Heading 1	16 Bold	Calibri Light (Headings)
Heading 2	15 Bold	Calibri Light (Headings)
Heading 3...	14 Bold	Calibri Light (Headings)
Normal	12	Calibri (Body)
List Paragraph	12 Bold	Calibri (Body)

Chapter X Intro

System Analysis

system analysis is a critical process for ensuring that a system meets the needs of its stakeholders and is designed and implemented in a way that is efficient, effective, and sustainable over time.

System analysis typically involves the following steps:

a- Requirements gathering:

This involves identifying the needs and requirements of the system's stakeholders, including users, customers, and other interested parties. And this includes our SRS.

- **SRS**

  stands for "Software Requirements Specification." It is a document that outlines the functional and non-functional requirements for a software project. The SRS document serves as a blueprint for the software development team, providing a clear and detailed description of what the software is supposed to do and how it should function.

  The SRS document typically includes sections such as an introduction, functional requirements, non-functional requirements, system architecture, user interface design, system constraints, and testing requirements. The document is usually created during the initial stages of the software development lifecycle and serves as a reference point throughout the development process.

  The purpose of the SRS document is to ensure that the software development team and stakeholders are on the same page with regards to the project requirements. It helps to prevent misunderstandings and miscommunication, which can lead to costly delays and errors in the development process. By having a clear and detailed SRS document, the development team can ensure that they are building the software that the stakeholders want and need.

- **OUR SRS**

  i. **Introduction:**

  The Smart Farming System is a software system that aims to improve the efficiency and productivity of farming operations. The system uses sensors and other technologies to monitor and control various aspects of the farming process, including soil moisture, temperature, and humidity.

ii. Functional Requirements:

1. **Sensor Integration:**

   The system should be able to integrate with various sensors, including soil moisture sensors, temperature sensors, humidity sensors, and other environmental sensors.

2. **Irrigation Control:**

   The system should be able to control irrigation systems based on the data collected from the sensors. This includes automatically turning on or off irrigation systems based on soil moisture levels.

3. **Pest Detection:**

   The Smart Farming System should be able to detect pests and diseases in crops using image processing and machine learning techniques.

4. **Image Processing:**

   The system should be able to process images captured by cameras installed in the fields to identify and classify pests and diseases.

5. **Machine Learning:**

   The system should use machine learning algorithms to train on a dataset of images of healthy and diseased crops to improve the accuracy of pest and disease detection.

6. **Notification:**

   The system should notify farmers when pests or diseases are detected in their crops, including the type of pest or disease and the severity of the infestation.

iii. Non-Functional Requirements:

1. **Security:**

   The system should be designed with strong security measures to protect the data collected from the sensors and ensure that only authorized users can access the system.

2. **Reliability:**

The system should be reliable, with a high degree of uptime and minimal downtime due to hardware or software failures.

3. **Scalability:**

The system should be able to scale to accommodate additional sensors and devices as needed.

4. **Usability:**

The system should be easy to use, with a user-friendly interface that allows farmers to easily monitor and control their farming operations.

iv. **System Architecture:**

The Smart Farming System will be a cloud-based system, with a central server that collects and analyzes data from the various sensors and devices.

v. **User Interface Design:**

The system's user interface will be designed to be intuitive and easy to use, with a dashboard that displays real-time data on soil moisture, temperature, humidity, and livestock health. The interface will also include controls for managing irrigation systems, monitoring crop growth, and managing livestock.

vi. **System Constraints:**

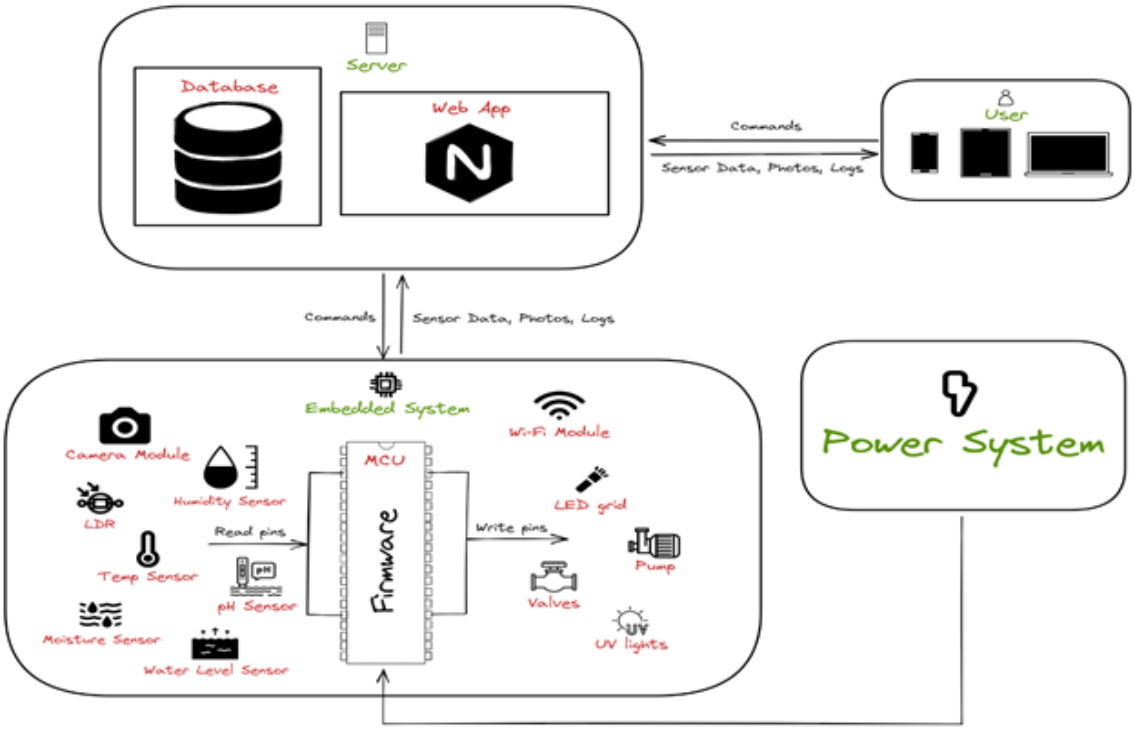
The Smart Farming System will require an internet connection to transmit data to the central server and receive commands from the mobile application. The system will also require compatible sensors and devices to be installed on the farm.

vii. **Testing Requirements:**

The Smart Farming System will undergo rigorous testing to ensure that it is reliable, secure, and meets all functional and non-functional requirements. This will include testing the system with various sensors and devices, as well as testing its scalability and security measures.

b- System modeling:

This involves creating models and diagrams to represent the system's components, functions, and interrelationships. This section will involve system block diagram to have an initial image of the whole system.



c- System design

This involves designing the system architecture, including hardware, software, and network components. This can include selecting the appropriate hardware and software platforms, designing the database schema, and designing the user interface. And this will be discussed later in the incoming chapters.

d- Implementation:

This involves building and testing the system, including writing code, configuring hardware and software components, and testing the system for functionality, performance, and security. And this will be discussed later in the incoming chapters.

e- Maintenance:

This involves maintaining and updating the system over time, including fixing bugs, adding new features, and addressing security vulnerabilities. And this will be discussed later in the incoming chapters.

Conclusion

Overall, system analysis is a critical process for ensuring that a system meets the needs of its stakeholders and is designed and implemented in a way that is efficient, effective, and sustainable over time.

Chapter x System Design

Intro

Before we start implementing the system, we first started by the whole system design as a step of project planning so that we have the complete concept of the project that make it easier for us to implement the system in a systematic way and make best use of our time to achieve our goal. in the following, we will discuss the design in more detail.

Our system consists of two major parts the hardware and software so, we can divide the system design into two categories:

2. Hardware Selection, Configuration, and Design:

The goal of hardware selection is to identify the best combination of hardware components that will meet the requirements of the system while also being cost-effective and easy to manufacture.

We start by choosing our microcontroller based the requirement of the system that we need after searching, we decided to choose a microcontroller based on ARM.

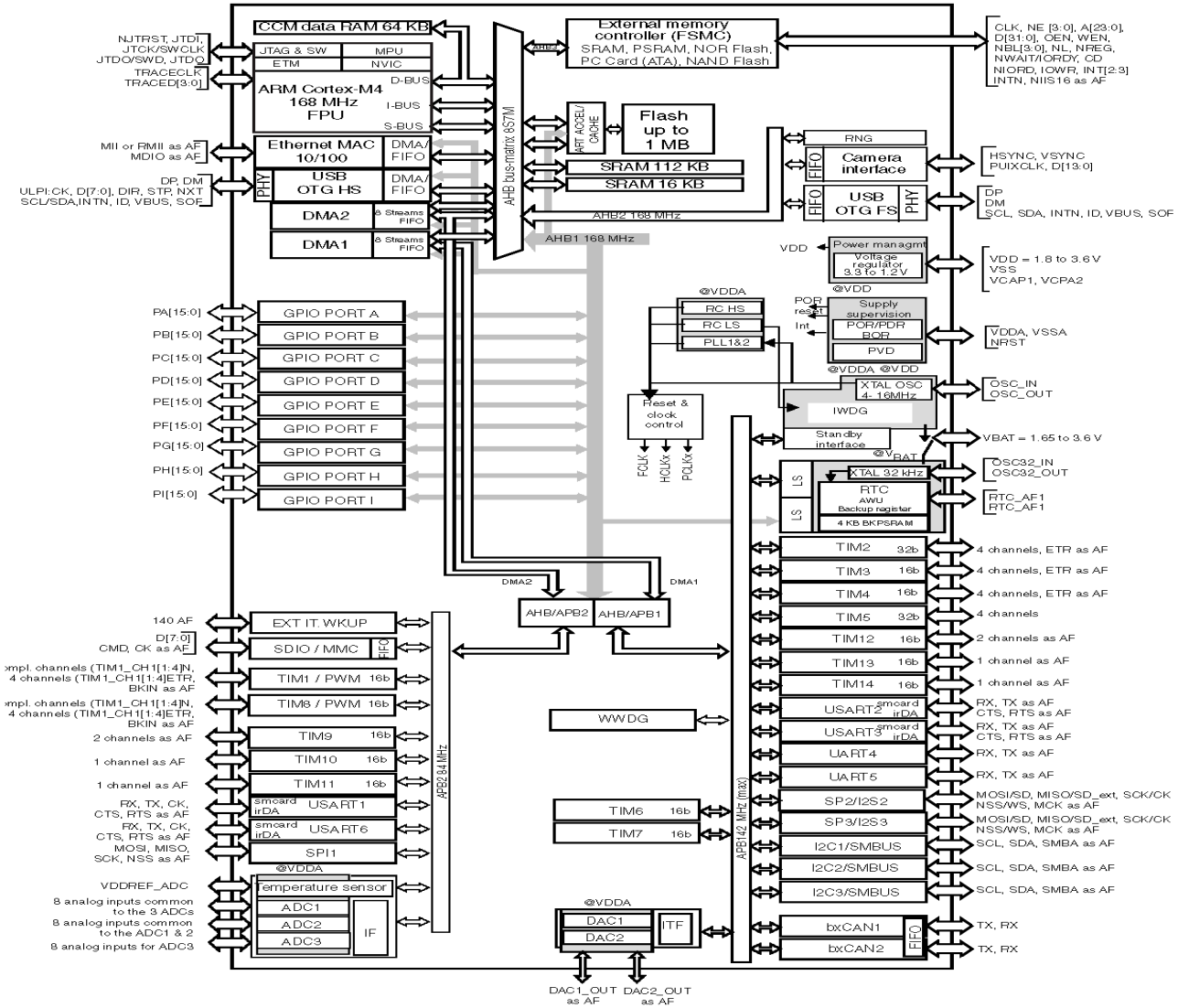
The reasons why we chose a MCU based on ARM are:

i. Minimal Cost and Power
ii. Customizable
iii. 32-bit address and data bus
iv. Ultra Low Power with High Performance
v. Very Powerful and east to use Interrupt Controller
    - Supports up to 240 external interrupt source and 15 internal.
vi. RTOS Friendly
vii. ARM provides lots of documentations, and technical references manuals.

