern of LEDs activated when the game resulted in a draw.
    - **Win Condition**: Specific LEDs illuminated to celebrate the winning player.
  - These actions provided immediate visual feedback, ensuring players were aware of the game's current state without solely relying on the LCD display.

❖ TIMER:

1. General Purpose Timer Overview

The TM4C123GH6PM microcontroller includes several General-Purpose Timer Modules (GPTM) designed to handle timing and counting tasks. These timers are highly versatile and can be configured for a variety of modes such as periodic, one-shot, and capture. In this project, the GPTM is utilized primarily for introducing precise delays in the game logic using **Timer2** in **32-bit periodic mode**. Additionally, timer interrupts are used to handle specific game conditions, ensuring efficient and non-blocking execution.

2. Timer Configuration in the XO Game

2.1. Timer Block Used

- **Timer Block**: 16/32-bit Timer2
- **Mode**: 32-bit periodic mode
- **Purpose**:
  - Generate precise delays for the game's logic flow.
  - Handle specific game conditions using interrupts, such as timeout signals.

2.2. Mode Selection

- **Periodic Mode**: Ensures the timer runs continuously, resetting itself after reaching the timeout value, which is ideal for delay generation.
- **Snapshot Mode**: Although more efficient for simple delays, it is not supported directly with 32-bit periodic mode in the KEIL environment when combined with UART functionality. Hence, 32-bit periodic mode was selected as a practical alternative.

3. Timer Implementation

3.1. Timer Initialization

The `Timer2_delay_ms()` function sets up and starts the timer for a specified delay in milliseconds. Here’s the detailed breakdown of the configuration:

General Methodology for Timer Configuration

The Timer module is used in the XO game to introduce precise delays for controlling game logic and flow. Below is the methodology we followed for configuring the General-Purpose Timer 2 (GPTM) as a **32-bit periodic timer** with interrupt support:

1. **Activate Timer2**:
   To begin, the clock for Timer2 is enabled by setting the corresponding bit in the `SYSCTL_RCGCTIMER_R` register. A short delay is introduced to ensure the clock stabilizes before further configuration.
2. **Disable Timer During Setup**:
   To avoid unintended behavior during the configuration process, Timer2A is disabled by clearing the `CTL` (Control) register bit. This step ensures that the timer does not begin counting prematurely.
3. **Configure Timer as 32-bit**:
   The timer is configured in 32-bit mode by setting the `CFG` (Configuration) register to `0x00`. This allows for extended timing ranges suitable for the game's logic.
4. **Set to Periodic Mode**:
   Periodic mode is selected by modifying the `TAMR` (Timer A Mode) register. This mode ensures that the timer will automatically reload and continue running after reaching the timeout value, making it ideal for generating repetitive delays.
5. **Reload Value Calculation**:
   The `TAILR` (Timer A Interval Load) register is loaded with a calculated value based on the desired delay. This value is computed as `(80000 * period_ms) - 1`, assuming an 80 MHz system clock, which translates to a clock period of 12.5 ns. The result ensures that the timer generates the required delay in milliseconds.
6. **Interrupt Configuration**:
   Interrupts are enabled to handle specific conditions when the timer expires. The following steps are performed:
   - **Clear Timeout Flag**: The timeout flag is cleared initially to ensure no pending interrupts interfere with operation.
   - **Enable Timeout Interrupt**: The timeout interrupt is enabled in the `IMR` (Interrupt Mask) register for Timer2A.
   - **Set Interrupt Priority**: The priority of the interrupt is configured using the NVIC (Nested Vectored Interrupt Controller) priority register.
   - **Enable NVIC IRQ**: The interrupt is enabled in the NVIC to allow it to trigger when the timer expires.
7. **Start Timer**:
   Once the timer is fully configured, it is started by enabling Timer2A in the `CTL` register. This allows the timer to begin counting and functioning as required.
8. **Semaphore Handling**:
   A global variable, `Semaphore`, is used to synchronize the timer's operation with the main program. The main program waits in a loop (`while(Semaphore == 0)`) until the interrupt handler signals that the delay has expired.

