Advanced Multi-Pump Control System: Comprehensive Guide for Optimal Pressure Management and Efficient Operation

**Overview of System :**

**System Composition:** The system is composed of a sophisticated control board, designed to interface with Variable Frequency Drives (VFDs) or directly with pumps. Its primary function is to regulate the operation of these VFDs or pumps, aiming to maintain a specific pressure set point with maximum efficiency and fast response time .

**Hardware and Connectivity:** The hardware facilitates seamless connectivity, allowing users to interact with the board via Bluetooth Low Energy (BLE) through various platforms, including mobile phones, laptops, and desktop computers. This system supports a diverse range of applications, including mobile apps for iOS, web applications, and desktop software, offering users flexibility in monitoring and managing the system.

hello

**User Interface and Control:** Users can effortlessly access comprehensive system diagnostics, real-time operational statuses, and modify system Parameters through the user-friendly interface of the apps. The system employs a Proportional-Integral-Derivative (PID) controller for precise speed and operational control. Additionally, it features a dual scheduler system: one to ensure balanced working hours across all pumps, and another to enable user-defined operational schedules based on specific days, times, or date-triggered events.

**Communication Protocol and Customizability:** Leveraging the Modbus communication protocol, the system offers compatibility with any Modbus-enabled machinery, thus providing users with the capability to transmit custom commands. Knowledgeable users can directly communicate with and control various devices connected to the system.

**Enhanced Accessibility and Monitoring:** Accessibility is further enhanced through a bar-code scanning feature, allowing quick and easy connection to the system. The system archives a detailed history of alarms, facilitating effective troubleshooting and maintenance. Additionally, a dedicated monitoring panel is available for each pump, displaying vital data such as current speed, actual pressure, and target set points through interactive graphs and visualizations.

**Purpose :**

The primary purpose of this documentation is to provide an exhaustive and comprehensive understanding of the Advanced Multi-Pump Control System. It is designed to detail the system's design, functionalities, operational procedures, and the underlying code modules. The document serves as an all-encompassing guide, covering both the theoretical underpinnings and practical implementations of the system, from its architectural composition to the intricate details of its software modules and control mechanisms.

**Scope:**

**This documentation includes:**

**System Architecture and Code Modules:** An exploration of the system's design and structure, including hardware components, connectivity, and a detailed breakdown of the code modules that govern its functionality.

**Operational Functionality:** In-depth insights into the system's functionalities such as pressure maintenance, PID control, scheduling mechanisms, Modbus communication, and user interface operations.

**User Interaction and Control:** Guidelines on system interaction through various platforms, available functionalities, and the role of each code module in user operations.

**Target Audience :**

This documentation is designed for a broad audience, including:

**System Operators and Technicians:** Individuals responsible for daily operation and maintenance, including monitoring and troubleshooting.

**Control System Engineers and Developers:** Professionals involved in system design, development, and enhancement, needing detailed knowledge of the system's architecture and code modules.

**Industrial Managers and Decision-Makers:** Personnel overseeing processes or facilities using the system, requiring a comprehensive understanding for effective management.

**Educational and Training Institutions:** Instructors and students in control systems and industrial automation, looking for practical examples and in-depth analyses.

**Technical Support Staff:** Individuals providing technical assistance, requiring a deep understanding of the system's functions and potential issues.

**General Public and Non-Technical Users:** People with a general interest in the system or those seeking basic knowledge about its operation and applications.