The next step was to choose a specific MCU that has the needed requirements. After searching we find that the best choice is STM32F4xx series that produced by ST and we found NUCLEO PCB board shown in the figure below that is easy to use and powerful.

The specifications of NUCLEO are that it has:

- ✓ Core: ARM Cortex-M4F with FPU
- ✓ Clock frequency: up to 180 MHz
- ✓ Flash memory: 512 KB
- ✓ SRAM: 128 KB
- ✓ Timers: up to 14 timers (including 3 32-bit timers, 2 16-bit timers, and 9 general-purpose timers)
- ✓ ADC: up to 24 channels of 12-bit ADC with a conversion rate of up to 2.4 MSPS
- ✓ DAC: up to 2 channels of 12-bit DAC with a conversion rate of up to 1 MSPS
- ✓ Communication interfaces: up to 7 USARTs, up to 4 SPIs, up to 3 I2Cs, up to 2 CANs, up to 2 SDIOs, and up to 2 USB OTG FS/HS
- ✓ 2 DMA controllers that acts as master in system and can transfer data up to 10 times faster than the processor
- ✓ DCMI that can work with digital camera and can process images with different extensions with help of DMA.
- ✓ Other peripherals: RTC, WDT, CRC, RNG, and more
- ✓ Operating voltage: 1.7V to 3.6V



After choosing the MCU, it's the time to choose the on-board devices that will be used to integrate the system. Now, we are searching for devices that has three major features:

1. **Low power consumption**
2. **Output with High Accuracy**
3. **Low Cost**

Based on these features we start our journey of searching for these components, and after taking our time and see lots of components and see its rate we have chosen our components that are:

1- ESP8266

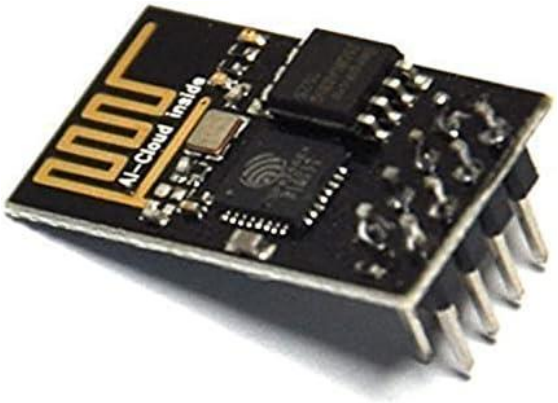
ESP8266 is a low-cost, highly integrated Wi-Fi microchip that is developed by Espressif Systems. It is designed for Internet of Things (IoT) applications and allows devices to connect to Wi-Fi networks, send and receive data over the internet, and communicate with other devices over a Wi-Fi network.

The ESP8266 microchip includes a powerful 32-bit RISC processor, 64KB of instruction RAM, and up to 1MB of data flash memory. It also has an integrated Wi-Fi radio, which supports 802.11 b/g/n standards and operates in the 2.4 GHz frequency band.

It also has a built-in TCP/IP protocol stack, which provides a complete TCP/IP protocol suite for easy integration with other devices and systems.

Due to its low cost, small size, and easy-to-use programming interface, the ESP8266 is widely used in a variety of IoT applications, such as smart home devices, industrial automation, and sensor networks. It also has a range of development boards and modules available, which make it easy for developers to integrate the ESP8266 into their projects.

We use it as our port to the internet and the key of making bar metal speak to server. It can be connected to USART and talk to it using AT commands that will be discussed later on implementation chapter.



3. Software design:

is the process of creating a plan or blueprint for a software system to meet specific requirements and objectives. It involves identifying the problem to be solved, analyzing the requirements, and then designing a solution that meets those requirements.

The main goal of software design is to create a high-quality software system that is reliable, efficient, maintainable, and scalable. We can divide this process into two subprocesses:

a- Static Design

Also known as structural design, is a type of software design that focuses on the overall structure and organization of a software system. It involves defining the static relationships and interactions between the different components of the system, including classes, objects, modules, and packages.

The primary goal of static design is to create a well-organized and modular software system that is easy to understand, maintain, and extend. This is achieved by defining a clear hierarchy of components and their relationships, as well as using standard design patterns and principles to ensure consistency and reusability.

Static design is typically done during the early stages of software development, before any code is written. It is often represented graphically using diagrams such as class diagrams, package diagrams, and component diagrams.

Some of the key principles and techniques used in static design include:

i. The Layered Architecture

is a type of software architecture that organizes the components of a software system into distinct layers, with each layer having a specific responsibility and interacting with adjacent layers through well-defined interfaces. This architecture is commonly used in enterprise applications, where the system needs to be scalable, modular, and maintainable.

The layered architecture typically consists of three or more layers, with each layer having a clear separation of concerns so that every layer has a specific responsibility.

ii. Description For Each Module APIs and Types:

In this step we define each module APIs that will be used to communicate with the module by the others.

So, we will start with our components starting with MCAL layer to APP layer.

1. MCAL

a. RCC

i. Types

Name	Rcc_PeripheralId_t
Type	Enum
Description	The ID of peripheral

Name	Rcc_ClkType_t
Type	Enum
Description	The system clock type

Name	Rcc_PllType_t
Type	Enum
Description	The function type of PLL

Name	Rcc_PllSrc_t
Type	Eanum
Description	PLL clock source

Name	Rcc_PllConfig_t
Type	Struct
Description	PLL configurations

ii. APIs

Name	Rcc_EnablePericlock
Syntax	Error_State_t Rcc_EnablePericlock(Rcc_PeripheralId_t Copy_PeripheralId, bool Copy_PeripheralClkMode);
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_PeripheralId, Copy_PeripheralClkMode
Parameters (out)	None
Return Value	ErrorState_t
Description	Enable The clock to a peripheral and choose if in low power mode for the peripheral to continue working while in sleep mode.

Name	Rcc_DisablePericlock
Syntax	Error_State_t Rcc_DisablePericlock(Rcc_PeripheralId_t Copy_PeripheralId)

Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_PeripheralId
Parameters (out)	None
Return Value	ErrorState_t
Description	Disable any clock connected to a peripheral.

Name	**Rcc_SetClkState**
Syntax	Error_State_t Rcc_SetClkState(Rcc_ClkType_t Copy_ClkType, bool Copy_ClkState, Rcc_PllConfig_t *Copy_PllConfigPtr)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_ClkType, Copy_ClkState, Copy_PllConfigPtr
Parameters (out)	None
Return Value	ErrorState_t
Description	Turn on/off different clock sources.

Name	**Rcc_SetSysClkSrc**
Syntax	Error_State_t Rcc_SetSysClkSrc(Rcc_ClkType_t Copy_ClkType)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_ClkType
Parameters (out)	None
Return Value	ErrorState_t
Description	Choose the source for the SYSCLK.

Name	**Rcc_DisablePericlock**
Syntax	Error_State_t Rcc_DisablePericlock(Rcc_PeripheralId_t Copy_PeripheralId)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_PeripheralId
Parameters (out)	None
Return Value	ErrorState_t
Description	Disable any clock connected to a peripheral.

b. GPIO

i. Types

Name	**Gpio_Port_t**
Type	Enum
Description	GPIO Port number

Name	**Gpio_PIN_t**
Type	Enum

Description	GPIO Pin number

Name	Gpio_PinMode_t
Type	Enum
Description	GPIO Pin operation mode

Name	Gpio_PinOutput_t
Type	Enum
Description	GPIO Pin output mode

Name	Gpio_OutputSpeed_t
Type	Enum
Description	GPIO Pin output speed

Name	Gpio_PinPullUpDown_t
Type	Enum
Description	GPIO Pin pullup\down

Name	Gpio_PinAltFunOption_t
Type	Enum
Description	GPIO Pin Alternate function number

Name	Gpio_PinState_t
Type	Enum
Description	GPIO Pin state

Name	Gpio_PinConfig_t
Type	Struct
Description	GPIO Pin configuration

ii. APIs

Name	Gpio_PinInit
Syntax	ErrorState_t Gpio_PinInit(const Gpio_PinConfig_t *Copy_PinConfig)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_PinConfig
Parameters (out)	None
Return Value	ErrorState_t
Description	The Function Initializes the Required Pin Configuration options.

Name	Gpio_SetPinValue

Syntax	ErrorState_t Gpio_SetPinValue(Gpio_Port_t Copy_Port,Gpio_PIN_t Copy_Pin,Gpio_PinState_t Copy_PinValue)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_Port, Copy_Pin, Copy_PinValue
Parameters (out)	None
Return Value	ErrorState_t
Description	The function sets an output value to the required pin.

Name	**Gpio_SetPortValue**
Syntax	ErrorState_t Gpio_SetPortValue(Gpio_Port_t Copy_Port,u16 Copy_PortValue);
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Copy_Port, Copy_PortValue
Parameters (out)	None
Return Value	ErrorState_t
Description	The function sets an output value to the required port.

Name	**Gpio_GetPinValue**
Syntax	ErrorState_t Gpio_GetPinValue(Gpio_Port_t Copy_Port,Gpio_PIN_t Copy_Pin,Gpio_PinState_t* Copy_PinValue)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_Port, Copy_Pin
Parameters (out)	Copy_PinValue
Return Value	ErrorState_t
Description	The function Reads an Input Value of the Required Pin.

Name	**Gpio_GetPortValue**
Syntax	ErrorState_t Gpio_GetPortValue(Gpio_Port_t Copy_Port,u16* Copy_PortValue)
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Copy_Port
Parameters (out)	Copy_PortValue
Return Value	ErrorState_t
Description	The function reads an Input value of the required port.

Name	**Gpio_TogglePinValue**

Syntax	ErrorState_t Gpio_TogglePinValue(Gpio_Port_t Copy_u8Port,Gpio_PIN_t Copy_Pin)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_Port, Copy_Pin
Parameters (out)	None
Return Value	ErrorState_t
Description	The function toggles the value of output pin.

c. NVIC

i. Types

Name	Nvic_IRQn_t
Type	Enum
Description	IRQ number

Name	Nvic_PrioGroup_t
Type	Enum
Description	Group and sub group priority

ii. APIs

Name	Nvic_EnableIRQ
Syntax	ErrorState_t Nvic_EnableIRQ(Nvic_IRQn_t Copy_IRQ)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	None
Return Value	ErrorState_t
Description	The function enables the required IRQ.

Name	Nvic_DisableIRQ
Syntax	ErrorState_t Nvic_DisableIRQ(Nvic_IRQn_t Copy_IRQ)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	None
Return Value	ErrorState_t
Description	The function Disables the required IRQ.

Name	Nvic_SetPendingIRQ
Syntax	ErrorState_t Nvic_SetPendingIRQ(Nvic_IRQn_t Copy_IRQ)
Sync/Async	Synchronous

Reentrancy	Non-Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	None
Return Value	ErrorState_t
Description	The function sets the pending flag of the required IRQ by software.

Name	**Nvic_ClearPendingIRQ**
Syntax	ErrorState_t Nvic_ClearPendingIRQ(Nvic_IRQn_t Copy_IRQ)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	None
Return Value	ErrorState_t
Description	The function resets the pending flag of the required IRQ by software.

Name	**Nvic_GetPendingIRQ**
Syntax	ErrorStateErrorState_t Nvic_GetPendingIRQ(Nvic_IRQn_t Copy_IRQ, bool *pState)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	pState
Return Value	ErrorState_t
Description	The function gets the pending flag state of the required IRQ.

Name	**Nvic_GetActive**
Syntax	ErrorState_t Nvic_GetActive(Nvic_IRQn_t Copy_IRQ, bool *pState)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	pState
Return Value	ErrorState_t
Description	The function gets the active flag state of the required IRQ.

