TM4C123GH6PM-based Pong Game

Embedded Systems Project

1 – Shady bahaa El-kiss

2- Mostafa Shokry Elkamel

3 - Mostafa Abdallah Mostafa

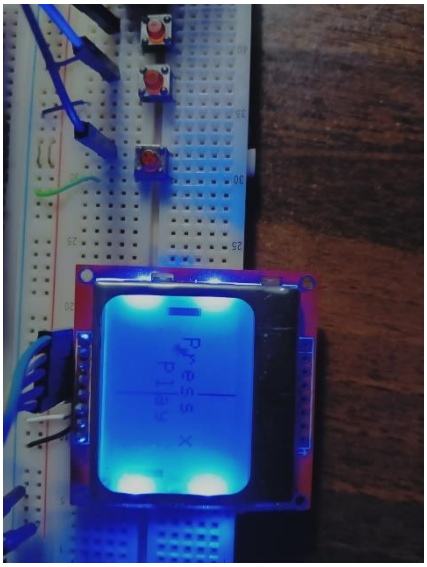
4 – Marwan Mohamed Shaaban

5- Eman Mohamed Abdel-hamid

About the project

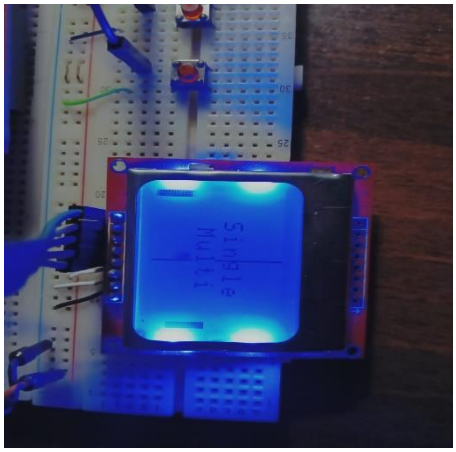
How to play it

Step 1: Initial Setup



1. **Power On the Device:** Ensure the TM4C123GH6PM board is powered on and connected to the display (Nokia 5110 in this case) and other peripherals (buttons, buzzer, etc.).
2. **Display the Opening Screen:** When the game starts, the opening screen displays an option to press a button (OK_PIN) to start the game. The message "Press x" and "Play" will alternate on the screen.

Step 2: Choose Game Mode



1. **Select Game Mode:**
    - Press the OK button (connected to OK_PIN) to navigate to the mode selection menu.
    - Use the UP and DOWN buttons (connected to

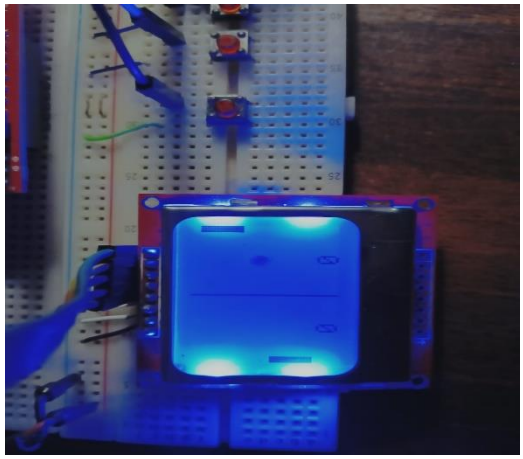
UP_PIN and DOWN_PIN) to switch between "Single" and "Multi" player modes.

- Press the OK button again to confirm the selection.

Step 3: Gameplay

Single Player Mode:

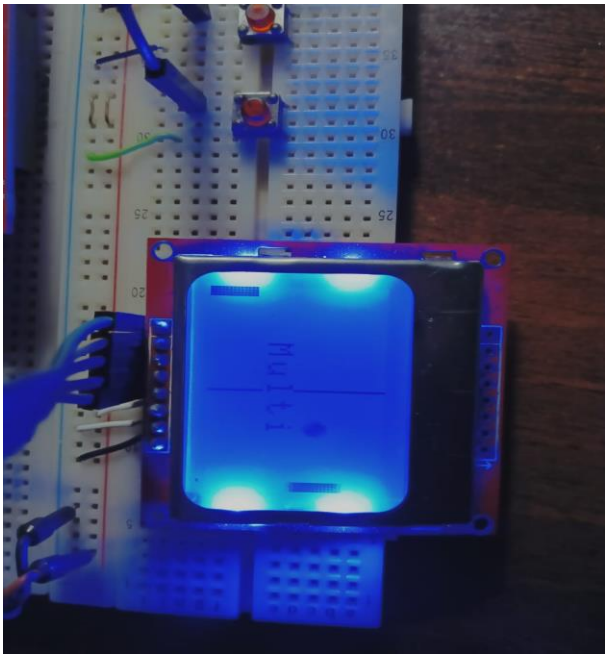
1. **Control Paddle:**
    - Use the UP and DOWN buttons to move your paddle (Paddle One) up and down on the screen.