**High-Level System Diagram and Description of System Components and Their Interactions:**

**Features:**

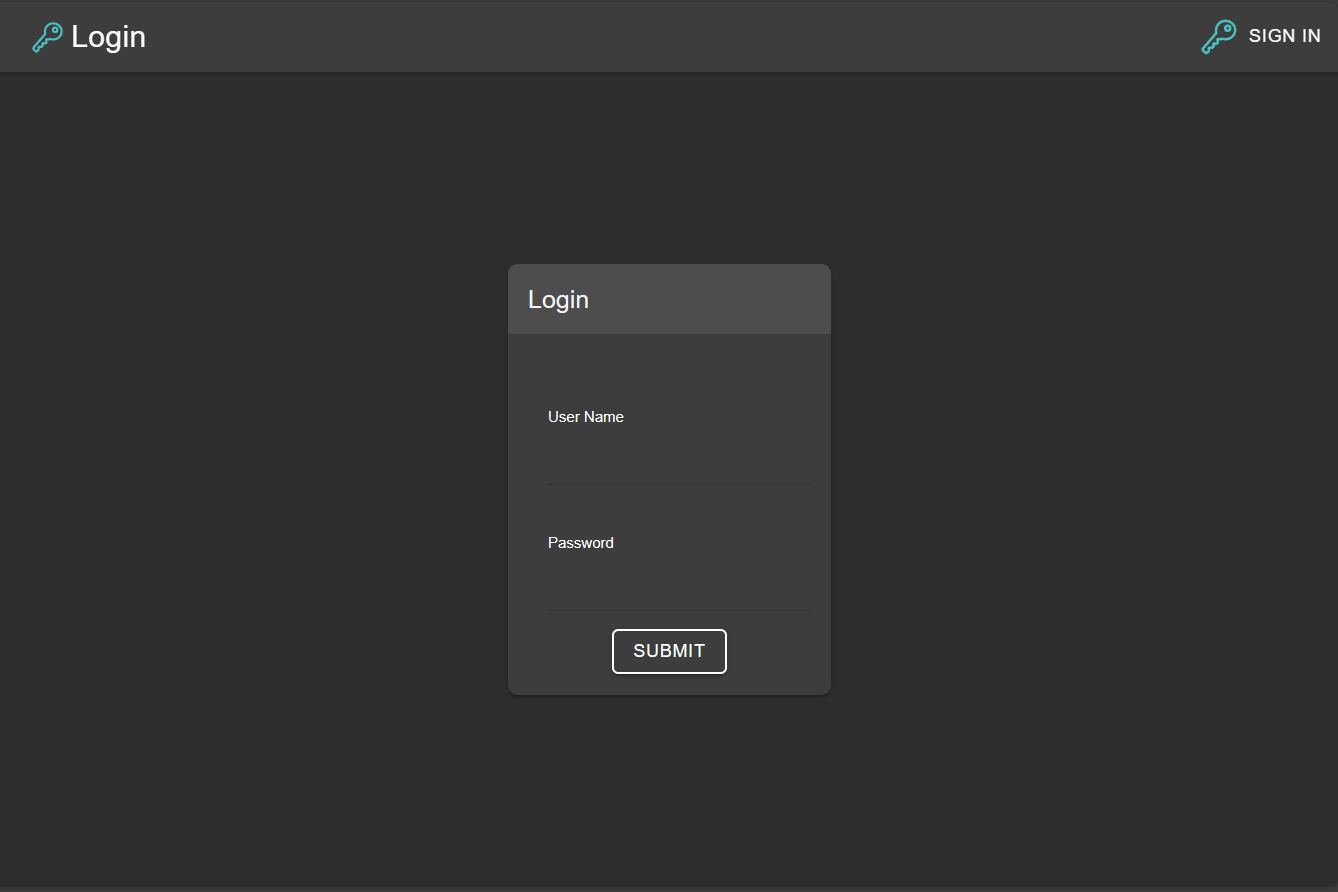
**User Interface:**

**Mobile App:** Provides on-the-go access and system control.

**Web App:** Allows access via web browsers for a more comprehensive view.

**Desktop App:** Offers robust functionalities for advanced users.

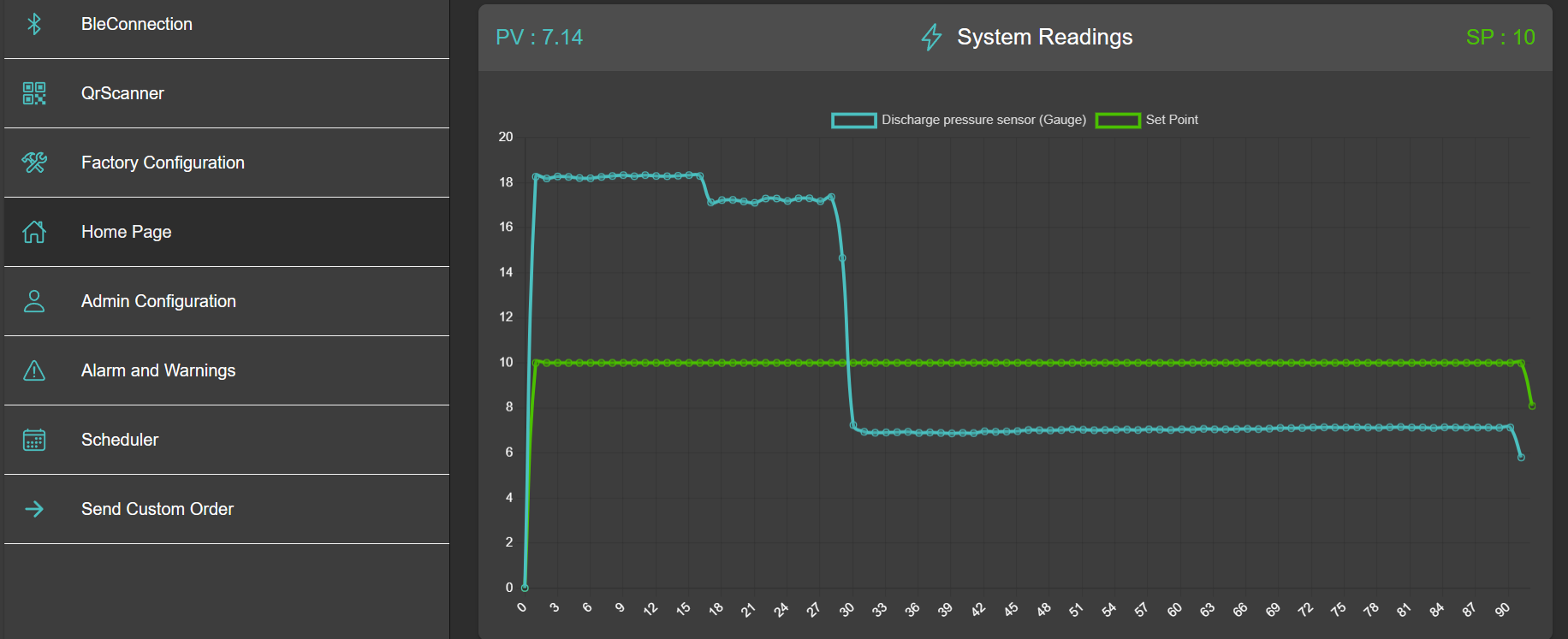
**FEATURES THAT THE SYSTEM PROVIDE:**

**DIFFERENT ACCESS LEVEL WITH EACH LEVEL HAVE IT ON SETTINGS :**

**FULL SYSTEM CONDITION MONITOR THROUGH A DEDICATED**

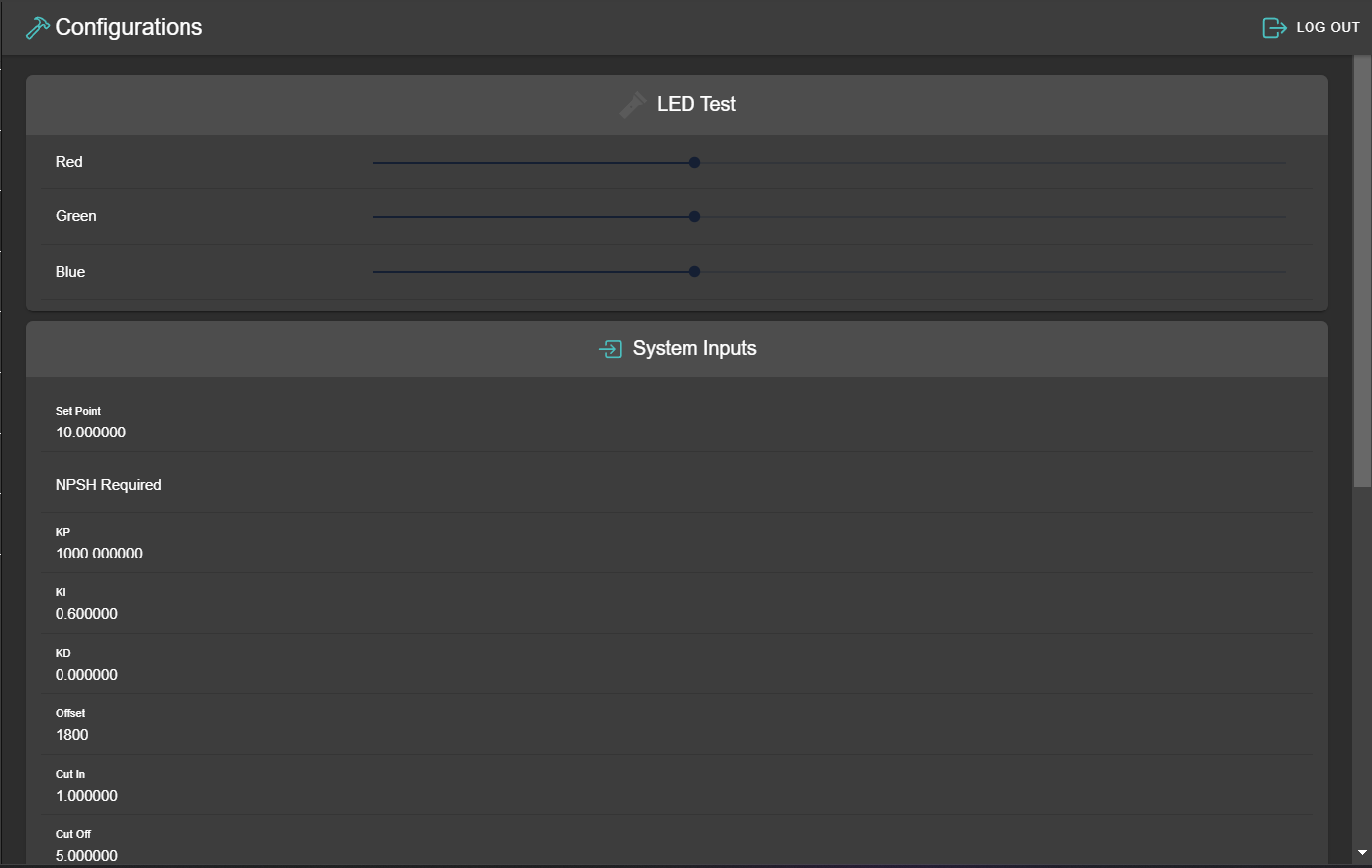
**CONTORL PANEL:**





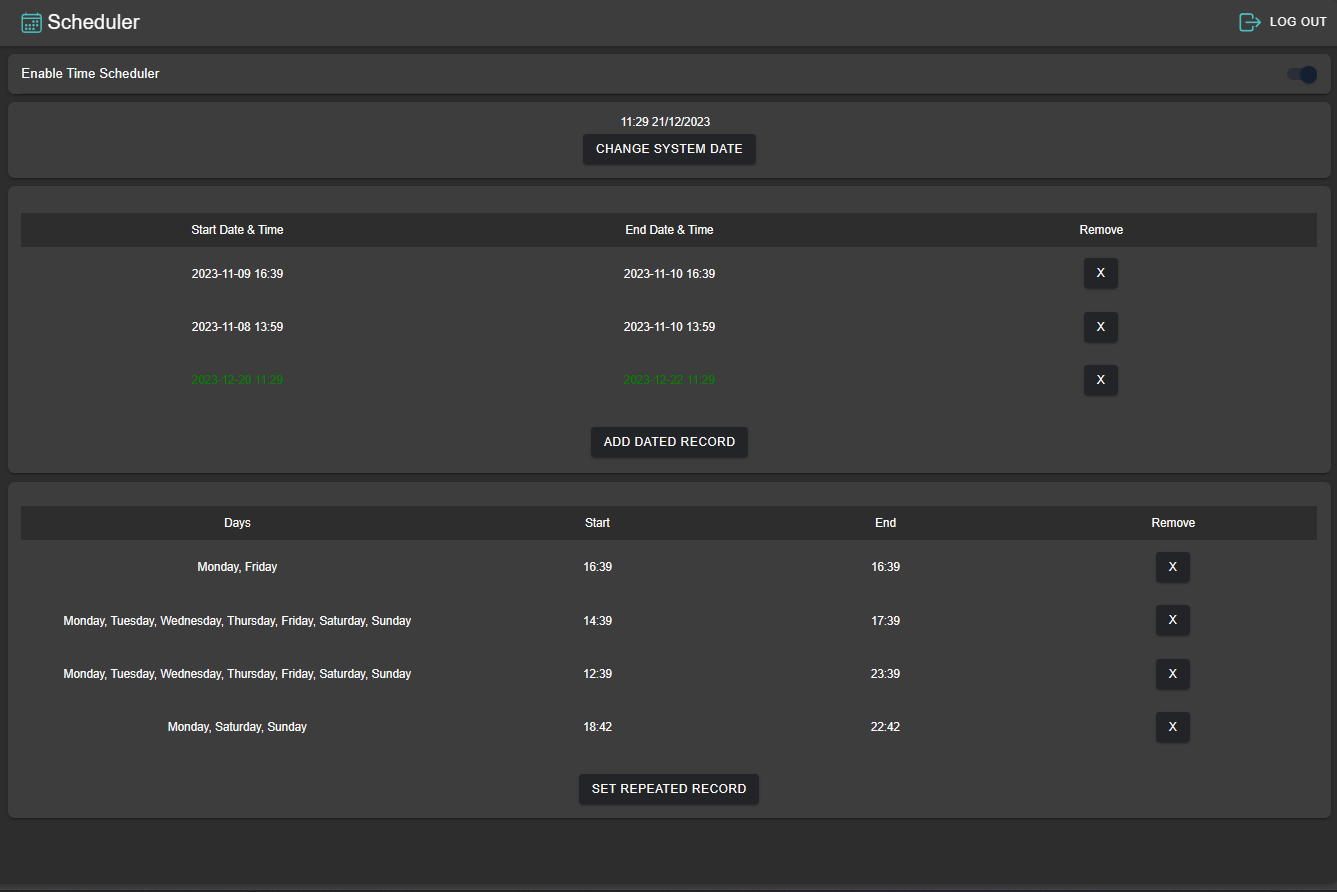
**MANUAL CONTROL OVER EACH INDIVIDUAL VFD:**