Name	**Nvic_SetPriorityGrouping**
Syntax	ErrorState_t Nvic_SetPriorityGrouping(Nvic_PrioGroup_t Copy_PrioGroup);
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_PrioGroup

Parameters (out)	None
Return Value	ErrorState_t
Description	The function sets the group and sub-group numbers.

Name	Nvic_SetPriority
Syntax	ErrorState_t Nvic_SetPriority(Nvic_IRQn_t Copy_IRQ, u8 Copy_Prio);
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Copy_IRQ, Copy_Prio
Parameters (out)	None
Return Value	ErrorState_t
Description	The function sets the priority of the required IRQ.

Name	Nvic_GetPriority
Syntax	ErrorState_t Nvic_GetPriority(Nvic_IRQn_t Copy_IRQ, u8 *pPrio)
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	pPrio
Return Value	ErrorState_t
Description	The function gets the priority of the required IRQ.

Name	Nvic_GenerateInterrupt
Syntax	ErrorState_t Nvic_GenerateInterrupt(Nvic_IRQn_t Copy_IRQ)
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Copy_IRQ
Parameters (out)	None
Return Value	ErrorState_t
Description	The function generates interrupt by software to the required IRQ.

d. DMA

i. Types

Name	Dma_Num
Type	Enum
Description	DMA number

Name	Dma_StreamNum
Type	Enum

Description	DMA Stream Number

Name	Dma_ChannelNum
Type	Enum
Description	DMA Stream channel Number

Name	Dma_StreamMode
Type	Enum
Description	DMA Stream Mode

Name	Dma_StreamFifoTreshold
Type	Enum
Description	DMA Stream FIFO Threshold

Name	Dma_ChannelMode
Type	Enum
Description	DMA Channel Mode

Name	Dma_TransDirction
Type	Enum
Description	DMA Transfer Direction

Name	Dma_StreamPriority
Type	Enum
Description	DMA Stream Priority

Name	Dma_SrcDesState
Type	Enum
Description	DMA Source and Destination Increment state

Name	Dma_PeripheralSize
Type	Enum
Description	DMA Peripheral Size

Name	Dma_MemorySize
Type	Enum
Description	DMA Memory Size

Name	Dma_Transfer_t

Type	Enum
Description	DMA Transfer type

Name	Dma_Config_t
Type	Struct
Description	DMA configuration structure

Name	Dma_StreamStartTrans
Type	Struct
Description	DMA Stream Transaction Configurations

ii. APIs

Name	Dma_Init
Syntax	ErrorState_t Dma_Init(Dma_Config_t* Copy_DmaConfig)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_DmaConfig
Parameters (out)	None
Return Value	ErrorState_t
Description	Initialize the required DMA with required configuration.

Name	Dma_StreamStartSynch
Syntax	ErrorState_t Dma_StreamStartSynch(Dma_StreamStartTrans* Copy_StartTrans)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_StartTrans
Parameters (out)	None
Return Value	ErrorState_t
Description	Initialize the transaction of DMA.

Name	Dma_StreamStartASynch
Syntax	ErrorState_t Dma_StreamStartASynch(Dma_StreamStartTrans* Copy_StartTrans, void(*Copy_NotificationFunc)(void));
Sync/Async	Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_StartTrans, Copy_NotificationFunc
Parameters (out)	None
Return Value	ErrorState_t
Description	Initialize the transaction of DMA.

Name	DMA_StopStream
Syntax	void DMA_StopStream(Dma_Num Copy_DmaNumber, Dma_StreamNum Copy_StreamNum)
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Copy_DmaNumber, Copy_StreamNum
Parameters (out)	None
Return Value	Void
Description	Stop the DMA Stream.

e. ADC

i. Types

Name	Adc_Num
Type	Enum
Description	ADC Number

Name	Adc_ChannelNum
Type	Enum
Description	ADC channel Number

Name	Adc_Resolution
Type	Enum
Description	ADC Resolution

Name	Adc_TriggerType
Type	Enum
Description	ADC Trigger Type

Name	Adc_ExtTriggerSense
Type	Enum
Description	ADC External Trigger edge Sense

Name	Adc_RegularExtTrigger
Type	Enum
Description	ADC Regular Channel External Trigger Source

Name	Adc_InjectedExtTrigger
Type	Enum
Description	ADC Injected Channel External Trigger Source

Name	Adc_DataAlignment
Type	Enum
Description	ADC Output Alignment

Name	Adc_ChannelSampleTime
Type	Enum
Description	ADC Sampling Clock Cycles

Name	Adc_ChannelType
Type	Enum
Description	ADC Channel Type Regular / Injected

Name	Adc_ChainConvType
Type	Enum
Description	ADC Group Conversion Type

Name	Adc_ConversionConfig_t
Type	Struct
Description	ADC Channel Conversion Configurations

Name	Adc_ChainConvConfig_t
Type	Struct
Description	ADC Chain of Channels Conversion Configurations

ii. APIs

Name	Adc_Init
Syntax	ErrorState_t Adc_Init(Adc_Num Copy_Adc, Adc_Resolution Copy_AdcRes, Adc_DataAlignment Copy_DataAlignment)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_Adc, Copy_AdcRes, Copy_DataAlignment
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Initialize the ADC.

Name	Adc_StartConversionSynch
Syntax	ErrorState_t Adc_StartConversionSynch(Adc_ConversionConfig_t* Copy_ConvConfig, u16* Copy_Reading)
Sync/Async	Synchronous

Reentrancy	Non-Reentrant
Parameters (in)	Copy_ConvConfig
Parameters (out)	Copy_Reading
Return Value	ErrorState_t
Description	Convert channel in a synchronous way.

Name	**Adc_StartConversionAsynch**
Syntax	ErrorState_t Adc_StartConversionAsynch(Adc_ConversionConfig_t* Copy_ConvConfig, u16* Copy_Reading, void(*Copy_NotificationFunc)(void))
Sync/Async	Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_ConvConfig, Copy_NotificationFunc
Parameters (out)	Copy_Reading
Return Value	ErrorState_t
Description	Convert channel in an asynchronous way.

Name	**Adc_StartChainConversionSynch**
Syntax	ErrorState_t Adc_StartChainConversionSynch (Adc_ChainConvConfig_t* Copy_ChainConfig, u16* Copy_Reading)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_ChainConfig
Parameters (out)	Copy_Reading
Return Value	ErrorState_t
Description	Convert chain of channels in a synchronous way.

Name	**Adc_StartChainConversionAsynch**
Syntax	ErrorState_t Adc_StartChainConversionAsynch (Adc_ChainConvConfig_t* Copy_ChainConfig, u16* Copy_Reading, void(*Copy_NotificationFunc)(void))
Sync/Async	Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_ChainConfig, Copy_NotificationFunc
Parameters (out)	Copy_Reading
Return Value	ErrorState_t
Description	Convert chain of channels in an asynchronous way.

f. DCMI

i. Types

Name	**Dcmi_InterruptId_t**
Type	Enum
Description	DCMI interrupt source

Name	Dcmi_InterruptState_t
Type	Enum
Description	DCMI interrupt state

ii. APIs

Name	Dcmi_Init
Syntax	`void Dcmi_Init(void)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	None
Parameters (out)	None
Return Value	Void
Description	initialize the DCMI peripheral.

Name	Dcmi_CaptureImage
Syntax	`void Dcmi_CaptureImage(void)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	None
Parameters (out)	None
Return Value	Void
Description	start the capture of frame.

Name	Dcmi_Init
Syntax	`ErrorState_t Dcmi_ControlInt(Dcmi_InterruptId_t Copy_IntId, Dcmi_InterruptState_t Copy_IntState, void(*Copy_CallBackFunc)(void))`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_IntId, Copy_IntState, Copy_CallBackFunc`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	control the state of the interrupts.

g. EXTI

i. Types

Name	Exti_Port_t
Type	Enum
Description	The EXTI Pin Port

Name	Exti_Pin_t
Type	Enum
Description	The EXTI Pin

Name	Exti_Trigger_t
Type	Enum
Description	EXTI Trigger Type

Name	Exti_PinConfig_t
Type	Struct
Description	EXTI Pin Configuration

ii. APIs

Name	Exti_PinInit
Syntax	`ErrorState_t Exti_PinInit(const Exti_PinConfig_t* Copy_PinConfig);`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_PinConfig
Parameters (out)	None
Return Value	ErrorState_t
Description	Initialize the pin as EXTI pin.

Name	Exti_IntEnable
Syntax	`void Exti_IntEnable(Exti_Pin_t Copy_Pin)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_Pin
Parameters (out)	None
Return Value	Void
Description	Enable The External Interrupt.

Name	Exti_IntDisable
Syntax	`void Exti_IntDisable(Exti_Pin_t Copy_Pin)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_Pin
Parameters (out)	None
Return Value	Void
Description	Disable The External Interrupt.

h. USART

i. Types

Name	Usart_Number_t

Type	Enum
Description	The required USART number to be configured

Name	Usart_DataLength_t
Type	Enum
Description	USART Data Length to be transmitted / received

Name	Usart_Parity_t
Type	Enum
Description	USART Parity Check State

Name	Usart_mode_t
Type	Enum
Description	USART Mode Synch/Asynch

Name	Usart_StopBit_t
Type	Enum
Description	USART Number of Stop Bits

Name	Usart_DataTransfer_t
Type	Enum
Description	USART Transfer Data type

Name	Usart_config_t
Type	Struct
Description	USART Configuration structure

ii. APIs

Name	Usart_Init
Syntax	`ErrorState_t Usart_Init(Usart_config_t* Copy_config)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_config`

Parameters (out)	None
Return Value	ErrorState_t
Description	initialize USART With Required Configuration.

Name	Usart_SendCharSynch
Syntax	ErrorState_t Usart_SendCharSynch(Usart_Number_t Copy_UsartNum, u16 Copy_Data)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_UsartNum, Copy_Data
Parameters (out)	None
Return Value	ErrorState_t
Description	Send a character through USART in Synchronous way.

Name	Usart_SendCharASynch
Syntax	ErrorState_t Usart_SendCharASynch(Usart_Number_t Copy_UsartNum, u16 Copy_Data , void (*Copy_NotificationFunc)(void))
Sync/Async	Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_UsartNum, Copy_Data, Copy_NotificationFunc
Parameters (out)	None
Return Value	ErrorState_t
Description	Send a character through USART in an asynchronous way.

Name	Usart_SendStringSynch
Syntax	ErrorState_t Usart_SendStringSynch(Usart_Number_t Copy_UsartNum, char* Copy_String)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_UsartNum, Copy_String
Parameters (out)	None
Return Value	ErrorState_t
Description	Send a String through USART in synchronous way.

Name	Usart_SendStringASynch
Syntax	ErrorState_t Usart_SendStringASynch(Usart_Number_t Copy_UsartNum, char* Copy_String, void (*Copy_NotificationFunc)(void))
Sync/Async	Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_UsartNum, Copy_String, Copy_NotificationFunc
Parameters (out)	None
Return Value	ErrorState_t

Description	Send a String through USART in an asynchronous way.

Name	Usart_SendBufferSynch
Syntax	`ErrorState_t Usart_SendBufferSynch(Usart_Number_t Copy_UsartNum, u8* Copy_Buffer, u16 Copy_BufferLen)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_Buffer, Copy_BufferLen`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Send a Buffer through USART in Synchronous way.

Name	Usart_SendBufferASynch
Syntax	`ErrorState_t Usart_SendBufferASynch(Usart_Number_t Copy_UsartNum, char* Copy_Buffer, u16 Copy_BufferLen, void (*Copy_NotificationFunc)(void))`
Sync/Async	Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_Buffer, Copy_BufferLen, Copy_NotificationFunc`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Send a Buffer through USART in an asynchronous way.