2. **Game Mechanics:**
    - The ball moves automatically. It bounces off the paddles and the top and bottom edges of the screen.
    - If the ball passes your paddle, the other paddle scores a point. If it passes the other paddle, you score a point.
3. **Winning the Game:**
    - The game continues until one player reaches 15 points.
    - The screen will display "Player 1 Wins" or "Player 2 Wins" based on the final score.

Multiplayer Mode:

1. **Control Paddles:**
    - Player 1 uses the UP and DOWN buttons to control Paddle One.
    - Player 2 sends commands via Bluetooth to move Paddle Two up (sending 'w' or 'W') or down (sending 's' or 'S').



2. **Game Mechanics:**
    - The ball behaves the same as in single-player mode, bouncing off paddles and screen edges.
    - Each time the ball passes a paddle, the opposing player scores a point.
3. **Winning the Game:**
    - The game ends when one player reaches 15 points.
    - The screen will display "Player 1 Wins" or "Player 2 Wins" based on the final score.

Step 4: Restart or Exit

1. **Restarting the Game:**

- After a game ends, the winner is displayed.
- Press the OK button to return to the main menu and choose the game mode again.

2. **Exiting:**
    - To turn off the game, simply power down the TM4C123GH6PM board.

Controls Summary:

- **OK Button:**
    - Navigate menu and confirm selections.
- **UP Button:**
    - Move Paddle One up.
- **DOWN Button:**
    - Move Paddle One down.
- **Bluetooth Commands (Player 2 in Multiplayer Mode):**
    - Send 'w' or 'W' to move Paddle Two up.
    - Send 's' or 'S' to move Paddle Two down.

Key Functions in the Code:

- **APP_voidPlaySingle():** Manages the single-player game logic.
- **APP_voidPlayMulti():** Manages the multiplayer game logic.
- **APP_u8DisplayOpening():** Displays the opening screen and handles the menu navigation.

- **APP_voidDrawFillRect():** Draws filled rectangles for the paddles.
- **APP_voidDrawFillCircle():** Draws the ball as a filled circle.
- **APP_u8CheckBallRectCollide():** Checks for collisions between the ball and paddles.
- **APP_voidRefreshIRQ():** Handles the refresh rate interrupt to update the game state.

APP Layer

1. APP_voidPlaySingle()

Purpose:

Manages the single-player game logic, including movement of the player's paddle, AI paddle, and ball.

Key Responsibilities:

- Initialize game variables.
- Handle paddle movements.
- Handle ball movement and collision detection.
- Update the score and check for a win condition.

2. APP_voidPlayMulti()

Purpose:

Manages the multiplayer game logic, including movement of both players' paddles and ball.

Key Responsibilities:

- Initialize game variables.
- Handle paddle movements for both players.
- Handle ball movement and collision detection.
- Update the score and check for a win condition.

3. APP_u8DisplayOpening()

Purpose:

Displays the opening screen and handles the menu navigation.

Key Responsibilities:

- Display the initial screen.
- Handle button input to navigate the menu.

4. APP_voidDrawFillRect()

Purpose:

Draws a filled rectangle on the display, used for drawing paddles.

Key Responsibilities:

- Draw rectangles (paddles) at specified positions.

5. APP_voidDrawFillCircle()

Purpose:

Draws a filled circle on the display, used for drawing the ball.

Key Responsibilities:

- Draw circles (ball) at specified positions.

6. APP_u8CheckBallRectCollide()

Purpose:

Checks for collisions between the ball and paddles.

Key Responsibilities:

- Determine if the ball has collided with any paddle.

7. APP_voidRefreshIRQ()

Purpose:

Handles the refresh rate interrupt to update the game state.

Key Responsibilities:

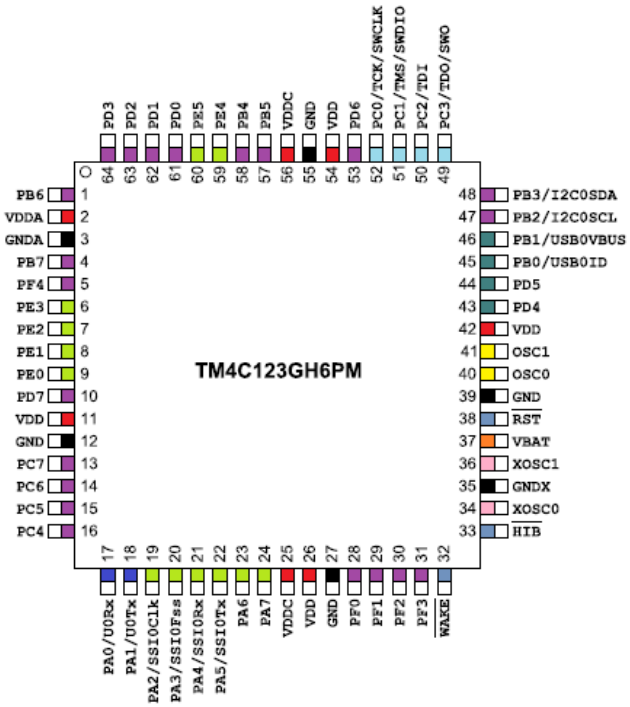
- Refresh the display at a set interval.

MCAL Layer

GPIO Driver

The TM4C123GH6PM microcontroller has a built-in peripheral called **General Purpose Input/Output (GPIO)**. This peripheral allows you to control the digital pins on the microcontroller. These pins can be configured to:

- **Input:** Receive digital signals from external devices like sensors or buttons.
- **Output:** Send digital signals to external devices like LEDs or actuators.



There are several key things to understand about GPIO in TM4C123GH6PM:

- **Multiple Ports:** The microcontroller has six GPIO ports labeled Port A, B, C, D, E, and F. Each port has a different number of pins.

- **Configurability:** You need to configure the GPIO pins before using them. This involves setting them as input or output and defining other parameters like pull-up resistors.
- **Muxing:** Some pins can be shared with other peripherals on the chip like timers or analog-to-digital converters (ADC). You need to configure the pin to be used by GPIO if you want to use it for digital input/output.

This driver helps to interact with the peripheral. Here is a breakdown of it:

Global Variables:

- `GPIO_PORTS[]`: This is an array of pointers to the GPIO registers for each port (A, B, C, D, E, F).
- `GPIOCfgs`: This is a pointer to a configuration structure that likely holds information about the state and configuration of each GPIO pin.
- `InterruptHandlers` This is a 2D array that stores interrupt handler functions for each pin.

Non-ISR Functions:

- `GPIO_Init()`: This function initializes the GPIO driver by enabling clock for all GPIO ports, configuring pins based on the configurations provided, and setting up interrupt handlers (if any).

- `GPIO_EnablePort()`: This function enables the clock for a specific GPIO port.
- `GPIO_DisablePort()`: This function disables the clock for a specific GPIO port.
- `GPIO_EnablePin()`: This function enables a specific GPIO pin.
- `GPIO_DisablePin()`: This function disables a specific GPIO pin.
- `GPIO_SetDirection()`: This function sets the direction (input or output) of a specific GPIO pin.
- `GPIO_GetDirection()`: This function gets the direction (input or output) of a specific GPIO pin.
- `GPIO_SetFunction()`: This function sets the alternate function for a specific GPIO pin (if applicable).
- `GPIO_GetFunction()`: This function gets the alternate function for a specific GPIO pin (if applicable).
- `GPIO_SetStrength()`: This function sets the drive strength of a specific GPIO pin (2mA, 4mA, or 8mA).
- `GPIO_GetStrength()`: This function gets the drive strength of a specific GPIO pin.
- `GPIO_SetPadConfig()`: This function sets the pad configuration for a specific GPIO pin (e.g., pull-up resistor, pull-down resistor).

- `GPIO_GetPadConfig()`: This function gets the pad configuration for a specific GPIO pin.
- `GPIO_SetDigitalAnalogSelect()`: This function selects between digital or analog mode for a specific GPIO pin.
- `GPIO_GetDigitalAnalogSelect()`: This function gets the selected mode (digital or analog) for a specific GPIO pin.
- `GPIO_SetInterruptEvent()`: This function configures the type of interrupt event (e.g., falling edge, rising edge) for a specific GPIO pin.
- `GPIO_GetInterruptEvent()`: This function gets the configured interrupt event type for a specific GPIO pin.
- `GPIO_GetInterruptStatus()`: This function checks the interrupt status (occurred or not) for a specific GPIO pin.
- `GPIO_ClearInterrupt()`: This function clears the interrupt flag for a specific GPIO pin.
- `GPIO_EnableInterrupt()`: This function enables interrupts for a specific GPIO pin.
- `GPIO_DisableInterrupt()`: This function disables interrupts for a specific GPIO pin.
- `GPIO_RegisterInterruptHandler()`: This function registers an interrupt handler function for a specific GPIO pin.