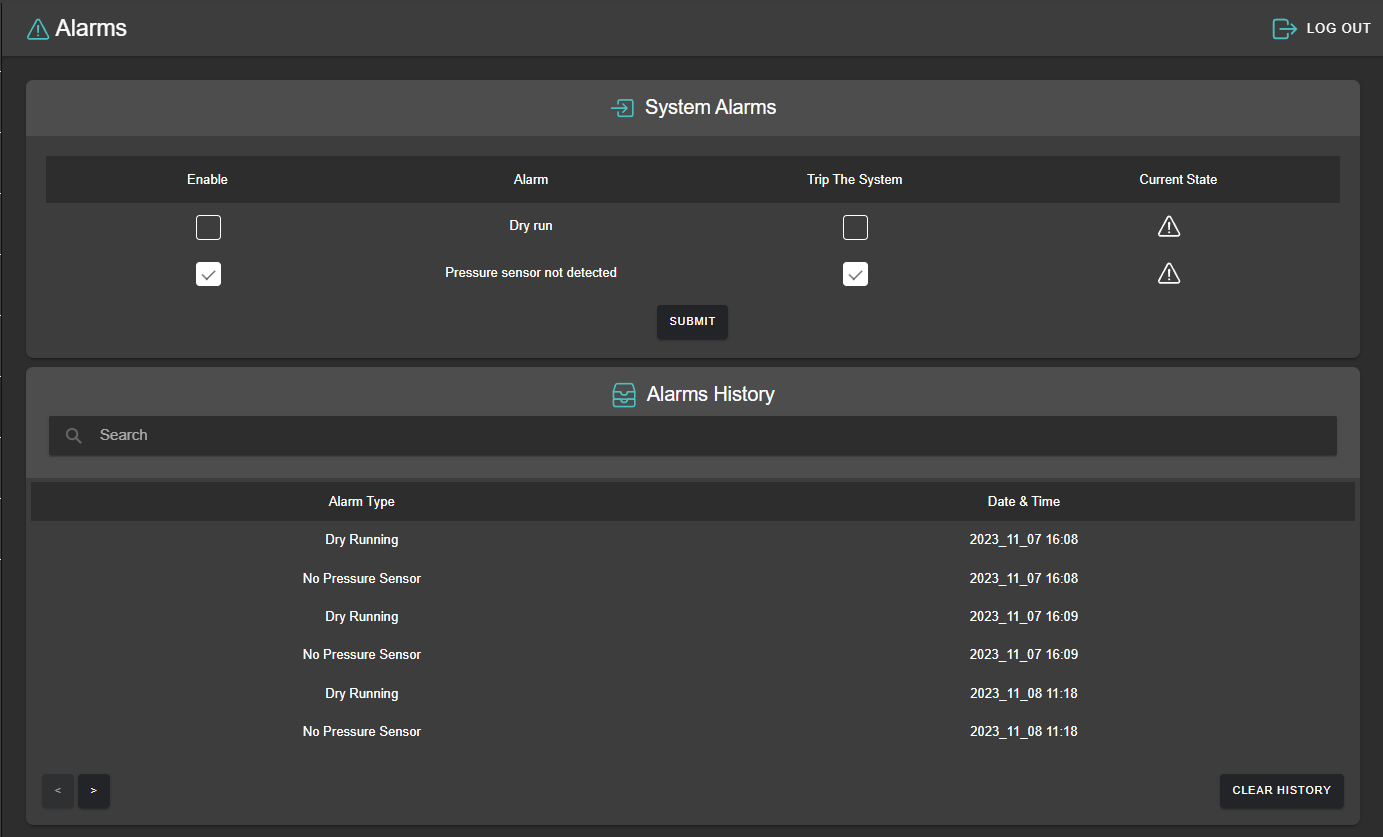
**THE SYSTEM USER HAVE ACCESS TO ALL THE SYSTEM PARAMETERS :**

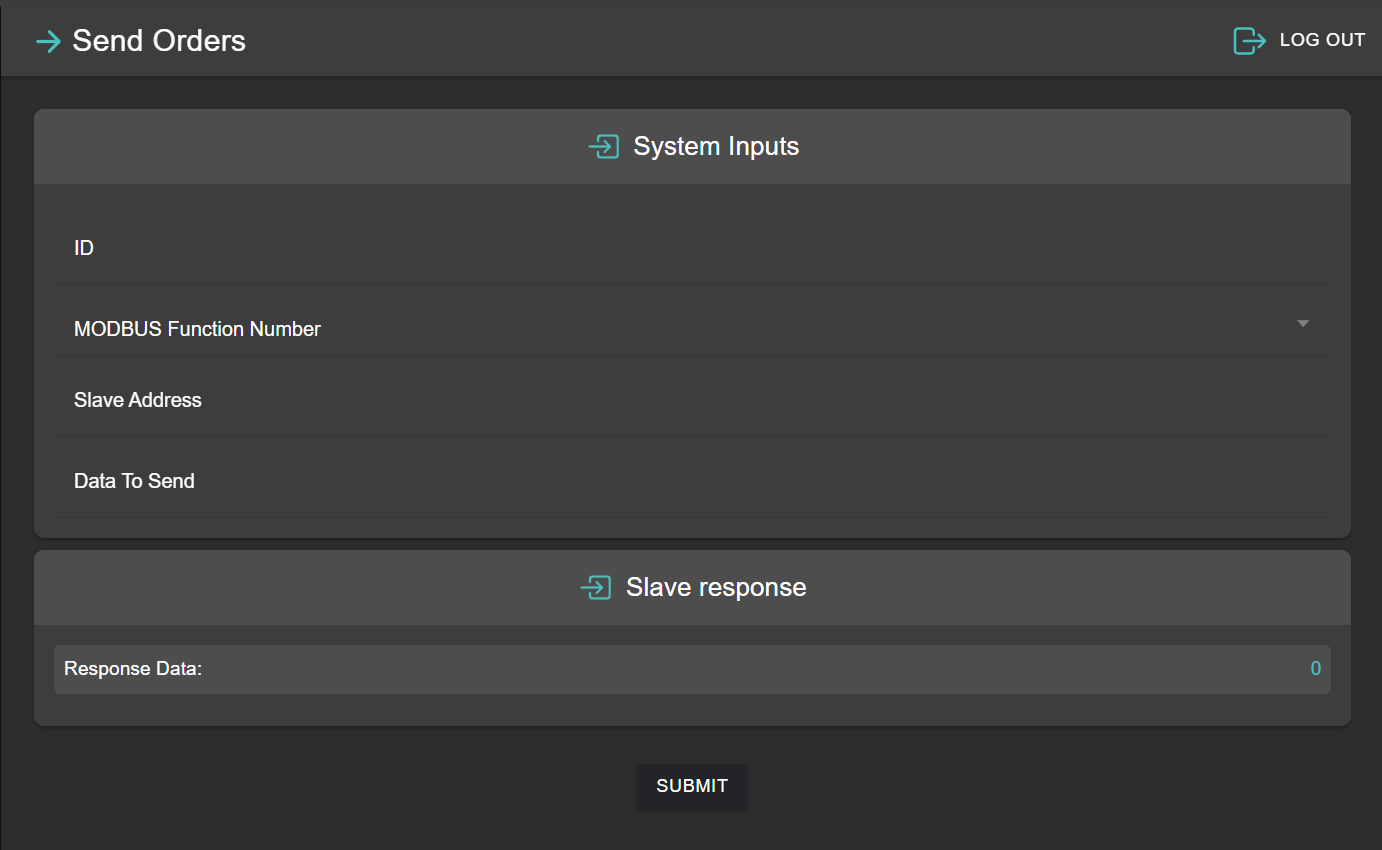


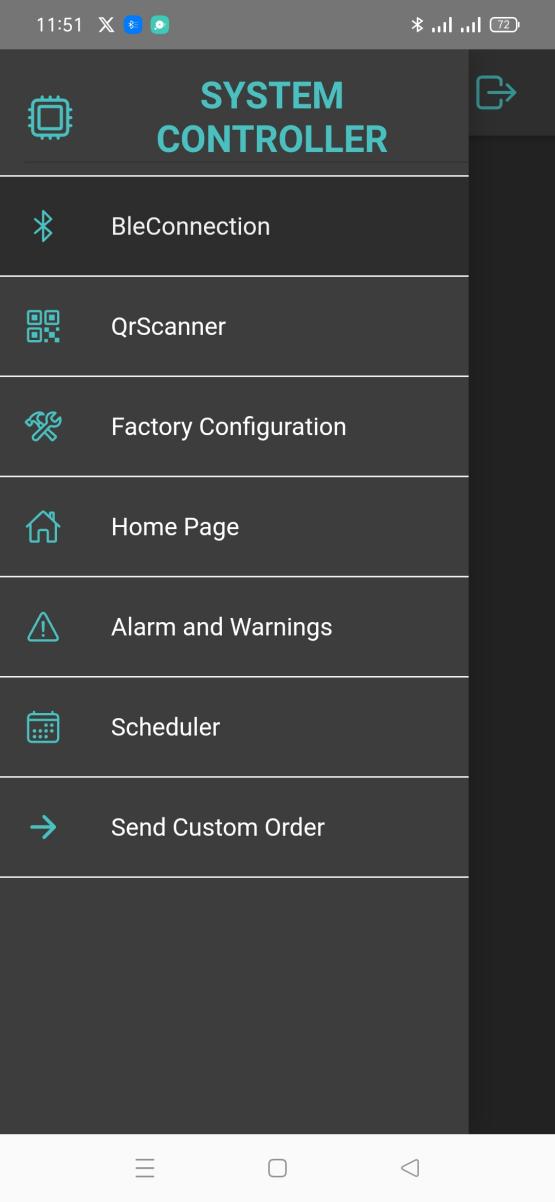
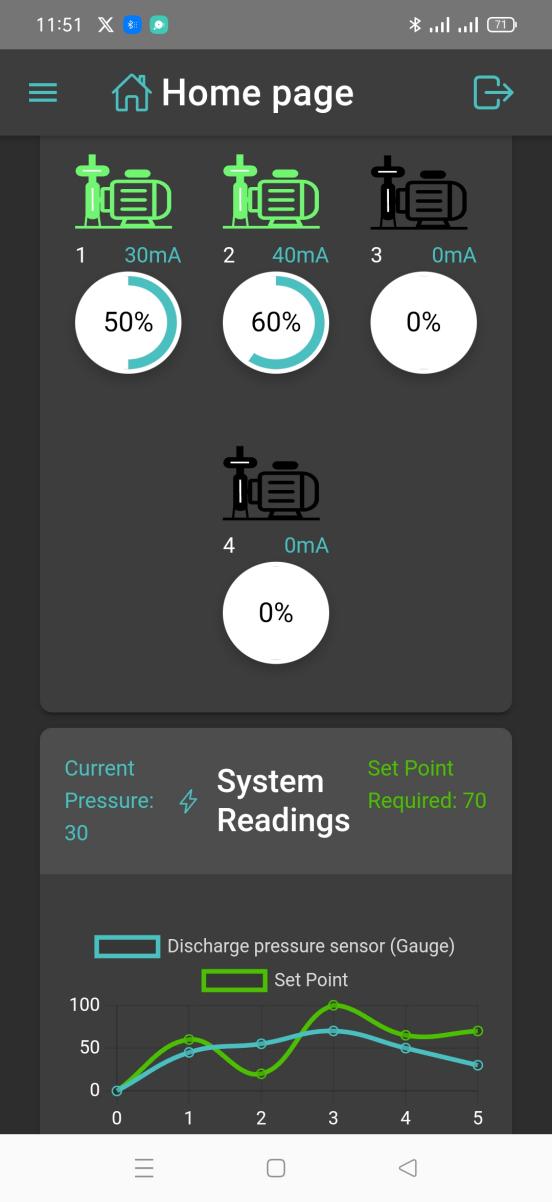
**SCHEDULER TO SET WHEN THE SYSTEM PID SHOULD WORK**

**ON A REGULAR BASIS OR ON DATED BASIS:**



**CURRENT SYSTEM ALARM AND WARNINGS AND HISTORY OF EACH EVENT HAPPENED ON THE SYSTEM:**

**THE ABILITY TO SEND ANY CUSTOM ORDER TO ANY MODBUS SLAVE OR READ DATA FROM IT :**

**DEDICATED MOBILE APP:**

**Communication Layer:**

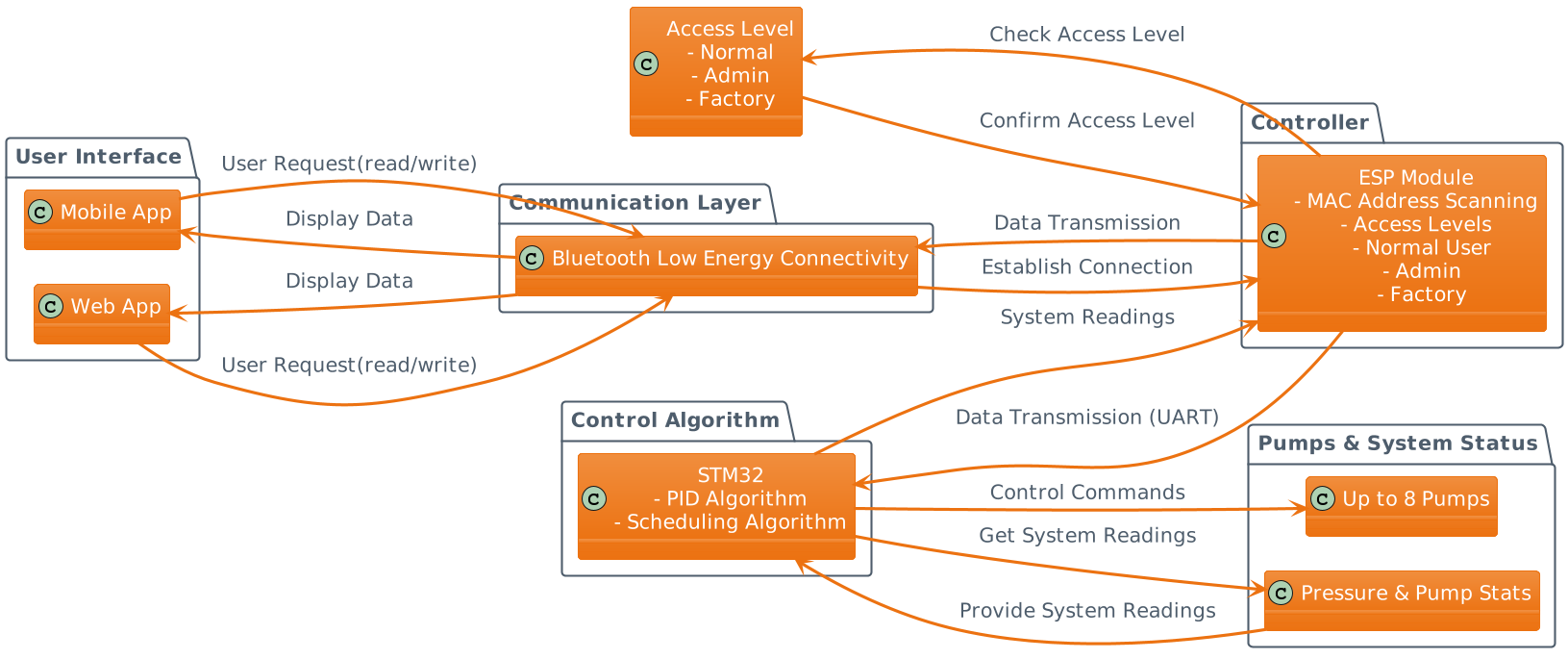
**Bluetooth Low Energy (BLE):** Facilitates wireless connectivity for user commands and data visualization.

**QR Code Scanning:** Provides a method for device identification and connection establishment using bar code scanning that connect to the board directly through Bluetooth Low Energy.

**ESP Module:** Acts as the central communication hub, managing access levels and data transmission.

**Control Algorithm:**

**STM32 Microcontroller:** Employs a PID algorithm for precise control of pressure levels, alongside a scheduling algorithm to balance pump operation hours. And this is Up to 8 Pumps so the system can manage multiple pumps, dynamically controlling their operation to maintain optimal pressure.



**Workflow:**

**User Access:**

Users initiate the system interaction through the mobile, web, or desktop application, requesting a connection to the control board via BLE or MAC address scanning.

**Authentication:**

To engage with the system, users are required to authenticate themselves, ensuring secure access.

**Access Levels:**

**General Access:** Users receive basic system readings and operational status.

**Admin Access:** Users obtain advanced data insights and control options.

**Factory Level:** Users are granted comprehensive data access and full control capabilities.

**Data Transmission:**Upon establishing a connection, the ESP module communicates system readings to the user interface through the STM32 micro controller via UART.

**Control and Efficiency Algorithms:**

The STM32 executes a PID algorithm to regulate pump activity, maintaining the set pressure point. An internal scheduling algorithm optimizes pump efficiency by evenly distributing operational hours.

