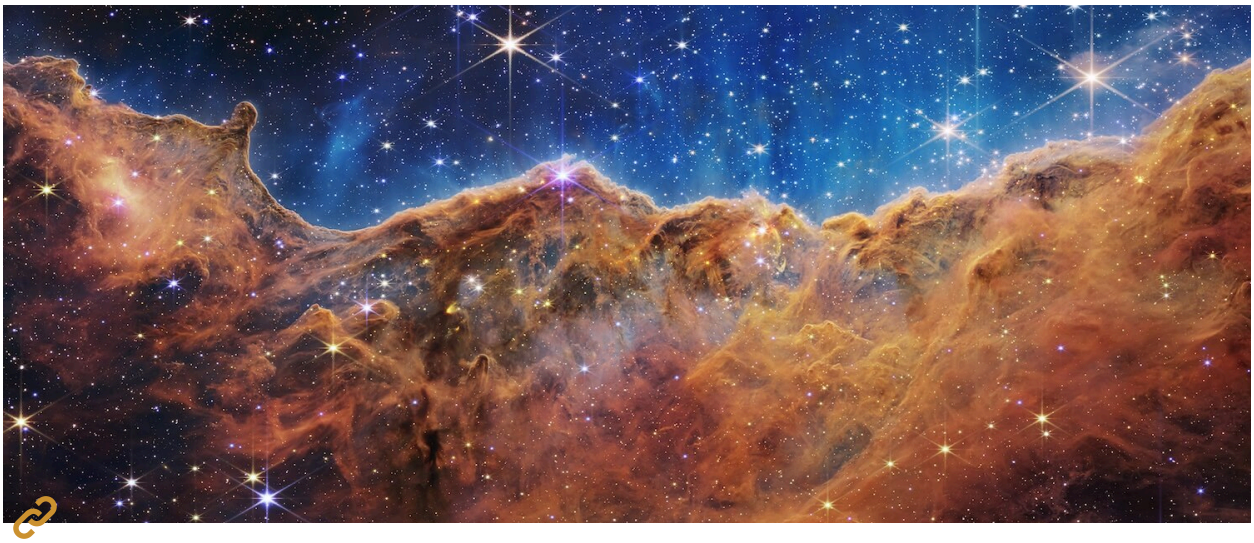


GPS Project Documentation

Welcome to the **GPS Project Documentation**! This page serves as a comprehensive guide to our project, which integrates a **GPS module** with the **TIVA-C** microcontroller for real-time location tracking and navigation.

‼️For Better Documentation Readability:

CLICK ME

🌐 GitHub Repo:

CLICK ME

📽️Project Video:

CLICK ME

👧 Team Members:

Name	ID	Contributions
Ziad Alaa Anis Mohamed Kamal Kassem	2200278	Team Leader, GPIO-driver, SYSTICK-driver, GPS-driver, Milestone-1&2.
Ahmad Ayman Ibrahim Ezz El Deen	2200563	UART-driver, Hardware Integration, Milestone-1&2.
Mustafa Tamer Mansour EL-Sherif	2200556	EEPROM-driver, Interrupt-driver, Milestone-3&4, Phase1-review.
Hussein Bahaa Hussein Abdelhamid	2200211	UART-driver, Repo documentation.
Randa Ayman Samir Osman Abdou	2200127	CLCD driver, LED driver, Switches driver and Documentation.
Jessica Samuel Nabih Aziz	2200923	Bitmath, STDtypes, helped in LED, Switches, Documentation.

🔵Modular Design

This project is structured using a **modular software architecture which is COTS(components on the shelf)**, organized into four key layers:

1. **MCAL (Microcontroller Abstraction Layer)**

Handles low-level hardware access. Modules include:

- **SYSTICK**
- **GPIO**
- **UART**
- **GPIO Interrupts**
- **EEPROM**

2. **HAL (Hardware Abstraction Layer)**

Builds on MCAL to provide high-level interfaces for:

- **LED**
- **Switch**
- **Character LCD (CLCD)**
- **GPS**

3. **LIB (Utility Layer)**

Provides general-purpose utilities:

- **Standard Types (** `STD_TYPES.h` **)**
- **Bit Manipulation Macros (** `BIT_MATH.h` **)**

4. **APP (Application Layer)**

Contains the application logic and helper functions:

- Organized into `APP.h` and `APP.c`
- Interacts with HAL and LIB to implement high-level behavior

---

Each peripheral or utility is implemented with a consistent structure:

- `private.h`
- `interface.h`
- `config.h`
- `program.c`

- `main.c` Initializes the system and controls the main program loop.

⚙️ Peripheral Drivers

- `GPIO` Configures and controls general-purpose input/output (LEDs, switches).
- `UART` Manages serial communication with both the GPS module and the PC.
- `Systick` Implements timing delays and periodic interrupts.
- `CLCD` Interfaces with a character LCD in 8-bit or 4-bit mode for data display.
- `EEPROM` Provides read/write access to non-volatile memory (used for storing coordinates).
- `Switches` Handles debounced input from pushbuttons.
- `LED` Controls indicator LEDs used for system feedback.

📡 Application-Specific Modules

- `GPS` Parses NMEA sentences (e.g., `$GPRMC`) to extract latitude and longitude data.
- `BitMath` Contains reusable macros for bit manipulation.
- `StdTypes` Defines standard data types (`u8`, `u16`, `s32`, etc.) to improve portability and clarity.

🎁 List of Bonus Features (9 bonuses)

✅**CLEAN CODE in Interrupt Driver got it** by *Eng. Ayman*

📺 **CLCD Driver**

🎛️**Potentiometer**– Adjustable brightness for CLCD

💾**EEPROM Driver**– Non-volatile memory integration

👁️Interrupt Driver - built the full generic driver

✨ Two Extra Features : Time Visiting Report based on EEPROM & Navigator to any chosen Place

---

🔊**Buzzer**– Audio feedback support implemented in APP layer

🧰Made All Drivers from scratch each in 4 files style driver , then Constructed the COTS

📦**Packaging**– Organized, clean project structure & Using portable energy source 🔋

📌 Project Overview

This project focuses on capturing real-time **GPS data** using the **Tiva-C** microcontroller. The system processes the received data and extracts key information such as:

✅ **Latitude & Longitude**

✅ **Real-time Position Tracking**

🔧 Key Features

- **Real-time GPS data acquisition** 📡
- **UART communication** between Tiva-C and the GPS module 🔁
- **Data parsing & processing** 📝
- **Display of coordinates & navigation info** 🗺️

📌Main Objectives:

- UART communication between Tiva-C and the GPS module 🔁
- Data parsing & processing 📝
- Display of coordinates & navigation info 🗺️

✅MILESTONE 1:

CODE: CLICK ME

VIDEO, “Making sure that the code flashes from Keil”: CLICK ME

“Tri-LED MultiMode Controller”

🎯 **Objectives:**

This code logic uses drivers that has GPIO functions implemented in them each serving different functionalities:

- Dependencies of SW
- Dependencies of LED
- Dependencies of CLCD

Hardware Used:

- 3 LEDs (LED1, LED2, LED3)
- 2 Switches
- SW1: Change Mode
- SW2: Perform Action (depending on current mode)
- CLCD: Shows current mode and info
- Tiva C GPIOs

Modes Overview (Cycle with SW1):

Mode 0 – Manual Control Mode:

- Press SW2 to toggle LEDs one by one (LED1 → LED2 → LED3 → back to LED1...)
- Each press toggles that LED ON/OFF
- CLCD shows: "Manual Mode", "LED X: ON/OFF"

Mode 1 – Auto Toggle Mode:

- LEDs toggle automatically in sequence:
- LED1 ON → LED2 ON → LED3 ON → OFF → repeat
- SW2: Pause/Resume the sequence
- CLCD shows: "Auto Toggle Mode", "Status: Running/Paused"

Mode 2 – Blink Speed Control:

- All LEDs blink together
- SW2 increases blink speed (3 speeds: Slow → Medium → Fast → loop)
- CLCD shows: "Blink Mode", "Speed: Medium"

Mode 3 – LED Binary Counter:

- 3 LEDs act as a 3-bit binary counter
- Count = 0 → 7 (000 to 111)
- LED1 = LSB, LED3 = MSB
- Count auto-increments every 1s
- SW2 resets the count to 0
- CLCD shows: "Counter Mode"

✅ MILESTONE 2:

CLICK ME

“UART Control of LEDs”

🎯 **Objective:**

Use UART to control three LEDs (RED, GREEN, YELLOW) through terminal commands. No physical switches are involved.

📡 **UART Setup:**

Configure UART4 for 9600 baud, 8N1 format

```
UART_CONFIG_t uart0configuration;

uart0configuration.Module = UART4;
uart0configuration.DataBits = DataBits8;
uart0configuration.StopBits = OneStopBit;
uart0configuration.ParOnOff = ParityDisable;
uart0configuration.Parity = OddParity; // Doesn't matter (disabled)
uart0configuration.BaudRate = 9600;

UART_StdErrorInit(&uart0configuration);
```

🔌 **GPIO Setup:**

Initialize GPIO ports and configure LEDs as outputs

```
GPIO_StdErrorInit(PortA);
GPIO_StdErrorInit(PortB);

LED_voidInitExternalLed(PortA, 7); // RED
LED_voidInitExternalLed(PortA, 5); // GREEN
LED_voidInitExternalLed(PortA, 6); // YELLOW
```

📺 **LCD Display (Optional UI feedback):**

```
CLCD_StdErrorDataPinsInit();
CLCD_VoidClearDisplay();
CLCD_voidDataSetCursor(LINE_1, 0);
CLCD_voidDataSendString("Milestone 2");
CLCD_voidDataSetCursor(LINE_2, 0);
```

⌨️ **UART Command Handling:**

Receive UART commands and control LEDs accordingly

```
u8 command;

if (UART_StdErrorReceiveByte(&uart0configuration, &command) == OK) {
   switch (command) {
      case 'r':
         LED_voidTurnOnLed(PortA, 7);
         Systick_StdErrorDelayIn_ms(500);
         break;

      case 'y':
         LED_voidTurnOnLed(PortA, 5);
         Systick_StdErrorDelayIn_ms(500);
         break;

      case 'g':
         LED_voidTurnOnLed(PortA, 6);
         Systick_StdErrorDelayIn_ms(500);
         break;

      case 'o':
         TurnOffAllLeds();
         Systick_StdErrorDelayIn_ms(1000);
         break;
   }

   TurnOffAllLeds(); // Ensures LED is not left on
   CLCD_StdErrorSendData(command); // Display pressed key
}
```

🚦 **LED OFF Utility Function:**

Helper function to switch off all LEDs

```
void TurnOffAllLeds(void){
   LED_voidTurnOffLed(PortA, 7);
   LED_voidTurnOffLed(PortA, 5);
   LED_voidTurnOffLed(PortA, 6);
}
```

✅MILESTONE 3:

CLICK ME

“ EEPROM Read/Write Verification “

🎯**Objective:**

Verify EEPROM write and read functionality by writing a value and lighting up the corresponding LED based on successful or failed data verification.