Interrupt Handler Methodology

The interrupt handler is responsible for managing the timer’s timeout event and signaling the main program. The methodology for the interrupt handler is as follows:

1. **Acknowledge Timeout**:
   When the timer expires, a timeout flag is set in the `RIS` (Raw Interrupt Status) register. This flag is acknowledged and cleared by writing to the `ICR` (Interrupt Clear) register. Clearing this flag prevents repeated interrupts from being triggered.
2. **Signal Timeout**:
   The `Semaphore` variable is set to 1 within the interrupt handler. This action signals the main program that the delay has completed and it can proceed with its next task.

❖ UART:

Purpose

The UART module was utilized to facilitate user control over the game grid. Through the UART interface, the user can navigate the cursor position up, down, right, or left across the XO grid. This module enables smooth communication between the user and the microcontroller.

Configuration Steps

1. **Activating UART and GPIO Ports**
   - The UART0 peripheral and GPIO Port A were enabled to establish the communication interface.
   - This configuration allows the UART module to send and receive data through pins PA0 (RX) and PA1 (TX).
2. **Baud Rate Setup**
   - The UART was configured for a baud rate of 19200, assuming a system clock frequency of 80 MHz.
   - The integer and fractional parts of the baud rate divisor were calculated using the formula:
     - Baud Divisor = System Clock Frequency / (16 × Baud Rate)
     - Integer part: Floor(Baud Divisor)
     - Fractional part: (Fraction x 64 + 0.5)
3. **Data Format Configuration**
   - The UART module was set to an 8-bit word length with no parity bits, one stop bit, and FIFOs enabled.
   - This ensures reliable and efficient data transfer between the user interface and the microcontroller.
4. **Pin Configuration**
   - PA0 and PA1 were configured for UART functionality through alternate functions.

    - Digital functionality was enabled on these pins, while the analog functionality was disabled to ensure compatibility with UART.

5. **Interrupt Setup**
    - The UART interrupt was enabled to handle incoming data.
    - The interrupt priority was set to ensure efficient processing within the system.
    - NVIC configurations were updated to allow interrupt-driven communication.

Operation of the UART Module

- **Receiving Data**:
  The UART module listens for incoming data from the user interface. When a character is received, it is stored in a mailbox variable for further processing.
- **Interrupt Handling**:
    - Upon receiving data, the UART interrupt handler is triggered.
    - The received character is processed, and a flag is set to indicate the availability of new data.
    - The interrupt flag is cleared to prepare for the next data reception.
- **Controlling Grid Navigation**:
  The received data is interpreted as navigation commands (up, down, right, left), which are subsequently used to update the cursor position on the game grid.

❖ ADC:

Overview

This part of report provides an analysis of the ADC (Analog-to-Digital Converter) module implemented on the TM4C123 microcontroller. The module enables sampling of an analog signal through ADC channel 1 and processes it for various applications. It features periodic sampling triggered by software and integrates interrupt handling for efficient data acquisition.

Features

1. **Software-Triggered Sampling:**

    - The ADC is configured to operate in software trigger mode for controlled sampling.

2. **Channel-Specific Input:**

    - Samples analog signals from channel 1 (PE3).

3. **Interrupt Handling:**
    - Utilizes interrupts to handle end-of-conversion events efficiently.

4. **Global Variable for Data Access:**
    - Stores sampled values in a global variable ADCvalue, enabling external modules to access the processed data.

5. **Integration with GPIO:**
    - Configures GPIO pins to support analog input functionality.

Detailed Analysis

1. Initialization

Function: ADC0_InitSWTriggerSeq3_Ch0

This function initializes ADC0 with the following steps:

1. **Clock Configuration:**
    - Enables the clock for GPIO Port E and ADC0.
    - SYSCTL_RCGCGPIO_R |= (1<<4);   // Enable Clock to GPIOE
    - SYSCTL_RCGCADC_R |= (1<<0);   // Enable ADC0 clock

2. **GPIO Configuration:**
    - Configures PE3 for analog input:
        - Enables the alternate function.
        - Disables digital functionality.
        - Enables analog functionality.
    - GPIO_PORTE_AFSEL_R |= (1<<3); // Enable alternate function
    - GPIO_PORTE_DEN_R &= ~(1<<3); // Disable digital function
    - GPIO_PORTE_AMSEL_R |= (1<<3); // Enable analog function

