Mansoura University

Faculty of Engineering

Computer Engineering and Control Systems Department

Graduation Project Handbook

OTA Update for smart vehicles

Supervisor

Dr. Hesham Helmy Gad

Students

1. Belal Mohamed Abd Elhamid Elmahdy

2. Sahar Hossam Elnagar

3. Nada Gamal Elsayed

4. Hager Mohamed Mohamed

5. Yomna Mohamed Mokhtar

Contents

Chapter 0

Introduction

OTA technology has increased in significance, as mobile devices evolve and applications emerge. Mobile operators and telecommunication third parties can send OTA updates through SMS to configure data updates in SIM cards; distribute system updates; or access services, such as wireless access protocol (WAP) or multimedia messaging service (MMS). OTA updates also enable mobile operators to activate user subscriptions. OEMs can use OTA updates to fix bugs through firmware and change the user interface.

The proliferation of IoT has led manufacturers to use OTA updates for autonomous vehicles, smart home speakers and other IoT devices.

An over-the-air update : is the wireless delivery of new software or data to mobile devices.

Wireless carriers and original equipment manufacturers (OEMs) typically use over-the-air (OTA) updates to deploy firmware and configure phones for use on their networks. The initialization of a newly purchased phone, for example, requires an over-the-air update. With the rise of smartphones, tablets and internet of things (IoT) devices, carriers and manufacturers have also turned to over-the-air updates for deploying new operating systems (OSes) to these devices.

0.1 Problem Statement

During the past twenty-five years, computer-based electronic control units (ECUs) have gradually replaced many of the mechanical and pneumatic control systems in vehicles. A 2013 study released by Frost & Sullivan found that mass market cars by then had at least 20 million to 30 million lines of software code, while premium cars could have as much as 100 million lines controlling essential systems. According to Frost & Sullivan, the average cost of the software code is $10 per line and it is steadily increasing. They estimate that by 2020 that the amount of software will increase by as much as 50 percent.

It is essential for vehicle OEMs to manage the software efficiently over the lifecycle of the vehicle, both to provide improvements in performance and to deliver corrections to faulty software that endanger lives or the environment and which could result in expensive product recalls. It is estimated that between 60 and 70 per cent of vehicle recalls in North America and Europe today are due to software problems, so this issue is clearly one that must be addressed aggressively by the OEMs. A topical case in point is Volkswagen and the eleven million diesel vehicles it has sold during the past eight years with 'faulty' emissions control software. A portion of these vehicles also require hardware changes, but if VW could have corrected the problem with only a software update, the process would have taken far less time, cost much less and reduced the environmental impact.

Over-the-air updating is already in use among some vehicle OEMs (e.g. Tesla and Mercedes-Benz) as an alternative to performing the software updates using a vehicle workshop system, however this practice is still very new and limited. In a future OTA scenario, whether the updates are performed in the workshop or at the dealer with OTA technology, or remotely, there remain both technical and business reasons for using the dealer network for performing the updates. Unlike a company like Tesla Motors, which sells cars directly to customers and which does not have a dealer network, vehicle OEMs are dependent on their dealers and National Sales

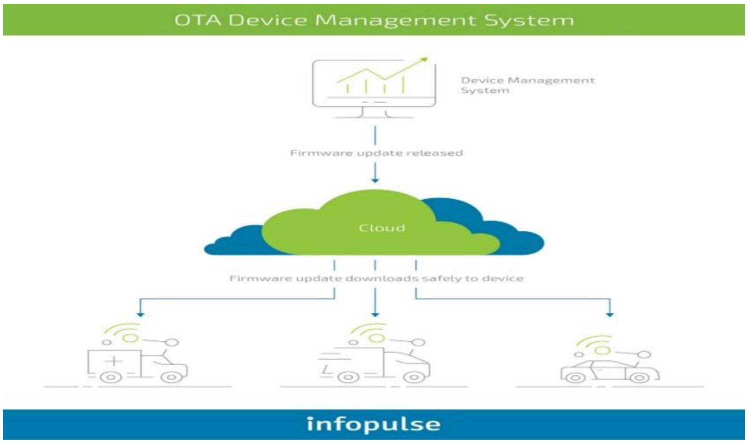
Companies for customer contact information. Many, if not most, car OEMs do not have a central database containing the names and most recent contact information on the owners or drivers of their vehicles. Dealers with workshops want to continue to have the direct relationships with customers in order to sell service and accessories, and to eventually sell the customer a new car. They will resist any attempt to short circuit them.

0.2 Over-The-Air (OTA) updates for Vehicles

As shown in Figure(0-1), the advantages of being able to perform in-the-field software updates to cars are well established: it will save manufacturers money, enable critical bugs to be patched immediately and allow compelling new features to be added to the vehicle at any time during its lifecycle.

At present however, few vehicles actually have the ability to receive updates. Even the cars that do support Over-The-Air (OTA) updates are only designed to update the infotainment or telematics systems. If an OEM discovers a fault in the firmware that controls a critical function such as the brakes or an airbag, then the vehicle has to be returned to the dealership. Electronic Control Units (ECUs) that control features like the engine, brakes, steering are deeper within the vehicle network architecture, and are typically Microcontrollers (MCUs) with small amounts of embedded flash and RAM. This brings constraints which require a different update approach.

Unlike a mobile phone or PC, car owners will not tolerate downtime of their vehicles while updates take place. Therefore, updates critical to vehicle operation should ideally take place seamlessly and invisibly in the background.


OTA Device Management System
Device Management System
Firmware update released
Cloud
Firmware update downloads safely to device
infopulse

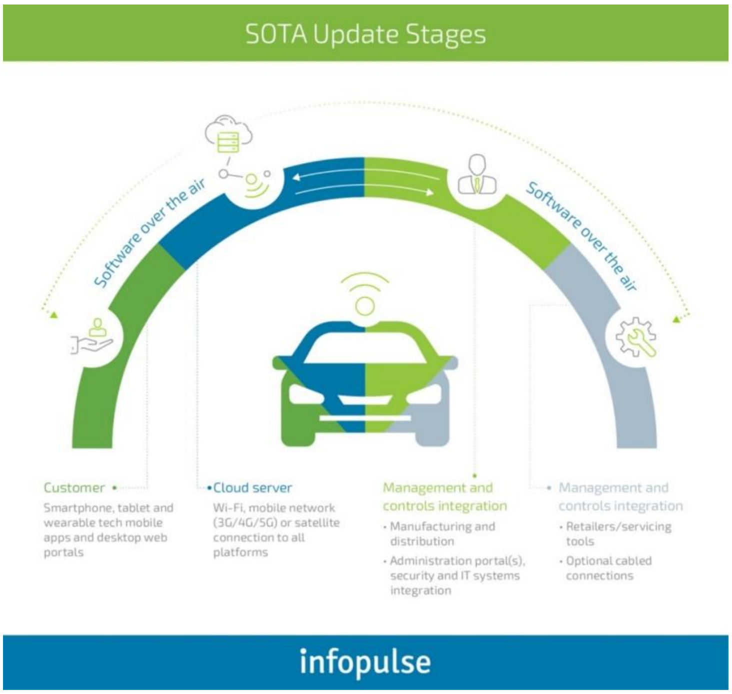

SOTA Update Stages
Software over the air
Software over the air
Customer
Smartphone, tablet and wearable tech mobile apps and desktop web portals
Cloud server
Wi-Fi, mobile network (3G/4G/5G) or satellite connection to all platforms
Management and controls integration
• Manufacturing and distribution
• Administration portal(s), security and IT systems integration
Management and controls integration
• Retailers/servicing tools
• Optional cabled connections
infopulse

Figure (0-1)

FOTA *VS.* SOTA :

Firmware and Software Release Packages are versioned separately, but they do interact with each other, hence dependencies between the two need to be considered.

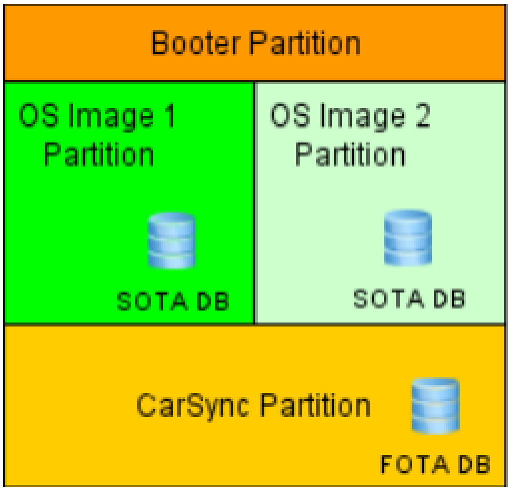
A Software Release Package describes the state of the software components that make up certain release.

Firmware Release Package of a component that is versioned with Software RPs has to be considered as a frozen snapshot of the Software RP.

•Update Manager must keep track of replaced versions of firmware and software packages for the use in offline rollback scenario. This requirement implies that considerable storage may be required.

•For the purpose of firmware update, it is required to have at least 2 partitions for keeping firmware versions, as shown in Figure(0-2).

•The bootloader shall be able to boot either partition as instructed.



Figure(0-2)

0.3 Major Challenges

Based on this high level description of the OTA update process, three major challenges arise that the OTA update solution must address. The first challenge relates to memory. The software solution must organize the new software application into volatile or nonvolatile memory of the client device so that it can be executed when the update process completes. The solution must ensure that a previous version of the software is kept as a fallback application in case the new software has problems. Also, we must retain the state of the client device between resets and power cycles, such as the version of the software we are currently running, and where it is in memory. The second major challenge is communication. The new software must be sent from the server to the client in discrete packets, each targeting a specific address in the client's memory. The scheme for packetizing, the packet structure, and the protocol used to transfer the data must all be accounted for in the software design. The final major challenge is security. With the new software being sent wirelessly from the server to the client, we must ensure that the server is a trusted party. This security challenge is known as authentication. We also must ensure that the new software is obfuscated to any observers, since it may contain sensitive information. This security challenge is known as confidentiality. The final element of security is integrity, ensuring that the new software is not corrupted when it is sent over the air.

Memory and other challenges:

In order to achieve reliability and stability when remotely updating ECUs additional RAM and Flash must be added to one or more of the ECUs within the vehicle; particularly if the process is transparent to the vehicle operations.

The Cloud Server distributing the releases must be design and implement as configuration management system that act as a distribution center with proper intelligence and knowledge.

-The Vehicles must have built-in intelligence to be capable of receiving and distributing incoming ECU releases

◦ This is particularly true if the ECUs are expected to have rollback capabilities and/or can receive updates via one-way communication mechanisms such as Satellite Radio or USB Drives

-The following functionality must be exposed by the bootloader:

◦ Means of reading the currently running software version

◦ Means of changing the software version to be run after reboot

◦ Means of reading whether an error has occurred when booting the firmware

-Partitioning and storage requirements :

◦ Update Manager must keep track of replaced versions of firmware and software packages for the use in offline rollback scenario. This requirement implies that considerable storage may be required.

-For the purpose of firmware update, it is required to have a least 2 partitions for keeping firmware versions.

◦ The bootloader shall be able to boot either partition as instructed.

0.4 OTA Requirements

Safety

◦ no image can be downloaded until it has been verified as the correct image for the specific ECU subcomponent; utilizing CRCs, checksums, and other release version compatibilities used.

Security

◦ data should be sent utilizing security protocol and encryption mechanisms.

Reliability

◦ should utilize data aggregation over one or more connection sequences (with no loss of data) with final reassembly done only after aggregation has completed.

Scalability

◦ Should be design to support large number of vehicles by providing multicast and broadcast capabilities in addition to unicast.

Flexibility

◦ Should support national and regional requirements.

◦ Should provide an efficient mechanism to support multiple car models and car groups.

0.5 OTA Update Benefits

1. **Cost efficient**. OEMs can seamlessly manage firmware updates across a fleet of IoT devices from one unified interface. The costs significantly decrease over the entire lifecycle of a car.

2. **Continuous improvements**. Bugs can be fixed and product behavior can be enhanced after the device lands in the hands of your consumers. This can potentially eliminate costly recalls and in-person maintenance.

3. **Improved scalability**. FOTA updates enable manufacturers to add new features to infrastructure after the release without physical access to upgrade firmware.

4. **Faster time-to-market**. Developers can test new features on selected devices and deploy frequently knowing that the products will remain stable. Firmware updates can be

dispatched even while the vehicle is still on the production line or in dealer center.

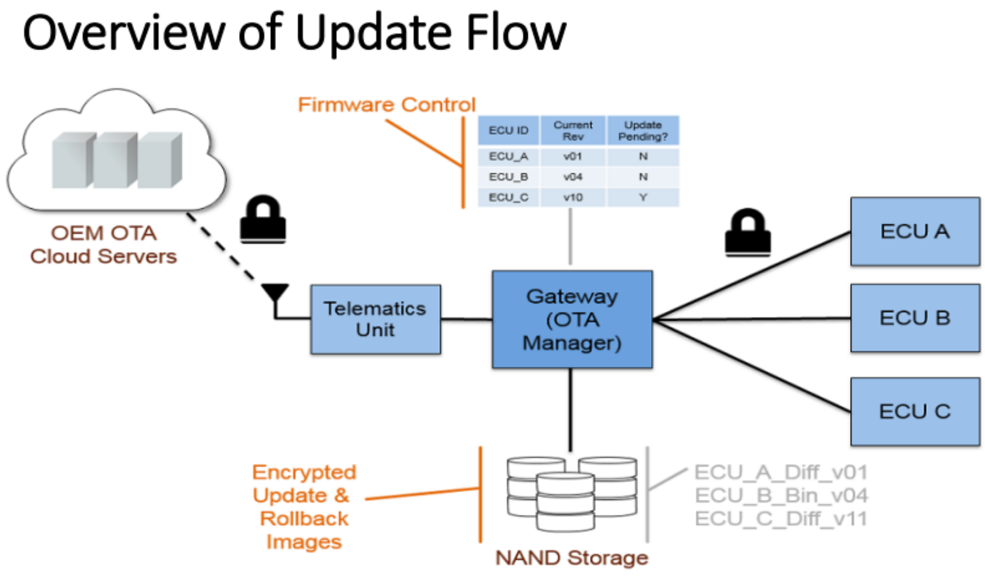
5. **Improved safety and compliance.** OEMs can use SOTA updates to patch the known vulnerabilities, e.g. defective adapter plugs, instead of recalling the vehicles. Your business can respond to the legal and regulatory responsibilities faster and more cost-effectively.

6. **Better software quality.** SOTA updates allow you to continuously amend your systems. Whenever you discover a new opportunity for improvement (e.g., a way to reduce vehicles fuel consumption), you can instantly deliver it to customers, instead of waiting to incorporate it in a new batch of vehicles.

7. **Timely updates.** Ensure that your users are receiving the latest products from your business — new add-ons to the infotainment systems or navigation. For autonomous driving, regular OTA software updates will become crucial to ensure safe and smooth navigation and routing.

8. **Two-way communication.** OTA data exchanges can happen both ways. By collecting data on the vehicle usage or performance and deploying analytics tools, OEMs can promote better customer experience and show customers that they care about them by issuing regular system updates. Additionally, you can gather intel for R&D and ensure preventive maintenance. Finally, you can also monetize your in-vehicle generated data using Caruso Dataplace — an innovative data marketplace for the mobility ecosystem. And you can enrich your solutions with additional data sources or white-label solutions purchased through this service.

CHAPTER ONE
SYSTEM COMPONENTS

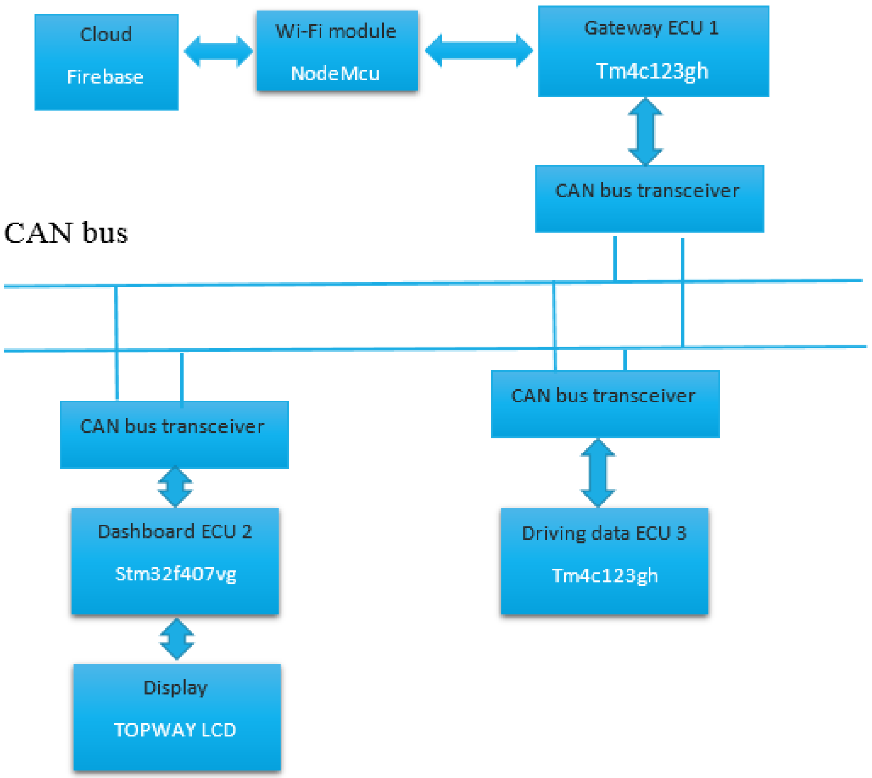


Figure(1-1)

1.1 Electronic equipment

- Tm4c123GH microcontroller.
- Stm32f407VG microcontroller.
- NodeMCU.
- CAN bus transceiver.
- TOPWAY Smart LCD.