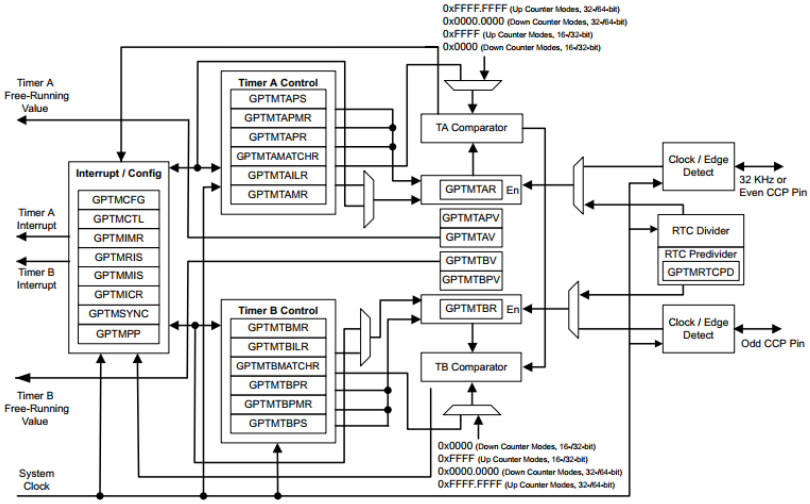
Interrupt Handler Functions:

- `GPIOPortA_Handler()`: This function handles interrupts from all pins of Port A. It checks the interrupt status for each pin, clears the interrupt flag, and calls the corresponding interrupt handler function if registered. Similar interrupt handler functions likely exist for Ports B to F

GPTM Driver

The TM4C123GH6PM microcontroller includes a peripheral called the **General Purpose Timer Module (GPTM)**. This module provides several timers that can be used for various purposes, including:

- **Timing Events:** Generate periodic interrupts at specific intervals. This is useful for creating delays, controlling program flow, or synchronizing with external events.



- **Counting Events:** Count external pulses or events like button presses or encoder rotations.
- **Generating waveforms:** Create simple waveforms like square waves or pulse-width modulation (PWM) signals for controlling motors or LEDs.

Here are some key features of the GPTM in TM4C123GH6PM:

- **Multiple Timers:** The microcontroller has multiple GPTM modules, each containing several timers (typically Timers A and B). This allows for creating independent timing or counting functions.
- **Configurable Modes:** Each timer can be configured in different modes, such as periodic mode, one-shot mode, or capture mode, depending on your needs.
- **Interrupt Capability:** Timers can generate interrupts when they reach a specific count value, allowing your program to react to timing events efficiently.

This driver helps to interact with the peripheral. Here is a breakdown of it:

Non-ISR Functions:

- `GPTM_VoidInit` : This function initializes the GPTM driver based on the configuration provided in a `GPTM_Config_t` structure. It iterates through all timers (TIMER_0 to TIMER_5) and checks their clock status in the configuration.
    - If the timer is enabled in the configuration, it calls functions to:
        - Set the timer clock source (enabled/disabled).
        - Set the timer mode (e.g., periodic, one-shot).

        - Configure other timer settings for each counter (A and B) within the timer module:
            - Enable/disable the counter.
            - Set the timer period (reload value).
            - Set the timer prescaler value.
            - Clear any pending timeout interrupt flags.
            - Enable/disable timeout interrupt based on the configuration.
            - Set the callback function to be called on timeout interrupt (if provided).
- `GPTM_VoidSetTimerClockStatus`: This function enables or disables the clock for a specific timer module based on the provided status.
- `GPTM_VoidSetTimerStatus`: This function enables or disables a specific timer counter (A or B) within a timer module.
- `GPTM_VoidSetTimerMode`: This function sets the operating mode (e.g., periodic, one-shot) for a specific timer module.
- `GPTM_VoidSetTimerPeriodicity`: This function sets the periodicity mode (periodic, one-shot) for a specific timer counter (A or B).
- `GPTM_VoidSetTimerCountingDirection`: This function sets the counting direction (up or down) for a specific timer counter (A or B).