3. **ADC Sequencer Setup:**

  - Configures sequencer 3 for software triggering.
  - Maps channel 0 (PE3) to sequencer 3.
  - Sets flags for single-sample acquisition.
  - ADC0_ACTSS_R &= ~(1<<3);   // Disable SS3 during configuration
  - ADC0_EMUX_R &= ~0xF000;   // Software trigger conversion
  - ADC0_SSMUX3_R = 0;   // Input from channel 0
  - ADC0_SSCTL3_R |= (1<<1)|(1<<2); // Single sample, set flag

4. **Interrupt Configuration:**
   - Unmasks ADC interrupt for sequence 3.
   - Enables NVIC interrupt for ADC0 sequence 3.
   - ADC0_IM_R |= (1<<3);   // Unmask interrupt
   - NVIC_EN0_R |= 0x00020000;   // Enable IRQ17 for ADC0SS3

5. **Enable ADC Sequencer:**
   - Reactivates sequencer 3 to begin sampling.
   - ADC0_ACTSS_R |= (1<<3);   // Enable sequencer 3
   - ADC0_PSSI_R |= (1<<3);   // Start sampling

---

2. Interrupt Service Routine

Function: ADC0Seq3_Handler

This ISR processes ADC conversion results and prepares the system for the next conversion cycle.

1. **Read Conversion Result:**
   - Stores the result from the sequence 3 FIFO into ADCvalue.
   - ADCvalue = ADC0_SSFIFO3_R; // Read result from SS3 FIFO

2. **Clear Interrupt Flag:**
   - Clears the interrupt flag to acknowledge the end of the current conversion.

    - ADC0_ISC_R |= 0x08; // Clear conversion complete flag
3. **Restart Conversion:**
    - Triggers a new conversion for continuous sampling.
    - ADC0_PSSI_R |= 0x08; // Start new conversion

Global Variables

volatile unsigned int ADCvalue

- Represents the digital value corresponding to the analog input on PE3.
- Scaled based on the input voltage:

Voltage (V)	ADCvalue
0.00	0
0.75	1024
1.50	2048
2.25	3072
3.00	4095

Integration with System

1. **GPIO Setup:**
    - PE3 is configured as the input channel for ADC0.
2. **Interrupt-Driven Sampling:**
    - Efficiently captures data without blocking the CPU.
3. **Access from External Modules:**
    - Other system components can use ADCvalue to make decisions based on the sampled input.

4. **Condition Handling:**
    - Resets ADCvalue when F2 is active, ensuring valid data representation.

Strengths and Benefits

1. **Efficient Sampling:**
    - Interrupt-driven approach ensures non-blocking operation.
2. **Scalable Design:**
    - Can be extended to additional channels or higher sampling rates.
3. **Real-Time Data:**
    - Provides continuous and up-to-date information about the analog input signal.
4. **Modular Initialization:**
    - Clear separation of concerns allows easy reuse and modification.

Areas for Improvement

1. **Error Handling:**
    - Add checks for overflow or invalid ADC states.
2. **Dynamic Configuration:**
    - Enable dynamic channel selection to support multiple inputs.
3. **Power Optimization:**
    - Implement low-power modes during idle periods to reduce energy consumption.
4. **Sampling Rate Adjustment:**
    - Allow runtime adjustment of the sampling frequency for adaptive operation.

❖ DAC:

Overview

This report explains the functionality, design, and implementation of the Sound.c and DAC.c modules for generating audio signals on a microcontroller (TM4C123GH6PM). The Sound module utilizes the DAC (Digital-to-Analog Converter) for producing periodic sound outputs based on specific parameters. Both modules integrate seamlessly to handle sound generation, including initialization, configuration, and interrupt-driven output updates.

Sound.c

Purpose

The Sound.c module is responsible for generating sound signals through the DAC. It uses the SysTick timer to produce periodic interrupts, enabling the DAC to output audio signals based on a predefined waveform.