1.2 Hardware Structure

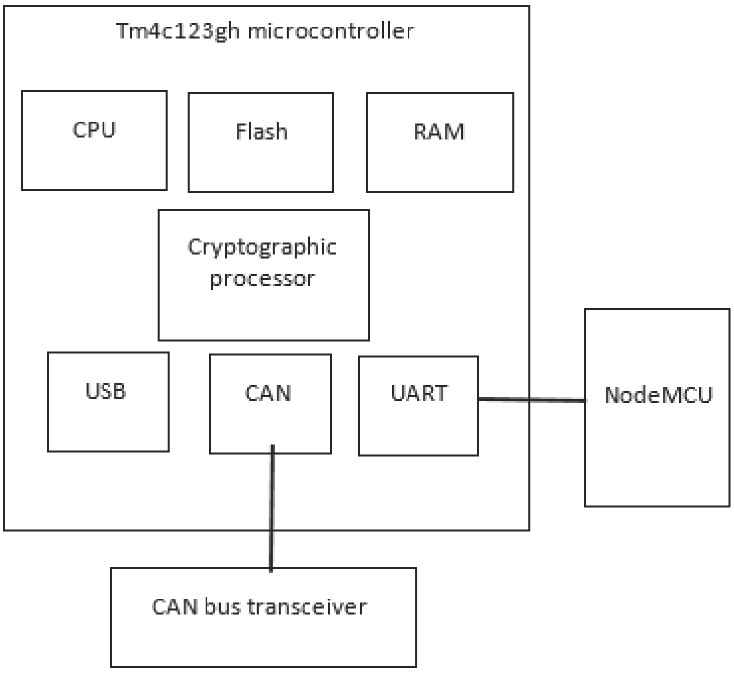


Figure(1-2)

1.2.1 ECU 1(The gateway):

As shown in figure(1-3) TM4C123gh microcontroller is connected to:

- NodeMCU to connect to the OEM cloud (from which the car will receive the update) through Wi-Fi.
- CAN bus transceiver to distribute the update to other ECUs.
- It is also contains decryption algorithm enables to develop high performance, data protected application through secure cloud connection.

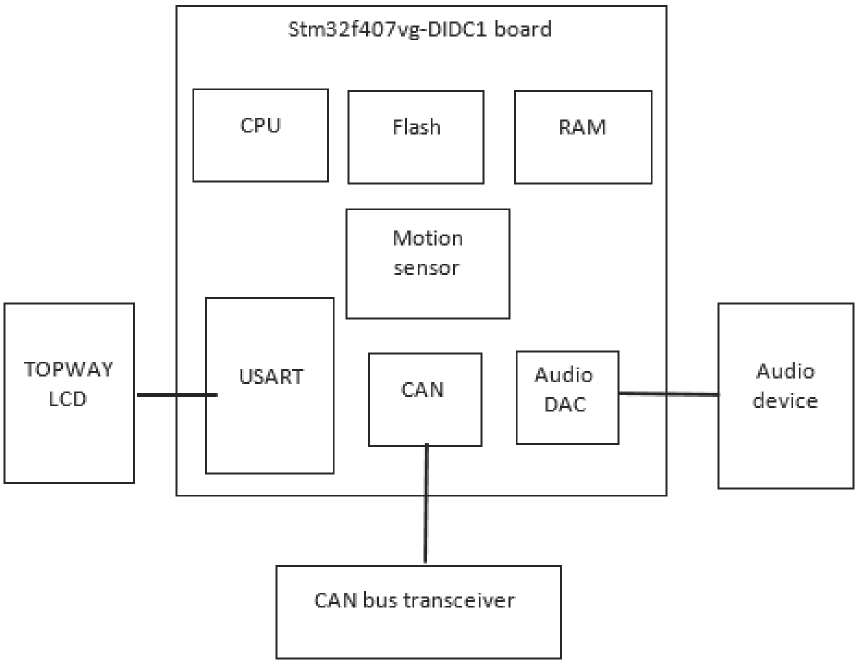


Figure(1-3)

1.2.2 ECU 2(dashboard):

As shown in figure(1-4) STM32F407VG microcontroller is connected to:

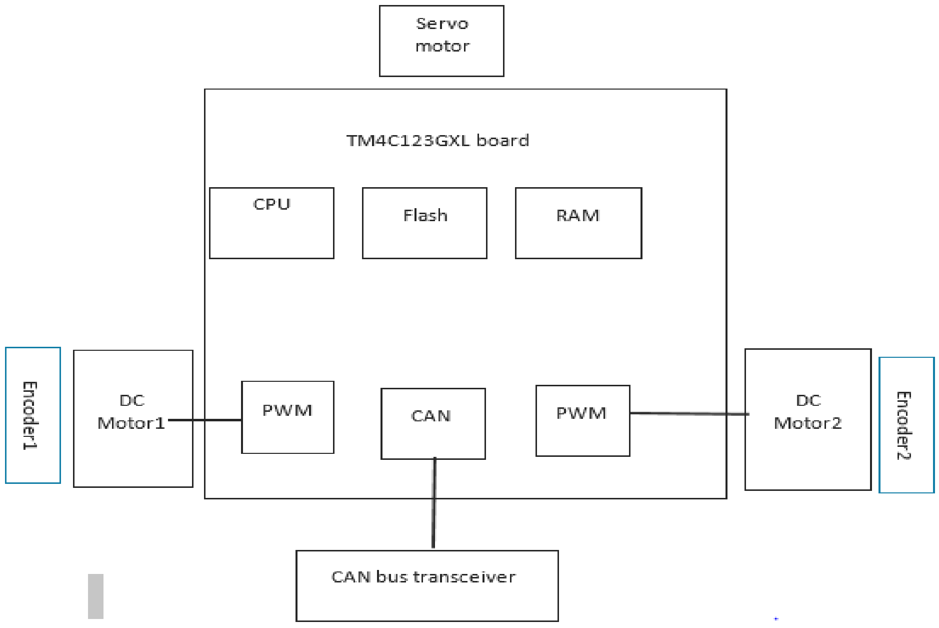
- TOPWAY LCD to display driving data and the update process.
- CAN bus transceiver to receive the update from the gateway and receive the data to display.
- Audio device.



Figure(1-4)

1.2.3 ECU3 (steering and driving):

Figure(1-5) shows an Example for other ECUs in the car.



.

Figure(1-5)

CHAPTER TWO NODEMCU AND CLOUD CONNECTION

In industrial environments, it is not feasible to update remote devices using the traditional method, which involves connecting each embedded device to a PC with a cable especially in automotive. If not updated, the devices can miss out on critical security improvements and bug fixes.

Ota update is a mechanism for remotely updating internet-connected hardware with new settings, software, and/or firmware.

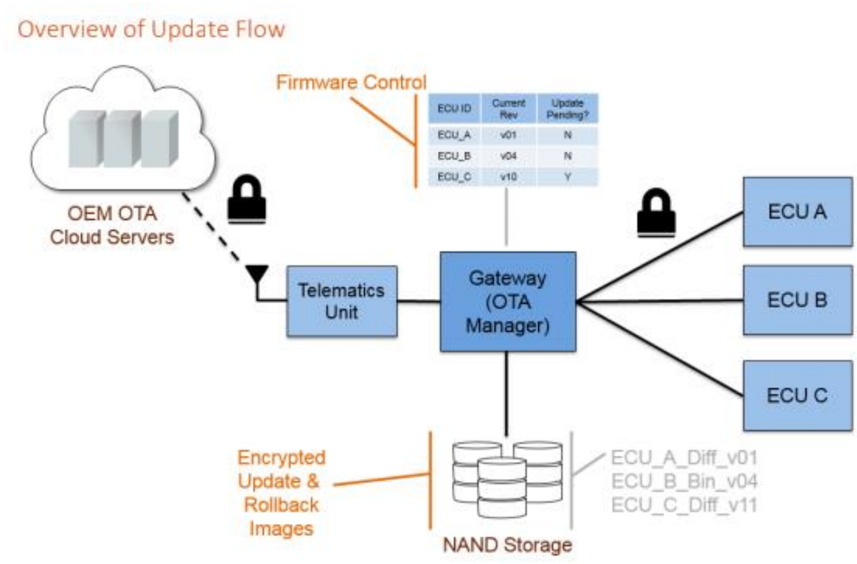
2.1 Different methods for OTA updates

There are different methods for OTA updates, but the most common are as follows:

1. Edge-to-cloud OTA updates: An ECU microcontroller installed in the vehicle can receive firmware OTA packages from a remote server. The package can include upgrades to both the microcontroller's underlying hardware capabilities (FOTA) and updates to applications running on those (SOTA).

2. Gateway-to-cloud OTA updates: An Internet-connected gateway (for instance, a telematics system), in charge of managing a set of local edge devices, can receive updates from a remote server. These updates can be aimed at improving all or some of the installed software applications, the app's host environment, and/or the gateway device's firmware.

3. Edge-to-gateway-to-cloud OTA updates: As shown in Figure(2-1), an internet-connected gateway manages a group of locally connected edge devices. These devices receive remote firmware updates via the gateway.



ECU ID	Current Rev	Update Pending?
ECU_A	v01	N
ECU_B	v04	N
ECU_C	v10	Y

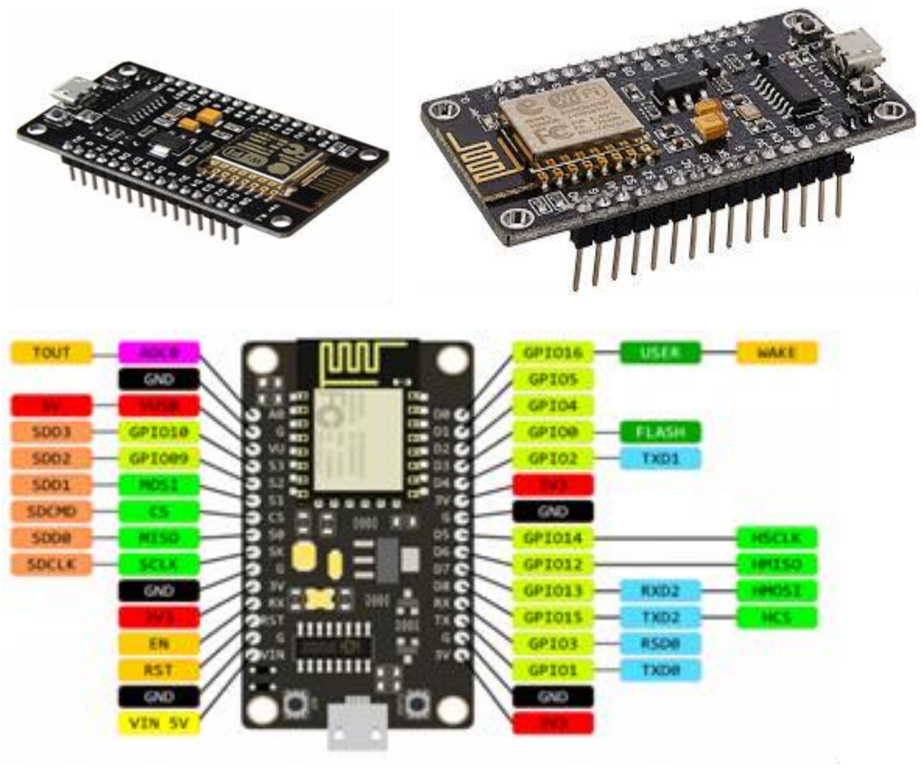
Figure(2-1)

In our project we are using **Edge-to-gateway-to-cloud OTA updates**, so we need a cloud connection to have internet-connected gateway to be able to upload firmware to the cloud and then download it with this gateway , then we will be able to send it to the central gateway and this is the ECU microcontroller installed in the vehicle to perform the update needed.

2.2 The Role of IoT Gateway in the OTA Update

A gateway is a bridge between edge devices and cloud; it manages traffic between networks that use different protocols. A cloud gateway performs protocol translation tasks. It also acts as a proxy server for devices on the field that do not feature wireless communication capabilities. The IoT gateway receives data from those devices and packages the information for transmission over TCP/IP.

In our project we are using an internet-connected gateway and a central gateway which is ECU microcontroller, that is why we choose Nodemcu -shown in Figure(2-2)- to represented in our project as internet-connected gateway.



Figure(2-2)

2.3 Let's talk about Nodemcu

NodeMCU is an open-source firmware and development kit that helps you to prototype or build IoT products. It includes firmware that runs on the ESP8266 Wi-Fi SoC from Espressif Systems, and hardware which is based on the ESP-12 module. The firmware uses the Arduino IDE.

Nodemcu is a Arduino based microcontroller with additional feature of ESP8266 Wifi chip with full TCP/IP stack, main advantage of Nodemcu is that it can directly connect to the internet without using any additional peripheral, and can connect to cloud using HTTP or MQTT protocols to send and receive data to help iot systems to be up to date and can monitor systems as well analyse data for the future developments.

We used **Nodemcu** to connect to firebase cloud using it's features to able to download firmware using **Nodemcu**.

Figure(2-3)

2.4 What is firebase and why we used it?

Firebase is a powerful platform for your mobile and web application. Firebase can power your app's backend, including data storage, user authentication, static hosting, and more.

Firebase lets you automatically run backend code in response to events triggered by Firebase features and HTTPS requests. Your code is stored in Google's cloud and runs in a managed environment. There's no need to manage and scale your own servers. It's supporting for Android, iOS, C++, Unity and Web Platform.

Firebase is google platform, free and easy to use, it have a real time database that we will explain later what we need it for.

2.5 Using Nodemcu with firebase platform

The main usage we need firebase for is uploading firmware file that we need to perform ota update, then connect with Nodemcu and downloading firmware.

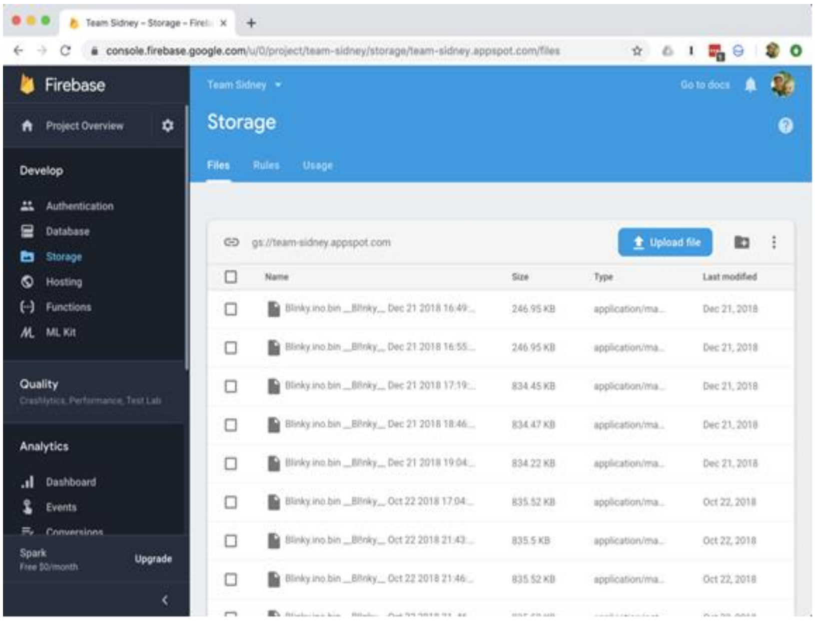
We use other firebase features like real time database , cloud messaging, data analysis and other features that we will mention and explain later.

2.5.1 Uploading a firmware file to Firebase

First, we need a way to upload a new firmware file to Firebase, so it can be downloaded by our Nodemcu over the Internet.

One of Firebase's key features is Firebase Cloud Storage, which lets you store files which can be retrieved over the Internet using a simple HTTP request.

A manual way to upload a new file to Firebase is to visit the Firebase console, click on "Storage" in the left nav bar, and then click on the "Upload file" button shown in Figure(2-4):

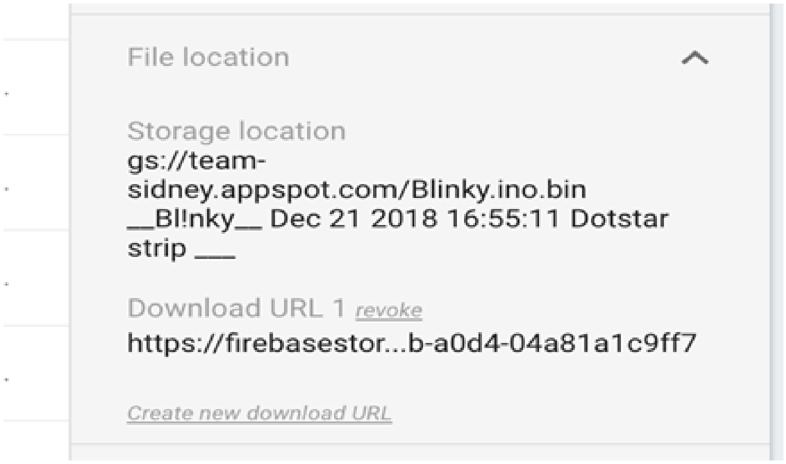


Name	Size	Type	Last modified
Blinky.ino.bin __Blinky__ Dec 21 2018 16:49:...	246.95 KB	application/ma...	Dec 21, 2018
Blinky.ino.bin __Blinky__ Dec 21 2018 16:55:...	246.95 KB	application/ma...	Dec 21, 2018
Blinky.ino.bin __Blinky__ Dec 21 2018 17:19:...	834.45 KB	application/ma...	Dec 21, 2018
Blinky.ino.bin __Blinky__ Dec 21 2018 18:46:...	834.47 KB	application/ma...	Dec 21, 2018
Blinky.ino.bin __Blinky__ Dec 21 2018 19:04:...	834.22 KB	application/ma...	Dec 21, 2018
Blinky.ino.bin __Blinky__ Oct 22 2018 17:04:...	835.52 KB	application/ma...	Oct 22, 2018
Blinky.ino.bin __Blinky__ Oct 22 2018 21:43:...	835.5 KB	application/ma...	Oct 22, 2018
Blinky.ino.bin __Blinky__ Oct 22 2018 21:46:...	835.52 KB	application/ma...	Oct 22, 2018

Figure(2-4)

2.5.2 Downloading a firmware file from Firebase

First, we need to get the URL for the file that we uploaded above. In the Firebase console, when you click on a file in the Cloud Storage list, look for the "Download URL" in the box that pops up to the right, as shown in Figure(2-5).



gs://team-sidney.appspot.com/Blinky.ino.bin
__Bl!nky__ Dec 21 2018 16:55:11 Dotstar strip ___

https://firebasestor...b-a0d4-04a81a1c9ff7

Figure(2-5)

This is going to give you a URL for the file we need to download, All we have to do is have our Nodemcu code pull the contents of this URL down over the Internet to download firmware file and store it in Nodemcu Memory.