📦 **Included Libraries (Only Relevant Ones):**

```
#include "GPIO_interface.h"
#include "SYSTICK_interface.h"
#include "EEPROM_interface.h"
```

- **GPIO_interface.h** – Handles pin initialization and control on Port F.
- **SYSTICK_interface.h** – Provides delay functions for timing.
- **EEPROM_interface.h** – Provides APIs for EEPROM read/write operations.

🧠 `main()` **Function Logic:**

```
u32 data_rx;
```

- Declares a variable `data_rx` to store the data read from EEPROM.

```
GPIO_StdErrorInit(PortF);
```

- Initializes **Port F** to enable GPIO functionality.

```
GPIO_StdErrorSetPinDir(PortF, PIN1, PIN_OUTPUT);
GPIO_StdErrorSetPinDir(PortF, PIN2, PIN_OUTPUT);
```

- Sets **PIN1** and **PIN2** of **Port F** as output to control LEDs.

```
EEPROM_STD_ERROR_Init();
```

- Initializes EEPROM module on the microcontroller.

```
EEPROM_STD_ERROR_WriteData(1,5);
```

- Writes the value `5` to EEPROM address `1`.

```
Systick_StdErrorDelayIn_ms(10);
```

- Waits 10 milliseconds to ensure write operation completes before reading.

```
data_rx = EEPROM_u32ReadData(1);
```

- Reads the value from EEPROM address `1` and stores it in `data_rx`.

```
if(data_rx == 5){
   GPIO_StdErrorWritePin(PortF, PIN1, PIN_HIGH);
}
else{
   GPIO_StdErrorWritePin(PortF, PIN2, PIN_HIGH);
}
```

- If the value read is correct (`5`), **PIN1 (LED1)** is turned ON (indicating success).

- If not, **PIN2 (LED2)** is turned ON (indicating failure).

```
while(1);
```

- Keeps the program running infinitely.

✅MILESTONE 4:

CLICK ME

“External Interrupt and GPIO Toggle”

🎯Objective:

Set up an external interrupt on **PIN4** of **Port F** to trigger a callback function, while toggling another LED in the main loop.

📦 Included Libraries (Only Relevant Ones):

```
#include "GPIO_interface.h"
#include "SYSTICK_interface.h"
#include "INT_interface.h"
```

- **GPIO_interface.h** – For GPIO initialization, direction, and writing.
- **SYSTICK_interface.h** – For millisecond-level delay functions.
- **INT_interface.h** – For configuring interrupts on GPIO pins.

🧠 Interrupt Callback Function:

```
void func (void) {
    GPIO_StdErrorWritePin(PortF, PIN2, PIN_HIGH);
    Systick_StdErrorDelayIn_ms(2000);
}
```

- Triggered when an interrupt occurs on **PortF PIN4**.
- Turns on **PIN2 (LED)** for 2 seconds using Systick delay.

🔁 Global Flag Declaration:

```
u8 flag;
```

- Used to toggle LED state in the main loop.

🔌 GPIO Initialization:

```
GPIO_StdErrorInit(PortF);
```

- Initializes GPIO settings for **Port F**.

```
GPIO_StdErrorSetPinDir(PortF, PIN4, PIN_INPUT);
GPIO_StdErrorSetPinDir(PortF, PIN1, PIN_OUTPUT);
GPIO_StdErrorSetPinDir(PortF, PIN2, PIN_OUTPUT);
```

- **PIN4** set as input (interrupt trigger pin).
- **PIN1** and **PIN2** set as output (LEDs).

🔌 Application Initial State:

```
flag = 0;
GPIO_StdErrorWritePin(PortF, PIN1, PIN_LOW);
GPIO_StdErrorWritePin(PortF, PIN2, PIN_LOW);
```

- Resets flag to 0.
- Turns off both LEDs at startup.

⚡ Interrupt Setup:

```
INT_voidPORTF_Enable(PIN4);
```

- Enables external interrupt on **PIN4**.

```
INT_voidFunc(PortF, PIN4, &func);
```

- Associates the `func()` callback with **PIN4** interrupt.

🔁 Main Loop Logic:

```
while(1){
   GPIO_StdErrorWritePin(PortF, PIN2, PIN_LOW);
```

- Turns off **PIN2** at the beginning of each loop cycle.

```
if(flag == 0){
   GPIO_StdErrorWritePin(PortF, PIN1, PIN_HIGH);
   flag = 1;
}
else{
   GPIO_StdErrorWritePin(PortF, PIN1, PIN_LOW);
   flag = 0;
}
```

- Toggles **PIN1 (LED)** on and off using the `flag`.

```
Systick_StdErrorDelayIn_ms(750);
```

- Adds a 750ms delay between each toggle.

🧩 main.c

CLICK ME

“Project Overview and Execution Logic”

This project demonstrates the integration of **EEPROM memory**, **external interrupts**, **GPIO control**, and **timing using SysTick** on the **Tiva C (TM4C123GH6PM)** microcontroller. The system performs the following key functions:

- Writes and reads data from EEPROM and verifies the result with LED indications.
- Uses an external interrupt on **Port F, PIN4** to trigger a custom callback routine.
- Utilizes SysTick for creating non-blocking delays.
- Implements a toggling LED pattern for visual feedback of system activity.
- Buzzer usage

This `main.c` file serves as the central hub for initializing modules, setting up interrupts, and running application-level logic.

🔧 Included Modules

```
#include "GPIO_interface.h"
#include "SYSTICK_interface.h"
#include "EEPROM_interface.h"
#include "INT_interface.h"
```

- **GPIO_interface.h** – Configures GPIO directions and controls digital outputs.
- **SYSTICK_interface.h** – Provides millisecond delay functions.
- **EEPROM_interface.h** – Enables persistent memory read/write operations.
- **INT_interface.h** – Handles external interrupt configuration and callbacks.

⚡ Interrupt Callback Function

```
void func (void) {
   GPIO_StdErrorWritePin(PortF, PIN2, PIN_HIGH);
   Systick_StdErrorDelayIn_ms(2000);
}
```

- Triggered by an interrupt on **Port F, PIN4**.
- Lights up **PIN2 (LED)** for 2 seconds to indicate interrupt activity.

📥 EEPROM Read/Write Verification

```
u32 data_rx;
EEPROM_STD_ERROR_Init();
EEPROM_STD_ERROR_WriteData(1,5);
Systick_StdErrorDelayIn_ms(10);
data_rx = EEPROM_u32ReadData(1);
if(data_rx == 5){
   GPIO_StdErrorWritePin(PortF, PIN1, PIN_HIGH);
} else {
   GPIO_StdErrorWritePin(PortF, PIN2, PIN_HIGH);
}
```

- Initializes EEPROM and writes the value `5` to address `1`.
- Reads the value back and lights **PIN1** if it matches, else **PIN2**.

🔌 GPIO Initialization

```
GPIO_StdErrorInit(PortF);
GPIO_StdErrorSetPinDir(PortF, PIN1, PIN_OUTPUT);
GPIO_StdErrorSetPinDir(PortF, PIN2, PIN_OUTPUT);
GPIO_StdErrorSetPinDir(PortF, PIN4, PIN_INPUT);
```

- Prepares **PIN1** and **PIN2** for LED output.
- Sets **PIN4** as input for external interrupt trigger.

⚙️ Interrupt Setup

```
INT_voidPORTF_Enable(PIN4);
INT_voidFunc(PortF, PIN4, &func);
```

- Enables GPIO interrupt on **PIN4** and links it to `func()`.

🔁 Main Loop

```
u8 flag = 0;
while(1){
   GPIO_StdErrorWritePin(PortF, PIN2, PIN_LOW);

   if(flag == 0){
      GPIO_StdErrorWritePin(PortF, PIN1, PIN_HIGH);
      flag = 1;
   } else {
      GPIO_StdErrorWritePin(PortF, PIN1, PIN_LOW);
      flag = 0;
   }

   Systick_StdErrorDelayIn_ms(750);
}
```

- Continuously toggles **PIN1** every 750ms.
- Ensures **PIN2** is off unless triggered by an interrupt.
- Uses `flag` to alternate LED state, giving a heartbeat-style activity signal.

🔔BUZZER

```
BuzzerTrigger();
```

- Alerts the user that the nearest place has been identified.
- Provides audio feedback after location processing is complete.
- Adds user-friendly interactivity (sound cue before displaying location/distance on LCD).
- Helps in visually-impaired scenarios or noisy environments where LCD might not be noticed quickly.