**User Interaction:**

Depending on their access level, users can either monitor system Parameters or actively control the system through the app's interface.

The system's layout is designed to provide a seamless experience from user access to pump management, emphasizing security, control precision, and operational efficiency. Each component and their interactions are integral to the system's robust performance and user satisfaction.

**code structure and modules:**

**ADC Module Documentation for STM32 Microcontroller**

**Overview**

The ADC module consists of adc.h and adc.c files, which interface with the STM32 microcontroller's ADC peripherals. They provide initialization, configuration, and a custom calibration routine essential for the Advanced Multi-Pump Control System.

**adc.h: Header File**

**Header Guard:** Prevents duplicate inclusions.

**C Linkage:** Ensures compatibility with C++ compilers.

**Includes:** References the main project header for shared definitions.

**External ADC Handlers:** Declares hadc1 and hadc2 for ADC operations.

**Function ParametersADC Initialization:** Functions MX\_ADC1\_Init() and MX\_ADC2\_Init() prepare ADC1 and ADC2.

**Calibration Function:** void CalibrateADC(); is declared for the ADC calibration process.

**adc.c:** Source File

CalibrateADC Implementation:

cCopy code

void CalibrateADC() {

// Calibration logic using ADC readings and coefficients coffA, coffB

}

**Global Variables:** Defines hadc1, hadc2, and DMA handlers for ADC data transfer.

**Initialization:** Implements the functions that initialize ADC1 and ADC2 with specified settings.

**Resource Management Functions:**

HAL\_ADC\_MspInit(): Prepares low-level hardware resources.