Name	Usart_ReceiveCharSynch
Syntax	`ErrorState_t Usart_ReceiveCharSynch(Usart_Number_t Copy_UsartNum, u8* Copy_Data)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum`
Parameters (out)	`Copy_Data`
Return Value	`ErrorState_t`
Description	Receive a character through USART in a synchronous way.

Name	Usart_ReceiveCharASynch

Syntax	`ErrorState_t Usart_ReceiveCharASynch(Usart_Number_t Copy_UsartNum, u16* Copy_Data, void (*Copy_NotificationFunc)(void))`
Sync/Async	Asynchronous
Reentrancy	`Copy_Data`
Parameters (in)	`Copy_UsartNum, Copy_NotificationFunc`
Parameters (out)	Copy_Reading
Return Value	`ErrorState_t`
Description	Receive a character through USART in an asynchronous way.

Name	**Usart_ReceiveBufferSynch**
Syntax	`ErrorState_t Usart_ReceiveBufferSynch(Usart_Number_t Copy_UsartNum, u8* Copy_Buffer, u16 Copy_BufferSize)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_BufferSize`
Parameters (out)	`Copy_Buffer`
Return Value	`ErrorState_t`
Description	Receive a Buffer through USART in a synchronous way.

Name	**Usart_ReceiveBufferASynch**
Syntax	`ErrorState_t Usart_ReceiveBufferASynch(Usart_Number_t Copy_UsartNum, u8* Copy_Buffer, u16 Copy_BufferSize, void (*Copy_NotificationFunc)(void))`
Sync/Async	Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_BufferSize, Copy_NotificationFunc`
Parameters (out)	`Copy_Buffer`
Return Value	`ErrorState_t`
Description	Receive a Buffer through USART in an asynchronous way.

i. **I2C**

i. **Types**

Name	**I2c_Id_t**
Type	Enum
Description	I2C Number

Name	**I2c_Mode_t**
Type	Enum
Description	I2C Mode of Operation

Name	**I2c_State_t**

Type	Enum
Description	I2C State

Name	I2c_Config_t
Type	Struct
Description	I2C Configuration Structure

ii. APIs

Name	I2c_Init
Syntax	`ErrorState_t I2c_Init(I2c_Handle_t *hi2c)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	hi2c
Parameters (out)	None
Return Value	ErrorState_t
Description	Initialize the I2C with required configuration.

Name	I2c_Master_Transmit
Syntax	`ErrorState_t I2c_Master_Transmit(I2c_Handle_t *hi2c, u16 Copy_DevAddress, u8 *pData, u16 Copy_Size, u32 Copy_Timeout)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	*hi2c, Copy_DevAddress, pData, Copy_Size, Copy_Timeout
Parameters (out)	Copy_Buffer
Return Value	ErrorState_t
Description	Master transmit to slave.

Name	I2c_Mem_Write
Syntax	`ErrorState_t I2c_Mem_Write(I2c_Handle_t *hi2c, u16 Copy_DevAddress, u16 Copy_MemAddress, u16 Copy_MemAddSize, u8 *pData, u16 Copy_Size, u32 Copy_Timeout)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	*hi2c, Copy_DevAddress, Copy_MemAddress, Copy_MemAddSize, pData, Copy_Size, Copy_Timeout
Parameters (out)	Copy_Buffer
Return Value	ErrorState_t
Description	Write a specific memory in slave.

Name	I2c_GetFlag
Syntax	`u8 I2c_GetFlag(I2c_Handle_t *hi2c, I2c_Flag_t Copy_Flag)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`hi2c, Copy_Flag`
Parameters (out)	None
Return Value	U8
Description	Get I2C Flag.

Name	I2c_Enable
Syntax	`ErrorState_t I2c_Enable(I2c_Handle_t* hi2c)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`hi2c`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Enable I2C.

Name	I2c_Disable
Syntax	`ErrorState_t I2c_Disable(I2c_Handle_t* hi2c)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`hi2c`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Disable the I2C.

j. SysTick

i. Types

Name	SysTick_ClkSrc_t
Type	Enum
Description	SysTick Timer Clock source

Name	SysTick_Config_t
Type	Struct
Description	SysTick Timer configuration

ii. APIs

Name	SysTick_Init
Syntax	`void SysTick_Init(SysTick_Config_t *pConfig);`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`pConfig`

Parameters (out)	None
Return Value	Void
Description	Enable The External Interrupt.

Name	SysTick_GetTick
Syntax	`u32 SysTick_GetTick(void)`
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	None
Parameters (out)	None
Return Value	U32
Description	gets the current tick count.

Name	SysTick_IncTick
Syntax	`void SysTick_IncTick(void);`
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	None
Parameters (out)	None
Return Value	Void
Description	Increments a local static variable by 1.

Name	SysTick_Delay
Syntax	`void SysTick_Delay(u32 Copy_DelayMs)`
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Copy_DelayMs
Parameters (out)	None
Return Value	Void
Description	blocking delay in ms.

2. **HAL**
    a. **ESP**
        i. **Types**

Name	Esp_UsartNum
Type	Enum
Description	The USART Number ESP Connected to.

Name	Esp_Recv_t
Type	Enum
Description	Receiving from ESP Type Sync/Asynch

ii. APIs

Name	Esp_Init
Syntax	`ErrorState_t Esp_Init(Esp_UsartNum Copy_UsartNum)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Initialize the ESP module.

Name	Esp_ConnectWifi
Syntax	`ErrorState_t Esp_ConnectWifi(Esp_UsartNum Copy_UsartNum, char* Copy_Username, char* Copy_Password)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_Username, Copy_Password`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Connect ESP to Wi-Fi.

Name	Esp_ConnectServer
Syntax	`ErrorState_t Esp_ConnectServer(Esp_UsartNum Copy_UsartNum, char* Copy_ServerIp, char* Copy_ConnectionType, u16 Copy_PortNum)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_ServerIp, Copy_ConnectionType, Copy_PortNum`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Connect the ESP module to Server.

Name	Esp_SendData
Syntax	`ErrorState_t Esp_SendData(Esp_UsartNum Copy_UsartNum, u8* Copy_Data, u16 Copy_DataLength)`
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_Data, Copy_DataLength`
Parameters (out)	None
Return Value	`ErrorState_t`
Description	Send Data to the server.

Name	Esp_ReceiveData
Syntax	`ErrorState_t Esp_ReceiveData(Esp_UsartNum Copy_UsartNum, u8* Copy_Data, u16 Copy_DataLength, Esp_Recv_t Copy_RecvType, void (*Copy_NotificationFunc)(void))`
Sync/Async	Synchronous / Asynchronous
Reentrancy	Non-Reentrant
Parameters (in)	`Copy_UsartNum, Copy_DataLength, Copy_RecvType, Copy_NotificationFunc`
Parameters (out)	`Copy_Data`
Return Value	`ErrorState_t`
Description	Receive data from the server.

b. **OV7670**

i. **Types**

Name	Ov7670_I2cId_t
Type	Enum
Description	The I2C Number Camera Connected to.

3. **Service**

a. **MQTT**

i. **Types**

Name	Mqtt_UsartNum
Type	Enum
Description	The USART Number ESP Connected to.

Name	Mqtt_Qos_t
Type	Enum
Description	QOS of Message publish / Subscribe.

Name	Mqtt_Connect_t
Type	Struct
Description	ESP Connection configuration.

Name	Mqtt_EspConnection
Type	Struct
Description	ESP Connected to MCU configuration.

Name	Mqtt_Publish_t
Type	Struct

Description	The publish message configuration.

ii. APIs

Name	Esp_SendData
Syntax	ErrorState_t Mqtt_Connect(Mqtt_UsartNum Copy_UsartNum ,Mqtt_Connect_t* Copy_Connect, Mqtt_EspConnection* Copy_EspConfig)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_UsartNum, Copy_Connect, Copy_EspConfig
Parameters (out)	None
Return Value	ErrorState_t
Description	Function initializes the whole Connection and Sends the Connect Packet to broker.

Name	Esp_SendData
Syntax	ErrorState_t Mqtt_Publish(Mqtt_UsartNum Copy_UsartNum, Mqtt_Publish_t* Copy_PubPacket)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_UsartNum, Copy_PubPacket
Parameters (out)	None
Return Value	ErrorState_t
Description	Function Sends the publish Packet to broker.

Name	Esp_SendData
Syntax	ErrorState_t Mqtt_Subscribe(Mqtt_UsartNum Copy_UsartNum, char* Copy_TopicName, Mqtt_Qos_t Copy_MaxQos)
Sync/Async	Synchronous
Reentrancy	Non-Reentrant
Parameters (in)	Copy_UsartNum, Copy_TopicName, Copy_MaxQos
Parameters (out)	None
Return Value	ErrorState_t
Description	Function Sends the subscribe Packet to broker.

4. LIB

a. Platform Types

Name	u8 \ s8
Type	Permeative
Description	Type definition of unsigned \ signed char.

Name	u16 \ s16
Type	Permeative
Description	Type definition of unsigned \ signed short.

Name	U32 \ s32
Type	Permeative
Description	Type definition of unsigned \ signed long.

Name	f32
Type	Permeative
Description	Type definition of float.

Name	f64
Type	Permeative
Description	Type definition of double.

b. STD Types

Name	Peripheral_State
Type	Enum
Description	Contains all states of peripherals.

Name	ISR_Src
Type	Enum
Description	Contains the possible ISR sources in the peripherals.

Name	Bool
Type	Enum
Description	Type definition for the Boolean.

Name	ErrorState_t
Type	Enum
Description	Contains all possible error states that can happen in peripherals.

b- Dynamic Design

Dynamic design, also known as behavioral design, is a type of software design that focuses on the dynamic behavior of a software system. It involves defining the behavior of the system in response to various events or stimuli, such as user input, external data, or system events.

The primary goal of dynamic design is to create a software system that executes correctly and efficiently in response to various scenarios and inputs. This is achieved by defining the

actions and interactions of the different components of the system, including classes, objects, and modules, as well as the flow of data and control between them.

Some of the key principles and techniques used in dynamic design include:

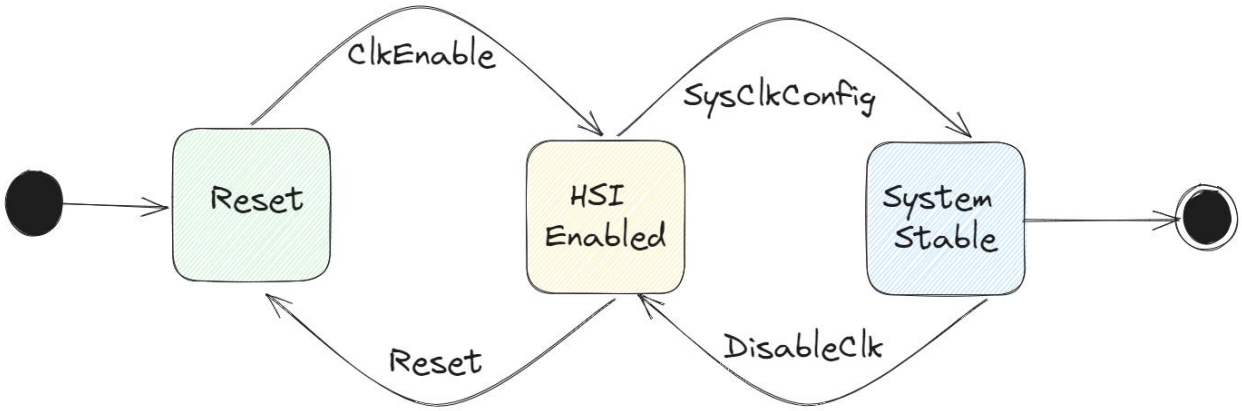
i. State Machine Diagram

These are graphical representations of the possible states of a system and the transitions between them, which help to define the behavior of the system in response to different events. And will be done for two scopes.