💻COTS:

1. MCAL:

⏱️ SYSTICK DRIVER

CODE: CLICK ME

📌 Systick_DelayInTicks(u32 ticks) :

Creates a delay for a specific number of system ticks.

ex:

```
Systick_DelayInTicks(1000000);
 // Creates a delay of 1 million ticks.
```

📌 Systick_DelayIn_ms(u32 ms) :

Delays the system for a specific number of milliseconds.

ex:

```
Systick_DelayIn_ms(500);
// Creates a delay of 500 milliseconds.
```

📌 Systick_DelayIn_us(u32 us):

Delays the system for a specific number of microseconds.
⚠️ Must be ≤ 0x00FFFFFF (2^24 - 1) ticks

ex:

```
Systick_DelayIn_us(100);
// Creates a delay of 100 microseconds.
```

📌 Systick_Reset():

Disables the Systick timer, clears its current and reload values.

ex:

```
Systick_Reset();
// Timer is disabled and reset.
```

📌 Systick_GetRemainingCounts():

Returns the remaining number of ticks before the counter reaches 0.

ex:

```
u32 remaining = Systick_GetRemainingCounts();
 // Useful for monitoring or profiling.
```

📌(SYSTICK_config.h)

```
#define STK_CTRL_CLK_SOURCE CTRL_CLK_SOURCE_INTERNAL
#define STK_CTRL_TICKINT    CTRL_TICKINT_DISABLE
```

- `CTRL_CLK_SOURCE_INTERNAL`: Use AHB (CPU clock)
- `CTRL_CLK_SOURCE_DIV4`: Use AHB / 4
- `CTRL_TICKINT_ENABLE`: Enable interrupt when counter reaches 0
- `CTRL_TICKINT_DISABLE`: Disable the interrupt

⚙️ GPIO:

CODE: CLICK ME

📌 Memory Mapping

The driver accesses GPIO hardware through memory-mapped registers based at the following addresses:

- PORTA: 0x40004000
- PORTB: 0x40005000
- PORTC: 0x40006000
- PORTD: 0x40007000
- PORTE: 0x40024000
- PORTF: 0x40025000

📌 Register Structure

The driver defines a comprehensive `GPIO_PORT_t` structure that maps all GPIO registers:

- Data registers
- Direction control
- Alternate function selection
- Pull-up/Pull-down configuration
- Slew rate control
- Digital enable
- Lock/commit mechanisms
- Analog mode selection
- Port control

📌 API Reference:

- **Initialization:**

```
STD_ERROR GPIO_StdErrorInit(u32 Copy_u32Port);
```

Description: Initializes a GPIO port, enabling clock, unlocking configuration, setting commit register, and enabling digital functionality.

Parameters: Port identifier (PortA to PortF)

Return: OK on success, NOK on failure

- **Pin Direction Control:**

```
STD_ERROR GPIO_StdErrorSetPinDir(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8PinDir);
```

Description: Sets a specific pin as input or output

Parameters: Port identifier, Pin number (0-7), Direction (PIN_INPUT/PIN_OUTPUT)

Return: OK on success, NOK on failure

- **Port Direction Control:**

```
STD_ERROR GPIO_StdErrorSetPortDir(u32 Copy_u32Port, u8 Copy_u8DirValue);
```

Description: Sets the direction for all pins in a port

Parameters: Port identifier, Direction value (PORT_INPUT/PORT_OUTPUT or custom bit pattern)

Return: OK on success, NOK on failure

- **Pin Write Operation:**

```
STD_ERROR GPIO_StdErrorWritePin(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8PinValue);
```

Description: Sets a specific pin to high or low

Parameters: Port identifier, Pin number (0-7), Value (PIN_HIGH/PIN_LOW)

Return: OK on success, NOK on failure

- **Pin Read Operation:**

```
STD_ERROR GPIO_StdErrorReadPin(u32 Copy_u32Port, u8 Copy_u8PinId, u8* Copy_u8PinReturnValue);
```

Description: Reads the current state of a specific pin

Parameters: Port identifier, Pin number (0-7), Pointer to store the read value

Return: OK on success, NOK on failure

- **Port Write Operation:**

```
STD_ERROR GPIO_StdErrorWritePort(u32 Copy_u32Port, u8 Copy_u8Copy_u8PortValue);
```

Description: Sets all pins in a port to a specific pattern

Parameters: Port identifier, Value pattern to write

Return: OK on success, NOK on failure

- **Port Read Operation:**

```
STD_ERROR GPIO_StdErrorReadPort(u32 Copy_u32Port, u8* Copy_ptru8ReturnValue)
```

Description: Reads the current state of all pins in a port

Parameters: Port identifier, Pointer to store the read value

Return: OK on success, NOK on failure

- **Alternate Function Selection:**

```
STD_ERROR GPIO_StdErrorSetPinAlternateFunction(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8PinValue, u8 Copy_u8AltFunc);
```

Description: Configures a pin to use an alternate function

Parameters: Port identifier, Pin number (0-7), Enable/disable value, Alternate function code

Return: OK on success, NOK on failure

- **Analog Mode Selection:**

```
STD_ERROR GPIO_StdErrorPinAnalogModeSelect(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8PinValue);
```

Description: Enables or disables analog functionality for a pin

Parameters: Port identifier, Pin number (0-7), Enable/disable value

Return: OK on success, NOK on failure

- **Pad Configuration**

```
STD_ERROR GPIO_StdErrorSetPinPadConfig(u32 Copy_u32Port, u8 Copy_u8PinId, GPIO_PadConfig_t* Copy_PtrGPIO_PadConfig_tConfig);
```

Description: Configures electrical characteristics for a pin

Parameters: Port identifier, Pin number (0-7), Pointer to configuration structure

Return: OK on success, NOK on failure

📌GPIO_PadConfig_t

Configuration structure for pin electrical characteristics:

```
typedef struct {
  GPIO_RESTYPE_t resType;// Pull-down=0, Pull-up=1
  GPIO_LOCKFLAG_t lockFlag;// 0=unlocked, 1=locked
  GPIO_SLEWRATE_t slewRate;// 0=disabled, 1=enabled
    } GPIO_PadConfig_t;
```

📌Internal Functions

The driver uses these static functions internally:

- **Port Unlock:**

```
static STD_ERROR GPIO_StdErrorPortUnlock(u32 Copy_u32Port, u8 Copy_u8PortValue);
```

Description: Unlocks port configuration using a special magic value (0x4C4F434B)

Parameters: Port identifier, Lock/unlock value

Return: OK on success, NOK on failure

- **Port Commit:**

```
static STD_ERROR GPIO_StdErrorPortCommit(u32 Copy_u32Port, u8 Copy_u8PortValue);
```

Description: Allows changes to protected registers

Parameters: Port identifier, Commit value

Return: OK on success, NOK on failure

- **Port Digital Enable:**

```
static STD_ERROR GPIO_StdErrorPortDigitalEnable(u32 Copy_u32Port, u8 Copy_u8PortValue);
```

Description: Enables digital functionality for pins

Parameters: Port identifier, Enable value

Return: OK on success, NOK on failure

⚠️Error Handling

All functions return a status code:

- `OK`: Operation completed successfully
- `NOK`: Operation failed

Functions perform parameter validation, checking for:

- Valid port identifier
- Valid pin number (0-7)
- Non-NULL pointers
- Valid parameter values

📝Usage Examples

- **Initializing a GPIO Port:**

```
 /* Initialize Port F */
if (GPIO_StdErrorInit(PortF) == OK) {
/* Port F initialized successfully */
} else {
/* Error occurred during initialization */
```

- **Configuring a Pin as Output and Setting it High:**

```
/* Configure Pin 3 on Port F as output */
GPIO_StdErrorSetPinDir(PortF, PIN3, PIN_OUTPUT);

/* Set Pin 3 on Port F high */
GPIO_StdErrorWritePin(PortF, PIN3, PIN_HIGH);
```

- **Reading a Pin State:**

```
u8 pinState;
/* Read the state of Pin 4 on Port B */
if (GPIO_StdErrorReadPin(PortB, PIN4, &pinState) == OK) {
/* pinState now contains the pin value (0 or 1) */
```

- **Configuring Pin with Pull-up and Slow Slew Rate:**

```
GPIO_PadConfig_t padConfig;
padConfig.resType = GPIO_PULL_UP;
padConfig.lockFlag = GPIO_UNLOCKED;
padConfig.slewRate = GPIO_SLEW_RATE_ENABLE;

/* Apply configuration to Pin 2 on Port A */
GPIO_StdErrorSetPinPadConfig(PortA, PIN2, &padConfig);
```

✅Best Practices

1. Always check return values from driver functions
2. Initialize ports before using them
3. Configure pin direction before reading or writing
4. Use appropriate pad configurations for your application needs
5. Unlock protected pins before modifying them
6. Re-lock pins after configuration to prevent accidental changes

📌Technical Notes

- The driver uses a bit-banding technique to efficiently access individual bits in registers
- Special handling is implemented for protected pins that require unlock/commit operations
- Alternate function configuration automatically sets necessary auxiliary registers
- Error handling provides robust operation even with invalid parameters

📌Memory Considerations

The driver is designed to be memory-efficient:

- Uses direct register access to minimize overhead
- Implements reusable internal functions
- Efficient port selection logic

🗺️Register Map

The hardware registers are accessed through the `GPIO_PORT_t` structure:

Register	Description
GPIODATA	register for reading pin values
GPIO_WRITE_DATA	Data register for writing pin values
GPIODIR	Direction control register
GPIOAFSEL	Alternate function select register
GPIOPUR	Pull-up resistor configuration register
GPIOPDR	Pull-down resistor configuration register
GPIOSLR	Slew rate control register
GPIODEN	Digital enable register
GPIOLOCK	Lock register (protects configuration)
GPIOCR	Commit register (allows configuration changes)
GPIOAMSEL	Analog mode select register
GPIOPCTL	Port control for alternate function

⚙️ UART

CODE: CLICK ME

UART (Universal Asynchronous Receiver/Transmitter) is a hardware communication protocol used to send and receive serial data between devices without needing a clock signal.