HAL\_ADC\_MspDeInit(): Releases hardware resources when not needed.

**ADC Configuration:** Sets up ADC channels, sampling times, and configurations.

**Additional Information**

**Code Structure:** Tailored for the STM32 using HAL library for abstraction.

User Code Protection: Custom user sections are safeguarded against overwrites during regeneration.

Conclusion

This concise documentation outlines the ADC module within the STM32 firmware, highlighting the calibration function CalibrateADC() and the structure for initializing and managing ADC peripherals. It is designed for developers needing to understand or modify the ADC behavior in the pump control system.

**backup\_registers\_operations.c**

**Module Information**

**Description:** This module provides functions for operating on backup registers as our micro controller contains 32 backup register that maintain their values even if the micro controller restart of loose power ,and this is because of the battery installed on the board. this module includes functionalities to read, write, store, restore, and delete various system and pump data from backup registers.

**Constants**

MAX\_NUM\_PUMPS: The maximum number of pumps supported.

NUM\_REGISTERS\_PER\_PUMP: Number of registers allocated per pump.

PUMPS\_FLAG\_REGISTER\_INDEX: Index of the register holding the flag for pumps data.

FLAG\_VALUE:Standard flag value indicating valid data.

DELETED\_FLAG\_VALUE: Flag value indicating data deletion.

FACTORY\_FLAG\_REGISTER\_INDEX: Index of the register for factory settings flag.

INTERNAL\_SYSTEM\_DATA\_FLAG\_INDEX: Index of the register for internal system data flag.

ALARM\_SAVING\_REGISTER: Index of the register for alarm settings.

**Function Descriptions**

**write\_backup\_register**

**Prototype:** void write\_backup\_register(uint32\_t register\_index, uint32\_t value);

**Purpose:** Writes a value to a specified backup register of the stm32 backup registers.

**Parameters:**

register\_index: Index of the backup register to write to.

value: The value to be written to the register.

**Return Value:** None.

**read\_backup\_register**

**Prototype:** uint32\_t read\_backup\_register(uint32\_t register\_index);

**Purpose:** Reads a value from a specified backup register.

**Parameters:**

register\_index: Index of the backup register to read from.

**Return Value:** The value read from the specified register.

**write\_pump\_data\_to\_backup\_registers**

**Prototype:** void write\_pump\_data\_to\_backup\_registers(void);

**Purpose:** Writes the system pumps data to the backup registers.

**Parameters:** None.

**Return Value:** None.

**delete\_pump\_data\_from\_backup\_registers**

**Prototype:** void delete\_pump\_data\_from\_backup\_registers(void);

**Purpose:** Deletes the pump data from the backup registers.

**Parameters:** None.

**Return Value:** None.

**restore\_pump\_data**

**Prototype:** void restore\_pump\_data(void);

**Purpose:** Restores the pumps data from the backup registers.

**Parameters:** None.

**Return Value:** None.

**store\_factory\_page\_data**

**Prototype:** void store\_factory\_page\_data(void);

**Purpose:** Stores the data of the application's factory page to the backup registers.

**Parameters:** None.

**Return Value:** None.

**delete\_factory\_data\_from\_backup\_registers**

**Prototype:** void delete\_factory\_data\_from\_backup\_registers(void);

**Purpose:** Deletes the factory settings data from the backup registers.

**Parameters:** None.

**Return Value:** None.

**restore\_factory\_page\_data**

**Prototype:** void restore\_factory\_page\_data(void);

**Purpose:** Restores the factory page data from the backup registers after a system reboot.

**Parameters:** None.

**Return Value:** None.

**store\_internal\_system\_data**

**Prototype:** void store\_internal\_system\_data(void);

**Purpose**: Stores the current PID data to the backup registers.

**Parameters**: None.

**Return Value:** None.

**restore\_internal\_system\_data**

**Prototype:** void restore\_internal\_system\_data(void);

**Purpose:** Restores the current PID data from the backup registers.

**Parameters:** None.

**Return Value:** None.

**delete\_internal\_system\_data\_from\_backup\_domain\_registers**

**Prototype:** void delete\_internal\_system\_data\_from\_backup\_domain\_registers(void);

**Purpose:** Deletes the current PID state data from the backup registers.

**Parameters:** None.

**Return Value:** None.

**store\_alarm\_data**

**Prototype:** void store\_alarm\_data(void);

**Purpose:** Stores the alarm configuration data to the backup registers.

**Parameters:** None.

**Return Value:** None.

**restore\_alarm\_data**

**Prototype:** void restore\_alarm\_data(void);

**Purpose:** Restores the alarm configuration data from the backup registers.

**Parameters:** None.

**Return Value:** None.

**Additional Notes**

This module interacts with hardware-specific functions (e.g., HAL\_RTCEx\_BKUPWrite, HAL\_RTCEx\_BKUPRead) and assumes proper initialization of relevant peripherals (e.g., RTC, backup power domain).

It's designed for a specific embedded environment and requires an appropriate hardware abstraction layer.

**VFD.h**

Overview

**File Name:** VFD.h

**Description:** This header file contains the definitions and addresses for various Variable Frequency Drives (VFDs) that are compatible with the system. Specifically, it provides details for the Danfoss 101 series VFD.

**Contents:**

Danfoss 101 Addresses (danfossAddress Enum):

This enumeration defines the addresses for different Parameters of the Danfoss 101 VFD.

Includes addresses for reading voltage, speed, current, and frequency.

Also defines addresses for control word and speed settings.

Example: danfoss101Voltage\_V\_Read is the address to read voltage (in Volts).

Danfoss 101 Control Words (danfossControlWord Enum):

This enumeration specifies control words used for operating the Danfoss 101 VFD.

Includes commands for stopping, running forward, running reverse, and resetting the VFD.

Example: danfoss101\_runForward is the control word to run the VFD in a forward direction.

**Usage**

The addresses and control words defined in this file are used to interact with the Danfoss 101 VFD through a communication interface (Modbus).

The enums provide a clear and readable way to reference specific functions and Parameters of the VFD, enhancing code readability and maintainability.

**Additional Notes**

The VFD.h file is specifically tailored for the Danfoss 101 series and may need to be extended or modified for compatibility with other VFD models or brands.

The inclusion of these addresses and control words suggests that the broader system includes functionality for VFD control, as part of an industrial automation or control system.

**ESP\_uart\_config.c**

**Module Information**

**Description:** This module provides functions for handling UART communication between esp32 and stm32. It includes functionalities for receiving data, checking its integrity, and preparing data for transmission.

Functions

**CheckData\_received**

**Prototype**: returned\_ST\_Data CheckData\_received(unsigned char\* Data);

**Purpose:** Parses a received data string and extracts 'value' and 'identifier' based on a CRC-like format. Validates the integrity of the data by checking the format, length, and initial characters.

**Parameters**:

Data: Pointer to the received data string.

Return Value: A returned\_ST\_Data struct containing the extracted 'value' and 'identifier'. Returns default values ('0') for both fields if the data format does not match.

**Usage:** Call this function with a string received over UART to parse and validate it.

**Example:**

returned\_ST\_Data receivedData = CheckData\_received(uartReceivedData);

**Preconditions:**

The Data pointer must not be NULL and should point to a null-terminated string.

**ReceiveData**

**Prototype**: returned\_ST\_Data ReceiveData(void);

**Purpose:** Initializes data structures and processes received USART data. Resets the flag indicating new data availability after processing.

**Return Value**: A returned\_ST\_Data struct with the processed data after applying the CRC check.

**Usage:** Invoke this function to receive and process data from UART. Example:

returned\_ST\_Data data = ReceiveData();

**Preconditions:**

USART data buffer (getBuffer) should be ready with the received data.

The usart1HaveNewData flag should be set to indicate new data is available.

**PrepareData**

**Prototype**: char\* PrepareData(char\* identifier, char\* value);

**Purpose:** Formats the provided 'value' and 'identifier' into a specific string format for transmission over USART.

**Parameters**:

identifier: Pointer to the identifier string.

Value: Pointer to the value string.

**Return Value:** : Pointer to a static buffer containing the formatted data string.

**Usage:** Use this function to prepare data for transmission over UART. Example:

char\* formattedData = PrepareData("sensor1", "dataValue");

// Send formattedData over UART

**Preconditions:**

'identifier' and 'value' should not be NULL and must be null-terminated strings.

**Additional Notes**

The functions are designed to work with specific data formats and UART configurations. Ensure that the UART is correctly configured before using these functions.

The module assumes a specific CRC-like format for data integrity checks. This format must be consistent across the system for proper communication.

The PrepareData function uses a static buffer, which means its content will be overwritten on each call. Copy the data elsewhere if needed before the next call to PrepareData.