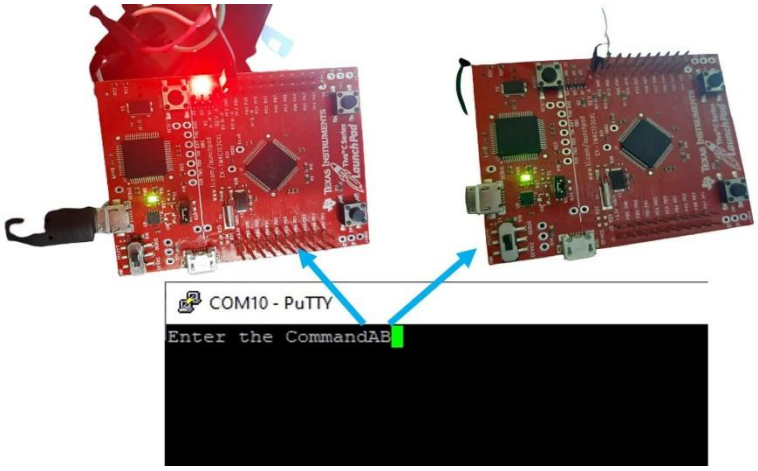
- `GPTM_VoidSetTimerReloadValue`: This function sets the reload value (period) for a specific timer counter (A or B). This value is loaded into the timer counter when it reaches the current value, causing it to restart counting.
- `GPTM_VoidSetTimerPrescalarValue`: This function sets the prescaler value for a specific timer counter (A or B). The prescaler divides the clock source frequency, effectively slowing down the timer.
- `GPTM_VoidClearTimerTimeOutRawInterrupt`: This function clears the timeout interrupt flag for a specific timer counter (A or B), acknowledging the interrupt.
- `GPTM_VoidSetTimerTimeOutRawInterruptStatus`: This function enables or disables the timeout interrupt for a specific timer counter (A or B).
- `GPTM_VoidSetTimeOutRawInterruptCallBack`: This function registers a user-provided callback function to be called when a specific timer counter (A or B) times out and generates an interrupt.
- `GPTM_VoidSetBusyWaitinMS`: This function implements a busy waiting loop using a specific timer counter (A or B) for a specified delay in milliseconds. It configures the timer for one-shot mode with a period based on the desired delay, starts the timer, and then waits in a loop until the timeout interrupt occurs.

Interrupt Handler Functions:

- `Timer0A_Handler` to `Timer5B_Handler`: These are individual interrupt handler functions for each timer counter (A or B) of all timer modules (TIMER_0 to TIMER_5).
    - Each handler clears the timeout interrupt flag for the corresponding timer counter.
    - It then checks if a callback function is registered for that timer counter using the `Raw_Interrupt_CallBacks` array.
    - If a callback function is registered, it calls the function.

UART Driver

The TM4C123GH6PM microcontroller boasts a built-in peripheral called the **Universal Asynchronous Receiver/ Transmitter (UART)**. This versatile peripheral facilitates serial communication between the microcontroller and other devices. It allows you to:



- **Transmit Data:** Send digital data byte by byte to external devices like computers, displays, or other microcontrollers.
- **Receive Data:** Receive digital data byte by byte from external devices.

Here are some key aspects of the UART peripheral in TM4C123GH6PM:

- **Multiple UARTs:** The microcontroller actually has **eight** UART modules (UART0 to UART7). This provides flexibility for connecting to multiple devices simultaneously.

- **Configurable Baud Rate:** You can configure the speed (baud rate) at which data is transmitted and received. This needs to match the baud rate of the connected device for proper communication.
- **Asynchronous Communication:** UART uses asynchronous communication, meaning the data bits are sent along with start and stop bits to define the beginning and end of each byte.
- **Error Detection:** The UART can detect some errors during transmission, like parity errors, to ensure data integrity.
- **FIFO Buffers:** The UART has built-in buffers (FIFO - First In, First Out) for temporary storage of data during transmission and reception. This helps to improve communication efficiency.

This driver helps to interact with the peripheral. Here is a breakdown of it:

Global Variables:

- `Reciever_Interrupt_CallBacks`: An array of function pointers, one for each UART unit. Used to store user-provided callback functions for receiving data interrupts.

`UART_VoidInit` Function:

- Initializes the UART driver based on a configuration provided in a `UART_Config_t` structure.
- Iterates through all UART units (UART_0 to UART_7).
- For each enabled UART unit in the configuration, it calls functions to:
    - Set the UART clock status (enabled/disabled).
    - Set the UART enable status (enabled/disabled).
    - Configure baud rate, parity, word length, FIFO, stop bits, and interrupt settings based on the configuration.
    - Clear any pending receive and transmit interrupt flags.

Other Configuration Functions:

- `UART_VoidSetUARTClockStatus`: Enables or disables the clock for a specific UART unit.
- Functions named `UART_VoidSet...`: These functions configure various aspects of a UART unit based on provided parameters (e.g., enable/disable receiver, set parity, set word length, etc.).

Interrupt Handler Functions:

- `UART0_Handler` to `UART7_Handler`: Individual interrupt handler functions for each UART unit's receive interrupt.
- Each handler:

    - Clears the receive interrupt flag for its corresponding UART unit.
    - Checks if a callback function is registered for that unit using the `Reciever_Interrupt_CallBacks` array.
    - If a callback function is registered, it calls the function.