⏱️ Clock frequency:

👉 #define UARTSystemClock 16000000.0

📌 Defining UART Peripherals:

👉

```
#define UART_PR_UART0 0
#define UART_PR_UART1 1
#define UART_PR_UART2 2
#define UART_PR_UART3 3
#define UART_PR_UART4 4
#define UART_PR_UART5 5
#define UART_PR_UART6 6
#define UART_PR_UART7 7

// macros for UART Peripherals
```

📌 Defining UART registers:

📌 #define UART_CTL_ENABLEBIT 0

// bit 0 of CTL register is the enable bit

📌 #define UART_CTL_TXEBIT 8

// bit 8 of CTL register is the transmission enable bit (TXE)

📌 #define UART_CTL_RXEBIT 9

// bit 9 of CTL register is the receive enable bit (RXE)

```
#define UART_FR_TXFFBIT 5
```

// Transmission FIFO is full bit

```
#define UART_FR_RXFEBIT 4
```

// Receive FIFO is empty

```
#define UART_LCRH_PENBIT 1
```

// Bit n1 in LCRH register is the PEN (Parity Enable) bit

```
#define UART_LCRH_EVENORODDBIT 2
```

// Bit n2 chooses between even or odd parity (EPS)

```
#define UART_LCRH_STPBIT 3
```

// Bit n3 chooses between one or two stop bits (STP)

```
#define UART_LCRH_FENBIT 4
```

// FIFO Enable bit

📌 UART Configuration Structure:

```
typedef struct{
    UART_TYPE_t *Module; // UARTnum (e.g. UART0, UART1, UART2 → UART7)
    UART_DATABITS_t DataBits; // How many data bits we are using (Choose: 1) DataBits5 2) DataBits6 3) DataBits7 4) DataBits8 )
    UART_STOPBITS_t StopBits; // How many stop bits (1 or 2)
    UART_PARITYONOFF_t ParOnOff; // Disable or Enable Parity
    UART_PARITY_t Parity; // Even or Odd Parity
    u32 BaudRate;

} UART_CONFIG_t;
```

```
STD_ERROR UART_StdErrorInit(UART_CONFIG_t *Copy_ptrConfig);
```

// Initialize and define the struct before calling the fn

```
STD_ERROR UART_StdErrorSendByte(UART_CONFIG_t *Copy_ptrConfig, u8 Copy_u8Data);
```

- Function : “UART_StdErrorSendByte”
  Sends one byte of data through a specific UART module.

- Parameters:
  Copy_ptrConfig : Pointer to UART configuration (which UART, settings).
  Copy_u8Data : The byte of data you want to send.

- Returns: “OK” on success, “NOK” on failure

📌 STD_ERROR UART_StdErrorReceiveByte(UART_CONFIG_t *Copy_ptrConfig, u8 *Copy_ptru8Data)

- Function : “UART_StdErrorReceiveByte”

  Receives one byte of data from a specific UART module.

- Parameters:

  Copy_ptrConfig : Pointer to UART configuration (which UART, settings).

  Copy_ptru8Data: Pointer to store the received data byte.

- Returns: “OK” on success, “NOK” on failure

⛔GPIO Interrupt:

CODE: CLICK ME

This documentation explains the GPIO interrupt driver for the TivaC TM4C123 microcontroller. The driver is split into four files: `INT_interface.h`, `INT_config.h`, `INT_private.h`, and `INT_program.c`. Together, they help you set up, configure, and handle interrupts on GPIO pins (like buttons or sensors) easily. Let’s break it down step by step.

📋 Overview

The driver lets you:

- Turn on interrupts for specific pins on GPIO ports (A, B, C, D, E, F).
- Decide how the interrupt triggers (e.g., when a signal rises or falls).
- Set a function to run when the interrupt happens (like turning on an LED).
- Turn off interrupts for a port or all interrupts at once.

Here’s how the files work together:

- **INT_interface.h** : The "menu" of functions you can use.
- **INT_config.h** : The "settings" for how interrupts behave.
- **INT_private.h** : The "secret details" like memory addresses.
- **INT_program.c** : The "engine" that makes everything work.

Let’s explore each file and its functions!

📌 INT_interface.h

This file lists the main functions you’ll use to control interrupts. Think of it as the driver’s public face.

1. **Functions to Enable Interrupts:**

These turn on interrupts for a specific pin on a GPIO port:

```
void INT_voidPORTA_Enable(u8 Copy_u8PinID)
```

- **What it does**: Turns on the interrupt for a pin on Port A (e.g., pin 3).
- **Input**: `Copy_u8PinID` is the pin number (0 to 7).
- **Example**: `INT_voidPORTA_Enable(3)` enables an interrupt on pin 3 of Port A.

```
void INT_voidPORTB_Enable(u8 Copy_u8PinID)
```

- Same as above, but for Port B.

```
void INT_voidPORTC_Enable(u8 Copy_u8PinID)
```

- Same, but for Port C.

```
void INT_voidPORTD_Enable(u8 Copy_u8PinID)
```

- Same, but for Port D.

```
void INT_voidPORTE_Enable(u8 Copy_u8PinID)
```

- Same, but for Port E.

```
void INT_voidPORTF_Enable(u8 Copy_u8PinID)
```

- Same, but for Port F.

2. **Functions to Disable Interrupts:**

These turn off interrupts:

```
void INT_voidGPIO_Disable(u8 Copy_u8Port)
```

- **What it does**: Turns off all interrupts for a specific port.
- **Input**: `Copy_u8Port` is the port (e.g., `PortA`, `PortB`, etc.).
- **Example**: `INT_voidGPIO_Disable(PortA)` disables interrupts on Port A.

```
void INT_voidGPIO_DisableGlobal(void)
```

- **What it does**: Turns off all interrupts across the microcontroller.
- **Input**: None.
- **Example**: `INT_voidGPIO_DisableGlobal()` stops all interrupts.

3. **Function to Set a Callback:**

```
void INT_voidFunc(u8 Copy_u8Port, u8 Copy_u8PinID, void(*Ptr_Func)(void))
```

- **What it does**: Sets a function to run when an interrupt happens on a pin.
- **Inputs**:
  - `Copy_u8Port`: The port (e.g., `PortA`).
  - `Copy_u8PinID`: The pin number (0 to 7).
  - `Ptr_Func`: The function to call (e.g., `myFunction`).
- **Example**: `INT_voidFunc(PortA, 3, myFunction)` runs `myFunction` when pin 3 on Port A triggers an interrupt.

📌 INT_config.h

This file sets up how interrupts work for each port. You can choose if they trigger on edges (signal changes) or levels (signal stays high/low), and whether they react to rising/high or falling/low signals.

1. **Settings for Each Port**

For each port (A to F), there are two settings:

1. **Sense**: Edge or Level

   ```
   #define PORTA_SENSE EDGE
   ```

   - Ports A, B, C, D, E use `EDGE`; Port F uses `LEVEL`.
   - **Edge**: Triggers when the signal changes (rising or falling).
   - **Level**: Triggers when the signal stays at a certain level (high or low).

2. **Event**: High or Low

   ```
   #define PORTA_EVENT HIGH
   ```

   - All ports use `HIGH`.
   - **High**: Triggers on a rising edge (if edge-triggered) or high signal (if level-triggered).
   - **Low**: Would trigger on a falling edge or low signal (not used here).

You can change these `#define` values to customize the driver!

📌 INT_private.h

This file holds private info like memory addresses and interrupt numbers. You don't call these directly, but they're used by the driver.

1. **Key Definitions**

```
#define NVIC_ISER0 *((volatile u32*)0xE000E100)
```

- A memory spot (`0xE000E100`) to enable interrupts.

2. **Interrupt Numbers**

```
#define GPIOA_IRQn 0
#define GPIOB_IRQn 1
#define GPIOC_IRQn 2
#define GPIOD_IRQn 3
#define GPIOE_IRQn 4
#define GPIOF_IRQn 30
```

These numbers tell the microcontroller which interrupt belongs to which port.

📌 INT_program.c

This file has the actual code for all the functions, plus helpers and interrupt handlers. It's the heart of the driver.

1. **Helper Functions – GPIO Register Control**

These low-level functions directly interact with GPIO registers:

```
STD_ERROR GPIO_stdErrorSetInterruptMask(u32 Copy_u32Port, u8 Copy_u8Value);
```

- Turns interrupt mask **on/off** for a given port.
- Inputs: `Copy_u32Port` (port), `Copy_u8Value` (e.g., `0xFF` to enable all pins).
- Sets the `GPIOIM` register.

```
STD_ERROR GPIO_stdErrorSetInterruptSense(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8Value);
```

- Selects **edge** (`0`) or **level** (`1`) triggering.
- Modifies the `GPIOIS` register.

```
STD_ERROR GPIO_stdErrorSetInterruptBothEdges(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8Value);
```

- Enables triggering on **both rising and falling** edges.
- `Copy_u8Value = 1` for both, `0` for single.
- Modifies `GPIOIBE`.

```
STD_ERROR GPIO_stdErrorSetInterruptEvent(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8Value);
```

- Sets event type: **rising/high** (`1`) or **falling/low** (`0`).
- Updates the `GPIOIEV` register.

```
STD_ERROR GPIO_stdErrorSetInterruptClear(u32 Copy_u32Port, u8 Copy_u8PinId, u8 Copy_u8Value);
```

- Clears the **interrupt flag**.
- Writes to `GPIOICR`.

```
STD_ERROR GPIO_stdErrorGetInterruptStatus(u32 Copy_u32Port, u8 Copy_u8PinId, u8* Copy_pu8Status);
```

- Checks whether an **interrupt occurred** on the pin.
- `Copy_pu8Status = 1` if triggered.
- Reads from `GPIORIS`.

📌 NVIC Control Functions

These handle enabling/disabling interrupts in the interrupt controller (NVIC):

```
void NVIC_EnableIRQ(u8 IRQn);
```

- Enables a specific interrupt (e.g., `GPIOA_IRQn`).

```
void NVIC_DisableIRQ(u8 IRQn);
```

- Disables a specific interrupt.

Enable Functions (e.g., `INT_voidPORTA_Enable`) – Step by Step:

Each port's enable function follows this sequence:

1. Enable the interrupt in NVIC (e.g., `NVIC_EnableIRQ(GPIOA_IRQn)`).
2. Temporarily disable the interrupt mask with `0x00`.
3. Configure edge or level triggering from `INT_config.h`.
4. Set to single-edge detection (no both-edges).
5. Define the trigger event (HIGH or LOW).
6. Clear any existing interrupt flag.
7. Enable the interrupt mask:
   - `0xFF` for Ports A, B, C, D, F.
   - `0x08` for Port E (only pin 3).
8. Globally enable interrupts using `__enable_irq()`.

📌 Callback Setup

```
INT_voidFunc(PORT, Pin, FunctionPtr);
```

- Registers your **callback function** in the relevant array:
  - e.g., `array_A[3] = myFunction`.

📌 Interrupt Handlers

(e.g., `GPIOA_Handler`)

What happens when an interrupt is triggered:

1. Loop through all 8 pins of the port.
2. Use `GPIO_stdErrorGetInterruptStatus` to check if an interrupt occurred.
3. If yes, run the corresponding callback, e.g., `array_A[3]()`.
4. Clear the interrupt flag with `GPIO_stdErrorSetInterruptClear`.