**Example Usage Scenario**

// Receiving and checking data from UART

returned\_ST\_Data receivedData = ReceiveData();

if (strcmp(receivedData.value, "0") != 0) {

// Process receivedData

}

// Preparing and sending data over UART

char\* dataToSend = PrepareData("sensorID", "45");

// Code to send dataToSend over UART

**Module Information**

**Description:** This module manages operations related to an external flash memory connected to an STM microcontroller via QSPI. It provides functionality to store, read, and manage various types of records, including alarms, scheduler records, and weekly records.

**Constants**

DATED\_SCHEDULAR\_RECORDS\_START\_ADDRESS: Start address in the external flash for dated scheduler records.

WEEKLY\_SCHEDULAR\_RECORDS\_START\_ADDRESS: Start address for weekly scheduler records.

ALARMS\_START\_ADDRESS: Start address for alarm records.

MAX\_RECORDS, MAX\_DAYS, DATED\_RECORD\_SIZE, etc.: Various Constants defining limits and sizes for records.

**Function Descriptions**

**store\_alarm**

**Prototype**: HAL\_StatusTypeDef store\_alarm(alarmType alarm, uint32\_t baseAddress);

**Purpose:** Stores an alarm event in the external flash memory.

**Parameters**:

alarm: Type of the alarm to store.

Baseaddress: Base address in the flash memory for storing alarm data.

**Return Value:** : HAL status indicating success or failure of the operation.

**read\_alarm\_records**

**Prototype**: HAL\_StatusTypeDef read\_alarm\_records(uint32\_t baseAddress, char (\*alarmsArray)[RECORD\_SIZE], uint32\_t \*numRecords);

**Purpose:** Reads alarm records from the external flash memory into a provided array.

**Parameters**:

Baseaddress: The starting address for reading.

alarmsArray: Array to store the read alarm records.

numRecords: Number of records to read.

**Return Value:** : HAL status indicating success or failure of the operation.

**compare\_weekly\_rtc**

**Prototype**: void compare\_weekly\_rtc(void);

**Purpose:** Compares the current RTC time with weekly schedules stored in the flash memory.

**Return Value:** : None. Updates global variables with matched record indices.

**compare\_rtc**

**Prototype**: void compare\_rtc(void);

**Purpose:** Compares the current RTC time with dated schedules stored in the flash memory.

**Return Value:** : None. Updates global variables with matched record indices.

**ExtractWeeklyRecords**

**Prototype**: void ExtractWeeklyRecords(QSPI\_HandleTypeDef \*Ctx);

**Purpose:** Extracts weekly records from the external flash memory and stores them in an array.

**Parameters**:

Ctx: Pointer to the QSPI handle.

**Return Value:** : None.

**ExtractRecords**

**Prototype**: void ExtractRecords(QSPI\_HandleTypeDef \*Ctx);

**Purpose:** Extracts dated records from the external flash memory and stores them in an array.

**Parameters**:

Ctx: Pointer to the QSPI handle.

**Return Value:** : None.

**sendMatchedIndices**

**Prototype**: void sendMatchedIndices(int \*matchedIndices, int \*matchedCount, const char \*id);

**Purpose:** Sends the indices of matched records over UART for the application to display them .

**Parameters**:

matchedIndices: Array of matched record indices.

matchedCount: Number of matched records.

id: Identifier to be sent along with the indices.

**Return Value:** : None.

**PrintValidRecords**

**Prototype**: void PrintValidRecords(void);

**Purpose:** Prints details of valid records stored in the system.

**Return Value:** : None.

**PrintWeeklyValidRecords**

**Prototype**: void PrintWeeklyValidRecords(void);

**Purpose:** Prints details of valid weekly records stored in the system.

**Return Value:** : None.

**DeleteRecordByIndex**

**Prototype**: void DeleteRecordByIndex(QSPI\_HandleTypeDef \*hqspi, int record\_index, uint32\_t base\_address, uint32\_t memory\_record\_size);

**Purpose:** Deletes a record from the external flash memory by overwriting its index with zeros.

**Parameters**:

hqspi: Pointer to the QSPI handle.

record\_index: Index of the record to delete.

base\_address: Base address in the memory for the records section.

memory\_record\_size: Size of each record in the memory.

**Return Value:** : None.

**externalFlashWrite**

**Prototype**: void externalFlashWrite(uint32\_t Baseaddress, char \*dataToWrite, uint32\_t \*index, uint8\_t writeMemorySize, uint8\_t saveRegisterNumber);

**Purpose:** Writes a record to the external flash memory and verifies its correctness.

**Parameters**:

Baseaddress: Base address in memory to start writing to.

dataToWrite: Data to be written.

index: Index where the data will be written.

writeMemorySize: Size of the memory to write.

saveRegisterNumber: Register number to update after successful write.

**Return Value:** : None.

**Additional Notes**

This module requires proper initialization and configuration of the QSPI interface and external flash memory.

It is designed to work with specific flash memory models ( S25FL256S) and assumes a certain memory layout for storing records.

The module is part of a larger system managing scheduler and alarm data, and it interacts closely with other system components like RTC and backup registers.

**RTC Module**

File Information

File: rtc.h

**Description:** This file contains the declarations for RTC (Real Time Clock) functionalities. It includes functions to initialize the RTC, set and get the current date and time. The file is part of a system that relies on accurate timekeeping.

**Global Variables**

RTC\_HandleTypeDef hrtc: Handler for RTC, used for RTC operations.

RTC\_TimeTypeDef sTime: Global variable to hold the current time.

RTC\_DateTypeDef sDate: Global variable to hold the current date.

Function Descriptions

MX\_RTC\_Init

**Prototype**: void MX\_RTC\_Init(void);

**Purpose:** Initializes the RTC peripheral with necessary configurations.

**Usage:** This function should be called during system initialization to set up the RTC.

**Get\_RTC\_Values**

**Prototype**: void Get\_RTC\_Values(void);

**Purpose:** Reads the current date and time from the RTC and updates the global variables sTime and sDate.

**Usage:** Use this function to obtain the current time and date from the RTC.

**Affected Globals:** Updates sTime and sDate with the current RTC values.

**Set\_RTC\_Values**

**Prototype**: void Set\_RTC\_Values(uint16\_t year, uint8\_t month, uint8\_t day, uint8\_t hour, uint8\_t minute, uint8\_t weekDay);

**Purpose:** Updates the RTC with new date and time values.

**Parameters**:

year: Year to set (e.g., 2023).

month: Month to set (1 - 12).

day: Day of the month to set (1 - 31).

hour: Hour to set (0 - 23).

minute: Minute to set (0 - 59).

weekDay: Day of the week (1 - 7, where 1 is Monday, 7 is Sunday).

**Usage:** Call this function to change the RTC to a new date and time. Useful for initial setup or updating the RTC.

Additional Information

The functions in this module interact directly with the hardware RTC of the STM32 microcontroller. They are essential for applications where time tracking is crucial.

Error handling is performed by the Error\_Handler() function, which should be defined elsewhere in the application to handle errors appropriately.

It is assumed that the RTC peripheral is correctly configured in the hardware settings, and the hrtc handler is properly initialized before these functions are used.

**S25FL256S Module**

File Information

File: S25FL256S.h

**Description:** This file contains Function Parameters for the S25FL256S QSPI Flash memory. It includes commands for reading, writing, erasing, and other memory operations.

**Function Descriptions**

**S25FL256S\_GetFlashInfo**

**Prototype**: int32\_t S25FL256S\_GetFlashInfo(S25FL256S\_Info\_t \*pInfo);

**Purpose:** Fetches basic information about the S25FL256S Flash memory like size, sector size, number of sectors, page size, and number of pages.

**Parameters**:

pInfo: Pointer to the structure where the flash information will be stored.

**Return:** Status of the operation.

**S25FL256S\_WriteEnable**

**Prototype**: int32\_t S25FL256S\_WriteEnable(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode);

**Purpose:** Enables writing operations on the flash memory.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

**Return:** Status of the operation.

**S25FL256S\_BlockErase**

**Prototype**: int32\_t S25FL256S\_BlockErase(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode, uint32\_t BlockAddress, S25FL256S\_Erase\_t BlockSize);

**Purpose:** Erases a specific block of the flash memory.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

BlockAddress: Address of the block to erase.

BlockSize: Size of the block to erase.

**Return:** Status of the operation.

**S25FL256S\_ChipErase**

**Prototype**: int32\_t S25FL256S\_ChipErase(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode);

**Purpose:** Erases the entire flash chip.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

**Return:** Status of the operation.

**S25FL256S\_PageProgram**

**Prototype**: int32\_t S25FL256S\_PageProgram(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode, uint8\_t \*pData, uint32\_t WriteAddr, uint32\_t Size);

**Purpose:** Programs data into a page of the flash memory.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

pData: Pointer to the data to be written.

WriteAddr: Start address for writing data.

Size: Size of the data to write.

**Return:** Status of the operation.

**S25FL256S\_ReadSTR**

**Prototype**: int32\_t S25FL256S\_ReadSTR(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode, uint8\_t \*pData, uint32\_t ReadAddr, uint32\_t Size);

**Purpose:** Reads a specified amount of data from the flash memory in STR mode.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

pData: Pointer to buffer where the data will be stored.