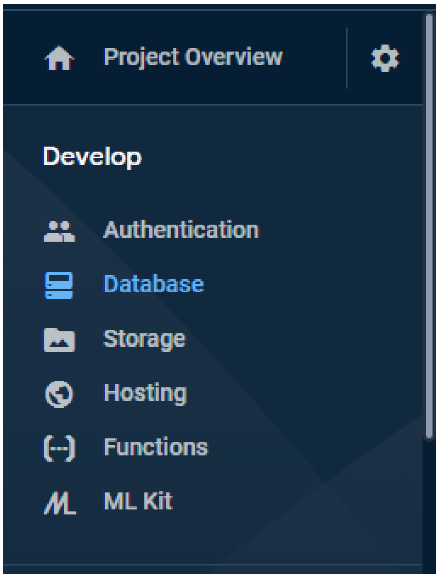
2.5.3 Using Real time database in our project

We discuss in last part that we need the file url to download it, so we can't include the url in our code every time we need to update since the Nodemcu will be installed in our vehicle.

That's why we need to use real time database to send a notification to the Nodemcu that there is a new update and also send the url needed to download the file let's see how we can do this in the next steps.

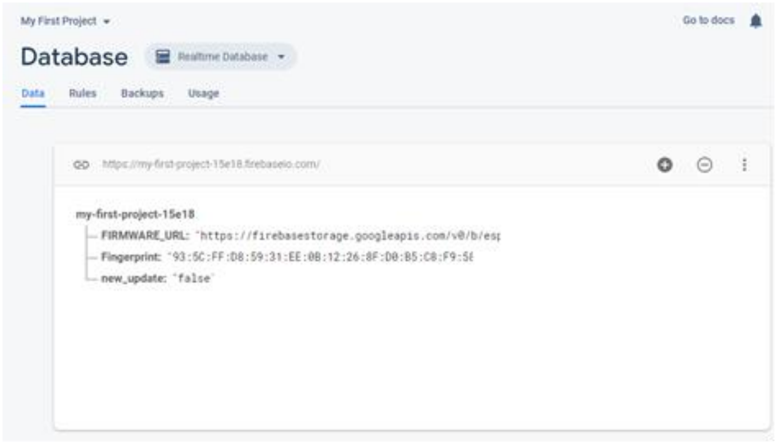
Let's explain this procedure on a simple example, we have a file uploaded on firebase cloud storage, so we have file's url.

First we will create a real time database from the left bar after creating a firebase account as shown in Figure(2-6).



Figure(2-6)

Second we will create a records in this database as we want and shown in the next picture for our example, we have a flag that tells Nodemcu to start receiving the url to start download the default will be “false” and then we will change it to “true” as shown in Figure(2-7).



```
my-first-project-15e18
├─ FIRMWARE_URL: "https://firebasestorage.googleapis.com/v0/b/es
├─ Fingerprint: "93:5C:FF:D8:59:31:EE:0B:12:26:8F:D0:B5:C8:F9:5
└─ new_update: "false"
```

Figure(2-7)

26 Firmware file AES Encryption

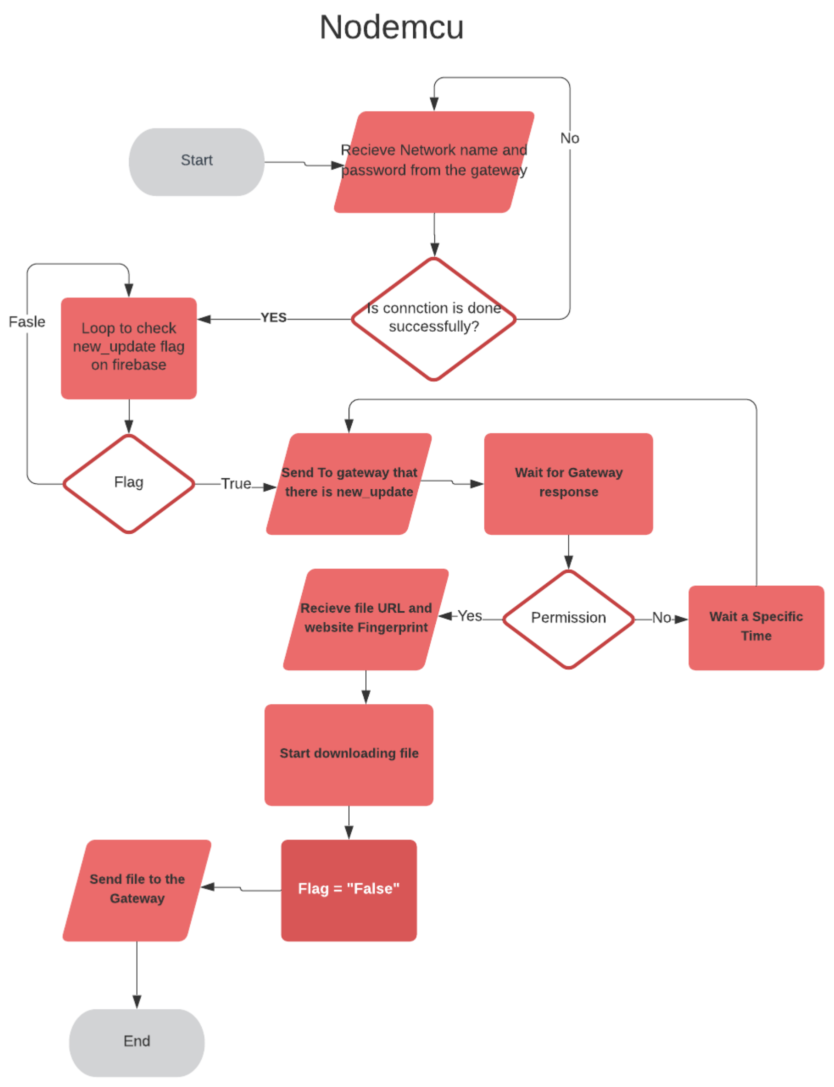
We explained previously that we need to upload the code file to Firebase cloud storage ,but we have to ensure that the code is safe and have a high security by encrypting the code file before uploading it and then the system download it.

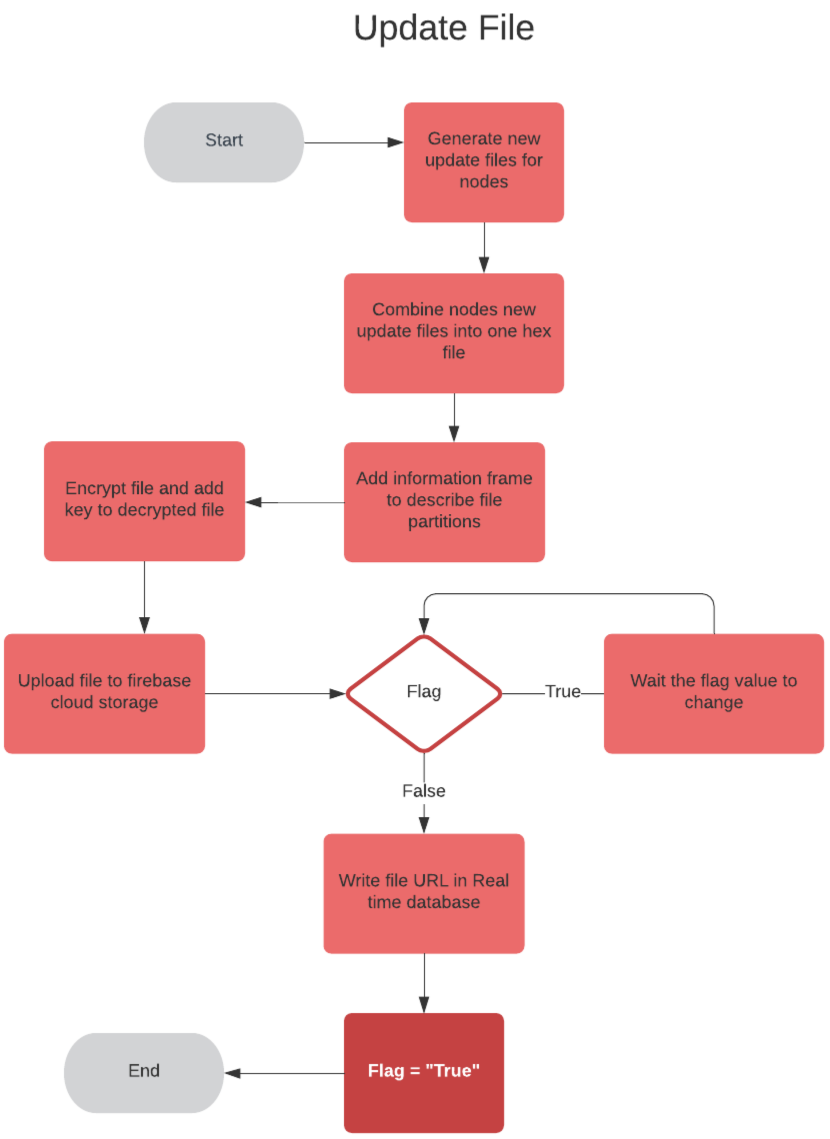
The algorithm we used to encrypt the file is AES 128 which refer to the Advanced Encryption Standard is a block cipher that uses shared secret encryption based on symmetric key algorithm.

AES-128 Features :

- Pure C library
- Input: 128-bit blocks
- Output: 128-bit blocks
- Key: 128 bits

2.7 IOT flow charts


Nodemcu
Start
Recieve Network name and password from the gateway
No
Is connction is done successfully?
YES
Loop to check new_update flag on firebase
Fasle
Flag
True
Send To gateway that there is new_update
Wait for Gateway response
Permission
Yes
No
Wait a Specific Time
Recieve file URL and website Fingerprint
Start downloading file
Flag = "False"
Send file to the Gateway
End


Update File
Start
Generate new update files for nodes
Combine nodes new update files into one hex file
Add information frame to describe file partitions
Encrypt file and add key to decrypted file
Upload file to firebase cloud storage
Flag
True
Wait the flag value to change
False
Write file URL in Real time database
Flag = "True"
End

CHAPTER THREE AUTOSAR MEMORY STACK

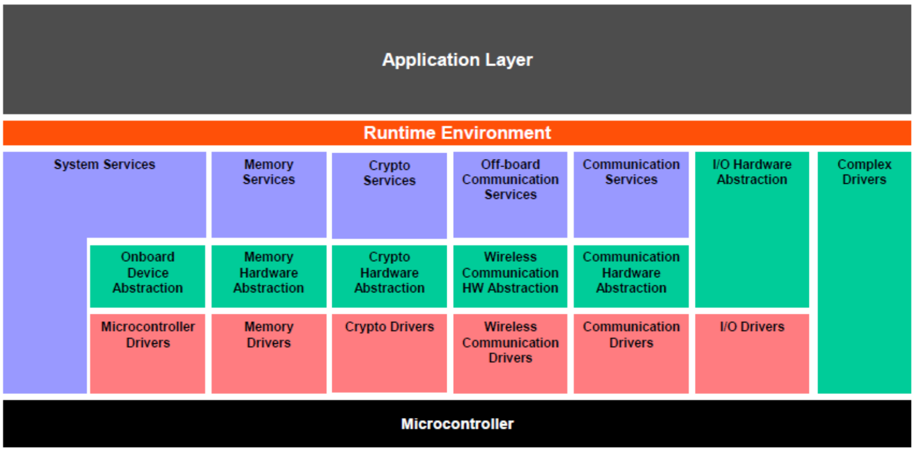
3.1 Introduction to AUTOSAR architecture:

AUTomotive Open System ARchitecture (**AUTOSAR**) is a global development partnership of automotive interested parties founded in 2003. It pursues the objective to create and establish an open and standardized software architecture for automotive electronic control units (ECUs). Goals include the scalability to different vehicle and platform variants, transferability of software, the consideration of availability and safety requirements, a collaboration between various partners, sustainable use of natural resources, and maintainability during the whole product lifecycle.

The AUTOSAR Architecture as shown in Figure(3-1) distinguishes on the highest abstraction level between three software layers: Application, Runtime Environment and Basic Software which run on a Microcontroller.

The AUTOSAR Basic Software layer is further divided in the layers: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers.

The Basic Software Layers are further divided into functional groups (Vertical Stacks). Examples of Services are System, Memory and Communication Services.



🟪 **Service Layer** 🟩 **ECU Abstraction Layer** 🟥 **Microcontroller Abstraction Layer**

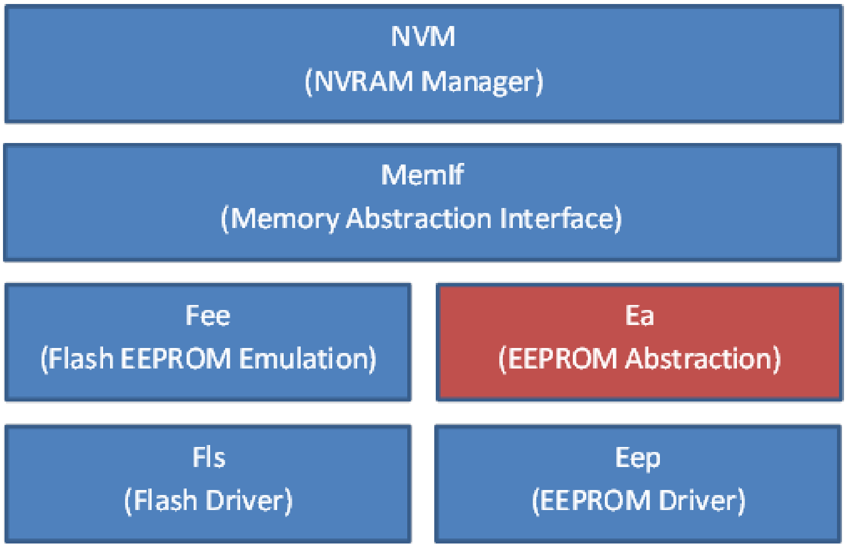
Figure(3-1)

In this project we are implementing the memory stack based on AUTOSAR Architecture, so Let's get into its details..

3.2 AUTOSAR Memory Stack:

Memory Stack provides basic memory management services to the upper Application layer and to the Basic Software Modules (BSW) of the AUTOSAR layered architecture.
The memory management services ensure access to the memory cluster, to the devices or software functions, for reading and writing data to non-volatile memory media like Flash or EEPROM.

Figure(3-2) shows various software modules and device drivers associated with AUTOSAR Memory Stack:



Figure(3-2)

Software Modules and Device Drivers in AUTOSAR MemStack :

Memory Stack in AUTOSAR layered architecture is a collection of software modules and device drivers.
Following is the list of modules in different layers of AUTOSAR:

- Non-Volatile Memory Manager (NvM) – it is part of the AUTOSAR Services Layer.
- Memory Interface (MemIf) – it is part of the AUTOSAR ECU Abstraction Layer.
- Flash EEPROM Emulation (Fee) – it is part of the AUTOSAR ECU Abstraction Layer.
- EEPROM Abstraction (Ea) – it is part of the AUTOSAR ECU Abstraction Layer.

- Flash Driver (Fls) – it is part of the AUTOSAR MCAL Layer.
- EEPROM Driver (Eep) – it is part of the AUTOSAR MCAL Layer.

Now, Let,s talk about the modules from down to top…

3.2.1 EEPROM Driver Module

The EEPROM driver provides services for reading, writing, erasing to/from an EEPROM. It also provides a service for comparing a data block in the EEPROM with a data block in the memory (e.g. RAM).

3.2.1.1 Further definitions of terms used throughout this module:

Data block	A data block may contain 1..n bytes and is used within the API of the EEPROM driver. Data blocks are passed with - Address offset in EEPROM - Pointer to memory location - Length to the EEPROM driver.
Data unit	The smallest data entity in EEPROM. The entities may differ for read/write/erase operation.