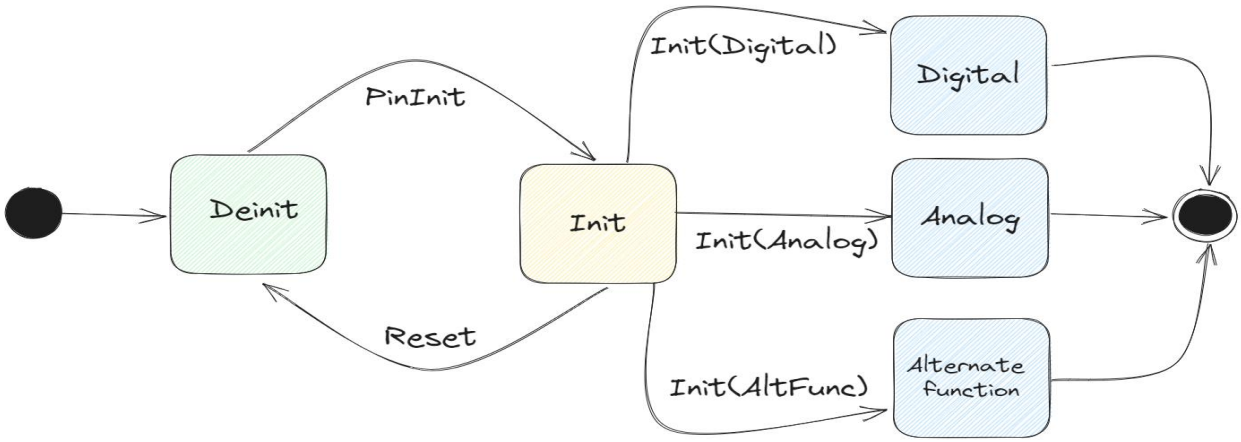
1. ECU Components

a. MCAL

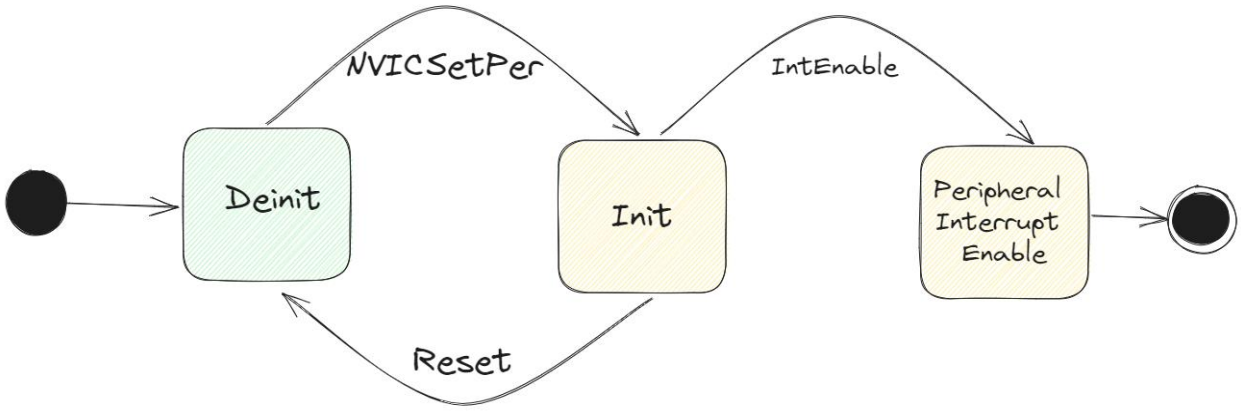
i. RCC



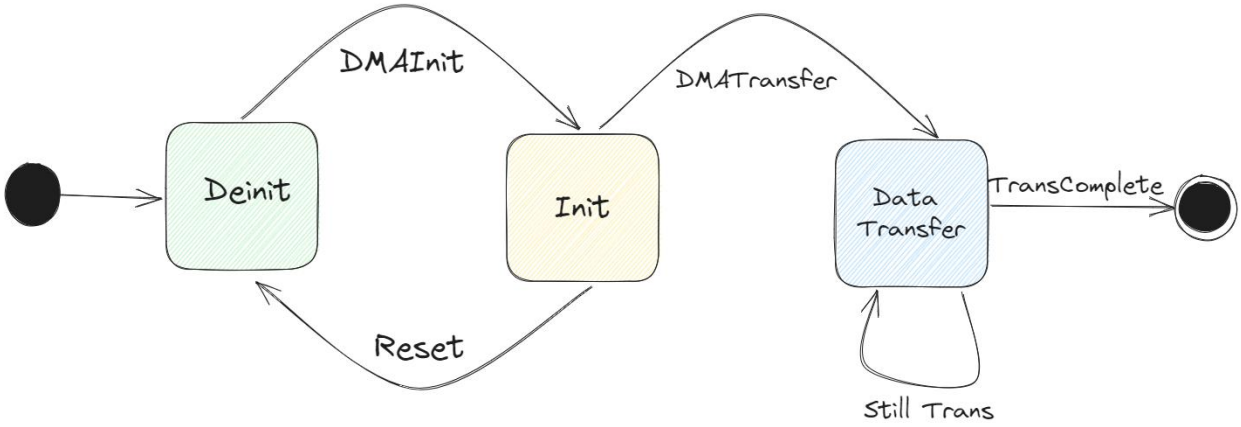
ii. GPIO



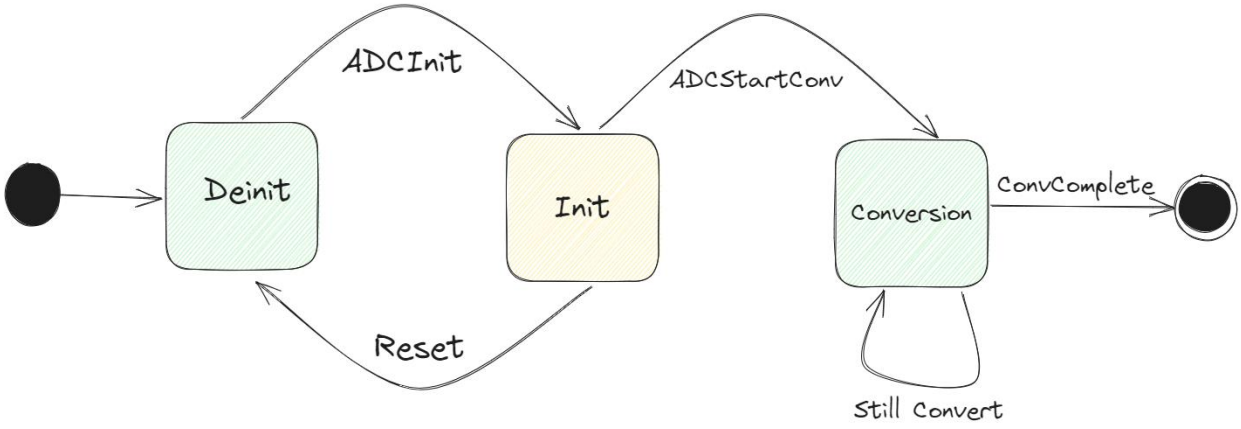
iii. NVIC



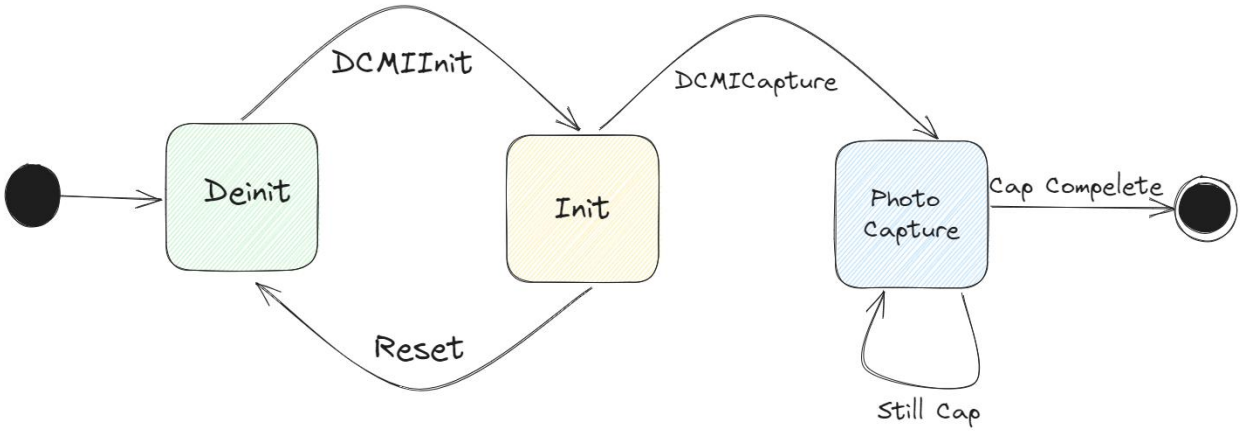
iv. DMA



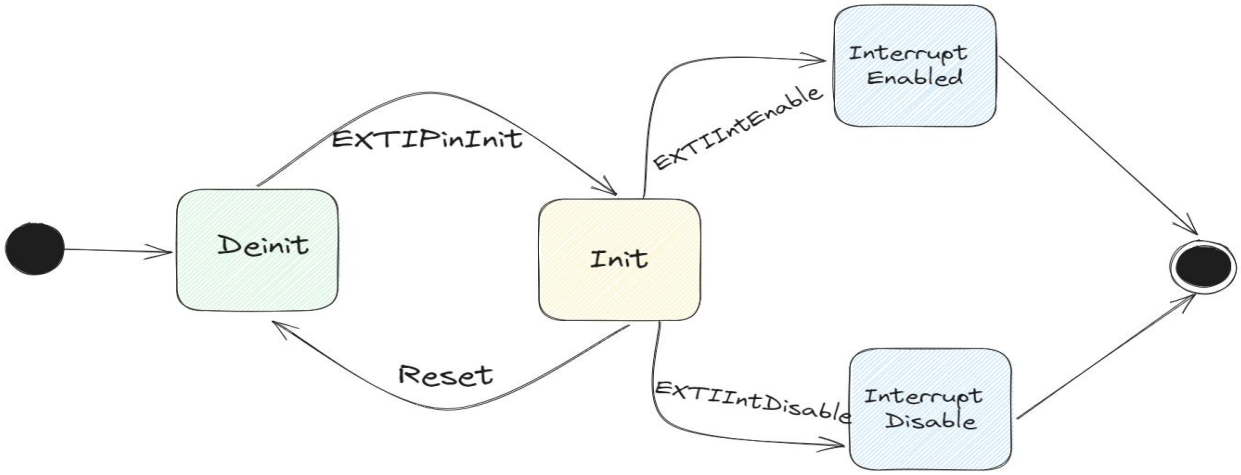
v. ADC



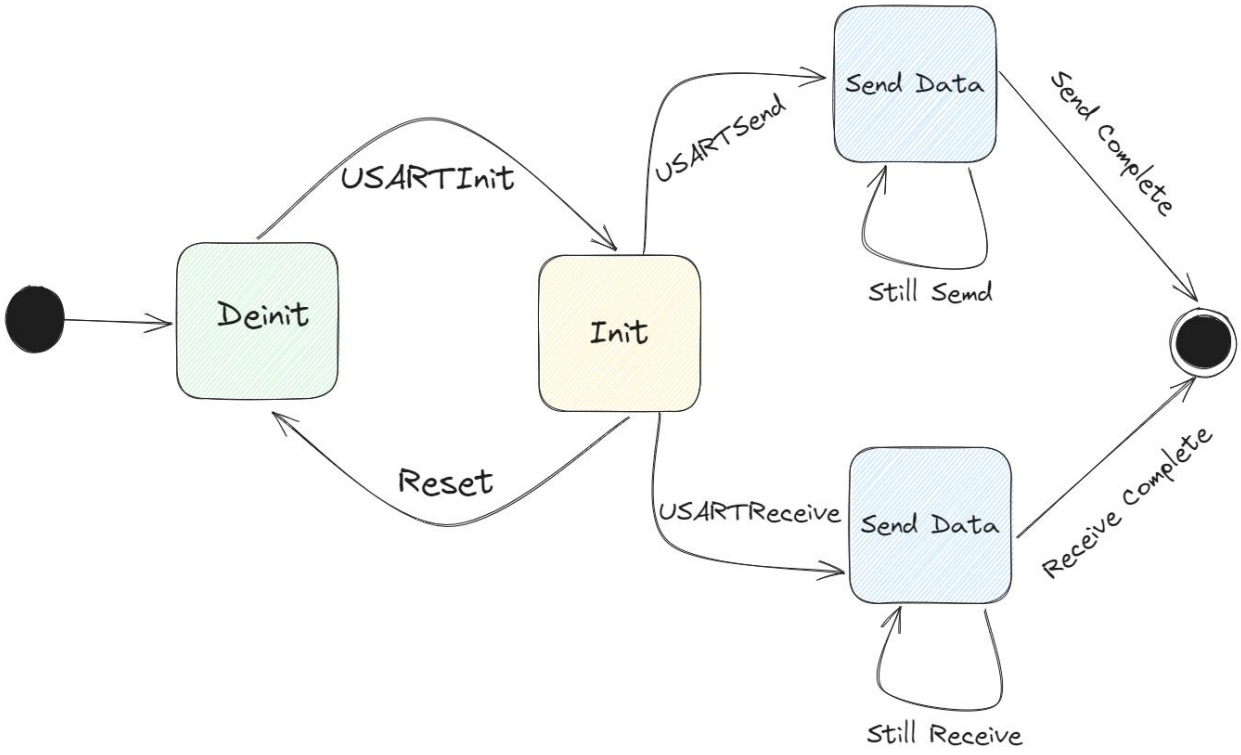
vi. DCMI



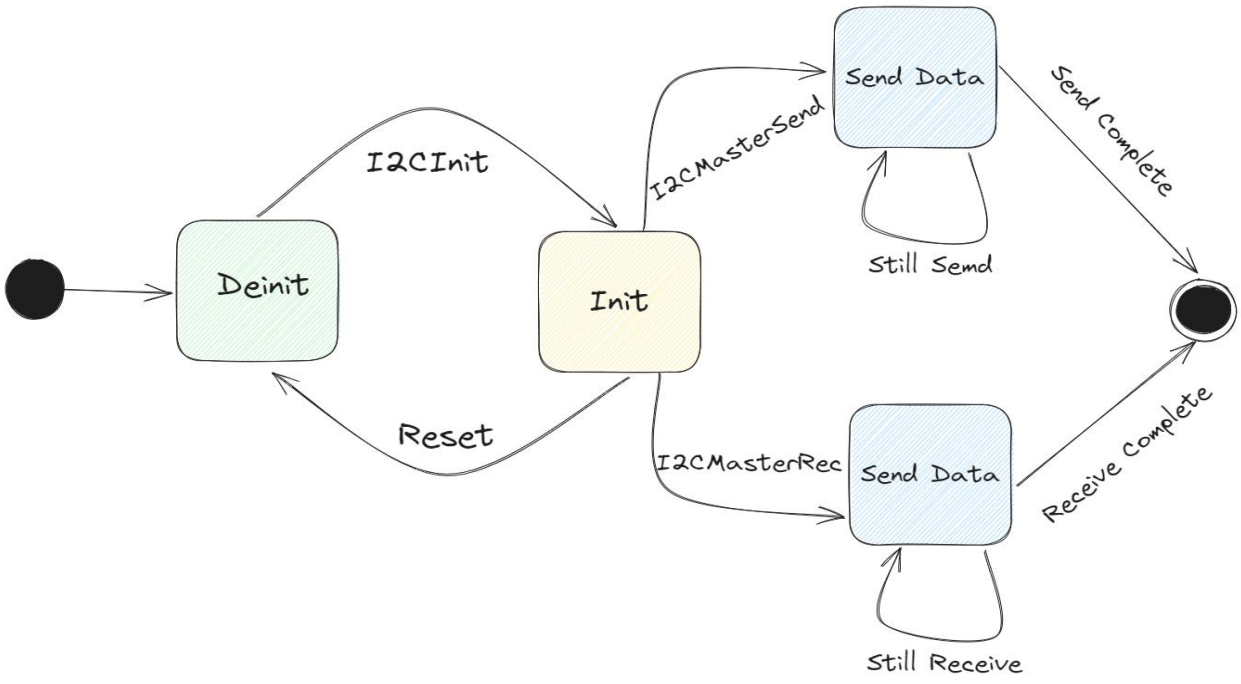
vii. EXTI



viii. USART

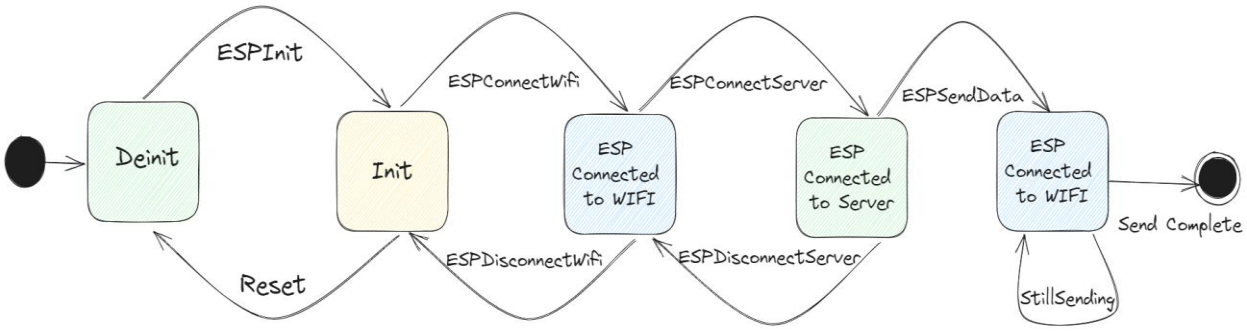

Send Data
Send Complete
USARTSend
USARTInit
Still Semd
Deinit
Init
Reset
USARTReceive
Send Data
Receive Complete
Still Receive

ix. I2C

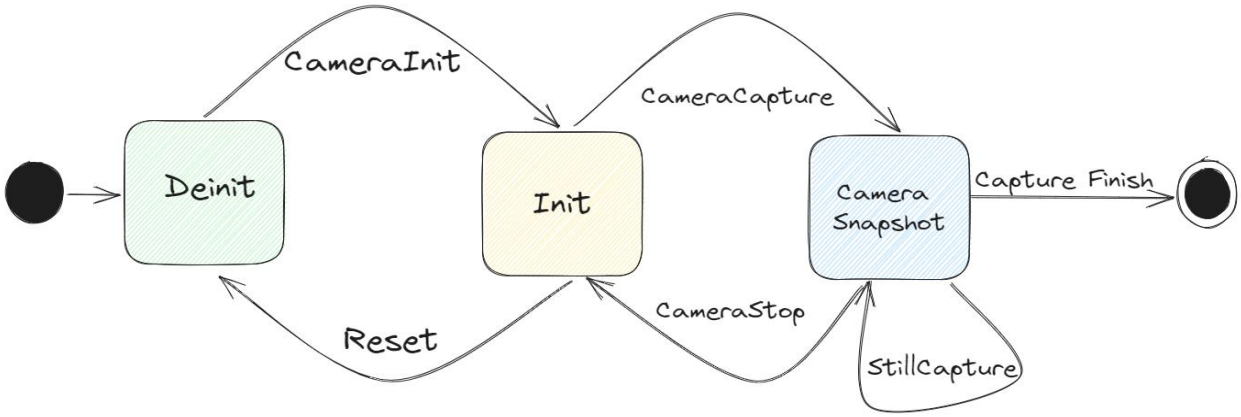

Send Data
Send Complete
I2CMasterSend
I2CInit
Still Semd
Deinit
Init
Reset
I2CMasterRec
Send Data
Receive Complete
Still Receive

b. HAL

i. ESP

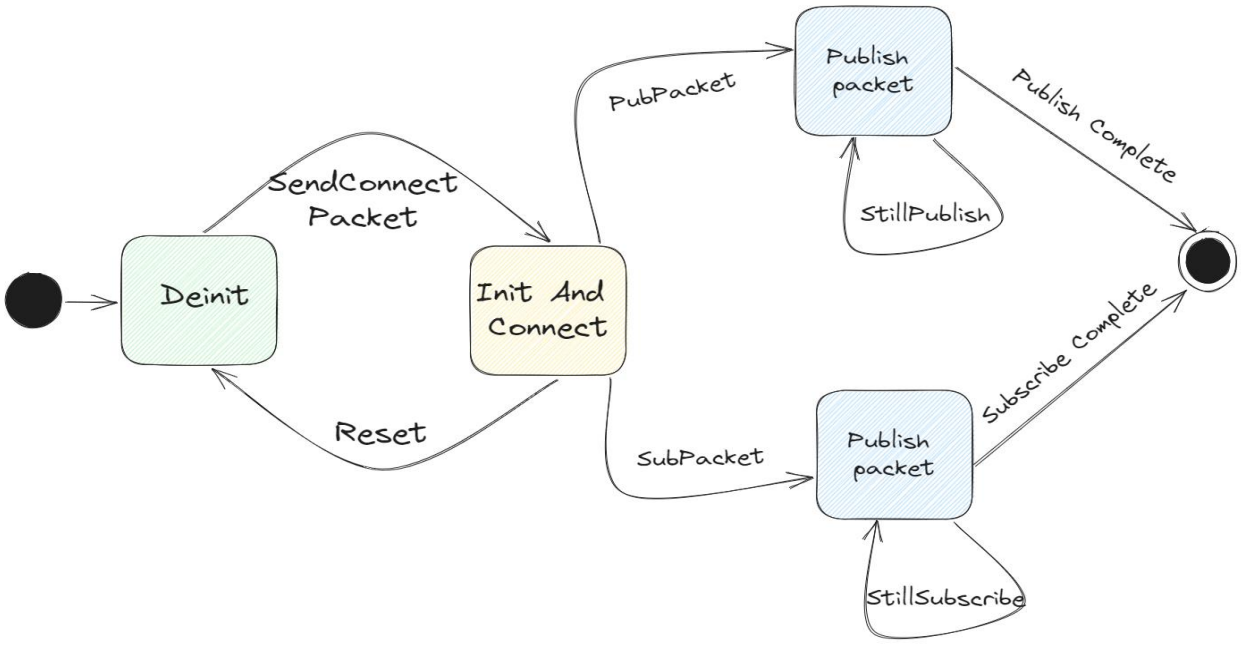


ii. OV7670



c. Service

i. MQTT



2. ECU Operation

ii. The Sequence Diagram for ECU

These are graphical representations of the interactions between objects in a system, which help to define the flow of control and data between them.

Conclusion

In conclusion, both software and hardware design are critical components of any technology solution, and both play important roles in ensuring the success of a project.

Software design involves the creation of software applications, systems, and programs that enable devices to perform specific functions and interact with users. It involves the development of algorithms, coding, testing, and debugging to ensure that the software operates as intended and meets the desired requirements.

Hardware design, on the other hand, is the process of creating physical components and systems that enable devices to function. It involves the design of circuits, components, and devices, as well as the selection of materials, manufacturing processes, and testing procedures to ensure that the hardware performs as intended and meets the desired specifications.

Both software and hardware design are closely intertwined, as software relies on hardware to function, and hardware requires software to control and operate. Therefore, it is essential to consider both software and hardware design together when developing a technology solution.

Ultimately, successful software and hardware design requires a multidisciplinary approach, with collaboration between software engineers, hardware engineers, and other stakeholders to ensure that the solution meets the desired specifications and performs as intended.