ReadAddr: Start address for reading data.

Size: Size of the data to read.

**Return:** Status of the operation.

**S25FL256S\_ReadStatusRegister**

**Prototype**: int32\_t S25FL256S\_ReadStatusRegister(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode, uint8\_t \*Value);

**Purpose:** Reads the status register of the flash memory.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

Value: Pointer to store the read value.

**Return:** Status of the operation.

**S25FL256S\_ReadConfigRegister**

**Prototype**: int32\_t S25FL256S\_ReadConfigRegister(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode, uint8\_t \*Value);

**Purpose:** Reads the configuration register of the flash memory.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

Value: Pointer to store the read value.

**Return:** Status of the operation.

**S25FL256S\_EnableMemoryMappedModeSTR**

**Prototype**: int32\_t S25FL256S\_EnableMemoryMappedModeSTR(QSPI\_HandleTypeDef \*Ctx, S25FL256S\_Interface\_t Mode);

**Purpose:** Enables the memory-mapped mode for easy access to the flash memory.

**Parameters**:

Ctx: QSPI handle.

Mode:Interface mode for the flash operation.

**Return:** Status of the operation.

**Additional Functions**

Several other functions are provided for specific operations like write disable, reading ID, resetting memory, entering deep power down, program/erase resume and suspend, reading SFDP, etc.

Additional Information

The functions in this module provide a comprehensive interface for interacting with the S25FL256S Flash memory over a QSPI interface.

The module abstracts complex flash operations and provides a simplified API for reading, writing, erasing, and other operations.

Proper initialization of the QSPI interface and handling of the QSPI handle is crucial for the correct operation of these functions.

**Modbus Module**

**Description:** This module implements Modbus communication protocol to interface with various types of Variable Frequency Drives (VFDs). It supports Modbus RTU over serial (USART) and TCP/IP for network communication. The module includes functionalities for reading and writing VFD Parameters, controlling VFD operations, and handling communication errors.

**Constants:**

USART\_HW: Identifier for USART hardware interface.

USB\_CDC\_HW: Identifier for USB CDC hardware interface.

TCP\_HW: Identifier for TCP/IP network interface.

USART\_HW\_DMA: Identifier for USART with DMA hardware interface.

MB\_SLAVE: Identifier for Modbus slave mode.

MB\_MASTER: Identifier for Modbus master mode.

**Data Structures:**

modbusRingBuffer\_t: Structure for Modbus ring buffer management.

MB\_FC: Enum for Modbus function codes.

modbus\_t: Structure for Modbus query management.

modbusHandler\_t: Main structure for handling Modbus communication.

**Function Descriptions:**

void ModbusInit(modbusHandler\_t \*modH): Initializes Modbus settings based on the specified handler configuration.

void ModbusStart(modbusHandler\_t \*modH): Starts Modbus communication based on the initialized settings.

void ModbusStartCDC(modbusHandler\_t \*modH): Starts USB CDC-based Modbus communication.

void ModbusQuery(modbusHandler\_t \*modH, modbus\_t telegram): Sends a Modbus query to a slave device.

void ModbusQueryInject(modbusHandler\_t \*modH, modbus\_t telegram): Injects a Modbus query directly into the query queue.

void StartTaskModbusSlave(void \*argument): Task function for Modbus slave operation.

void StartTaskModbusMaster(void \*argument): Task function for Modbus master operation.

uint16\_t calcCRC(uint8\_t \*Buffer, uint8\_t u8length): Calculates CRC for Modbus packets.

void ModbusCloseConn(struct netconn \*conn): Closes a TCP connection.

void ModbusCloseConnNull(modbusHandler\_t \*modH): Closes a TCP connection and resets the handler.

**Additional Notes**:

The module supports various hardware configurations and requires appropriate hardware abstraction layer (HAL) for different microcontrollers.

It is designed for robust and efficient communication with VFDs, providing both master and slave functionalities.

Proper initialization of peripherals (e.g., USART, TCP/IP stacks) is required before using this module.

This module integrates with FreeRTOS for task management and uses its features like queues, semaphores, and timers for efficient operation.

Error handling is implemented to manage communication faults and to ensure reliable data exchange with VFDs.

This module provides a flexible and scalable solution for controlling VFDs in industrial applications, offering both serial and network communication options.

**Scheduler Module**

This module contains Function Parameters and structures for managing and scheduling pump operations. It includes functionalities for sorting pumps based on working hours, updating working times, and managing pump states.

**Function Descriptions**

**Sort\_Pumps\_By\_Working\_Hours\_and\_Minutes**

**Prototype**: void

Sort\_Pumps\_By\_Working\_Hours\_and\_Minutes(PumpInfo \*array, uint8\_t size);

**Purpose:** Sorts an array of PumpInfo structures in ascending order based on working hours and minutes.

**Parameters**: array: Pointer to the first element of the PumpInfo array. size: Number of elements in the array.

**Return:** None.

**Sort\_Pumps\_By\_changeOverTime**

**Prototype**: void

Sort\_Pumps\_By\_changeOverTime(PumpInfo \*array, uint8\_t size, uint8\_t changeOverTime);

**Purpose:** Sorts pumps based on calculated change-over time units in ascending order, moving the pump with the highest units to the end of the array.

**Parameters**: array: Pointer to the first element of the PumpInfo array. size: Number of elements in the array. changeOverTime: Change-over time in minutes used for calculation.

**Return:** None.

**Update\_Working\_Time**

**Prototype**: void

Update\_Working\_Time(uint8\_t id, CountAction action);

**Purpose:** Updates the working time of a specific pump based on its ID and the specified action (START\_COUNTING, STOP\_COUNTING, or UPDATE\_TIME).

**Parameters**: id: ID of the pump to update.

**action:** Action to perform (START\_COUNTING, STOP\_COUNTING, or UPDATE\_TIME).

**Return:** None.

**Sort\_Pumps\_By\_Working\_Hours**

**Prototype**: void

Sort\_Pumps\_By\_Working\_Hours(PumpInfo \*array, uint8\_t size);

**Purpose:** Sorts an array of PumpInfo structures based solely on working hours in ascending order.

**Parameters**: array: Pointer to the first element of the PumpInfo array. size: Number of elements in the array.

**Return:** None.

**Additional Functions**

The module includes additional internal utility functions for calculating elapsed days, finding pump by ID, and updating pump information based on real-time clock data.

**Additional Information**

The functions in this module provide essential utilities for managing pump operations, particularly for scheduling and tracking working hours. Proper handling of pump IDs and understanding the pump state logic is crucial for the correct operation of these functions.

**System Application and Tasks**

**UART1\_task Function**

Overview

**Function Name: UART1\_task**

**Purpose:** This FreeRTOS Task is implemented as a thread to manage UART (Universal Asynchronous Receiver-Transmitter) communications in an embedded system, specifically through UART1. It is primarily responsible for receiving, parsing, and acting upon data transmitted from external devices, such as an ESP32 module, to the main board microcontroller (STM32H750).

Key Functionalities

**UART Data Reception:**

Initiates UART reception in interrupt mode to handle asynchronous data reception.

Continuously checks for new data received from the connected device.

**Data Parsing and Processing:**

Upon receiving new data, it parses this data to extract relevant commands or settings.

Handles various data types, converting strings to appropriate formats (integer, float) for system use.

**System Parameter Updates:**

Updates system Parameters based on the received data.

**This includes settings for:**

RGB LED color intensity.

System setpoints like pressure thresholds.

Operational modes and pump speeds.

Real-time clock adjustments.

Scheduler and alarm configurations.

**Conditional Logic and Actions:**

Implements conditional checks to determine the type of received data and performs corresponding actions.

Includes logic to handle manual and automated operational modes, system alarms, and scheduling.

**Error Handling and Data Validation:**

Incorporates error handling for data parsing and validation to ensure valid data reception.

**Usage in the System**

The UART1\_task is a critical component in systems where real-time communication with ESP32 and data reception from it .

**Additional Implementation Details**

The function uses osDelay within its loop, indicating integration with an RTOS and ensuring cooperative multitasking.

The task's design implies a high dependency on accurate and timely data reception from external sources, highlighting its role in the broader context of system operation and management.

The use of string comparison (strcmp) and data type conversion functions (like atoi and atof) suggests a text-based communication protocol, which might require additional consideration for efficiency and robustness in data handling.

**ADC\_4\_20mApolling Task**

**Overview**

Function Name: ADC\_4\_20mApolling

**Purpose:** This function is a dedicated thread for polling and processing data from 4-20mA industrial sensors using ADC1. It is designed to continuously monitor sensor outputs, perform calibration, and calculate current and pressure values.

**Key Functionalities**

**Continuous ADC Polling:**

Regularly initiates and polls ADC conversions to acquire digital values from analog 4-20mA signals.

**Sensor Error Detection:**

Implements logic to detect sensor errors or disconnections based on predefined ADC value thresholds.

**Calibration Data Acquisition:**

Captures calibration data sent from an external interface (like an app) and stores it for further calculations.

**Calibration and Current Calculation:**

Processes calibration data to calculate coefficients (coffA and coffB).

Computes calibrated current values using these coefficients.

**Pressure Calculation:**

Derives the current pressure value from the calibrated current, considering the sensor's operating range.