3.2.1.2 Functions Supported in this Module:

Initialization:

Eep_Init

Operations:

Eep _Erase
Eep _Write
Eep _Cancel
Eep _Read
Eep _Compare

Information:

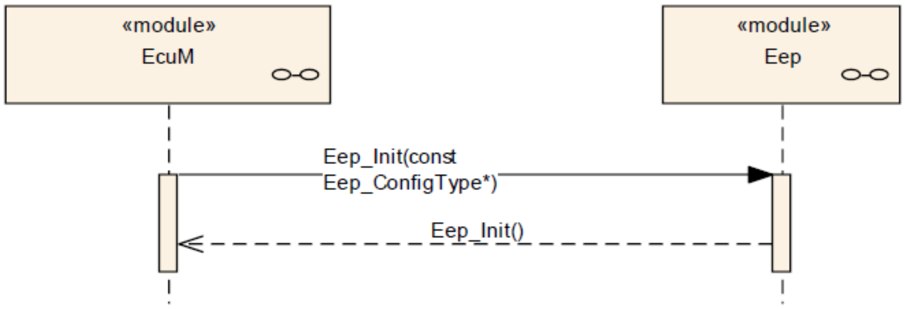
Eep _GetStatus
Eep _GetJobResult
Eep _SetMode
Eep _GetVersionInfo

Internal Operations:

Eep _MainFunction

3.2.1.3 Sequence Diagrams

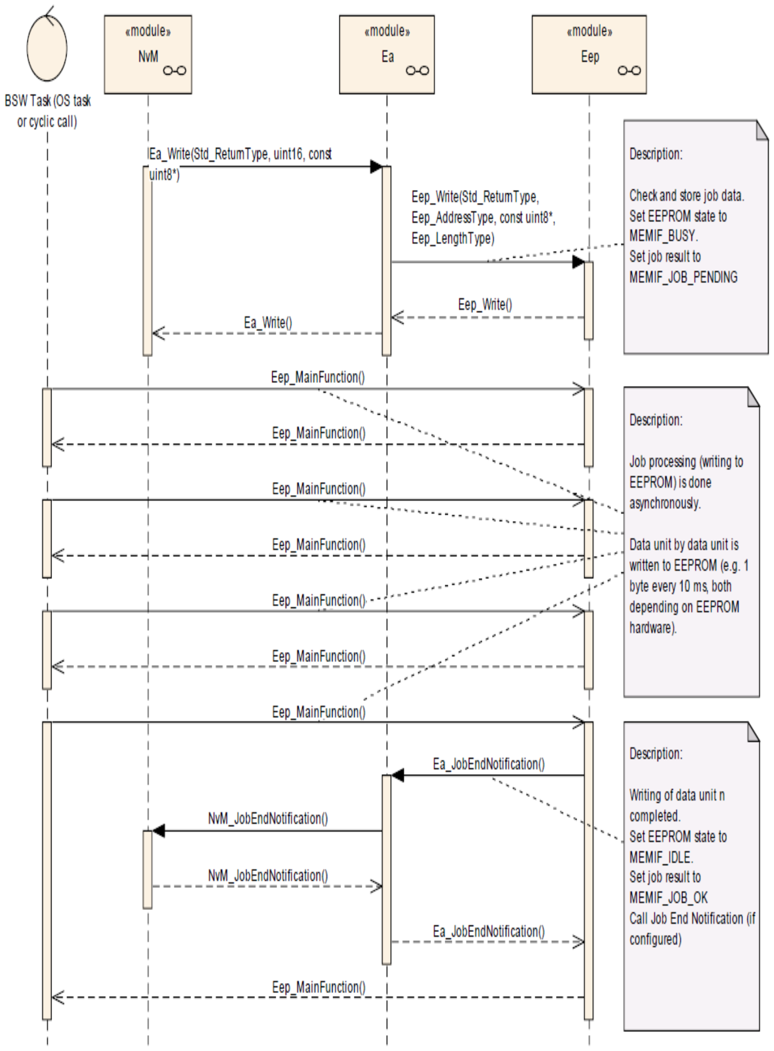
1. **Initialization**



Figure(3-3)

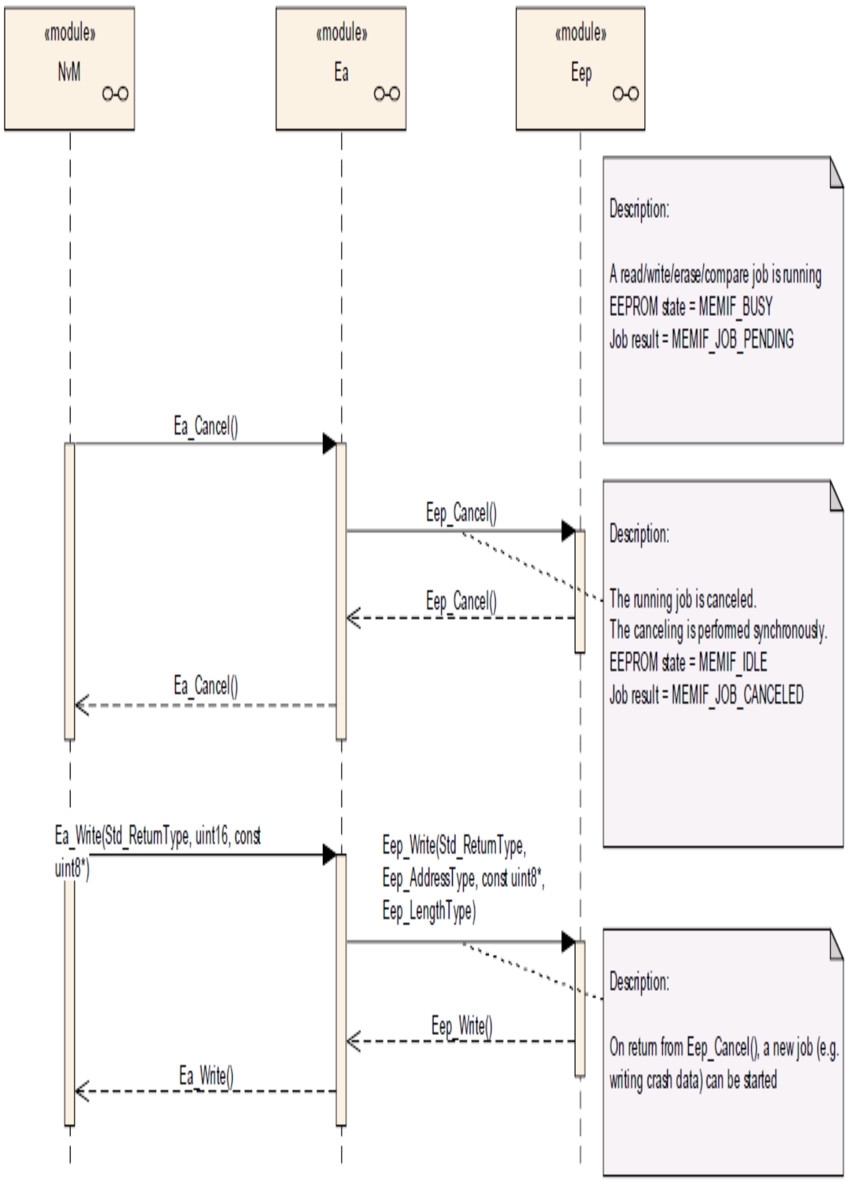
2. **Read/Write/Erase/Compare**

   Figure(3-4) shows the sequence diagram of write function as an example. The sequence for read, compare and erase is the same, only the processed block sizes may vary.


BSW Task (OS task or cyclic call)
«module» NvM
«module» Ea
«module» Eep
Ea_Write(Std_ReturnType, uint16, const uint8*)
Eep_Write(Std_ReturnType, Eep_AddressType, const uint8*, Eep_LengthType)
Eep_Write()
Ea_Write()
Description:
Check and store job data.
Set EEPROM state to MEMIF_BUSY.
Set job result to MEMIF_JOB_PENDING
Eep_MainFunction()
Description:
Job processing (writing to EEPROM) is done asynchronously.
Data unit by data unit is written to EEPROM (e.g. 1 byte every 10 ms, both depending on EEPROM hardware).
Ea_JobEndNotification()
NvM_JobEndNotification()
Description:
Writing of data unit n completed.
Set EEPROM state to MEMIF_IDLE.
Set job result to MEMIF_JOB_OK
Call Job End Notification (if configured)

Figure(3-4)

3. Cancelling of a running job


«module»
NvM
«module»
Ea
«module»
Eep
Description:
A read/write/erase/compare job is running
EEPROM state = MEMIF_BUSY
Job result = MEMIF_JOB_PENDING
Ea_Cancel()
Eep_Cancel()
Description:
The running job is canceled.
The canceling is perfomed synchronously.
EEPROM state = MEMIF_IDLE
Job result = MEMIF_JOB_CANCELED
Eep_Cancel()
Ea_Cancel()
Ea_Write(Std_ReturnType, uint16, const uint8*)
Eep_Write(Std_ReturnType, Eep_AddressType, const uint8*, Eep_LengthType)
Description:
On return from Eep_Cancel(), a new job (e.g. writing crash data) can be started
Eep_Write()
Ea_Write()

Figure(3-5)

3.2.2 Flash Driver Module

The flash driver provides services for reading, writing and erasing flash memory and a configuration interface for setting / resetting the write / erase protection if supported by the underlying hardware. It's used by the Flash EEPROM emulation module for writing data.

3.2.2.1 Further definitions of terms used throughout this module:

Flash Sector	A flash sector is the smallest amount of flash memory that can be erased in one pass. The size of the flash sector depends upon the flash technology and is therefore hardware dependent.
Flash page	A flash page is the smallest amount of flash memory that can be programmed in one pass. The size of the flash page depends upon the flash technology and is therefore hardware dependent.

3.2.2.2 Functions Supported in this Module:

Initialization:

Fls_Init

Operations:

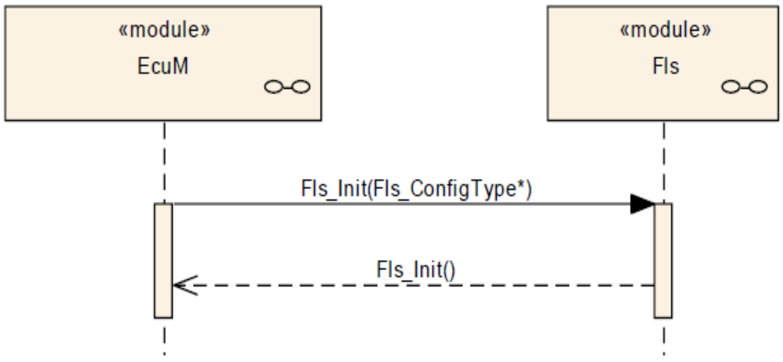
Fls_Erase
Fls_Write
Fls_Cancel
Fls_Read
Fls_Compare
Fls_BlankCheck

Information:
Fls_GetStatus
Fls_GetJobResult
Fls_SetMode
Fls_GetVersionInfo

Internal Operations:
Fls_MainFunction

3.2.2.3 Sequence Diagrams

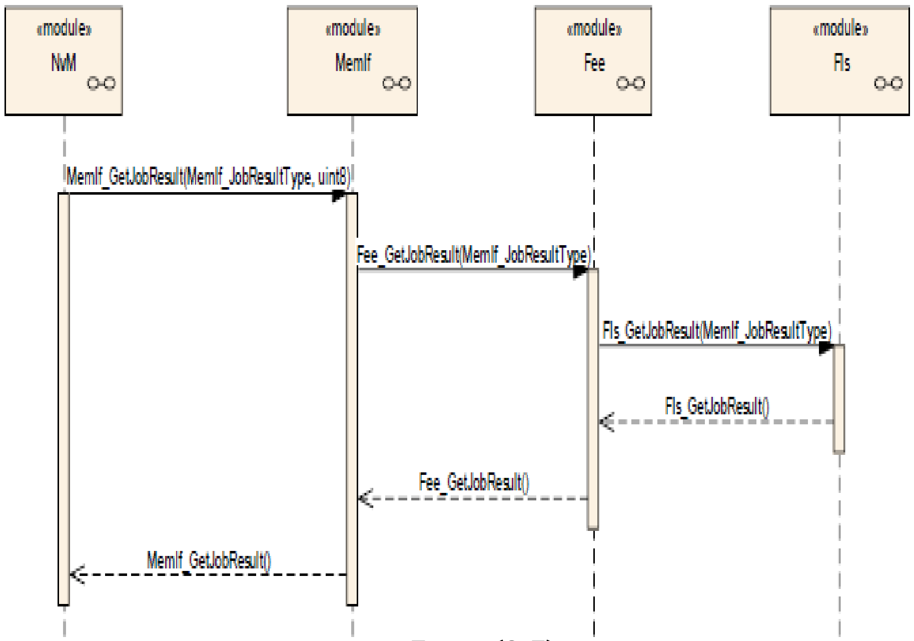
1. Initialization



Figure(3-6)

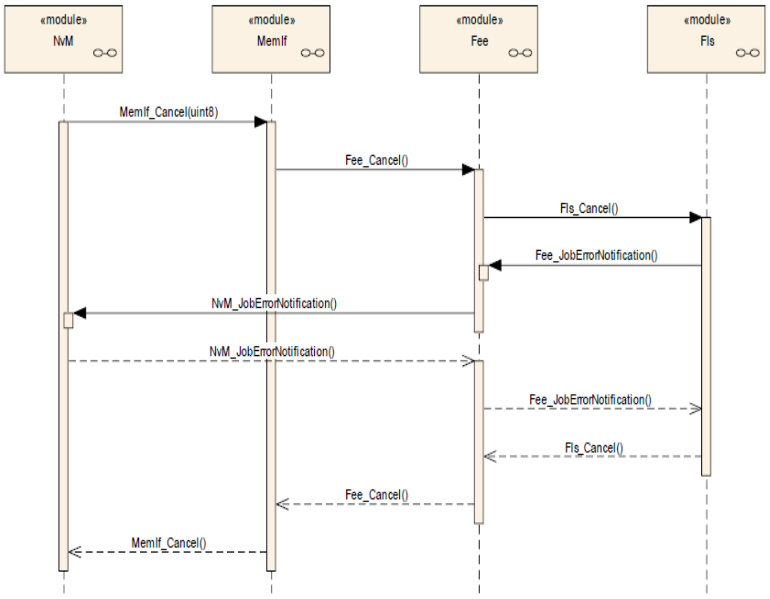
2. Synchronous functions

Figure(3-6) shows the sequence diagram of the function Fls_GetJobResult as an example for the synchronous functions of this module. The same sequence applies also to the functions Fls_GetStatus and Fls_SetMode.


«module»
NvM
«module»
MemIf
«module»
Fee
«module»
Fls
MemIf_GetJobResult(MemIf_JobResultType, uint8)
Fee_GetJobResult(MemIf_JobResultType)
Fls_GetJobResult(MemIf_JobResultType)
Fls_GetJobResult()
Fee_GetJobResult()
MemIf_GetJobResult()

Figure(3-7)

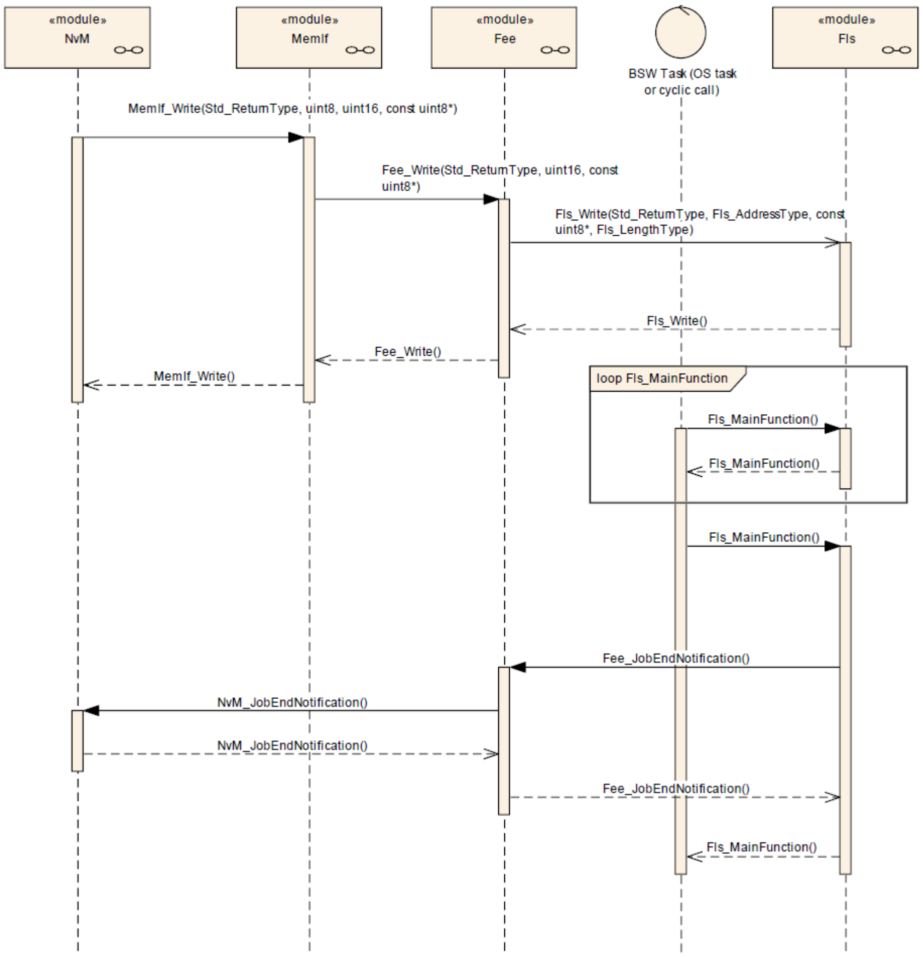
3. Canceling a running job



Figure(3-8)

4. Asynchronous functions

Figure(3-9) shows the sequence diagram of the flash write function as an example for the asynchronous functions of this module. The same sequence applies to the erase, read and compare jobs.



Figure(3-9)

3.2.3 EA Module

The EEPROM Abstraction (EA) module abstracts from the device specific addressing scheme and segmentation and provides the upper layers with a virtual addressing scheme and segmentation as well as a “virtually” unlimited number of erase cycles.