Chapter X Make Bare Metal Talking to The World Using MQTT

Intro

In This chapter we will take a dive in IoT world to learn more about how to talk to servers and websites, what is MQTT, why using it, and how to configure it in our MCU.

First, we will talk about IoT and what is it, why IoT and what we need from it.

IoT

The Internet of Things (IoT) is a network of physical devices, vehicles, buildings, and other objects that are embedded with sensors, software, and connectivity, allowing them to collect and exchange data with other devices and systems over the internet. The IoT is transforming the way we live and work, enabling new levels of automation, efficiency, and convenience across a wide range of industries and applications. There are several key components of the IoT ecosystem, including:

1. **Devices:**

   IoT devices can be anything from simple sensors to complex machines, and they are usually designed to collect data, monitor conditions, or perform specific tasks. Examples of IoT devices include smart thermostats, fitness trackers, industrial sensors, and autonomous vehicles.

2. **Connectivity:**

   IoT devices rely on various forms of connectivity to transmit data to other devices and systems. This can include wireless networks such as Wi-Fi, cellular networks, and low-power wide-area networks (LPWANs).

3. **Cloud Platforms:**

   IoT data is often processed and analyzed in cloud-based platforms that provide scalable and secure storage and computing resources. These platforms can be used to store, analyze, and visualize IoT data, as well as to manage and control IoT devices remotely.

4. **Applications:**

   IoT applications are software programs that are designed to interact with IoT devices and data. These applications can be used to control devices, monitor performance, automate processes, and provide insights and alerts based on IoT data.

Some of the key benefits of the IoT include:

1. **Increased Efficiency:**

   IoT devices and applications can automate and optimize many processes, reducing waste, improving productivity, and lowering costs.

2. **Improved Safety and Security:**

   IoT devices can monitor and alert users to potential safety hazards or security breaches, enabling them to take action quickly and prevent accidents or losses.

3. **Better Decision Making:**

   IoT data can provide valuable insights into customer behavior, product performance, and operational efficiency, enabling businesses to make more informed decisions.

4. **Enhanced User Experience:**

   IoT devices can provide personalized and convenient experiences for users, such as smart home devices that adjust lighting and temperature based on user preferences.

However, there are also several challenges associated with the IoT, including security and privacy concerns, interoperability issues, and the need for reliable connectivity and data management. As the IoT continues to evolve and mature, these challenges will need to be addressed to ensure that the IoT can deliver on its promise of a more connected and efficient world. We have lots of IoT protocols that can be used to connect our MCU to the internet. After searching we found two protocols that can be used in our case MQTT and COAP that are Application Layer protocols in internet stack and we will compare between them to choose one of Them.

Protocol Name	Network Layer	Transport Layer Protocol
COAP	APP	UDP
MQTT	APP	TCP

So, the main factor that we will choose the protocol based on it is the Transport Layer Protocol that used to send the Data. So, we need to know more about TCP and UDP.

1. **TCP:**

   Transmission Control Protocol is a widely used transport layer protocol in computer networking, providing reliable, ordered, and error-checked delivery of data between applications running on different hosts. TCP is a connection-oriented protocol, which means that it establishes a virtual connection between two endpoints before transmitting data. When two hosts want to communicate using TCP, they first establish a connection using a three-way handshake.

2. **UDP:**

   User Datagram Protocol is a transport layer protocol in computer networking that provides lightweight, low-overhead communication between applications running on different hosts. Unlike TCP, UDP is a connectionless protocol, meaning that it does not establish a virtual connection between two endpoints before transmitting data.

As TCP is a reliable and efficient protocol that is widely used in a variety of applications and industries. Its reliability and error-checking mechanisms make it suitable for applications that require accurate data transmission, while its flow control and congestion control mechanisms make it suitable for applications that require efficient use of network resources. We will choose MQTT to Send Data to server. So let's know more about MQTT and its configurations.

MQTT

Message Queuing Telemetry Transport is a lightweight and efficient messaging protocol that is widely used in the Internet of Things (IoT) and other low-power networked devices. So, we can say that MQTT connecting anything anywhere anytime. To make a bare metal system talk to the world using MQTT, you will need to implement the following steps:

1. **Choose an MQTT broker:**

   An MQTT broker is the central messaging server that is responsible for receiving and distributing messages between the bare metal system and other devices on the network. There are many public and private MQTT brokers available, such as Mosquitto, AWS IoT Core, Thingspeak, and Azure IoT Hub.

2. **Choose an MQTT client library:**

   An MQTT client library provides the necessary functions and APIs to send and receive MQTT messages over the network. There are many MQTT client libraries available for different programming languages and platforms, such as Paho MQTT for C/C++ and Eclipse for Python.

3. **Implement the MQTT client code:**

   Once you have chosen an MQTT client library, you can implement the MQTT client code in your bare metal system. This code will typically include the following steps:

   a. Initialize the MQTT client library and connect to the MQTT broker
   b. Subscribe to specific MQTT topics to receive messages from other devices
   c. Publish MQTT messages to specific topics to send data to other devices
   d. Handle incoming MQTT messages and perform the necessary actions based on the message content.

4. **Integrate with the bare metal system:**

   To make the MQTT client code work with the bare metal system, you will need to integrate it with the other system components, such as the sensors, actuators, and other peripherals. This integration will typically involve reading data from the sensors, processing the data, and sending it to other devices using MQTT messages.

5. **Test and deploy the system:**

Once you have implemented and integrated the MQTT client code with the bare metal system, you can test the system by sending and receiving MQTT messages and verifying that the system is working as expected. You can then deploy the system in a real-world environment and monitor its performance and reliability over time.

Overall, making a bare metal system talk to the world using MQTT requires careful planning, implementation, and testing to ensure that the system is reliable and efficient. However, with the right tools and techniques, it is possible to build powerful and scalable IoT systems that can communicate seamlessly with other devices on the network.

How MQTT Works

We will Talk about the basic concepts needed to start working with MQTT and know how it works. We will start with the definitions of the most important concepts needed.

1. **Clients and Brokers:**

   MQTT uses a client-server architecture, where clients are the devices or applications that send and receive messages, and brokers are the servers that receive and distribute messages between clients. Clients can be publishers (sending messages) or subscribers (receiving messages).

2. **Topics:**

   MQTT messages are organized by topics, which are hierarchical strings that represent a specific category or type of message. Topics are used to filter and route messages between clients.

3. **QoS Levels:**

   MQTT supports three levels of Quality of Service (QoS), which determine the level of reliability and delivery guarantees for messages. The three levels are:

   - **QoS 0 (At most once):**

     Messages are delivered at most once, without any guarantee of delivery. This level is used for messages that are not critical or require real-time delivery.

   - **QoS 1 (At least once):**

     Messages are guaranteed to be delivered at least once, but may be delivered multiple times due to network errors or message loss. This level is used for messages that require reliable delivery, but can tolerate duplicates.

- **QoS 2 (Exactly once):**

  Messages are guaranteed to be delivered exactly once, with no duplicates or losses. This level is used for messages that require the highest level of reliability and accuracy.

4. **Message Flow:**

   The typical flow of MQTT messages involves the following steps:

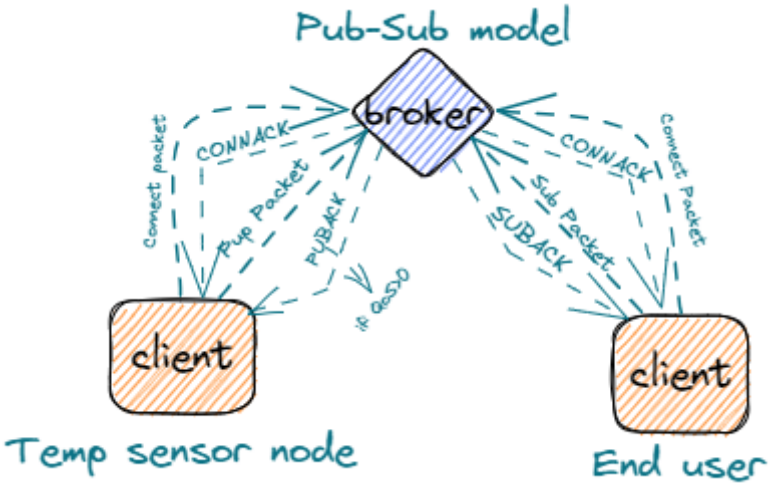
   - A client publishes a message to a specific topic, including the topic name, message payload, and QoS level.
   - The broker receives the message and stores it in a message queue for the specified topic.
   - The broker distributes the message to all subscribed clients for the topic, based on the QoS level and the availability of the clients.
   - The subscribed clients receive the message and process it according to their specific needs.

5. **Keep Alive:**

   MQTT includes a keep alive mechanism, where clients and brokers periodically exchange ping messages to ensure that their connection is still active and responsive.

   Overall, MQTT provides a simple and efficient way for devices and applications to communicate over a network, using lightweight and flexible messaging with a variety of QoS levels and topic-based routing.

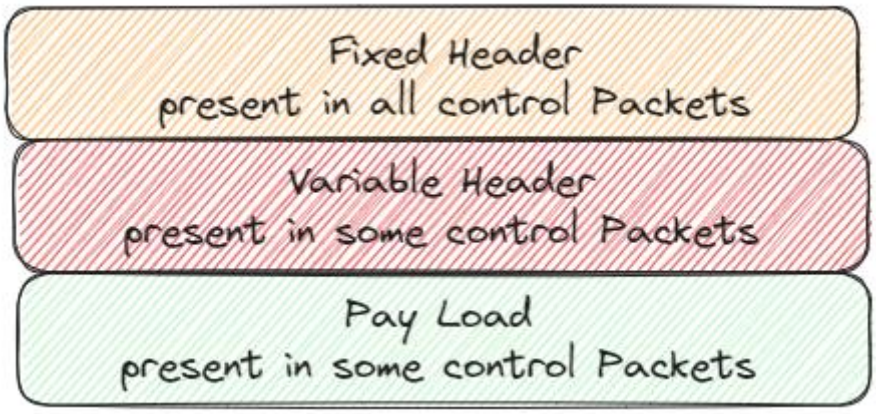
In embedded system we use QoS 0 if the information isn't important and use QoS 1 if the information is important and the subscriber must tell the broker max supported QoS. In the below figure you can see the complete PUBSUB model of MQTT.



We know how MQTT works and the important concepts that we need. And now we will have a dive in the packets that we will need to send data but first we will talk about the frame format of the MQTT packets. So, we will talk about the MQTT packet below.

MQTT Packet:

Also called control packets, used to send commands and data to broker. So, we can say that it's the way of the communication between client and broker. So, what are these packets consist of what are the different packets send what is the usage of each one we will talk about that below. Show the figure below to know the content of Packets.



Content of Packet

The packet consists of from one to three sections based on the packet type and usage. The three sections of packets are:

1. **Fixed Header:**

   Consist of two to five bytes. All packets must have the fixed header as it contains:

   1. **Packet Type:**

      The packet type is the MSB of first byte that holds number express the sent packet type. The table below list some of Packet Types

Name	Value	Direction of follow	Description
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Two Directions	Publish message
PUBACK	4	Two Directions	Publish acknowledgment
SUBSCRIBE	8	Client to Server	Client subscribes request
SUBACK	9	Server to Client	Subscribe acknowledgment

2. **Flags:** The 4 LSB of first byte and it's specific to each control packet.
3. **Remaining Length:**

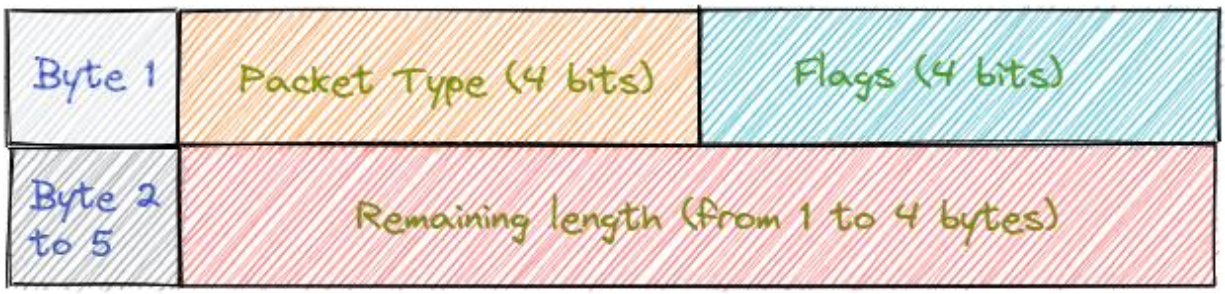
indicate the length of the variable header and payload. The Remaining Length field is a variable-length field that can range from 1 to 4 bytes, depending on the size of the packet. The format of the Remaining Length field is as follows:

- The first byte contains the least significant 7 bits of the Remaining Length value, with the most significant bit (bit 7) used as a continuation bit.
- If the continuation bit is set to 1, this indicates that there are additional bytes in the Remaining Length field. The next byte contains the next 7 bits of the value, with the continuation bit set to 1 if there are additional bytes.
- This process continues until the continuation bit is set to 0, indicating that the last byte of the Remaining Length field has been reached.