How It All Fits Together:

1. **Setup**: Configure the GPIO pin as input (done via separate GPIO functions).
2. **Enable**: Call `INT_voidPORTA_Enable(3)` for pin 3 on Port A.
3. **Set Callback**: Use `INT_voidFunc(PORTA, 3, myFunction)` to attach your function.
4. **Interrupt Occurs**: If triggered, `GPIOA_Handler` runs `myFunction`.
5. **Disable**: Use `INT_voidGPIO_Disable(PORTA)` or `INT_voidGPIO_DisableGlobal()` to stop interrupts.

💾EEPROM Driver

CODE: CLICK ME

This guide explains the EEPROM driver for the TivaC TM4C123 microcontroller. The driver is made up of four files that work together to let you store, read, and erase data in the EEPROM memory. We'll break it down into small, simple pieces and explain each part clearly.

📋 Overview

- **Initialize** the EEPROM.
- **Read** a 32-bit value from any EEPROM address.
- **Write** a 32-bit value to any EEPROM address.
- **Erase** a block of EEPROM memory.
- **Recover** from errors automatically.

📁 Driver File Structure

1. `EEPROM_interface.h` – Public functions you can call.
2. `EEPROM_config.h` – Configuration (currently empty).
3. `EEPROM_private.h` – Low-level details (registers/macros).
4. `EEPROM_program.c` – Actual implementation of all functions.

1. `EEPROM_interface.h`

```
STD_ERROR EEPROM_STD_ERROR_Init(void)
```

- Powers on and checks the EEPROM
- Returns `OK` or `NOK`
- Call this once before anything else

```
u32 EEPROM_u32ReadData(u32 Copy_u32Address)
```

- Reads a 32-bit number from EEPROM
- Input: `Copy_u32Address` – target address
- Returns the 32-bit data at that address

```
STD_ERROR EEPROM_STD_ERROR_WriteData(u32 Copy_u32Address, u32 Copy_u32Data)
```

- Writes a 32-bit number to EEPROM
- Inputs:
  - `Copy_u32Address` – where to write
  - `Copy_u32Data` – the value to store
- Returns `OK` or `NOK`

```
STD_ERROR EEPROM_STD_ERROR_MassEraseData(u32 Copy_u32Address)
```

- Erases the EEPROM block that includes the specified address
- Input: Any address within the block
- Returns `OK` or `NOK`

```
void EEPROM_voidConfigBlockOffset(u32 Copy_u32Address)
```

- Internally selects the correct EEPROM block and offset
- Used internally by read/write/erase functions

2. `EEPROM_config.h`

- **Currently empty**
- Future use: customizable settings (EEPROM size, default block addresses)

3. `EEPROM_private.h`

📌 Key Registers

- `SYSCTL_SREEPROM_R`: Resets EEPROM module
- `EEPROM_EEBLOCK_R`: Selects current memory block
- `EEPROM_EEOFFSET_R`: Selects word offset in the block
- `EEPROM_EERDWR_R`: Read/write data register
- `EEPROM_EEDONE_R`: Status info (busy/done/error)
- `EEPROM_EESUPP_R`: Error flags
- `EEPROM_EEDBGME_R`: Mass erase trigger register

📌 Important Status Bits

- `EEPROM_EEDONE_WRBUSY`: Write still in progress
- `EEPROM_EEDONE_WORKING`: EEPROM is busy
- `EEPROM_EESUPP_PRETRY`: Write retry required
- `EEPROM_EESUPP_ERETRY`: Erase retry required

📌 Handy Macros

- `EEPROM_WAITING`: Waits until EEPROM is ready
- `EEPROM_PRETRY` / `EEPROM_ERETRY`: Error recovery helpers

- `EEPROM_program.c` :

Helper Functions

```
void EEPROM_voidSoftwareReset(void)
```

- Resets the EEPROM module

- Steps:
  1. Set reset bit
  2. Wait 5 ms
  3. Clear reset bit
  4. Wait 5 ms again

```
void EEPROM_voidFixError(void)
```

- Attempts to fix failed writes/erases
- Steps:
  1. Detect retry errors
  2. Set repair bit
  3. Wait 20 ms
  4. Repeat up to 10 times
  5. If failed, reset EEPROM

```
void EEPROM_voidConfigBlockOffset(u32 Copy_u32Address)
```

- Calculates EEPROM block and offset
- Formula:
  - Block = address >> 4
  - Offset = address & 0xF
- Example: Address `0x0010` → Block 1, Offset 0

Main Functions:

```
STD_ERROR EEPROM_STD_ERROR_Init(void)
```

- Initializes the EEPROM module
- Steps:
  1. Enable EEPROM clock
  2. Wait 10 ms
  3. Check for busy/error
  4. If error: reset → fix → check again
- Returns `OK` or `NOK`

```
u32 EEPROM_u32ReadData(u32 Copy_u32Address)
```

- Reads a 32-bit word from the EEPROM
- Steps:
  1. Wait until EEPROM is ready
  2. Set block/offset
  3. Read from `EEPROM_EERDWR_R`

```
STD_ERROR EEPROM_STD_ERROR_WriteData(u32 Copy_u32Address, u32 Copy_u32Data)
```

- Writes a 32-bit word to EEPROM
- Steps:
  1. Wait until EEPROM is ready
  2. Set block/offset
  3. Write to `EEPROM_EERDWR_R`
  4. Check write status
- Returns `OK` or `NOK`

```
STD_ERROR EEPROM_STD_ERROR_MassEraseData(u32 Copy_u32Address)
```

- Erases a whole EEPROM block

- Steps:
  1. Wait until EEPROM is ready
  2. Set block/offset
  3. Write erase code to `EEPROM_EEDBGME_R`
  4. Check erase status
- Returns `OK` or `NOK`

How to Use This Driver

1. **Initialize**:

```
EEPROM_STD_ERROR_Init();
```

2. **Write Data**:

```
EEPROM_STD_ERROR_WriteData(0x0000, 1234);
```

3. **Read Data**:

```
u32 data = EEPROM_u32ReadData(0x0000);
```

4. **Erase Block**:

```
EEPROM_STD_ERROR_MassEraseData(0x0000);
```

---

2. HAL:

⚙️ LED:

CODE: CLICK ME

🔲 Overview

This LED driver abstracts and manages both **internal** and **external** LED control on the microcontroller using GPIO interfaces. It provides functions to initialize, turn on, turn off, and toggle LEDs.

This header file declares the public API for LED control:

- `LED_VoidInitInternalLeds()`
- `LED_StdErrorSetInternalLedValue(u8 LedId, u8 State)`
- `LED_voidInitExternalLed(u8 Port, u8 Pin)`
- `LED_voidTurnOnLed(u8 Port, u8 Pin)`
- `LED_voidTurnOffLed(u8 Port, u8 Pin)`
- `LED_voidToggleExternalLed(u8 Port, u8 Pin, u8 DelayMs)`

Constants Defined:

```
#define RED_LED_PIN     PIN1
#define BLUE_LED_PIN    PIN2
#define GREEN_LED_PIN   PIN3
// These macros map the RGB LED pins on Port F (PF1, PF2, PF3) of the TM4C123G LaunchPad.
#define LED_ON          1
#define LED_OFF         0
// LED_ON and LED_OFF are used to control LED states (1 = ON, 0 = OFF).
```

Implementing the LED control functions using GPIO and Systick drivers.

📌 Internal LED Initialization

```
void LED_VoidInitInternalLeds(void);
```

- Configures RED, GREEN, BLUE internal LEDs on `PortF` as output.
- Initializes all internal LEDs to `OFF`.

📌 Internal LED State Setter

```
STD_ERROR LED_StdErrorSetInternalLedValue(u8 LedId, u8 State);
```

- Sets individual internal LED state.
- Returns `OK` or `NOK` based on success.

📌 External LED Initialization

```
void LED_voidInitExternalLed(u8 Port, u8 Pin);
```

- Sets the given pin on a specified port as output and turns off the LED.

📌 External LED Control

```
void LED_voidTurnOnLed(u8 Port, u8 Pin);
void LED_voidTurnOffLed(u8 Port, u8 Pin);
```

- Turns on/off external LEDs by writing `1` or `0`.

📌 Toggle External LED

```
void LED_voidToggleExternalLed(u8 Port, u8 Pin, u8 DelayMs);
```

- Reads current pin state.
- Toggles its value and adds delay using `Systick`.

📌 Dependencies

- `GPIO_Interface.h` :GPIO_StdErrorInit, GPIO_StdErrorSetPinDir, GPIO_StdErrorWritePin, GPIO_StdErrorReadPin
- `SYSTICK_interface.h` :Systick_StdErrorDelayIn_ms
- `STD_TYPES.h` STD_ERROR LED_StdErrorSetInternalLedValue

⚙️SW:

CODE: CLICK ME

🗒️ Overview

This driver handles both **internal** and **external** switch (button) inputs. It abstracts the GPIO configuration, pull-up/down behavior, and press detection logic.

Configuration header for internal and external switch resistance.

```
#define INTERNAL_SW_RES    GPIO_PULL_UP
#define EXTERNAL_SW_RES    GPIO_PULL_UP
```

Defining public functions and constants:

Constants:

```
#define Internal_SW1    PIN4
#define Internal_SW2    PIN0
 // Specify the pin numbers for the onboard switches on Port F.
#define SW_PRESSED      1
#define SW_NOT_PRESSED  0
 // Represent the logical states for switch input detection.
```

Functions:

```
STD_ERROR SW_StdErrorInitInternal(u8 Copy_u8Sw);
 // Initializes an internal switch with the specified pin.
u8 SW_u8InternalSWIsPressed(u8 Copy_u8Sw);
 // Returns the state (pressed/not pressed) of an internal switch.

STD_ERROR SW_StdErrorInitExternal(u8 Copy_u8Port, u8 Copy_u8Pin);
 // Initializes an external switch connected to a specific port and pin.
u8 SW_u8ExternalSWIsPressed(u8 Copy_u8Port, u8 Copy_u8Pin);
 // Returns the state of an external switch (pressed/not pressed) based on port and pin
```

⚙️ Implementation Details in `SW_program.c`

🚨 Internal Switch Initialization

```
STD_ERROR SW_StdErrorInitInternal(u8 Copy_u8Sw);
```

- Initializes internal switches (typically onboard on PortF).
- Sets direction to input.
- Applies pull-up/down resistance as configured.