3.2.3.1 General Behavior of EA module

- The EEPROM Abstraction (EA) shall only accept one job at a time, i.e. the module shall not provide a queue for pending jobs (that’s the job of the NVRAM Manager).

- The EA module shall execute the read, write and Invalidate operations asynchronously within the EA module’s main function.
- The EEPROM Abstraction (EA) provides upper layers with a 32bit virtual linear address space and uniform segmentation scheme. This virtual 32bit addresses consists of
    - a 16bit block number – allowing a (theoretical) number of 65536 logical blocks
    - a 16bit block offset – allowing a (theoretical) block size of 64Kbyte per block
- The Ea module shall manage for each block the information, whether this block is “correct” from the point of view of the EA module or not. This consistency information shall only concern the internal handling of the block, not the block’s contents.

When a block write operation is started the EA module shall mark the corresponding block as inconsistent1. Upon the successful end of the block write operation, the EA module shall mark the block as consistent (again).

3.2.3.2 Blocks Configurations

Each block in EA module has an EaBlockConfiguration Container contains the values for the following parameters associated for this block:

- EaBlockNumber : Block Identifier.
- EaBlockSize : Size of a logical block in bytes.
- EaImmediateData : Marker for high priority data.
  - true: Block contains immediate data.
  - false: Block does not contain immediate data.
- EaNumberOfWriteCycles : Number of write cycles required for this block.
- EaDeviceIndex : Reference to the device this block is stored in.

*Hint : These configurations is done at Pre-Compile time.

3.2.3.3 Functions Supported in this module

Initialization:

Ea_Init

Operations:

Ea _EraseImmediateBlock
Ea _Write
Ea _Cancel
Ea _Read
Ea_InvalidateBlock

Information:

Ea_GetStatus
Ea _GetJobResult
Ea_SetMode
Ea_GetVersionInfo

Internal Operations:

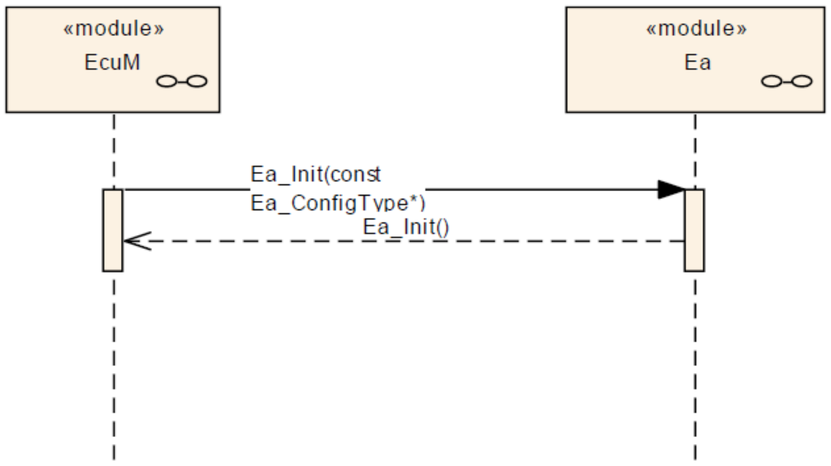
Ea _MainFunction

Call-back notifications:

Ea_JobEndNotification
Ea_JobErrorNotification

3.2.3.4 Sequence diagrams

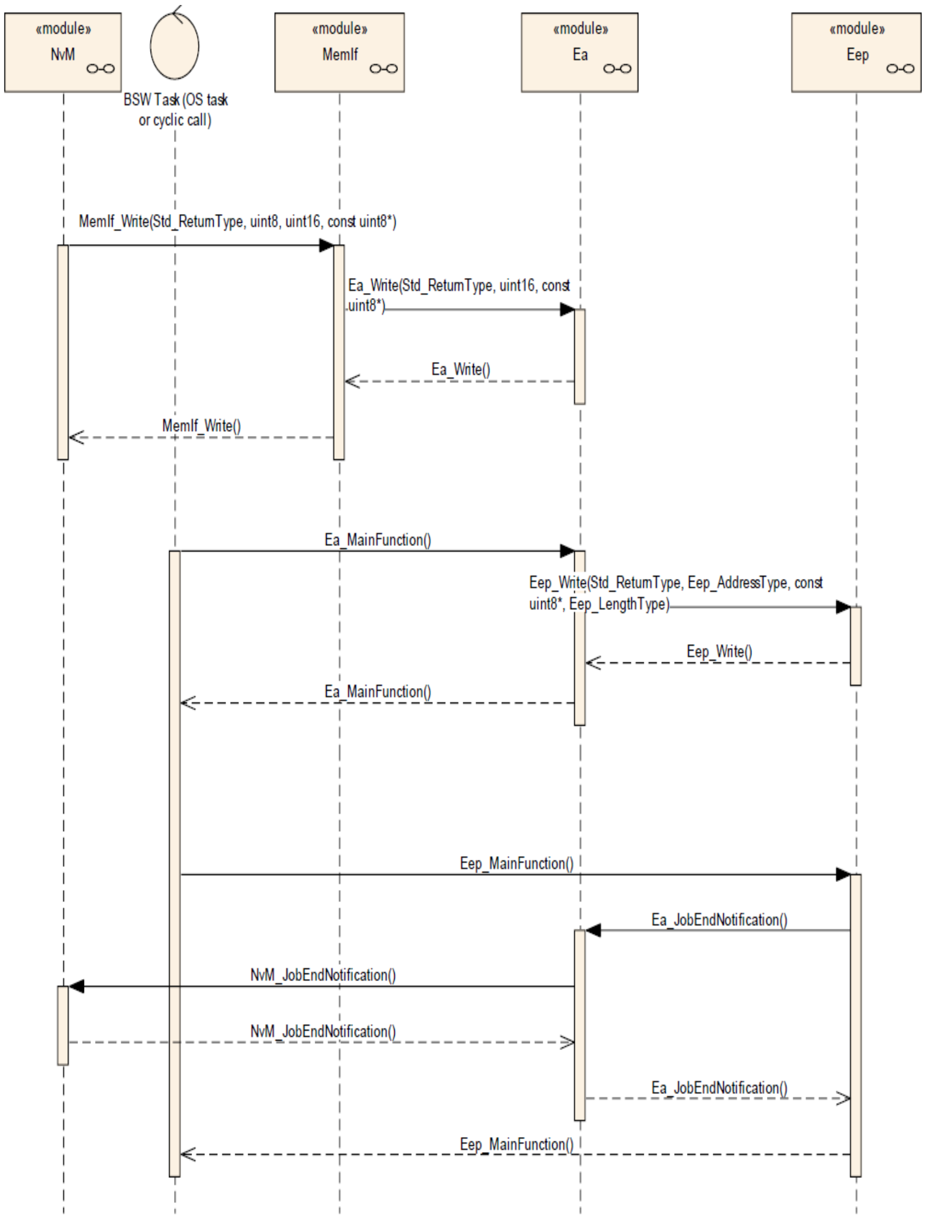
1. Initialization



Figure(3-10)

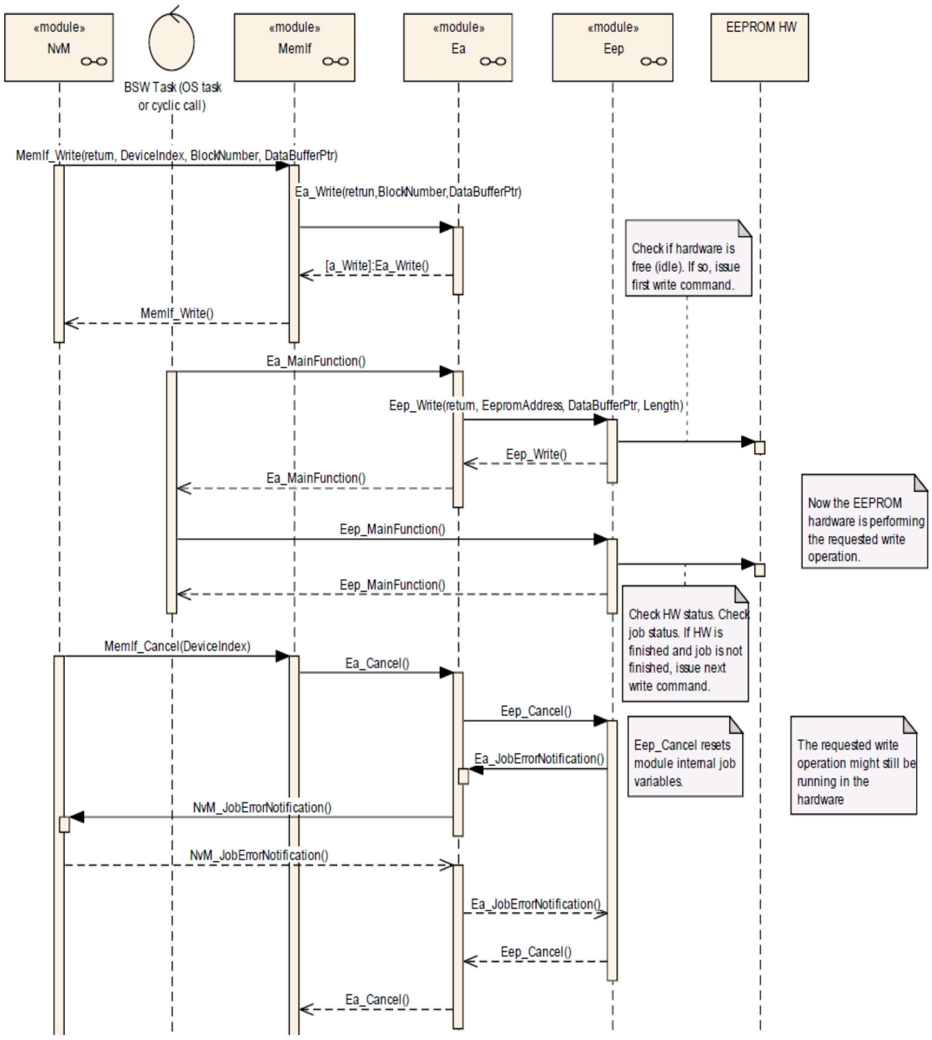
2. Ea_Write

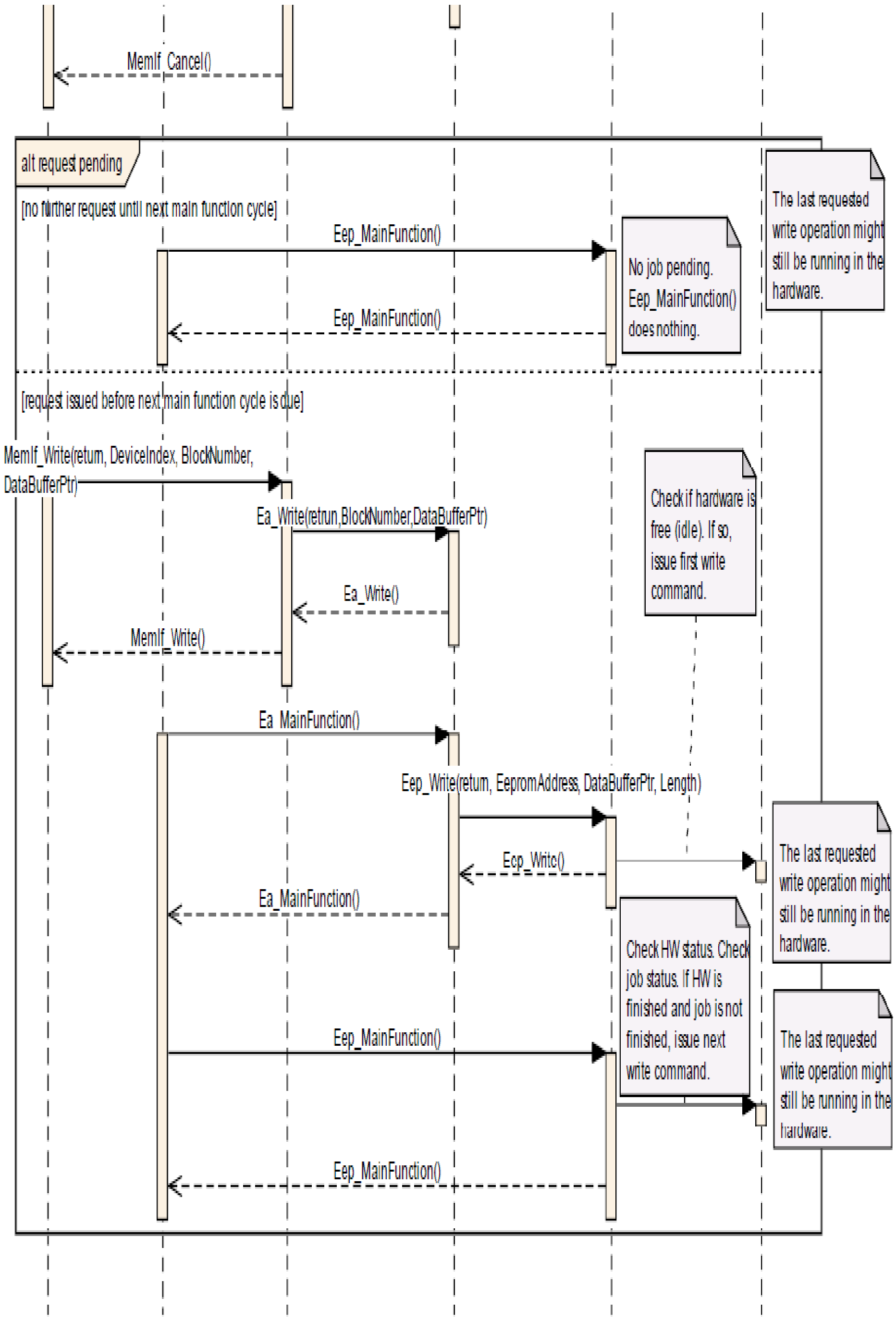
The following figure shows as an example the call sequence for the Ea_Write service. This sequence diagram also applies to the other asynchronous services of this module.


«module»
NvM
BSW Task (OS task or cyclic call)
«module»
MemIf
«module»
Ea
«module»
Eep
MemIf_Write(Std_ReturnType, uint8, uint16, const uint8*)
Ea_Write(Std_ReturnType, uint16, const uint8*)
Ea_Write()
MemIf_Write()
Ea_MainFunction()
Eep_Write(Std_ReturnType, Eep_AddressType, const uint8*, Eep_LengthType)
Eep_Write()
Ea_MainFunction()
Eep_MainFunction()
Ea_JobEndNotification()
NvM_JobEndNotification()
NvM_JobEndNotification()
Ea_JobEndNotification()
Eep_MainFunction()

3. EA_Cancel

Figure(3-11) shows as an example the call sequence for a canceled Ea_Write service. This sequence diagram shows that Ea_Cancel is asynchronous w.r.t. the underlying hardware while itself being synchronous.


«module» NvM
BSW Task (OS task or cyclic call)
«module» MemIf
«module» Ea
«module» Eep
EEPROM HW
MemIf_Write(return, DeviceIndex, BlockNumber, DataBufferPtr)
Ea_Write(retrun,BlockNumber,DataBufferPtr)
[a_Write]:Ea_Write()
Check if hardware is free (idle). If so, issue first write command.
MemIf_Write()
Ea_MainFunction()
Eep_Write(return, EepromAddress, DataBufferPtr, Length)
Eep_Write()
Ea_MainFunction()
Now the EEPROM hardware is performing the requested write operation.
Eep_MainFunction()
Eep_MainFunction()
Check HW status. Check job status. If HW is finished and job is not finished, issue next write command.
MemIf_Cancel(DeviceIndex)
Ea_Cancel()
Eep_Cancel()
Ea_JobErrorNotification()
Eep_Cancel resets module internal job variables.
The requested write operation might still be running in the hardware
NvM_JobErrorNotification()
NvM_JobErrorNotification()
Ea_JobErrorNotification()
Eep_Cancel()
Ea_Cancel()


MemIf_Cancel()
alt request pending
[no further request until next main function cycle]
Eep_MainFunction()
No job pending. Eep_MainFunction() does nothing.
The last requested write operation might still be running in the hardware.
Eep_MainFunction()
[request issued before next main function cycle is due]
MemIf_Write(return, DeviceIndex, BlockNumber, DataBufferPtr)
Ea_Write(retrun,BlockNumber,DataBufferPtr)
Check if hardware is free (idle). If so, issue first write command.
Ea_Write()
MemIf_Write()
Ea_MainFunction()
Eep_Write(return, EepromAddress, DataBufferPtr, Length)
Eep_Write()
The last requested write operation might still be running in the hardware.
Ea_MainFunction()
Check HW status. Check job status. If HW is finished and job is not finished, issue next write command.
Eep_MainFunction()
The last requested write operation might still be running in the hardware.
Eep_MainFunction()

Figure(3-11)

3.2.3.5 Error Handling

The Ea module shall detect the following errors and exceptions depending on its configuration (development/production):

- **Development Errors**

Type or error	Error code	Value(Hex)
API service called while module is not (yet) initialized.	EA_E_UNINIT	0x01
API service called with invalid block number.	EA_E_INVALID_BLOCK_NO	0x02
API service called with invalid block offset.	EA_E_INVALID_BLOCK_OFS	0x03
API service called with invalid pointer argument.	EA_E_PARAM_POINTER	0x04
API service called with invalid block length information.	EA_E_INVALID_BLOCK_LEN	0x05
Ea_Init failed.	EA_E_INIT_FAILED	0x09

- **Runtime Errors**

Type or error	Error code	Value(Hex)
API service called while module is busy.	EA_E_BUSY	0x06
Ea_Cancel called while no job was pending.	EA_E_INVALID_CANCEL	0x08

*Hint: These error codes when detected by the Ea module, are raised to the **Development Error Tracer (DET)** module.