Data Sending/Receiving Functions:

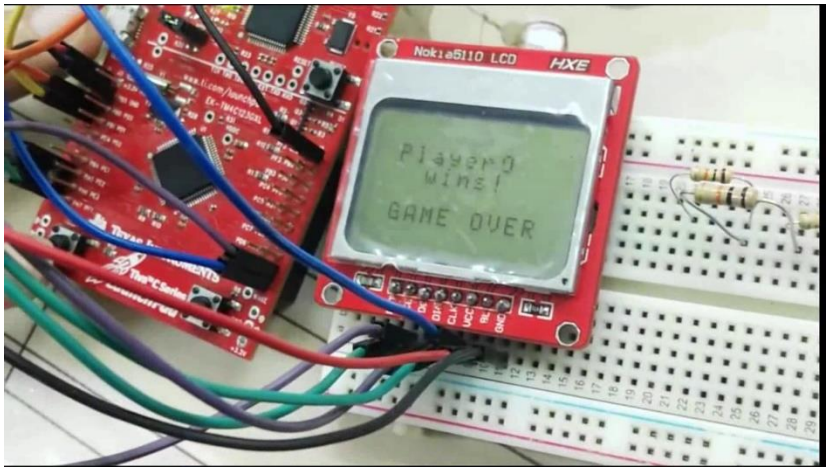
- `UART_UnsignedCharGetDataBlocking/NonBlocking`: These functions read received data from a UART unit. The blocking version waits until data is available, while the non-blocking version checks if data is available and returns immediately.
- `UART_VoidSendDataBlocking/NonBlocking`: These functions send data to a UART unit. The blocking version waits until the transmitter is ready, while the non-blocking version checks if the transmitter is ready and returns immediately.
- `UART_VoidSendStringBlocking`: Sends a null-terminated string of characters through a UART unit.

Interrupt Callbacks:

- `UART_VoidSetUARTInterruptCallBack`: Allows registering a user-provided callback function to be called when a receive interrupt occurs on a specific UART unit.

SSI Driver

The TM4C123GH6PM microcontroller has a peripheral called the **SSI (Synchronous Serial Interface)**, though it's functionally equivalent to a **Serial Peripheral Interface (SPI)**.



This peripheral enables synchronous serial communication between the microcontroller and other devices. Here's a breakdown of what the SSI can do:

- **Full-Duplex Communication:** It allows for simultaneous transmission and reception of data over a single data line, unlike UART which is half-duplex (sends or receives at a time).
- **Master-Slave Configuration:** The SSI can be configured as either a master device that initiates communication or a slave device that responds to a master's requests. The TM4C123GH6PM typically operates as the master in most applications.
- **Synchronized Data Transfer:** A clock signal synchronizes the data transfer between the master and slave devices, ensuring both sides are in agreement on when to send and receive each bit.

- **Multiple Data Sizes:** You can configure the SSI to transmit and receive data in various sizes, typically ranging from 4 bits to 16 bits per word.

Here are some key things about SSI in TM4C123GH6PM:

- **Shared Functionality:** The SSI peripheral actually supports three different protocols: SPI, TI Synchronous Serial Interface (SSI), and Microwire. By configuring specific settings, you can enable the desired protocol (SPI mode is most commonly used).
- **Separate Clock Pin:** Unlike UART, the SSI uses a dedicated clock pin to synchronize data transfer. You need to configure a GPIO pin as the SSI clock for proper communication.
- **Chip Select (CS):** The communication between the master and slave is typically controlled by a separate chip select (CS) line. This line is used by the master to indicate when it wants to communicate with a specific slave device.

This driver helps to interact with the peripheral. Here is a breakdown of it:

`MSSI_vInit` Function:

- Initializes the SSI peripheral.

- Enables the SSI clock by setting the corresponding bit in the `SYSCTL_RCGC1_R` register.
- Adds a small delay to allow clock stabilization (common practice).
- Disables the SSI module by clearing the `SSE` bit in the `SSI0_CR1_R` register.
- Sets the SSI mode (Master/Slave) based on the configuration in `SSI_cfg.h`.
- Selects the clock source (e.g., system clock, PIOSC) based on the configuration in `SSI_cfg.h`.
- Sets the SSI prescaler divisor based on the configuration in `SSI_cfg.h`.
- Sets the serial clock rate by calculating the appropriate value based on system clock and prescaler and writing it to the `SSI_CR0_R` register.
- Sets the clock polarity (active high/low) based on the configuration in `SSI_cfg.h`.
- Sets the clock phase (data sampled on first/second edge) based on the configuration in `SSI_cfg.h`.
- Sets the frame format (Freerun/Framed) based on the configuration in `SSI_cfg.h`.
- Sets the data size (8-bit/16-bit) based on the configuration in `SSI_cfg.h`.