📌 Read Internal Switch State

```
u8 SW_u8InternalSWIsPressed(u8 Copy_u8Sw);
```

- Reads switch logic level.
- Interprets the press based on the configured pull-up/pull-down.
- Returns:
  - `SW_PRESSED` if the switch is actively pressed.
  - `SW_NOT_PRESSED` otherwise.

🚨 External Switch Initialization

```
STD_ERROR SW_StdErrorInitExternal(u8 Copy_u8Port, u8 Copy_u8Pin);
```

- General-purpose function to set up external switches.
- Configures direction, resistance, and disables analog function.

📌 Read External Switch State

```
u8 SW_u8ExternalSWIsPressed(u8 Copy_u8Port, u8 Copy_u8Pin);
```

- Reads external pin value.
- Returns switch state in the same logic as internal ones.

📌 Dependencies

- `GPIO_Interface.h` :GPIO_StdErrorInit, GPIO_StdErrorSetPinDir, GPIO_StdErrorSetPinPadConfig, GPIO_StdErrorPinAnalogModeSelect, GPIO_StdErrorReadPin,

```
GPIO_PadConfig_t Local_PadConfig = {
      INTERNAL_SW_RES,
      GPIO_UNLOCKED,
      GPIO_SLEW_RATE_DISABLE}
```

- `STD_TYPES.h`

📺 CLCD:

CODE: CLICK ME

📋 Overview

The CLCD driver provides control over a character LCD using **8-bit mode**. It supports displaying characters, strings, and numbers on a 2-line LCD module, with full initialization, command, and data transmission features.

Configuration file for assigning GPIO ports/pins to the LCD:

```
#define DATA_PORT   PortB  // Port used for 8-bit data lines (D0–D7).
#define CTRL_PORT   PortA  // Port used for control signals (RS, RW, ENA).

#define RS         PIN2   // Register Select pin – selects between command (0) and data (1).
#define RW          PIN3   // Read/Write pin – selects between write (0) and read (1).
#define ENA         PIN4   // Enable pin – triggers data/command latching on the LCD.
```

Public interface with user-accessible functions and constants:

Line Selection Constants:

```
#define LINE_1   0
#define LINE_2   1
```

Function Prototypes:

```
STD_ERROR CLCD_StdErrorDataPinsInit(void)
void CLCD_voidDataSendString(u8 *Copy_u8Str);            //to send each character
void CLCD_voidDataSetCursor(u8 Copy_u8LineNum, u8 Copy_u8Location); // Move cursor to a specific position on the LCD
void CLCD_voidClearDisplay(void);
void CLCD_voidSendNum(u16 Copy_u16Num);
```

Private internal functions used only by the driver implemented in `CLCD_private.h`

```
STD_ERROR CLCD_StdErrorDataPinssendCommand(u8 Copy_u8Command);
STD_ERROR CLCD_StdErrorSendData(u8 Copy_u8Data);
```

⚙️ **Implementation Details in** `CLCD_program.c`

📌 Initialization

```
STD_ERROR CLCD_StdErrorDataPinsInit(void);
```

- Configures data port and control pins(RW,EN.RS) as outputs.
- Sends CLCD setup commands:
  - `0x38` : 8-bit, 2 lines, 5×8 font
  - `0x0F` : Display ON
  - `0x01` : Clear screen
  - `0x06` : Entry mode

📌 Send Command to LCD

```
STD_ERROR CLCD_StdErrorDataPinssendCommand(u8 Copy_u8Command);
```

- Sets RS=0, RW=0 and writes a command byte to the LCD.

📌 Send Data (Character) to LCD

```
STD_ERROR CLCD_StdErrorSendData(u8 Copy_u8Data);
```

- Sets RS=1, RW=0 and writes a data byte (character) to the LCD.

🔤 Display String

```
void CLCD_voidDataSendString(u8 *Copy_u8Str);
```

- Sends a null-terminated string to the display.

📌 Set Cursor Position

```
void CLCD_voidDataSetCursor(u8 Copy_u8LineNum, u8 Copy_u8Location);
```

- Moves the cursor to a specific row and column.
  - Line 1: `0x80 + column`
  - Line 2: `0xC0 + column`

📌 Clear Screen

```
void CLCD_voidClearDisplay(void);
```

- Clears the entire display.

🔢 Display Number

```
void CLCD_voidSendNum(u16 Copy_u16Num);
```

- Converts a 16-bit integer to characters and displays it.
- Handles digit extraction and reverse order display.

📌 Dependencies

- `GPIO_Interface.h` :GPIO_StdErrorWritePin, GPIO_StdErrorWritePort, GPIO_StdErrorSetPinDir
- `SYSTICK_interface.h` :Systick_StdErrorDelayIn_ms

- `STD_TYPES.h`

🛰️ GPS

CODE: CLICK ME

📋 Overview

The GPS Module is a Hardware Abstraction Layer (HAL) component that provides functionality to interface with GPS modules, process NMEA sentences, extract location data, and perform geographical calculations. This module is part of the Ain Shams University Embedded Systems project.

📚 File Structure

📄 GPS_config.h

Configuration header file for GPS module parameters.

- Currently empty, ready for future configuration options

📄 GPS_interface.h

Public interface header file that declares all functions and data types exposed to other modules.

- Contains function declarations and type definitions

📄 GPS_private.h

Private header file containing internal macros, types, and function declarations.

- Defines constants such as PI and EARTH_RADIUS_M
- Declares array sizes for parsing NMEA sentences
- Declares private utility functions

📄 GPS_program.c

Implementation file containing the actual code for all GPS functions.

- Includes functions for parsing NMEA sentences
- Contains geographical calculation algorithms
- Implements data conversion utilities

📝 Data Types

✅❌ GPS_Status_t

An enumeration representing the possible status values when getting GPS data:

```
typedef enum {
    GPS_NO_DATA = 0,      // No data received from GPS module
    GPS_INVALID_DATA = 1, // Data received but invalid
    GPS_VALID_DATA = 2    // Valid GPS data received
} GPS_Status_t;
```

🔍 Constants & Macros

Constant	Value	Description
`PI`	3.14159265358979323846	Mathematical constant pi
`EARTH_RADIUS_M`	6371000	Earth's radius in meters
`MAX_SENTENCE_FIELDS`	15	Maximum number of fields in a NMEA sentence
`SENTENCE_SIZE`	100	Maximum size of a NMEA sentence

🧩 Public Functions

GPS Coordinate Conversion

`GPS_ConvertToDecimalDegrees`

Converts GPS coordinates from DDMM.MMMM format to decimal degrees.

```
f64 GPS_ConvertToDecimalDegrees(f64 Copy_f64CoordInGPSFormat);
```

Parameters:

- `Copy_f64CoordInGPSFormat`: GPS coordinate in DDMM.MMMM format

Returns:

- Coordinate in decimal degrees

Example:

- Input: 3007.038 (30 degrees, 7.038 minutes)

- Output: 30.1173 degrees

Angular Conversion

`GPS_f64ToRadians`

Converts an angle from degrees to radians.

```
f64 GPS_f64ToRadians(f64 degree);
```

Parameters:

- `degree` : Angle in degrees

Returns:

- Angle in radians

Distance Calculation

`GPS_f64GetDistance_ArgInDegrees`

Calculates the great-circle distance between two points on Earth using the Haversine formula.

```
f64 GPS_f64GetDistance_ArgInDegrees(f64 Copy_f64LatGPS, f64 Copy_f64LongGPS,
                                    f64 Copy_f64LatSaved, f64 Copy_f64LongSaved);
```

Parameters:

- `Copy_f64LatGPS` : Current latitude in decimal degrees
- `Copy_f64LongGPS` : Current longitude in decimal degrees
- `Copy_f64LatSaved` : Target latitude in decimal degrees
- `Copy_f64LongSaved` : Target longitude in decimal degrees

Returns:

- Distance in meters between the two points

GPS Data Acquisition

`GPS_stdStatusGetLocation`

Parses NMEA sentences to extract location data from GPS.

```
GPS_Status_t GPS_stdStatusGetLocation(UART_CONFIG_t *Copy_ptrUartConfig,
                                      f64 *Copy_ptrf64Latitude,
                                      f64 *Copy_ptrf64Longitude,
                                      f64 *Copy_ptru8Time);
```

Parameters:

- `Copy_ptrUartConfig` : Pointer to UART configuration structure
- `Copy_ptrf64Latitude` : Pointer to store the extracted latitude
- `Copy_ptrf64Longitude` : Pointer to store the extracted longitude
- `Copy_ptru8Time` : Pointer to store the extracted time

Returns:

- GPS status (NO_DATA, INVALID_DATA, or VALID_DATA)

`GPS_stdStatusWaitForValidData`

Waits for valid GPS data with a timeout.

```
GPS_Status_t GPS_stdStatusWaitForValidData(UART_CONFIG_t *Copy_ptrUartConfig,
                                           f64 *Copy_ptrf64Latitude,
                                           f64 *Copy_ptrf64Longitude,
                                           f64 *Copy_ptru8Time,
                                           u16 Copy_u16Timeout);
```

Parameters:

- `Copy_ptrUartConfig` : Pointer to UART configuration structure
- `Copy_ptrf64Latitude` : Pointer to store the extracted latitude
- `Copy_ptrf64Longitude` : Pointer to store the extracted longitude
- `Copy_ptru8Time` : Pointer to store the extracted time
- `Copy_u16Timeout` : Number of attempts before timeout

Returns:

- GPS status (NO_DATA, INVALID_DATA, or VALID_DATA)

Location Comparison

`GPS_IsWithinRadius`

Checks if a current position is within a specified radius of a target position.

```
u8 GPS_IsWithinRadius(f64 Copy_f64CurrentLat, f64 Copy_f64CurrentLong,
                      f64 Copy_f64TargetLat, f64 Copy_f64TargetLong,
                      f64 Copy_f64RadiusInMeters);
```

Parameters:

- `Copy_f64CurrentLat`: Current latitude in decimal degrees
- `Copy_f64CurrentLong`: Current longitude in decimal degrees
- `Copy_f64TargetLat`: Target latitude in decimal degrees
- `Copy_f64TargetLong`: Target longitude in decimal degrees
- `Copy_f64RadiusInMeters`: Radius in meters