3.2.4 Fee Module

The Flash EEPROM Emulation (FEE) shall abstract from the device specific addressing scheme and segmentation and provide the upper layers with a virtual addressing scheme and segmentation as well as a “virtually” unlimited number of erase cycles.

3.2.4.1 General behavior of FEE Module

- The Flash EEPROM Emulation (FEE) shall emulate the behavior of the EEPROM Abstraction Layer on flash memory technology. Thus it shall have the same functional scope and API as the EEPROM Abstraction Layer and allow for a similar configuration based on that of the underlying flash driver and flash device.

- Wear leveling algorithm to increase emulated EEPROM cycling capability.

- The objective of FEE Driver is to provide a set of software functions intended to use a Sector of Flash memory as the emulated EEPROM.

3.2.4.2 Functions supported in this module

Initialization:
Fee_Init

Operations:
Fee_Write
Fee_Read
Fee_InvalidateBlock

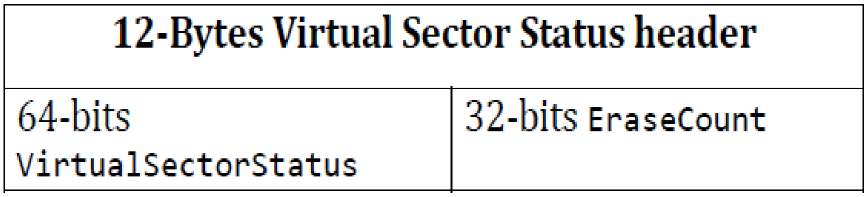
Information:
Fee_GetStatus
Fee_GetJobResult

Internal Operations:
Fee_MainFunction

Sector Operations:
GetSectorStatus
SetSectorStatus
Erase_Virtualsector
ReadVariable
WriteVariable
FindActiveSector
TransferSector
FindValidBlocks

3.2.4.3 Sectors and Blocks Organizations

- The Virtual Sector is the basic organizational unit used to partition the EEPROM Emulation Flash Bank. This structure can contain one or more contiguous Flash Sectors contained within one Flash Bank.
- Virtual Sector size depends on the size of all configured blocks.
- Virtual sector size equation :
- **VIRTUAL_SECTOR_**SIZE = (NUM_OF_PHYSICAL_SECTORS* PHYSICAL_SECTOR_SIZE)
- **NUM_OF_PHYSICAL_SECTORS** = (((SECTOR_HEADER_SIZE + ALL_BLOCKS_HEADER_SIZE + BLOCKS_SIZE)\PHYSICAL_SECTOR_SIZE) +(1U))
- **Virtual Sector Header** :



12-Bytes Virtual Sector Status header	
64-bits `VirtualSectorStatus`	32-bits `EraseCount`

- **Block Headre :**

18-Bytes Block header			
64-bits `BlockStatus`	32-bits `WriteCycleCount`	32-bits `NextBlockAddress`	16-bits `BlockNumber`

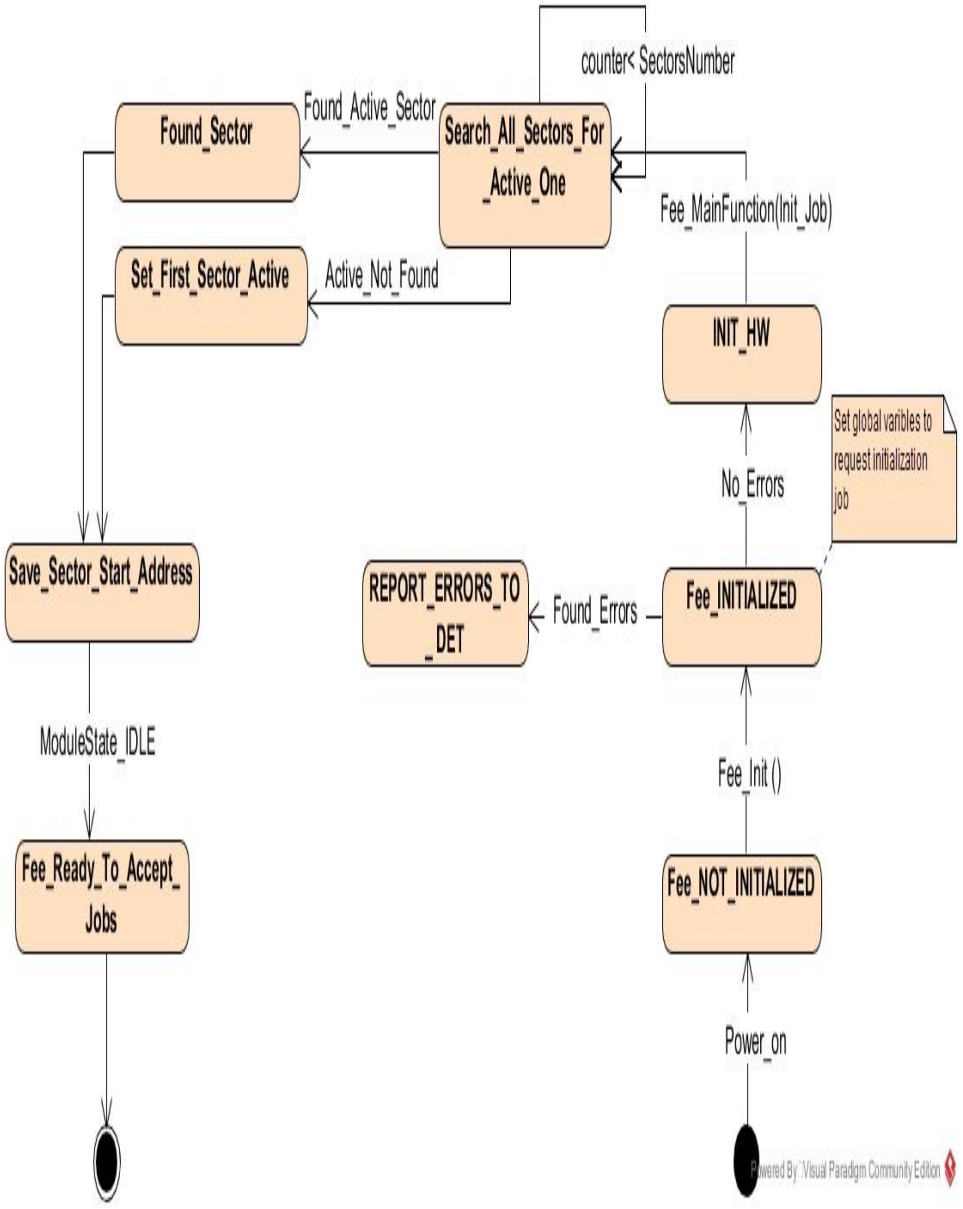
3.2.4.4 Module Global Variables

Variable Name	Description
ActiveSectorInfo	Sector Lookup table
ModuleInternalManagement	Save Current Module state , Job result, InternalProcessingState
BlockIndex	Save Block ID in blocks configurations container
ParametersCopy	Save Asynchronous functions parameters
JobDone	Save underlying layer end job notification
JobError	Save underlying layer job error notification
rtn_val1, rtn_val2, rtn_val3, rtn_val4, rtn_val5	Save internal states return values (E_OK, E_NOT_OK, E_PENDING)

3.2.4.5 Module State Machines

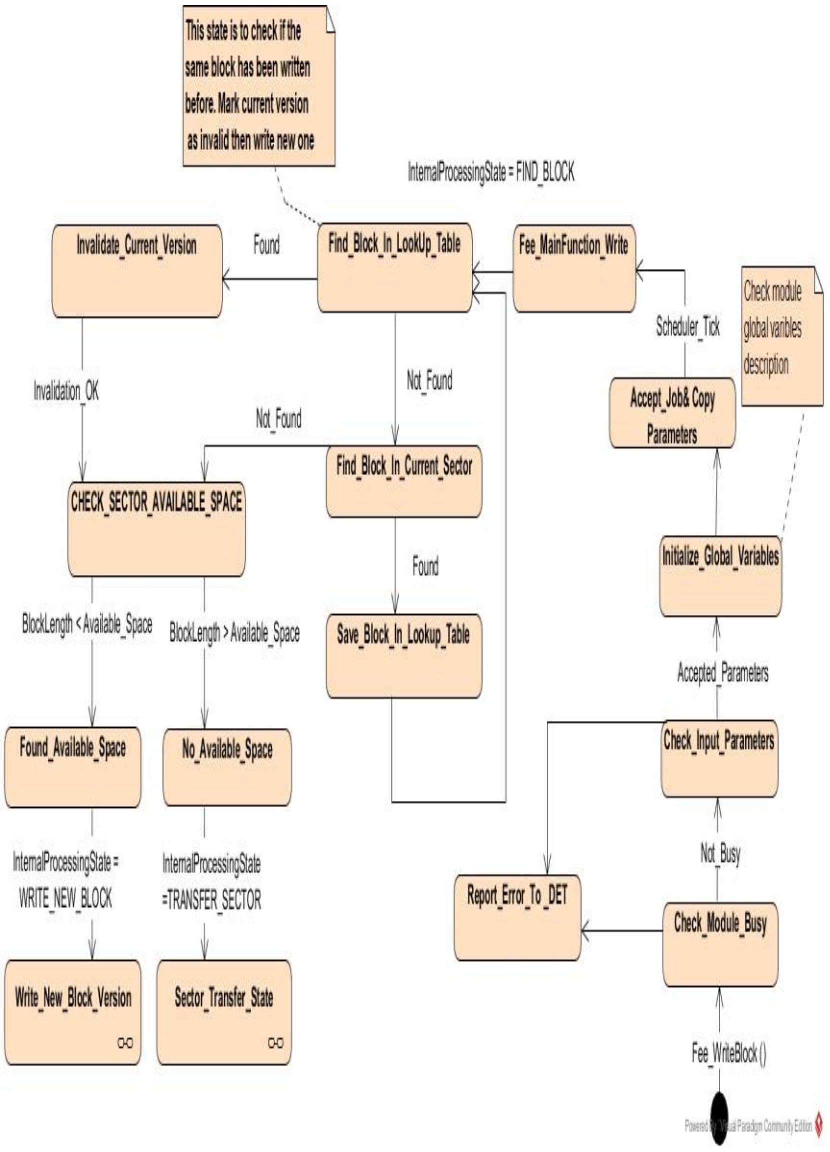
1. Initialization State machine

- As shown in Figure(3-12), after Initialization state Fee save the Active sector start address in Lookup table.


counter< SectorsNumber
Found_Active_Sector
Found_Sector
Search_All_Sectors_For _Active_One
Fee_MainFunction(Init_Job)
Set_First_Sector_Active
Active_Not_Found
INIT_HW
Set global varibles to request initialization job
No_Errors
Save_Sector_Start_Address
REPORT_ERRORS_TO _DET
Found_Errors
Fee_INITIALIZED
ModuleState_IDLE
Fee_Init ()
Fee_Ready_To_Accept_ Jobs
Fee_NOT_INITIALIZED
Power_on
Powered By Visual Paradigm Community Edition

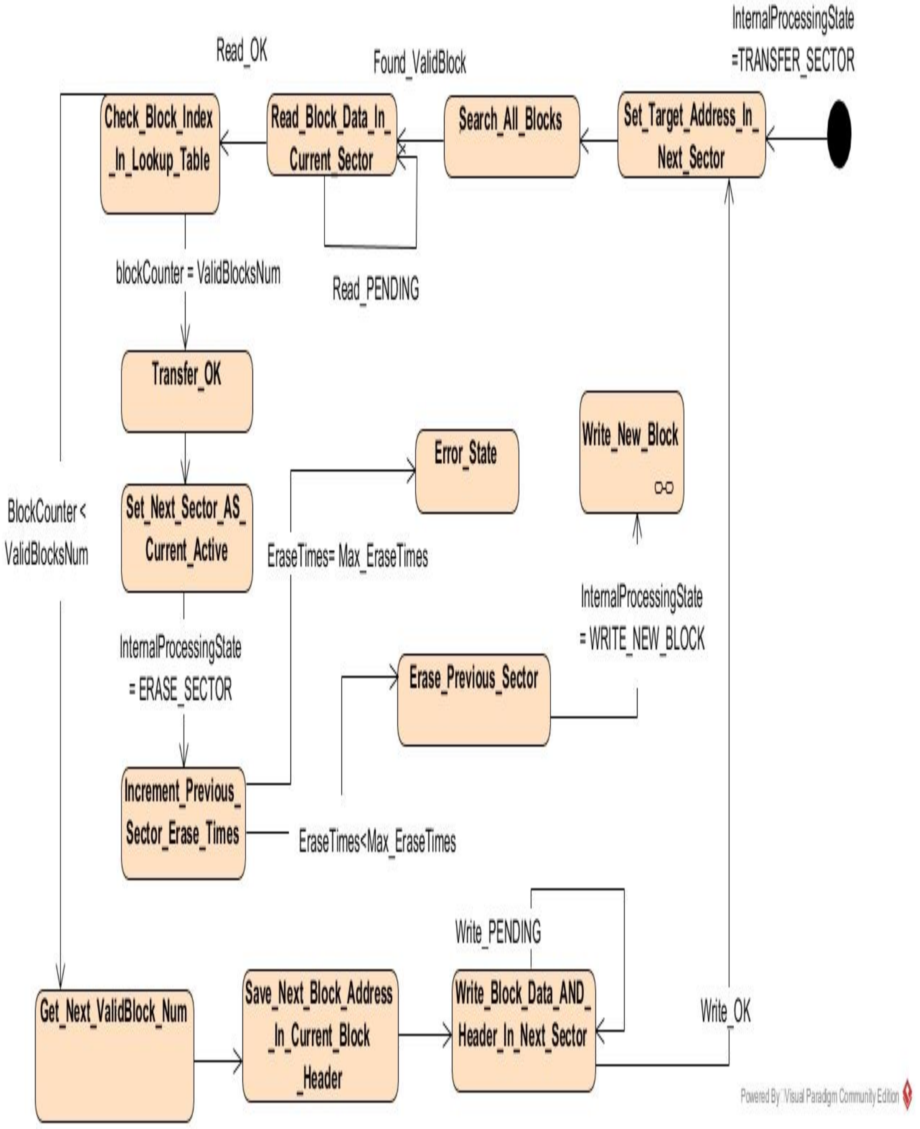
Figure(3-12)

2. Write Job State Machine


This state is to check if the same block has been written before. Mark current version as invalid then write new one
InternalProcessingState = FIND_BLOCK
Invalidate_Current_Version
Found
Find_Block_In_LookUp_Table
Fee_MainFunction_Write
Check module global varibles description
Scheduler_Tick
Invalidation_OK
Not_Found
Not_Found
Accept_Job& Copy Parameters
CHECK_SECTOR_AVAILABLE_SPACE
Find_Block_In_Current_Sector
Initialize_Global_Variables
Found
BlockLength < Available_Space
BlockLength > Available_Space
Save_Block_In_Lookup_Table
Accepted_Parameters
Found_Available_Space
No_Available_Space
Check_Input_Parameters
InternalProcessingState = WRITE_NEW_BLOCK
InternalProcessingState =TRANSFER_SECTOR
Not_Busy
Report_Error_To_DET
Check_Module_Busy
Write_New_Block_Version
Sector_Transfer_State
Fee_WriteBlock ()
Powered By Visual Paradigm Community Edition

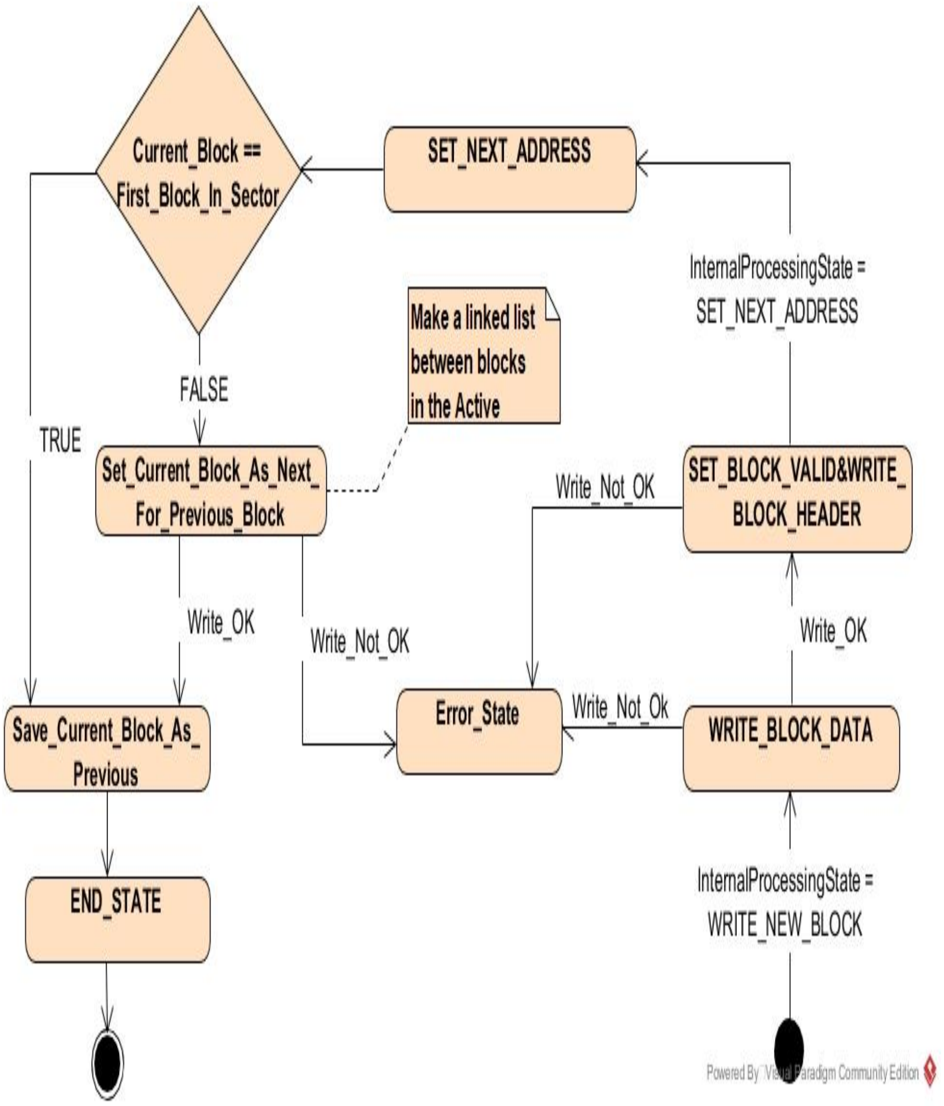
Figure(3-13)