- Finally, enables the SSI module by setting the `SSE` bit in the `SSI0_CR1_R` register.

Data Transmission Functions:

- `MSSI_vTransmit_single`: transmits a single 16-bit data word.
    - It waits for the SSI to be not busy (`BSY` bit cleared in `SSI0_SR_R`).
    - Writes the data to the `SSI0_DR_R` register.
    - Waits again for the SSI to be not busy to ensure transmission is complete.
- `MSSI_vTransmit_continuous`: transmits a single 16-bit data word in continuous mode.
    - It waits for the transmit FIFO to have space available (`TNF` bit set in `SSI0_SR_R`).
    - Writes the data to the `SSI0_DR_R` register.

Data Reception Functions:

- `MSSI_u16Recieve_single`: receives a single 16-bit data word.
    - It waits for the SSI to be not busy (`BSY` bit cleared in `SSI0_SR_R`).

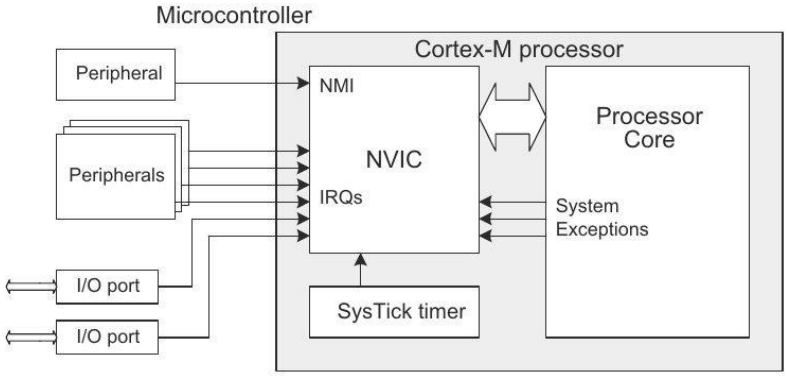
    - Reads the received data from the `SSI0_DR_R` register and returns it.
- `MSSI_u16Recieve_continuous`: receives a single 16-bit data word in continuous mode.
    - It waits for the receive FIFO to have data available (`RNE` bit set in `SSI0_SR_R`).
    - Reads the received data from the `SSI0_DR_R` register and returns it.

Status Check Functions:

- `MSSI_u8Check_Busy`: Checks if the SSI is busy by reading the `BSY` bit in `SSI0_SR_R`. Returns 1 if busy, 0 if not.
- `MSSI_u8Check_NotEmpty`: Checks if the transmit FIFO is not full by reading the `TNF` bit in `SSI0_SR_R`. Returns 1 if not full, 0 if full.

NVIC Driver

The TM4C123GH6PM microcontroller doesn't actually have a peripheral called NVIC. **NVIC stands for Nested Vectored Interrupt Controller**. It's not a separate physical component but rather a **built-in unit within the ARM Cortex-M4 core** that the TM4C123GH6PM uses.



The NVIC acts as a central hub for managing all interrupts within the microcontroller. Here's a breakdown of its key functions:

- **Interrupt Prioritization:** When multiple interrupts occur simultaneously, the NVIC determines which interrupt has the highest priority and services that one first. This ensures critical tasks are handled promptly.
- **Interrupt Vectoring:** The NVIC maintains a table that maps each interrupt source (like a timer or GPIO) to a specific function (Interrupt Service Routine - ISR) in your program. When an interrupt occurs, the NVIC fetches the corresponding ISR address from the table and executes it.

- **Interrupt Masking and Enabling:** The NVIC allows you to enable or disable specific interrupts. This helps to control which interrupts are allowed to disrupt the main program flow.

Here are some additional points to consider about the NVIC:

- **Configurable Priority Levels:** You can configure the priority level for most interrupt sources using NVIC registers. This allows you to fine-tune how interrupts are handled based on their importance.
- **Nested Interrupts:** The NVIC supports nested interrupts. This means a higher-priority interrupt can interrupt a lower-priority interrupt that is already being serviced. Nesting allows for handling critical events even during ongoing interrupt routines.

This driver helps to interact with the peripheral. Here is a breakdown of it:

`MNVIC_vEnableINTPeripheral:`

- Enables a specific peripheral interrupt.
- Calculates the register index and bit position within the register based on the provided `A_u8IntId` (interrupt ID).
- Sets the corresponding bit in the NVIC Interrupt Enable Register (`NVIC->EN[register_index]`) using bit manipulation macros.