**Task Synchronization:**

Uses osDelay for task synchronization and to maintain a controlled polling rate.

**Usage in the System**

The ADC\_4\_20mApolling function is essential in systems where continuous and accurate monitoring of physical Parameters (like pressure) is crucial, especially in industrial and automation environments.

It ensures real-time data acquisition and processing, contributing to the system's overall responsiveness and control accuracy.

**Additional Notes**

This function assumes the availability of a properly configured ADC peripheral (hadc1) and relies on HAL library functions for ADC operations.

The calculation logic and thresholds should be tailored to the specific sensor and application requirements.

Integration with a real-time operating system (RTOS) suggests this function is part of a larger, multitasking embedded application.

**UART\_Send Task for multi drive system**

**Function Name:** UART\_Send

**Purpose:** UART\_Send is a dedicated thread for transmitting various system Parameters and states via UART to ESP32. This function is essential for ensuring real-time communication and feedback to external interfaces such as user applications.

Periodic Data Transmission:

Regularly sends data like current pressure, system errors, and configuration settings over UART.

Includes conditional logic to transmit relevant data based on the current system state or user interface activity.

**System Parameters and Error State Transmission:**

Formats and transmits a wide range of data including operational Parameters, error states, calibration data, and RTC values.

Utilizes sprintf for data formatting and HAL\_UART\_Transmit for UART transmission.

**User Interface Interaction Handling:**

Responds to specific requests from external interfaces (e.g., a mobile app) for system data, such as factory settings, alarm history, and real-time clock information.

Manages data transmission based on user interface page, ensuring relevant data is sent for the specific context.

**Operational Mode Status Communication:**

Sends the current operational mode (manual or automatic) for appropriate display and interaction on the user interface.

**Task Synchronization and Rate Control:**

Employs osDelay to control the rate of data transmission, ensuring consistent communication without overwhelming the UART channel.

**Usage in the System**

The UART\_Send function is crucial for systems requiring continuous feedback and interaction with external interfaces, particularly in embedded control systems with user interfaces.

It facilitates real-time monitoring, system configuration, and user interaction by providing necessary system data to external devices.

**Additional Notes**

This function assumes the availability of a properly configured UART peripheral (huart1) and depends on the HAL library for UART operations.

The data transmission logic should be tailored to match the specific requirements of the external interface and the overall system architecture.

Integration with an RTOS implies that this function is part of a larger, multitasking embedded application, where real-time data transmission is a key requirement.

**PIDControl Task For multiDrive System**

**Purpose:** The PIDControl function is designed to control multiple Variable Frequency Drives (VFDs) in an industrial setting. It regulates the speed of these drives based on a PID (Proportional, Integral, Derivative) algorithm in automatic mode, and allows for manual control adjustments based on user inputs.

**Key Functionalities**

PID Algorithm in Automatic Mode:

Implements a PID control loop to adjust the speed of VFDs based on process control Parameters such as pressure.

Calculates and applies the appropriate control action to each VFD to maintain the desired setpoint.

**Manual Mode Operation:**

Allows for direct user control over individual VFD speeds.

Adjusts VFD speeds based on manual input commands, bypassing the automatic PID control.

**VFD Communication via Modbus:**

Utilizes the Modbus protocol to communicate with each VFD, sending speed control commands and receiving operational data.

Ensures synchronization and timely response in the control of multiple VFDs.

**System Parameter Monitoring and Updating:**

Monitors key Parameters like current pressure and VFD operational status.

Updates system settings such as calibration data, alarm states, and operational modes.

**Data Storage and Retrieval:**

Handles storage of system Parameters for continuity in operations, especially during system resets.

Retrieves stored data to maintain operational consistency.

**Usage in the System**

The PIDControl function is essential in multi-VFD systems for achieving precise and coordinated control of multiple drives.

It is particularly relevant in industrial automation, process control, and applications where multiple motors or pumps need to operate in unison or require precise control  
to maintain certain set point of pressure .

‘

**Additional Notes**

Integration with an RTOS suggests the task operates as part of a larger system, likely involving real-time monitoring and control.

The control algorithm and operational logic should be tailored to the specific requirements of the application and the characteristics of the VFDs in use.

**PIDControl Task in a Single VFD System**

Overview

**Function Name:** PIDControl

**Purpose:** The PIDControl function is designed for a single VFD system controlling a base load pump, with additional pumps connected directly for full efficiency operation. It regulates the base load pump's speed using a PID algorithm and manages the on/off state of other pumps to maintain the desired system setpoint.

**Key Functionalities**

**Base Load Pump VFD Control:**

Implements a PID control algorithm to adjust the speed of the base load pump VFD, ensuring precise pressure control.

Modulates the VFD speed based on real-time pressure measurements and setpoints.

**Direct Control of Additional Pumps:**

Manages the direct on/off control of additional pumps, operating them at full efficiency.

Coordinates the operation of these pumps with the base load pump for optimal system performance.

**System Monitoring and Parameter Updates:**

Continuously monitors system Parameters like current pressure and pump statuses.

Updates operational settings and responds to changes in system demands or user inputs.

**Pump State Management and Synchronization:**

Manages the states of all pumps in the system, ensuring synchronized operation to meet process requirements.

Applies changes in pump states based on the PID control output and system conditions.

**Scheduler and Manual Mode Handling:**

Responds to scheduler inputs for pump arrangement based on working hours.

Allows for manual control adjustments in specific operational modes.

**Usage in the System**

The PIDControl function is essential in systems with a single VFD controlling a base load pump, alongside multiple direct online pumps.

It is particularly relevant in industrial automation, process control, and applications where precise pressure control and efficient pump operation are required.

**Additional Notes**

Integration with an RTOS suggests the task operates as part of a larger, multitasking system, likely involving real-time monitoring and control.

The control algorithm and operational logic should be tailored to the specific requirements of the application and the characteristics of the pumps and VFD in use.

**PIDControl Task in a Direct Online Pump System**

Overview

**Function Name: PIDControl**

**Purpose:** The PIDControl function is designed for systems utilizing direct online pumps, controlling each pump's on/off state based on specific pressure set points. It ensures efficient system operation by activating pumps at designated pressure levels to maintain or reach the desired set point.

**Key Functionalities**

**Pressure-Based Pump Control:**

Activates and deactivates pumps based on the difference between current pressure and predefined set points.

Uses error calculation (difference between set point and current pressure) to determine pump states.

**Set Point Management:**

Manages a series of pressure set points, each corresponding to the activation of an additional pump.

Ensures that pumps are activated or deactivated at the right moments to maintain system pressure.

**Prioritization of Pump Activation:**

Prioritizes pumps with the lowest working hours for activation to ensure even usage and maintenance of pump efficiency.

**System Monitoring and Operational Adjustment:**

Continuously monitors the current system pressure and adjusts the operation of pumps accordingly.

Adapts to changing pressure conditions to maintain the desired system performance.

**Efficiency and Simplicity in Control:**

Focuses on a simple yet effective method of controlling multiple pumps, suitable for systems where direct online operation is preferred.

**Usage in the System**

The PIDControl function is essential in systems that utilize direct online pumps for pressure management, commonly seen in water distribution, HVAC, and other industrial applications.

It provides an efficient and straightforward method of maintaining desired pressure levels using multiple pumps.

**Additional Notes**

Integration with an RTOS implies that the task is part of a larger, multitasking control system, with real-time monitoring and adjustment capabilities.

The control logic should be customized based on the specific requirements of the application, considering factors like pump capacity, system dynamics, and operational constraints.

**Pumps\_Scheduler Task**

Overview

**Function Name:** Pumps\_Scheduler

**Purpose:** The Pumps\_Scheduler function is designed for systems with multiple pumps, where it is crucial to balance the operating hours among pumps. The task dynamically adjusts the operation of the pumps based on their accumulated working hours, ensuring balanced usage and system efficiency.

Key Functionalities

**Pump Working Time Monitoring:**

Regularly updates and monitors the working time of each pump in the system.

Ensures accurate tracking of pump usage for operational decision-making.

**Pump Arrangement Based on Working Hours:**

Sorts pumps in ascending order based on their working hours.

Aims to activate pumps with lower working hours and deactivate those with higher usage.

**Dynamic Pump Operation Management:**

Adjusts pump operation dynamically based on the scheduler and system conditions.

Balances pump usage to prevent overuse of individual pumps and enhance system efficiency.

**Scheduler and Manual Mode Compatibility:**

Responds to scheduler inputs and manual mode operations, adjusting pump operation accordingly.

Manages pump states based on both time-based schedules and operational demands.

**Efficiency and Longevity of Pumps:**

Aims to extend the lifespan of pumps and maintain system efficiency through balanced operation.

Reduces wear and tear on individual pumps by evenly distributing operational load.

**Usage in the System**

The Pumps\_Scheduler function is vital in multi-pump systems, particularly in industrial, environmental, and water management applications.

It ensures efficient operation and longevity of pumps by balancing their usage and operational hours.

**Additional Notes**

Integration with an RTOS suggests the task operates within a larger, multitasking system, requiring real-time response and coordination.

The logic for pump arrangement and operation should be tailored to the specific requirements of the application, considering factors like pump capacity, system dynamics, and maintenance schedules.