3. Sector Transfer Sate Machine


InternalProcessingState =TRANSFER_SECTOR
Set_Target_Address_In_Next_Sector
Search_All_Blocks
Found_ValidBlock
Read_Block_Data_In_Current_Sector
Read_PENDING
Read_OK
Check_Block_Index_In_Lookup_Table
blockCounter = ValidBlocksNum
Transfer_OK
Set_Next_Sector_AS_Current_Active
InternalProcessingState = ERASE_SECTOR
Increment_Previous_Sector_Erase_Times
EraseTimes= Max_EraseTimes
Error_State
EraseTimes<Max_EraseTimes
Erase_Previous_Sector
InternalProcessingState = WRITE_NEW_BLOCK
Write_New_Block
BlockCounter < ValidBlocksNum
Get_Next_ValidBlock_Num
Save_Next_Block_Address_In_Current_Block_Header
Write_Block_Data_AND_Header_In_Next_Sector
Write_PENDING
Write_OK
Powered By Visual Paradigm Community Edition

Figure(3-14)

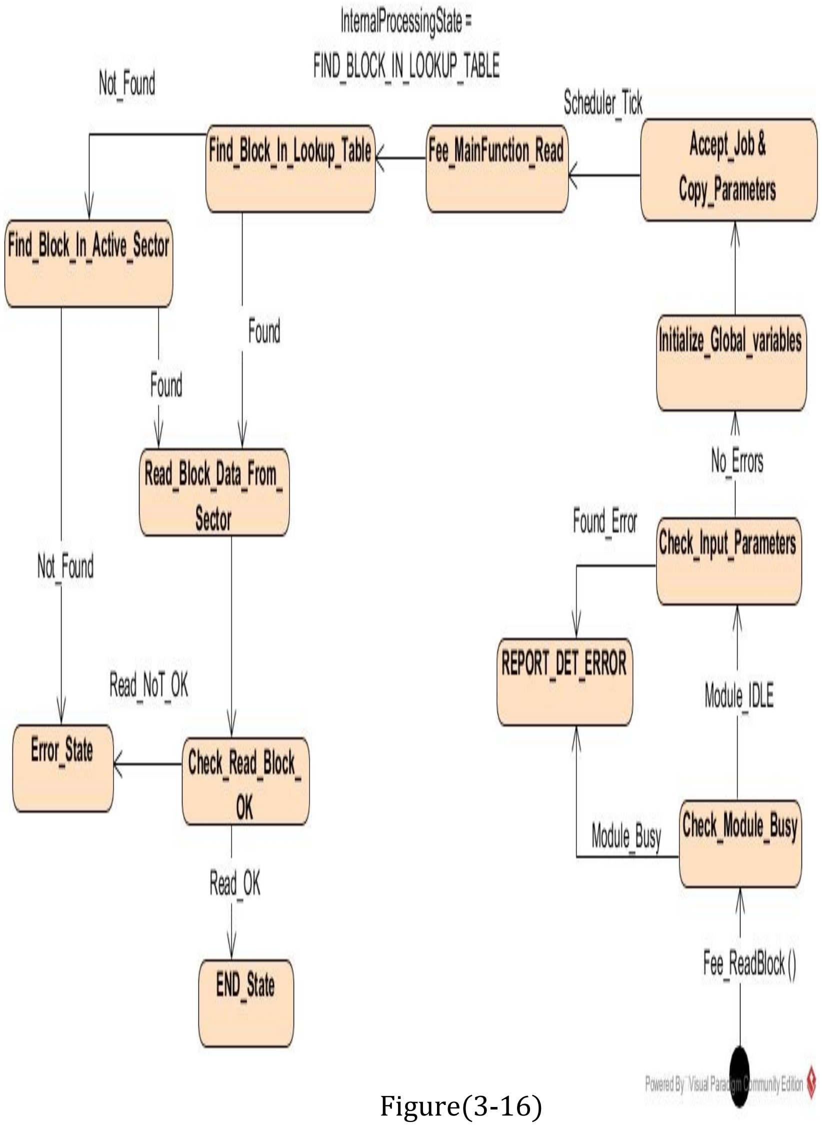
4. Write new block in active sector



Current_Block == First_Block_In_Sector
SET_NEXT_ADDRESS
InternalProcessingState = SET_NEXT_ADDRESS
Make a linked list between blocks in the Active
FALSE
TRUE
Set_Current_Block_As_Next_For_Previous_Block
Write_Not_OK
SET_BLOCK_VALID&WRITE_BLOCK_HEADER
Write_OK
Write_Not_OK
Write_OK
Save_Current_Block_As_Previous
Error_State
Write_Not_Ok
WRITE_BLOCK_DATA
END_STATE
InternalProcessingState = WRITE_NEW_BLOCK
Powered By Visual Paradigm Community Edition

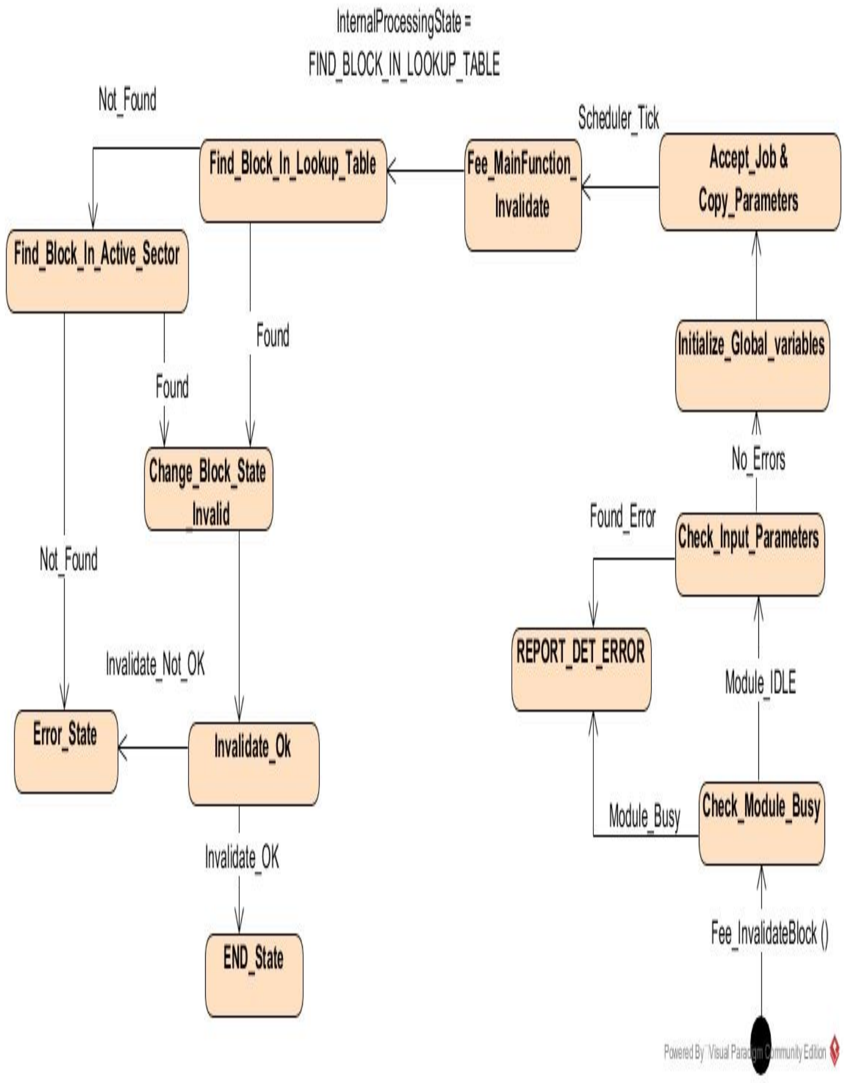
Figure(3-15)

5. Read Job State Machine



Figure(3-16)

6. Invalidate Block State Machine

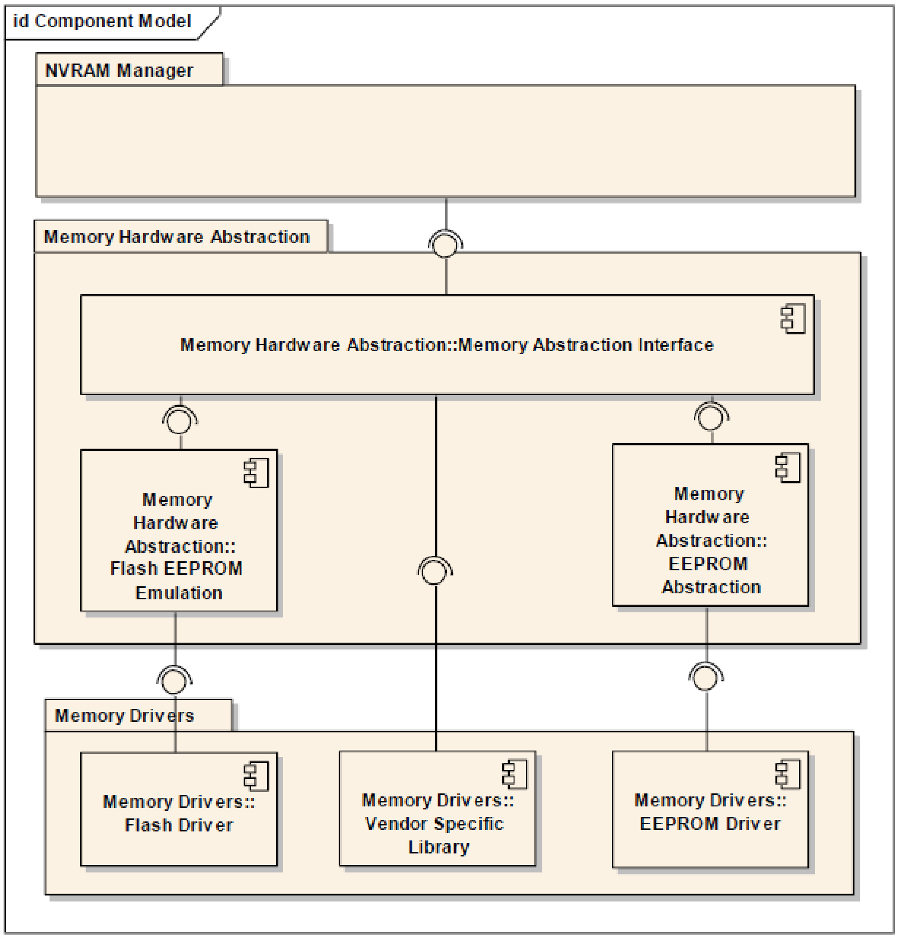

InternalProcessingState = FIND_BLOCK_IN_LOOKUP_TABLE
Not_Found
Scheduler_Tick
Find_Block_In_Lookup_Table
Fee_MainFunction_Invalidate
Accept_Job & Copy_Parameters
Find_Block_In_Active_Sector
Found
Found
Initialize_Global_variables
Change_Block_State Invalid
No_Errors
Found_Error
Check_Input_Parameters
Not_Found
REPORT_DET_ERROR
Invalidate_Not_OK
Module_IDLE
Error_State
Invalidate_Ok
Module_Busy
Check_Module_Busy
Invalidate_OK
Fee_InvalidateBlock ()
END_State
Powered By Visual Paradigm Community Edition

Figure(3-17)

3.2.5 MemIf Module

"Memory Abstraction Interface" (MemIf) module shown in Figure(3-18), allows the NVRAM manager to access several memory abstraction modules (FEE or EA modules).

It shall abstract from the number of underlying FEE or EA modules and provide upper layers with a virtual segmentation on a uniform linear address space.


id Component Model
NVRAM Manager
Memory Hardware Abstraction
Memory Hardware Abstraction::Memory Abstraction Interface
Memory Hardware Abstraction:: Flash EEPROM Emulation
Memory Hardware Abstraction:: EEPROM Abstraction
Memory Drivers
Memory Drivers:: Flash Driver
Memory Drivers:: Vendor Specific Library
Memory Drivers:: EEPROM Driver

Figure(3-18)

3.2.5.1 General Behavior

- The API specified in this chapter shall be mapped to the API of the underlying memory abstraction modules.
- The parameter DeviceIndex shall be used for selection of memory abstraction modules (and thus memory devices). If only one memory abstraction module is configured, the parameter DeviceIndex shall be ignored.

- If only one memory abstraction module is configured, the Memory Abstraction Interface shall be implemented as a set of macros mapping the Memory Abstraction Interface API to the API of the corresponding memory abstraction module. ⌋ (SRS_SPAL_12078, SRS_MemHwAb_14021)
  Example:
  ```
  #define MemIf_Write(DeviceIndex, BlockNumber, DataPtr) \
                              Fee_Write(BlockNumber, DataPtr)
  ```

- If more than one memory abstraction module is configured, the Memory Abstraction Interface shall use efficient mechanisms to map the API calls to the appropriate memory abstraction module.

- In our case, We are using array of pointers to functions where the parameter DeviceIndex is used as array index.

3.2.5.2 Functions supported in the MemIf Module

The same functions of FEE and EA modules with the same parameters plus another parameter (Device index) to select the appropriate device.

3.2.6 NvM Module

The NVRAM Manager (NvM) module shall provide services to ensure the data storage and maintenance of NV (non volatile) data according to their individual requirements in an automotive environment. The NvM module shall be able to administrate the NV data of an EEPROM and/or a FLASH EEPROM emulation device.

The NvM module shall provide the required synchronous/asynchronous services for the management and the maintenance of NV data (init/read/write/control).

3.2.6.1 Addressing Scheme for the memory hardware abstraction

- The Memory Abstraction Interface, the underlying Flash EEPROM Emulation and EEPROM Abstraction Layer provide the NvM module with a virtual linear 32bit address space which is composed of a 16bit block number and a 16bit block address offset.

- The NvM module allows for a (theoretical) maximum of 65536 logical blocks, each logical block having a (theoretical) maximum size of 64 Kbytes.

- The NvM module shall further subdivide the 16bit Fee/Ea block number into the following parts:
  - NV block base number (NVM_NV_BLOCK_BASE_NUMBER) with a bit width of (16 - NVM_DATASET_SELECTION_BITS)
  - Data index with a bit width of (NVM_DATASET_SELECTION_BITS)

3.2.6.2 Basic storage objects in NvM

A “Basic Storage Object” is the smallest entity of a “NVRAM block”. Several “Basic Storage Objects” can be used to build a NVRAM Block. A “Basic Storage Object” can reside in different memory locations (RAM/ROM/NV memory).
These basic storage objects are described below:

- **RAM Block**

The “RAM Block” is a “Basic Storage Object”. It represents the part of a “NVRAM Block” which resides in the RAM.
It is composed of user data and (optionally) a CRC value and (optionally) a NV block header. It is used to hold the live data. This is an optional part of NVRAM block.

- **ROM Block**

The “ROM Block” is a “Basic Storage Object”. It represents the part of a “NVRAM Block” which resides in the ROM. The “ROM Block” is an optional part of a “NVRAM Block”.
Contents of ROM Block are of persistent nature, which can’t be modified during program execution and resides in ROM/Flash. It is used to provide default data in case of an empty or damaged NV block.

- **NV Block**

The “NV Block” is a “Basic Storage Object”. It represents the part of a “NVRAM Block” which resides in the NV memory. The “NV Block” is a mandatory part of a “NVRAM Block”.
Contents of NV Block are of persistent nature that can be modified during program execution and resides in the Flash. It is composed of NV user data and (optionally) a CRC value and (optionally) a NV block header. It is used to hold the live data that are stored periodically/on request.

- **Administrative Block**

The “Administrative Block” is a “Basic Storage Object”. It resides in RAM. The “Administrative Block” is a mandatory part of a “NVRAM Block”.
Contents of Administrative Block are of non-persistent nature and resides in the RAM.
It is used to hold attribute/error/status information of the corresponding NVRAM blocks well as the block indices specifically for NVRAM blocks of type 'Dataset’. This is a mandatory part of NVRAM block.

3.2.6.3 Block Management Types in NvM

The following NVRAM Block Management types are supported by the NvM:

- **Native NVRAM block**

The Native NVRAM block is the simplest block management type. It allows storage to/retrieval from NV memory with a minimal overhead.

NVM_BLOCK_NATIVE type of NVRAM storage consists of the following basic storage objects:
- NV Blocks: 1
- RAM Blocks: 1
- ROM Blocks: 0..1
- Administrative Blocks:1

- **Redundant NVRAM block**

In addition to the Native NVRAM block, the Redundant NVRAM block provides enhanced fault tolerance, reliability and availability. It increases resistance against data corruption. The Redundant NVRAM block consists of two NV blocks, a RAM block and an Administrative block.
In case NV Block associated with a Redundant NVRAM block is deemed invalid (e.g. during read), an attempt is made to recover the NV Block using data from the incorrupt NV Block.