For example, if the Remaining Length field is 200, it would be encoded as follows:

- The first byte would be 0xC8 (11001000 in binary), indicating that the value is 200 and there are additional bytes in the field.
- The second byte would be 0x01 (00000001 in binary), indicating that the next 7 bits of the value are 1 and there are no additional bytes in the field.

The maximum value of the Remaining Length field is 268,435,455 (0xFFFFFF), which requires 4 bytes to encode. However, in practice, most MQTT packets have much smaller Remaining Length values, typically ranging from a few bytes to a few kilobytes. The complete fixed header is shown below.



2. **Variable Header:**

contains additional information about the MQTT packet beyond the fixed header. The structure of the variable header depends on the type of MQTT packet, but typically includes information such as the message ID, QoS level, topic name, and connection flags.

The most popular usage of variable header is message ID that used in publish and PUBACK packet when QoS is more than zero. The figure shown below show the ID message bytes

3. **Payload**

is the actual data that is being transmitted in the MQTT message. The payload is sent as part of the PUBLISH packet, which is used to send a message from a publisher client to the broker, or from the broker to a subscriber client.

We know about frame format of the packets and now we will talk about the control packets that we will use in our scope. So, let's start our journey with our control packets.

Control Packets

1. **CONNECT Packet**

The first packet sent by the client and it's used by a client to establish a connection with the broker after TCP connection is established. The CONNECT packet sent one time and if sent more than one time the connection is closed. The CONNECT packet has a fixed header, a variable header, which contains fields that specify various connection parameters, and payload. Below the frame format of connect packet in details.

1.1. Fixed Header

Bytes	Value	Description
Byte 1	0x10	The type is 1 and the flags are reserved
Bytes 2:5	X	The Remaining Length

The remaining length can be calculated from the below equation

$$Remaining\ Length\ =\ 10\ +\ 2\ +\ Client\ length\ +(2+\ username\ length\ if\ username)+(2+Password\ Length\ if\ Password)$$

1.2. Variable Header

Bytes	Value	Description
Bytes 1:2	0x04	The Protocol Name Size
Bytes 3:6	MQTT	The Protocol Name
Byte 7	0x04	The Protocol Level
Byte 8	X	Connect Flags
Bytes 9:10	X	KEEP ALIVE

- **Connect Flags:** set of flags that control the behavior of the connection. The below table Show usage of them

Bits	Usage	Description
Bit 0	Reserved	Must be 0
Bit 1	Clean Session	Clean the last messages sent when I was offline
Bit 2	Will Flag	Set the flag when there is a will message
Bits 3:4	QoS Retain	Send the will message with QoS x
Bit 5	Will Retain	Save the will message
Bit 6	Username flag	Set when Broker has username
Bit 7	Password Flag	Set when Broker has Password

1.3. **Payload:** The Payload consist of UTF-8 Encoded strings

Bytes	Value	Description
Bytes 1:2	X	The Client ID Length
Bytes 3: n	X	The Client ID as String
Bytes n: n+1	X	The Username Length
Bytes n+2: k	X	The Username as String
Bytes k: k+1	X	The Password Length
Bytes k+2: L	X	The Password as String

2. CONNACK Packet

used to acknowledge the receipt of a CONNECT packet from a client to a broker. When a client sends a CONNECT packet to a broker to establish a connection, the broker sends a CONNACK packet to the client indicating whether the connection request has been accepted or rejected.

It consists of fixed and variable headers as follow

2.1. Fixed Header

Bytes	Value	Description
Byte 1	0x20	The type is 2 and the flags are reserved
Byte 2	0x02	The Remaining Length, the length of variable header

2.2. Variable Header

Bytes	Value	Description
Byte 1	0x00	Reserved
Byte 2	X	Connect return code that indicates connection state

❖ Connect return code

Value	Connection State
0x00	Connection Accepted
0x01	Connection Refused, unacceptable protocol version
0x02	Connection Refused, identifier rejected
0x03	Connection Refused, server unavailable
0x04	Connection Refused, bad user name or password

3. PUBLISH Packet

used in the MQTT protocol to publish a message from a client to a broker or from a broker to a client. It is the most commonly used MQTT packet type and is used to send and receive messages in the MQTT network. It consists of fixed header, variable header, and payload.

3.1. Fixed Header

Bytes	Value	Description
Byte 1	0x3X	The type is 3 and the flags can be used
Bytes 2:5	X	The Remaining Length

❖ **The flags**

Bit	Set	Reset
DUP	re-delivery of the pub packet	The first trail to send the packet
QoS Level	QoS 0	QoS 1 or 2
RETAIN	The server saves the packet until the subscriber need it	The server saves the last recently sent packet before it

$$RemainingLength = 2 + Topic\ Length + Message\ Length + (2\ if\ QoS > 0)$$

3.2. Variable Header

Bytes	Value	Description
Bytes 1:2	X	Topic Length
Bytes 3: n	X	Topic Name
Bytes n+1: n+2	X	Packet ID used when $\boldsymbol{QoS > 0}$

3.3. **Payload:** actual data that is being transmitted in message.

4. PUBACK Packet

used to acknowledge the receipt of a PUBLISH packet from a client to a broker. When a client sends a PUBLISH packet to a broker, the broker sends a PUBACK packet to the client to acknowledge that the message has been received and processed successfully. Sent when $\boldsymbol{QoS > 0}$.

4.1. Fixed Header

Bytes	Value	Description
Byte 1	0x40	The type is 4 and the flags are reserved
Byte 2	0x02	The Remaining Length, the length of variable header

4.2. Variable Header: The Packet ID being Acknowledged.

5. SUBSCRIBE Packet

used in the MQTT protocol to subscribe a client to one or more topics on the broker. When a client sends a SUBSCRIBE packet to the broker, it requests to receive messages published to the specified topics.

5.1. Fixed Header

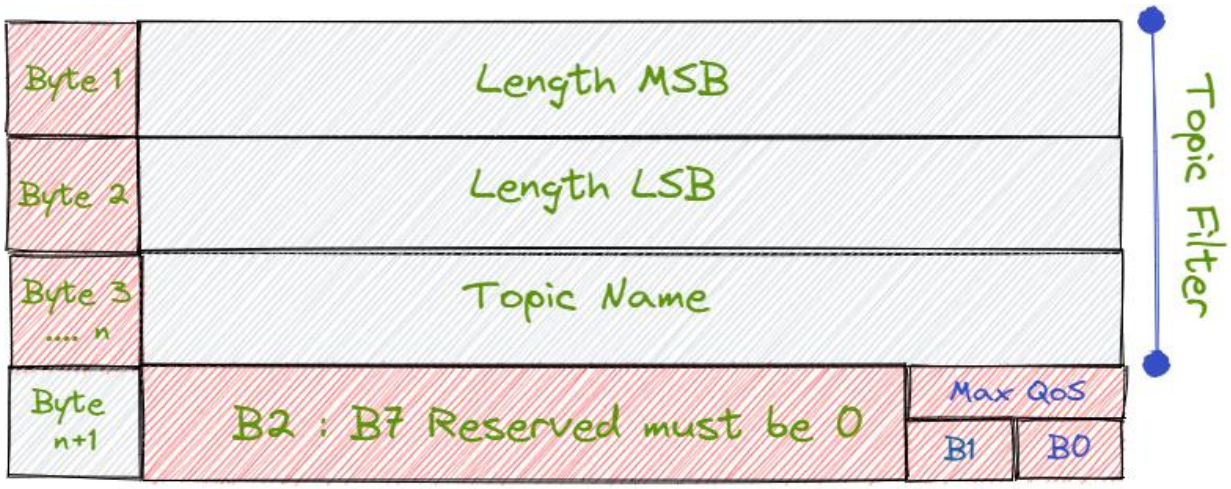
Bytes	Value	Description
Byte 1	0x82	The type is 8 and the flags are reserved
Byte 2	X	The Remaining Length

$$RemainingLength = 2 + (3 + Topic\ Length) * Number\ of\ Topics$$

5.2. Variable Header: The Packet ID used in SUBACK

5.3. Payload:

contains one or more topic filters and their corresponding QoS levels. The topic filters are strings that identify the topics to which the client wants to subscribe, and the QoS levels are the desired quality of service levels for the subscriptions. The figure below shows the payload.



6. SUBACK:

used to acknowledge the receipt of a SUBSCRIBE packet from a client to a broker. When a client sends a SUBSCRIBE packet to the broker to subscribe to one or more topics, the broker sends a SUBACK packet to the client indicating the status of the subscription request.

6.1. Fixed Header:

Bytes	Value	Description
Byte 1	0x90	The type is 9 and the flags are reserved
Byte 2	X	The Remaining Length, the length of variable header

$$RemaingLength = 2 + Number\ of\ topic\ Filters\ being\ Acked$$

6.2. Variable Header: The Packet ID of SUBSCRIBE Packet

6.3. Payload: Contains the Return codes of SUB topic filters

- **Return Codes:**

Value	Connection State
0x00	Success - Maximum QoS 0
0x01	Success - Maximum QoS 1
0x02	Success - Maximum QoS 2
0x80	Failure

Conclusion

In conclusion, MQTT is a powerful messaging protocol that can enable communication between bare metal devices and the outside world. By utilizing a lightweight and efficient approach to message transmission, MQTT can facilitate real-time communication and data exchange between devices, even in low-bandwidth or unstable network environments.

To make bare metal devices talk to the world using MQTT, you would need to first establish a connection to the MQTT broker server. This can be achieved through a variety of means, such as using an MQTT client library or implementing the protocol directly in your device's firmware.

Once connected, your device can subscribe to specific topics on the broker to receive messages or publish messages to specific topics to communicate with other devices or applications. This can enable various use cases, such as IoT applications, sensor networks, and machine-to-machine communication.

Overall, MQTT offers a flexible and scalable solution for bare metal devices to communicate with the world, and its popularity continues to grow as more and more applications require real-time communication and data exchange.

Future Plans

1. **Data Collection and Analysis:**

   The system should be able to collect data from the sensors and analyze it to provide insights into the farming process. This includes analyzing soil moisture levels, temperature, and humidity to determine the optimal conditions for crop growth.

2. **Crop Management:**

   The system should be able to manage crop growth, including monitoring growth rates, identifying pests and diseases, and providing recommendations for fertilizer and pesticide applications.

3. **Livestock Monitoring:**

   The system should be able to monitor the health and well-being of livestock, including tracking their movements, feeding schedules, and health indicators.

4. **Mobile App:**

   The system should include a mobile application that allows farmers to monitor and control their farming operations from anywhere.

5. **Internal Monitor and Control System:**

   The system should include internal control unit that allows farmers monitor and control the farm internally from the farm itself.