Returns:

- 1 if within radius, 0 if outside radius

🔒 Private Functions

`GPS_CustomStringToFloat`

Converts a string to a floating-point number.

```
f64 GPS_CustomStringToFloat(u8 *str);
```

Parameters:

- `str`: String to convert

Returns:

- Floating-point representation of the string

`GPS_stdStatusReadNMEASentence` (static)

Reads a complete NMEA sentence from UART.

```
static GPS_Status_t GPS_stdStatusReadNMEASentence(UART_CONFIG_t *Copy_ptrUartConfig,
                                                  u8 *Copy_arru8Buffer);
```

Parameters:

- `Copy_ptrUartConfig`: Pointer to UART configuration structure
- `Copy_arru8Buffer`: Buffer to store the NMEA sentence

Returns:

- GPS status (NO_DATA, INVALID_DATA, or VALID_DATA)

🔄 Implementation Details

NMEA Sentence Parsing

The module focuses on parsing GPRMC (Recommended Minimum Specific GPS/Transit data) sentences, which have the following format:

```
$GPRMC,time,status,latitude,N/S,longitude,E/W,...
```

Example: `$GPRMC,123519,A,3007.038,N,09733.701,W,...`

- 123519 = Time (12:35:19 UTC)
- A = Data valid (A=valid, V=invalid)
- 3007.038,N = Latitude 30° 7.038' North
- 09733.701,W = Longitude 97° 33.701' West

Coordinate Conversion

GPS coordinates are typically provided in DDMM.MMMM format (degrees and minutes). This module converts them to decimal degrees for easier calculations:

```
Decimal Degrees = Degrees + (Minutes / 60)
```

Distance Calculation

The module uses the Haversine formula to calculate the great-circle distance between two points on Earth:

1. Convert all coordinates from degrees to radians

2. Calculate using the formula:
   - a = sin²(Δlat/2) + cos(lat1) × cos(lat2) × sin²(Δlong/2)
   - c = 2 × atan2(√a, √(1−a))
   - distance = R × c (where R is Earth's radius)

💻 Usage Examples

1. Getting the current GPS location

```
UART_CONFIG_t gps_uart_config = {
    // Configure UART parameters for GPS module
    .BaudRate = BAUD_9600
    // Add other required settings
};

f64 latitude, longitude, time;
GPS_Status_t status;

status = GPS_stdStatusGetLocation(&gps_uart_config, &latitude, &longitude, &time);

if (status == GPS_VALID_DATA) {
    // Use the obtained coordinates
    printf("Latitude: %f, Longitude: %f\n", latitude, longitude);
} else {
    printf("Could not get valid GPS data\n");
}
```

2**. Calculating distance between two points**

```
f64 current_lat = 30.12345;  // Current position
f64 current_long = 31.54321;

f64 destination_lat = 30.10000;  // Destination
f64 destination_long = 31.50000;

f64 distance = GPS_f64GetDistance_ArgInDegrees(
    current_lat, current_long,
    destination_lat, destination_long
);

printf("Distance to destination: %.2f meters\n", distance);
```

3. Checking if within a radius of a point

```
f64 current_lat = 30.12345;  // Current position
f64 current_long = 31.54321;

f64 target_lat = 30.12300;   // Target position
f64 target_long = 31.54300;

f64 radius = 50.0;  // 50 meters radius

if (GPS_IsWithinRadius(current_lat, current_long, target_lat, target_long, radius)) {
    printf("You've reached your destination!\n");
} else {
    printf("Keep moving toward the destination.\n");
}
```

📌 Dependencies

- **UART Module**: Required for communication with the GPS hardware
- **STD_TYPES.h**: For standardized data types (u8, u16, f64, etc.)
- **BIT_MATH.h**: For bit manipulation macros
- **math.h**: For mathematical functions (sin, cos, atan2, etc.)
- **string.h**: For string manipulation functions (strstr, strtok_r)
- **Optional Dependencies**:
  - CLCD_interface.h (Commented out, for debugging)
  - LED_interface.h (Commented out, for debugging)

- SYSTICK_interface.h (For delay functions)

🛠️ Configuration Options

Currently, the GPS_config.h file is empty but could be extended with the following configuration options:

- GPS module type selection
- UART configuration defaults for GPS
- Default timeout values
- Debug mode settings

⚠️ Error Handling

The module uses the GPS_Status_t enumeration to report errors:

- `GPS_NO_DATA`: No data could be received from the GPS module
- `GPS_INVALID_DATA`: Data was received but was invalid or corrupted
- `GPS_VALID_DATA`: Valid GPS data was successfully parsed

Functions return appropriate status codes to indicate success or failure.

🔍 Notes and Limitations

1. The module specifically parses GPRMC sentences only
2. A valid GPS fix is required for accurate readings
3. Earth is assumed to be perfectly spherical for distance calculations
4. Altitude is not considered in distance calculations
5. Debug LCD code is included but commented out

🔄 Version History

Version	Date	Changes
1.0	Unknown	Initial release
2.0	Unknown	Unknown changes
3.0	October 5, 2025	Current version with improved NMEA parsing and distance calculations

📚 References

1. NMEA 0183 Standard: https://www.nmea.org
2. Haversine Formula: https://en.wikipedia.org/wiki/Haversine_formula
3. GPS Coordinate Systems: https://en.wikipedia.org/wiki/Geographic_coordinate_system
4. TM4C123G LaunchPad User's Guide: https://www.ti.com/lit/pdf/spmu296
5. TM4C123GH6PM Data Sheet: https://www.ti.com/lit/ds/spms376e/spms376e.pdf
6. NEO-6 Data Sheet: https://content.u-blox.com/sites/default/files/products/documents/NEO-6_DataSheet_(GPS.G6-HW-09005).pdf

3. LIB:

⚙️ STD_TYPES

CODE: CLICK ME

```
📌 #ifndef STD_TYPES_H_
   #define STD_TYPES_H_
   ...
   #endif
```

- Ensures that `STD_TYPES_H_` is **only included once per compilation unit**, avoiding **redefinition errors**.

📌 Unsigned Integer Data Types

```
typedef unsigned char      u8;
typedef unsigned short int u16;
typedef unsigned long int  u32;
typedef unsigned long long int u64;
```

- Defines **unsigned integer types** with **fixed sizes** to ensure consistency across different platforms.

  ex:

```
u8  var1 = 255;                          // 8-bit (0 to 255)
u16 var2 = 65535;                        // 16-bit (0 to 65,535)
u32 var3 = 4294967295;                   // 32-bit (0 to 4,294,967,295)
u64 var4 = 18446744073709551615ULL;      // 64-bit large number
```

📌 Signed Integer Data Types

```
typedef signed char       s8;
typedef signed short int  s16;
typedef signed long int   s32;
typedef signed long long int s64;
```

- Defines **signed integer types** to store both positive and negative values.

  ex:

```
s8  var5 = -128;                         // 8-bit (-128 to 127)
s16 var6 = -32768;                       // 16-bit (-32,768 to 32,767)
s32 var7 = -2147483648;                  // 32-bit (-2,147,483,648 to 2,147,483,647)
s64 var8 = -9223372036854775807LL;       // 64-bit range
```

📌 Floating-Point Data Types

```
typedef float  f32;
typedef double f64;
```

- Provides **floating-point** types for handling decimal values.

  ex:

```
f32 temperature = 36.5f;                 // 32-bit float
f64 pi = 3.141592653589793;              // 64-bit double precision
```

📌 Error Status Enum

```
typedef enum {
   NOK = 1,
   OK  = 0
} STD_ERROR;
```

- Defines **standard return values** for **function success or failure**.

  ex:

```
STD_ERROR checkStatus() {
return OK;  // Function returns 0 for success }

if (checkStatus() == OK) {
// Operation successful }

else {

// Operation failed }
```

📌 #define NULL ((void*)0)

- Defines a **null pointer constant**, used to represent a pointer that doesn't point to any valid memory location.
- Often used to **initialize**, **reset**, or **check** pointers safely.

  ex:

```
char* name = NULL;  // Pointer initialized to null

if (name == NULL) {
// Condition is true, pointer is null (not pointing anywhere) }
```

1️⃣ vs 0️⃣ BIT_MATH

CODE: CLICK ME

📌 SET_BIT(reg, bit)

reg |= (1 << bit)

- Sets a specific bit in a register to `1` without affecting other bits.

  ex:

  ```
  // Initial value of PORTA: 0000 0000 (All bits cleared)
  SET_BIT(PORTA, 1);
  // PORTA after execution: 0000 0010 (Bit 1 set)
  ```

📌 CLR_BIT(reg, bit)

reg &= ~(1 << bit)

- Clears (sets to `0`) a specific bit in a register without affecting other bits.

  ex:

  ```
  // Initial value of PORTA: 0000 0010 (Bit 1 set)
  CLR_BIT(PORTA, 1);
  // PORTA after execution: 0000 0000 (Bit 1 cleared)
  ```

📌 TOG_BIT(reg, bit)

reg ^= (1 << bit)

- Toggles (flips) the state of a specific bit (0 → 1 or 1 → 0).

  ex:

  ```
  // Initial value of PORTA: 0000 0010 (Bit 1 set)
  TOG_BIT(PORTA, 1);
  // PORTA after execution: 0000 0000 (Bit 1 toggled to 0)
  TOG_BIT(PORTA, 1);
  // PORTA after execution: 0000 0010 (Bit 1 toggled back to 1)
  ```

📌 GET_BIT(reg, bit)

((reg & (1 << bit)) >> bit)

- Retrieves the value (0 or 1) of a specific bit.

  ex:

  ```
  // Initial value of PORTA: 0000 0010 (Bit 1 set)
  u8 value = GET_BIT(PORTA, 1);
  // value = 1 (Bit 1 is set)
  ```

📌 IS_BIT_SET(reg, bit)

((reg & (1 << bit)) != 0)

- Checks if a specific bit is `1` (set) and returns `1` if true, `0` otherwise.

  ex:

  ```
  // Initial value of PORTA: 0000 0010 (Bit 1 set)
  if (IS_BIT_SET(PORTA, 1)) {
  // Condition is true, since Bit 1 is set (1)
  ```

📌 IS_BIT_CLR(reg, bit)

((reg & (1 << bit)) == 0)

- Checks if a specific bit is `0` (cleared) and returns `1` if true, `0` otherwise.

ex:

```
// Initial value of PORTA: 0000 0000 (All bits cleared)
if (IS_BIT_CLR(PORTA, 1)) {
// Condition is true, since Bit 1 is cleared (0)}
```

4. APP:

⚙️APP.h

CODE: CLICK ME

📊 Threshold Definitions

```
#define RED_LED_THRESHOLD     50
#define YELLOW_LED_THRESHOLD  25
#define GREEN_LED_THRESHOLD   10
#define MAX_DISTANCE_THRESHOLD 1000
```

- These constants define the **distance (or metric) thresholds** for turning on:
  - 🔴 Red LED if value ≥ 50
  - 🟡 Yellow LED if value ≥ 25
  - 🟢 Green LED if value ≥ 10
- `MAX_DISTANCE_THRESHOLD` sets an upper limit on how far a location can be considered for evaluation.