Key Components

1. **Sine Wave Array**:
2. const unsigned char sinWave[16] = {255, 0, 255, 0, 255, 0, 255, 0, 255, 0, 255, 0, 255, 0, 255, 0};
   - Represents a simple square wave pattern used to generate sound.
   - The array is cyclically accessed to simulate a waveform.
3. **Global Variables**:
   - Index: Tracks the current position in the sinWave array.
   - turn_Mark: Indicates the current player's turn (X or O).
   - flagDraw: Indicates whether the game is in a draw state, influencing sound output behavior.
4. **Functions**:

a. Sound_Init():

- Initializes the SysTick timer and DAC.
- Configures SysTick for periodic interrupts.

b. Sound_Tone():

- Configures the SysTick timer reload value to set the sound frequency.
- Updates the game state variables (turn_Mark, flagDraw).

c. Sound_Off():

- Stops sound output by disabling SysTick interrupts.

d. SysTick_Handler():

- Interrupt Service Routine (ISR) executed at each SysTick interrupt.
- Cyclically updates the Index variable to output the next value in the sinWave array.
- Sends the waveform data to the DAC via DAC_Out().

DAC.c

Purpose

The DAC.c module handles the output of digital signals to the hardware DAC, which converts them into analog signals. This conversion is necessary for generating audio signals.

Key Components

1. **GPIO Port Configuration**:
   - The 4-bit DAC utilizes GPIO Port D for output, controlled via specific pin configurations (e.g., GPIO_PORTD_DATA_R).
2. **Functions**:

a. DAC_Init():

- Prepares the hardware GPIO pins for DAC output.
- Ensures proper digital functionality by disabling alternate functions and analog modes.

b. DAC_Out():

- Outputs the waveform data to the DAC.
- Takes additional parameters (turnMark, FlagDraw) to differentiate game states:
  - **Draw State (FlagDraw)**:
    - Outputs a fixed signal pattern based on the game state.
  - **Player Turns (turnMark)**:

        - Adjusts the output signal for X or O.
    - Controls Port D pins based on the output data and state.

Integration

1. **Sound Generation Process**:
    - Sound_Init() initializes the system by setting up SysTick and DAC.
    - During gameplay, Sound_Tone() configures the sound frequency and updates the state.
    - The SysTick timer triggers periodic interrupts, and SysTick_Handler() cycles through the sinWave array, sending data to DAC_Out() for sound output.
2. **Waveform Output**:
    - The DAC receives digital values (0-255) from the sinWave array.
    - The corresponding analog signal is generated and sent to connected speakers or audio devices.
3. **Game State Influence**:
    - The DAC_Out() function dynamically adjusts the output based on game states (turnMark or flagDraw), adding an interactive element to the sound.

System Design

Hardware

- **Microcontroller:** TM4C123GH6PM.
- **Peripherals:**
    - GPIO Port D for DAC output.
    - SysTick timer for periodic interrupts.

Software

- **Modules:**
    - Sound.c handles sound generation and timing.

    - DAC.c manages digital-to-analog conversion.
- **Interrupts:** SysTick timer ISR drives waveform updates.

Code Flow Summary

1. **Initialization:**
    - Sound_Init() initializes the DAC and SysTick timer.
    - DAC is configured to output signals through GPIO pins.
2. **Sound Playback:**
    - Sound_Tone() adjusts the SysTick reload value to control sound frequency.
    - SysTick_Handler() outputs the waveform data to the DAC at each interrupt.
3. **Output Control:**
    - DAC_Out() translates digital data into analog signals based on the game state.
4. **Game Interaction:**
    - Game states (turnMark, flagDraw) influence the sound output, providing audio feedback for player actions and outcomes.

❖ GAME LOGIC:

Features

1. **Graphical Interface:**
    - Utilizes the Nokia 5110 display to render the game board and various statuses.
    - Bitmap-based rendering for hover effects and cell states (X, O, empty).
2. **Player Interaction:**
    - Supports both human input (via ADC or switches) and computer-controlled gameplay.
    - Displays current player's turn and tracks scores.

3. **Game Modes:**
    - Potentiometer-based selection (RunGameADC).
    - UART-based gameplay (RunGameUART).
4. **Game Outcomes:**
    - Checks for winning conditions and highlights winning sequences.
    - Displays a draw when the board is full without a winner.
5. **Sound Effects:**
    - Different sound tones for actions like winning or completing a game.
6. **Replay Options:**
    - Prompts the user to replay or end the game after a round.