MNVIC_vDisableINTPeripheral:

- Disables a specific peripheral interrupt.
- Similar to `MNVIC_vEnableINTPeripheral` but uses the NVIC Interrupt Disable Register (`NVIC->DIS[register_index]`) to clear the corresponding bit.

MNVIC_vSetPending (for Testing):

- Sets the pending state of an interrupt (for testing purposes).
- Calculates the register index and bit position based on `A_u8IntId`.
- Sets the corresponding bit in the NVIC Interrupt Pending Set Register (`NVIC->PEND[register_index]`) to simulate a pending interrupt.

MNVIC_vClearPending:

- Clears the pending state of an interrupt.
- Similar to `MNVIC_vSetPending` but uses the NVIC Interrupt Pending Unset Register (`NVIC->UNPEND[register_index]`) to clear the corresponding bit.

MNVIC_u8GetActive:

- Checks if a specific interrupt is currently active (in the process of being serviced).
- Calculates the register index and bit position based on `A_u8IntId`.
- Reads the corresponding bit from the NVIC Interrupt Active Register (`NVIC->ACTIVE[register_index]`) and returns its value (1 if active, 0 if not).

MNVIC_vSetpriority:

- Sets the priority of an interrupt.
- Calculates the register index and bit position within the register based on `A_u8IntId`.
- Clears the previous priority bits for the interrupt in the NVIC Interrupt Priority Register (`NVIC->PRI[register_index]`).
- Sets the new priority value (`A_u8IntPri`) in the correct bit position within the register

HAL Layer

Nokia 5110 Driver

The Nokia 5110 LCD is a small, monochrome graphic LCD screen originally used in the Nokia 5110 cell phone. It's become a popular choice for hobbyists and electronics projects due to its:

- **Simple design:** Easy to use and solder for DIY projects
- **Readability:** Small (around 1.5 inches diagonal) but with good clarity
- **Monchrome display:** Shows text and basic graphics in black and white
- **Availability:** Still relatively inexpensive and easy to find from electronics stores

Here are some key features of the Nokia 5110 LCD:

- Resolution: 84 x 48 pixels
- Backlight: Typically white
- Power supply voltage: 2.7V-3.3V

This driver helps to interact with the screen. Here is a breakdown of it:

Global Variable:

- `G_u8N5110Pixels`: A 2D array of u8 to store the pixel data for the Nokia 5110 display (each element represents a byte for 8 pixels).

Command/Data Write Functions:

- `HN5110_vWriteCommand`: Sends a command byte to the Nokia 5110 display.
    - Waits for the SSI to be not busy (`MSSI_u8Check_Busy`).
    - Sets the DC (Data/Command) pin to command mode using a bit manipulation macro or GPIO function.
    - Transmits the command byte using the `MSSI_vTransmit_single` function.
- `HN5110_vWriteData`: Sends a data byte to the Nokia 5110 display.
    - Waits for the SSI transmit FIFO to have space available (`MSSI_u8Check_NotEmpty`).
    - Sets the DC pin to data mode.

    - Transmits the data byte using the `MSSI_vTransmit_continuous` function (assuming continuous data transfer mode).

Display Initialization:

- `HN5110_vInit`: Initializes the Nokia 5110 display.
    - Initializes the SSI peripheral using `MSSI_vInit`.
    - Resets the display by pulsing the RST pin using GPIO functions or bit manipulation (commented out).
    - Sends a series of command bytes to configure the display (bias, temperature, etc.) using `HN5110_vWriteCommand`.

Character/String Printing Functions:

- `HN5110_vPrintChar`: Prints a single character to the display.
    - Sends a blank column data byte for left padding.
    - Iterates through the character's pixel data in the `ASCII` array and sends each byte using `HN5110_vWriteData`.
    - Sends another blank column data byte for right padding.
- `HN5110_vPrintString`: Prints a null-terminated string to the display.
    - Calls `HN5110_vPrintChar` for each character in the string until the null terminator is reached.

Number Printing Function:

- `HN5110_vPrintNumber`: Prints a signed 32-bit integer to the display.
    - Handles negative numbers by adding a minus sign and negating the value.
    - Converts the number to a string of digits in reverse order.
    - Calls `HN5110_vPrintChar` for each digit character.

Cursor Control/Display Functions:

- `HN5110_vSetCursor`: Sets the cursor position on the display within valid boundaries.
    - Sends commands to define the X and Y coordinates based on the character width (7 pixels).
- `HN5110_vClear`: Clears the entire display by sending blank data bytes for all pixels.
    - Also calls `HN5110_vHome` to set the cursor to the top-left corner.
- `HN5110_vDrawFullImage`: Draws a complete image from a provided data array to the display.
    - Sets the cursor to the top-left corner (`HN5110_vHome`).
    - Iterates through the image data and sends