💡 State and Mode Flags

```
#define TURNED_ON   1
#define TURNED_OFF  0

#define SLIDE_SHOW_MODE TURNED_ON
```

- General use flags for **on/off states**.
- `SLIDE_SHOW_MODE` is pre-set to enabled (`TURNED_ON`).

📍 Saved Location Structure

```
typedef struct {
    u8 name[15];
    f64 Lat;
    f64 Long;
} SavedLocations_t;
```

- Holds one saved location:
  - `name` (e.g., “Library”)
  - `Lat` = latitude
  - `Long` = longitude
- Typically used as an **array** to store multiple favorite or known locations.

🧭Closest Location Structure

```
typedef struct {
    u8 name[15];
    f64 distance;
} ClosestPlace_t;
```

- Stores info about the **closest location** relative to current position.
- Contains:
  - `name` of location
  - `distance` to that place

🧠 Application-Level Function Prototypes

```
void PeripheralsInit(void);
void BuzzerTrigger(void);
void TurnOffAllLeds(void);
```

```
void EEPROM_TimesVisitedIncrement(u32 Copyu32_EEPROMAddresses);
void ConvertUTCtoLocalTime(f64 utcTime, u8* timeStr);
void StringCopy(u8 *destination, u8 *source, u8 maxLength);
```

- `PeripheralsInit()` – Initializes UART, GPIOs, switches, LEDs, buzzer, etc.
- `BuzzerTrigger()` – Plays a simple tone pattern.
- `TurnOffAllLeds()` – Turns off all status LEDs.
- `EEPROM_TimesVisitedIncrement()` – Increments the visit counter at a given EEPROM address.
- `ConvertUTCtoLocalTime()` – Converts `hhmmss.ss` UTC to `hh:mm:ss` format in UTC+3.
- `StringCopy()` – Safe string copying utility with a max length limit.

⚙️APP.c

CODE: CLICK ME

“contains helper functions for the main.c code”

🔧Peripheral Initialization

```
void PeripheralsInit(void) {
    UART_CONFIG_t uart2configuration;
    GPIO_PadConfig_t GPIO_PF2;

    uart2configuration.Module = UART2;
    uart2configuration.DataBits = DataBits8;
    uart2configuration.StopBits = OneStopBit;
    uart2configuration.ParOnOff = ParityDisable;
    uart2configuration.Parity = OddParity;
    uart2configuration.BaudRate = 9600;

    GPIO_PF2.resType = GPIO_PULL_DOWN;
    GPIO_PF2.lockFlag = GPIO_UNLOCKED;
    GPIO_PF2.slewRate = GPIO_SLEW_RATE_DISABLE;

    // GPIO Ports
    GPIO_StdErrorInit(PortA);
    GPIO_StdErrorInit(PortB);
    GPIO_StdErrorInit(PortC);
    GPIO_StdErrorInit(PortF);

    // Switches
    SW_StdErrorInitExternal(PortC, PIN4); // Exit Button (Brown)
    SW_StdErrorInitExternal(PortC, PIN5); // Enter Button (Green)
    SW_StdErrorInitExternal(PortC, PIN6); // Main Functionality (Red)
    SW_StdErrorInitExternal(PortC, PIN7); // Slide Show Select (Blue)
    SW_StdErrorInitExternal(PortE, PIN3); // Report Interrupt Trigger

    INT_voidPORTE_Enable(PIN3); // Enable Interrupt

    // LEDs
    LED_voidInitExternalLed(PortA, PIN5); // RED
    LED_voidInitExternalLed(PortA, PIN6); // YELLOW
    LED_voidInitExternalLed(PortA, PIN7); // GREEN

    CLCD_StdErrorDataPinsInit();          // LCD
    UART_StdErrorInit(&uart2configuration); // UART2
    EEPROM_STD_ERROR_Init();              // EEPROM

    // Buzzer on PF2
    GPIO_StdErrorSetPinDir(PortF, PIN2, PIN_OUTPUT);
    GPIO_StdErrorSetPinPadConfig(PortF, PIN2, &GPIO_PF2);
}
```

🔔Buzzer Function

```
void BuzzerTrigger(void) {
    u8 i;
    for (i = 0; i < 2; i++) {
        GPIO_StdErrorWritePin(PortF, PIN2, PIN_HIGH);
        Systick_StdErrorDelayIn_ms(300);
```

```
        GPIO_StdErrorWritePin(PortF, PIN2, PIN_LOW);
        Systick_StdErrorDelayIn_ms(100);
    }
    GPIO_StdErrorWritePin(PortF, PIN2, PIN_HIGH);
    Systick_StdErrorDelayIn_ms(1000);
    GPIO_StdErrorWritePin(PortF, PIN2, PIN_LOW);
}
```

- Plays a short melody using the buzzer on **PortF Pin2**
- Delays created using **SysTick**

💡 Turn Off All LEDs

```
void TurnOffAllLeds(void) {
    LED_voidTurnOffLed(PortA, 5); // red
    LED_voidTurnOffLed(PortA, 6); // yellow
    LED_voidTurnOffLed(PortA, 7); // green
}
```

- Utility function to shut down all indicators.

💾 EEPROM Counter Increment

```
void EEPROM_TimesVisitedIncrement(u32 Copyu32_EEPROMAddresses) {
    u32 Local_TimesVisited = EEPROM_u32ReadData(Copyu32_EEPROMAddresses);
    EEPROM_STD_ERROR_WriteData(Copyu32_EEPROMAddresses, ++Local_TimesVisited);
}
```

- Reads the number of times a location was visited and increments the count.
- Stores result in EEPROM at given address.

🌍 UTC to Local Time Conversion

```
void ConvertUTCtoLocalTime(f64 utcTime, u8* timeStr) {
    u32 hours = (int)(utcTime / 10000);
    u32 minutes = (int)((utcTime - hours * 10000) / 100);
    u32 seconds = utcTime - hours * 10000 - minutes * 100;

    hours += 3;
    if (hours >= 24) {
        hours -= 24;
    }

    timeStr[0] = (hours / 10) + '0';
    timeStr[1] = (hours % 10) + '0';
    timeStr[2] = ':';
    timeStr[3] = (minutes / 10) + '0';
    timeStr[4] = (minutes % 10) + '0';
    timeStr[5] = ':';
    timeStr[6] = (seconds / 10) + '0';
    timeStr[7] = (seconds % 10) + '0';
    timeStr[8] = '\0';
}
```

- Converts **UTC time in format hhmmss.ss** to local time (UTC+3).
- Stores the result as a string in `hh:mm:ss` format.

📝 String Copy Utility

```
void StringCopy(u8 *destination, u8 *source, u8 maxLength) {
    u8 i;
    for (i = 0; i < maxLength && source[i] != '\0'; i++) {
        destination[i] = source[i];
    }
    destination[i] = '\0';
}
```

- Copies up to `maxLength` characters from `source` to `destination`.

🔥 How to Burn / Simulate:

Step 1

Download Keil V4:

https://engasuedu-my.sharepoint.com/personal/ayman_wagih_eng_asu_edu_eg/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Fayman%5Fwagih%5Feng%5Fasu%5Fedu%5Feg%2FDocuments%2FMicrosoft%20Teams%20Chat%20Files%2FMDK474%2EEXE&parent=%2Fpersonal%2Fayman%5Fwagih%5Feng%5Fasu%5Fedu%5Feg%2FDocuments%2FMicrosoft%20Teams%20Chat%20Files&ga=1

Step 2

Setup:

Step 3

Download this Driver & Technical Guide if needed

https://www.ti.com/tool/STELLARIS_ICDI_DRIVERS

Step 4

Download Launch Pad for simulations:

https://drive.google.com/file/d/1qZrwf3wPQY6IZAuxOuejcoE7X-c_LXYv/view

Step 5

Get the needed system file from C:Keil/ARM/Startup/TI/tm4c123

-Find this file system_TM4C123

-Copy this file to project files

Step 6: For burning

-Search for this icon: → Utilities→ disable debug driver→ in this drop down list select: stellaris ICDI

-In the same tab select debug aside utilities → enable use as seen in the picture below→ then select stellaris ICDI

-Then OK

-Finally to burn → all save files → build all files (first icon)→ download(second icon)

-Click Reset button on tiva then enjoy you project

Step 6: For simulation

-Copy the launchpad file

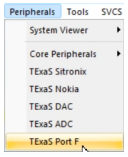
-Open C:keil/ARM/BIN → paste the file

-Search for this icon: → Debug→ Enable use simulator→ Add this to parameter -dLaunchPadDLL (making the full text like this

"-pCM4 -dLaunchPadDLL"

-Save all file files → Build all files(3rd icon) → Start debug (4th icon)

-Open peripherals → select used port → you will find a new tab opened with a simulation to the port



-Click run and Enjoy 🙂

💵 Project Requirements & Prices

🧩 Component	💵 Price (EGP)
Resistors	5×0.2=1.00 L.E
Pin Header male	3.50 L.E
Rotary Potentiometer	5.50 L.E
Round Press Button	7.00 L.E
Buzzer	7.00 L.E
LEDS	9.00 L.E
Bread Board	42 L.E
CLCD 16×2	65.00 L.E
Jumper Wires	90 L.E
External Power source	2X60=120 L.E
UBlox GPS	420 L.E
TM4C123GH6PM Microcontroller	2500 L.E
TOTAL:	3270 L.E