Detailed Analysis

1. Initialization

GameIntro

- Introduces the game with a sequence of screens displaying project credits and team details.
- Includes a welcoming animation using the Nokia 5110 display.
- Delays (Timer2_delay_ms) are used to create a dynamic visual effect.
- Provides detailed acknowledgments, including supervisor and team member names, creating a professional and engaging introduction.

PlayXO

- Allows players to select the game mode:
    - Potentiometer mode (selected via SW1).
    - Computer mode (selected via SW2).
- Displays relevant messages and transitions based on user choice.
- Implements debounce handling by waiting for a stable input before proceeding with mode selection.

- Clearly informs players of the selected mode and initial turn using visual feedback.

GameInitialization

- Resets the game state:
    - Clears the game board (GameMatrix) by setting all cells to empty (' ').
    - Sets the initial turn to X, ensuring consistency in starting rules.
    - Configures sound to the normal playing rate (rateOrdaniry).
    - Sets the currentCell to -1, indicating no cell is currently hovered or selected.

DrawClearGameMatrix

- Draws the empty game grid using the square bitmap for each cell.
- Updates the display buffer iteratively for all nine cells with appropriate delays for visual effect.
- Finalizes the grid by adding lines to separate cells and initializes the hover position at the first cell.
- Ensures the visual status of the game board reflects the reset state.

---

2. Gameplay Logic

RunGameADC

- Uses ADC values to determine the currently hovered cell based on dividing the ADC reading by 420:
    - Limits the hover range to valid cell indices (0-8).
- Updates the hover display by clearing the previous hover state and highlighting the new cell.
- Handles player moves when Sw2Flag is triggered:
    - Validates the selected cell is empty before marking it.
    - Updates the game state (GameMatrix) and redraws the cell with the appropriate player's mark (X or O).
- Checks for game outcomes after every move:
    - Invokes checkWinner to identify winning conditions.

    - Calls displayDraw when all cells are filled without a winner.
- Switches turns between players and provides hover feedback for the new turn.

RunGameUART

- Enables UART-based gameplay where players input commands:
    - 6: Move right.
    - 4: Move left.
    - 8: Move up.
    - 2: Move down.
    - 5: Select cell.
- Updates the hovered cell based on commands and reflects changes on the display.
- Similar gameplay mechanics to RunGameADC for marking cells, switching turns, and checking outcomes.
- Provides UART-based textual feedback for all actions, enhancing debugging and user clarity.

---

3. Game Outcome Handling

checkWinner

- Evaluates all eight possible winning combinations (rows, columns, diagonals).
- Highlights winning cells with a visual overlay (selectedX or selectedO) for emphasis.
- Draws a strike line over the winning sequence using Nokia5110_SetPixel for a clear visual effect.
- Returns 1 if a winner is found, 0 otherwise, allowing seamless integration with other functions.

Display_Winner

- Increments the score of the winning player (xWins or oWins) and updates the display.
- Announces the winner with a clear message and delays for user acknowledgment.
- Displays updated scores, ensuring players are informed of the current standings.

displayDraw

- Displays a "Game is a draw" message when no winner is determined and all cells are filled.
- Clears the screen after a brief delay to prepare for replay or game end.

CheckPlayAgain

- Prompts the user to decide whether to replay or exit.
- Displays options for Yes (SW1) or No (SW2) and waits for valid input.
- Returns 1 for replay and 0 for exit, ensuring smooth control flow.

EndGame

- Displays a "Goodbye" message if the player chooses not to replay.
- Clears the screen to signify the end of the session.

Graphics and Interaction

- **Nokia 5110 Display:**
    - Handles all visual elements, including the game board, player status, and endgame messages.
    - Uses bitmap rendering for cell states, hover effects, and winning highlights.
- **Switches (SW1, SW2):**
    - Serve dual purposes: mode selection and gameplay input.
    - Debounced to prevent accidental triggers.
- **Sound Feedback or Light Feedback:**
    - Provides auditory cues for actions like marking a cell, winning, or ending the game.
    - Differentiates sounds based on context (e.g., winning vs. draw).