- **Dataset NVRAM block**

The Dataset NVRAM block is an array of equally sized data blocks. The application can at one-time access exactly one of this data block.
NVM_BLOCK_DATASET type of NVRAM storage consists of the following basic storage objects:
- NV Blocks: 1..NvMNvBlockNum
- RAM Blocks: 1
- ROM Blocks: 0..NvMRomBlockNum
- Administrative Blocks: 1

The total number of configured datasets (NV+ROM blocks) must be in the range of 1..255.
A specific dataset element is accessed by setting the corresponding index using the API NvM_SetDataIndex. Elements with an index from 0 up to NvMNvBlockNum - 1 represent the NV Blocks, while the ones with an index from NvMNvBlockNum up to NvMNvBlockNum + NvMRomBlockNum - 1 represent the ROM blocks. The NVRAM Block user has to ensure that a valid dataset index is selected before accessing data elements.

3.2.6.4 CRC based comparison

- The NvM module internally uses CRC generation routines (8/16/32 bit) to check and to generate CRC for NVRAM blocks as a configurable option.

- The NvM module provides an option to skip writing of unchanged data by implementing a CRC based compare mechanism. CRC based compare mechanism can be enabled by setting configuration parameter NvMBlockUseCRCCompMechanism.

3.2.6.5 Error recovery

The NvM module provides implicit error recovery on read for NVRAM block management types NATIVE and REDUNDANT by loading default values (if configured via either the parameter NvMRomBlockDataAddress or the parameter NvMInitBlockCallback).
The explicit retrieval of ROM data is available for all block management types by calling the API NvM_RestoreBlockDefaults. For DATASET, the related index must be set (pointing at a ROM block) prior to calling this API.

3.2.6.6 Functions supported in this module

Synchronous requests:

- NvM_Init : Service for resetting all internal variables.
- NvM_SetDataIndex : Service for setting the DataIndex of a dataset NVRAM block.
- NvM_GetDataIndex : Service for getting the currently set DataIndex of a dataset NVRAM block.
- NvM_GetErrorStatus : Service to read the block dependent error/status information.
- NvM_SetRamBlockStatus : Service for setting the RAM block status of a permanent RAM block or the status of the explicit synchronization of a NVRAM block.

Asynchronous single block requests:

- NvM_ReadBlock : Service to copy the data of the NV block to its corresponding RAM block.
- NvM_WriteBlock : Service to copy the data of the RAM block to its corresponding NV block.

- NvM_RestoreBlockDefaults : Service to restore the default data to its corresponding RAM block.
- NvM_InvalidateNvBlock : Service to invalidate a NV block.

Asynchronous multi block requests :

- NvM_ReadAll : Initiates a multi block read request.
- NvM_WriteAll : Initiates a multi block write request.

Scheduled functions :

- NvM_MainFunction : Service for performing the processing of the NvM jobs.

Callback notification of the NvM module:

- NvM_JobEndNotification : Function to be used by the underlying memory abstraction to signal end of job without error.
- NvM_JobErrorNotification : Function to be used by the underlying memory abstraction to signal end of job with error.

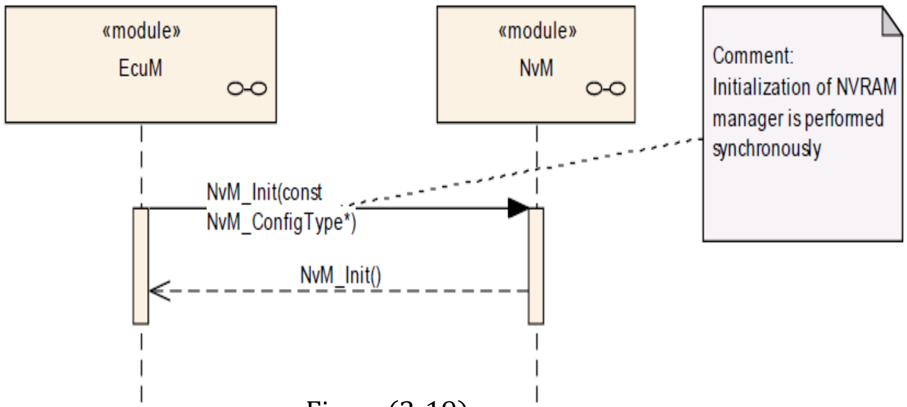
3.2.6.7 NvM Block Descriptor

Container for a management structure to configure the composition of a given NVRAM Block Management Type. Its multiplicity describes the number of configured NVRAM blocks, one block is required to be configured. The NVRAM block descriptors are condensed in the NVRAM block descriptor table.

3.2.6.8 Sequence Diagrams

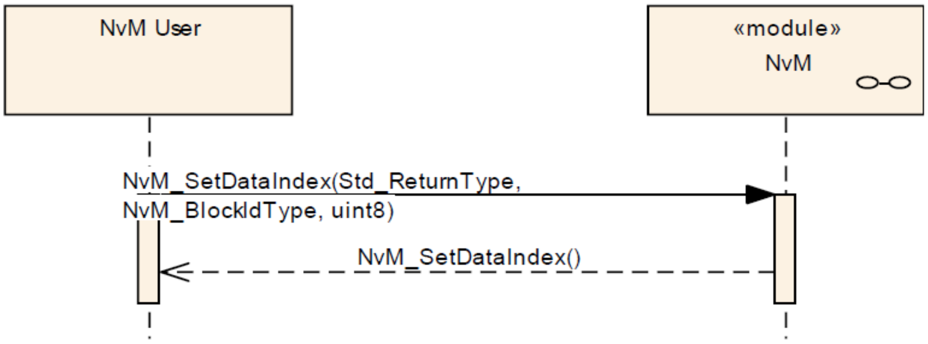
1. Synchronous calls

1. NvM_Init


«module»
EcuM
«module»
NvM
Comment:
Initialization of NVRAM manager is performed synchronously
NvM_Init(const NvM_ConfigType*)
NvM_Init()

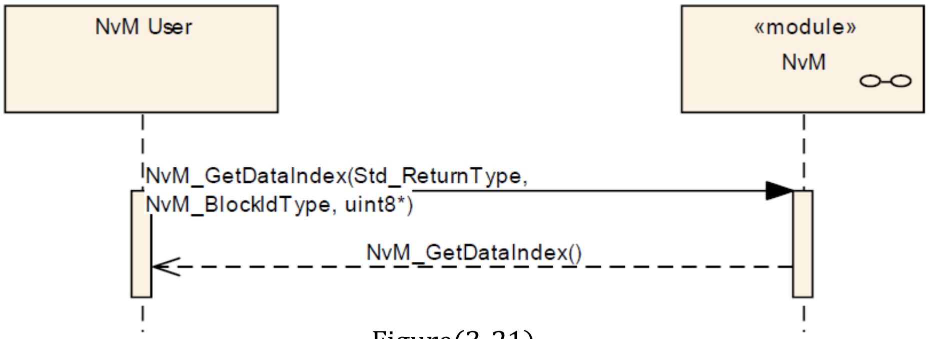
Figure(3-19)

2. NvM_SetDataIndex



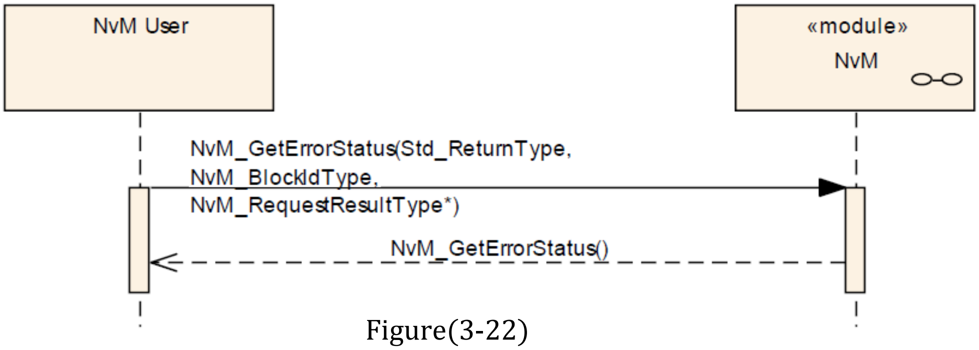
Figure(3-20)

3. NvM_GetDataIndex



Figure(3-21)

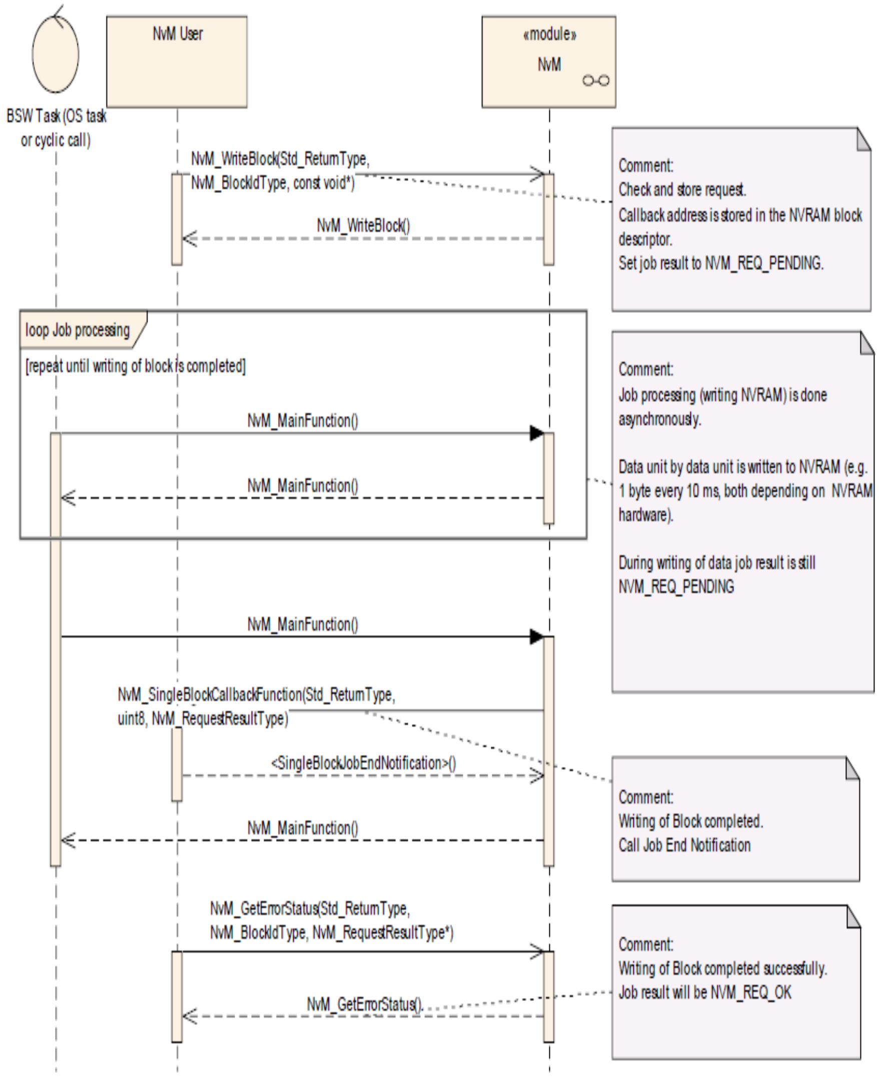
4. NvM_GetErrorStatus



Figure(3-22)

2. Asynchronous call with callback

Figure(3-23) shows the function NvM_WriteBlock as an example of a request that is performed asynchronously. The sequence for all other asynchronous functions is the same, only the processed number of blocks and the block types may vary. The result of the asynchronous function is obtained after an asynchronous notification (callback) by requesting the error/status information.


BSW Task (OS task or cyclic call)
NvM User
«module» NvM
NvM_WriteBlock(Std_ReturnType, NvM_BlockIdType, const void*)
NvM_WriteBlock()
Comment: Check and store request. Callback address is stored in the NVRAM block descriptor. Set job result to NVM_REQ_PENDING.
loop Job processing
[repeat until writing of block is completed]
NvM_MainFunction()
NvM_MainFunction()
Comment: Job processing (writing NVRAM) is done asynchronously. Data unit by data unit is written to NVRAM (e.g. 1 byte every 10 ms, both depending on NVRAM hardware). During writing of data job result is still NVM_REQ_PENDING
NvM_MainFunction()
NvM_SingleBlockCallbackFunction(Std_ReturnType, uint8, NvM_RequestResultType)
<SingleBlockJobEndNotification>()
NvM_MainFunction()
Comment: Writing of Block completed. Call Job End Notification
NvM_GetErrorStatus(Std_ReturnType, NvM_BlockIdType, NvM_RequestResultType*)
NvM_GetErrorStatus()
Comment: Writing of Block completed successfully. Job result will be NVM_REQ_OK

Figure(3-23)

CHAPTER FOUR

COMMUNICATION PROTOCOL CAN

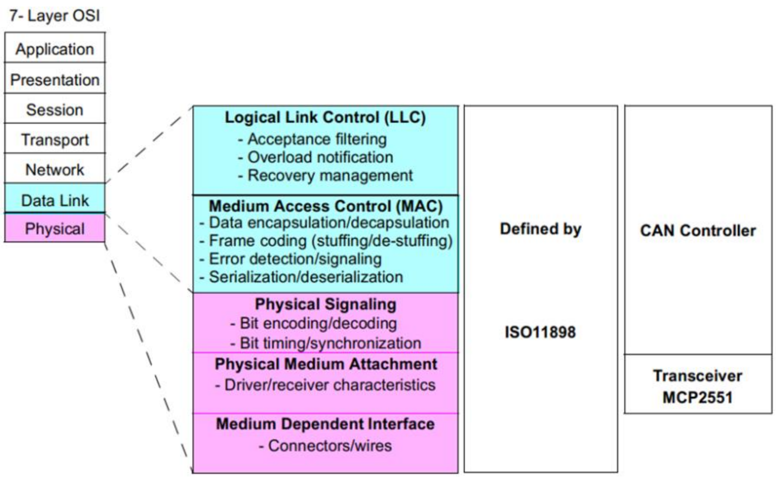
Abstract

In order to make the car smart and more safety new ECUs were added to the cars, it is not a problem. The problem was how to connect between these different ECUs to deliver the data from ECU to another. In order to solve the problem of wiring in the cars, Bosch started to develop a new serial communication protocol in the early1980s. In 1986, Bosch introduced The Controller Area Network (CAN) bus to the world , The first use of CAN bus was in 1988 BMW 8 Series cars. The last version of CAN bus is CAN with Flexible Data-Rate (CAN FD) and it released from Bosch in 2012, this version speed reach to 8 Megabit/sec and allows to sending 64 byte of data in every message. The CAN bus is also using to run diagnostics on the cars to report the status of each ECU in the car this feature make the fix of car's problem easier. Today all cars in the world use CAN bus as a main serial communication protocols between most of ECUs for several reasons :

- Message priority assignment and guaranteed maximum latencies.
- Multicast communication with bit-oriented synchronization.
- System-wide data consistency.
- Bus multi master access.
- Error detection and signaling with automatic retransmission of corrupted messages.
- Detection of permanent failures in nodes, and automatic switch-off to isolate faulty node.

4.1 Layers Structure of CAN

Many network protocols are described using the seven layer Open System Interconnection (OSI) model, as shown in Figure(4-1) . The Controller Area Network (CAN) protocol defines the Data Link Layer and part of the Physical Layer in the OSI model. The remaining physical layer (and all of the higher layers) are not defined by the CAN specification. These other layers can either be defined by the system designer, or they can be implemented using existing non-proprietary Higher Layer Protocols (HLPs) and physical layers .


7- Layer OSI
Application
Presentation
Session
Transport
Network
Data Link
Physical
Logical Link Control (LLC)
- Acceptance filtering
- Overload notification
- Recovery management
Medium Access Control (MAC)
- Data encapsulation/decapsulation
- Frame coding (stuffing/de-stuffing)
- Error detection/signaling
- Serialization/deserialization
Physical Signaling
- Bit encoding/decoding
- Bit timing/synchronization
Physical Medium Attachment
- Driver/receiver characteristics
Medium Dependent Interface
- Connectors/wires
Defined by
ISO11898
CAN Controller
Transceiver
MCP2551

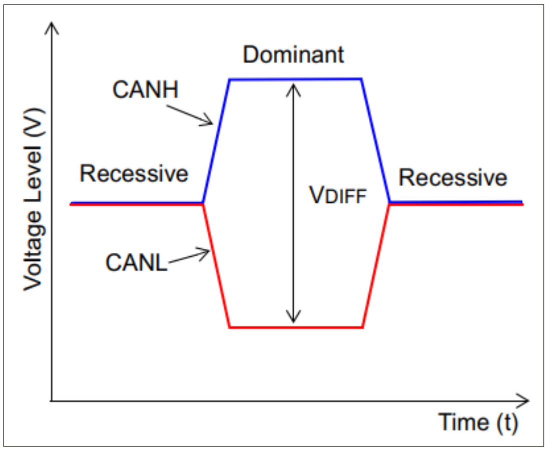
Figure(4-1)

4.1.1 Physical layer

Is the basic hardware required for a CAN network, i.e. the ISO 11898 electrical specifications. It converts 1's and 0's into electrical pulses leaving a node, then back again for a CAN message entering a node. Although the other layers may be implemented in software or in hardware as a chip function, the Physical Layer is always implemented in hardware
The impact on the physical, which includes the transceiver, the network and the interface between microcontroller and transceiver , will be discussed in this part .

4.1.1.1 Bus Levels

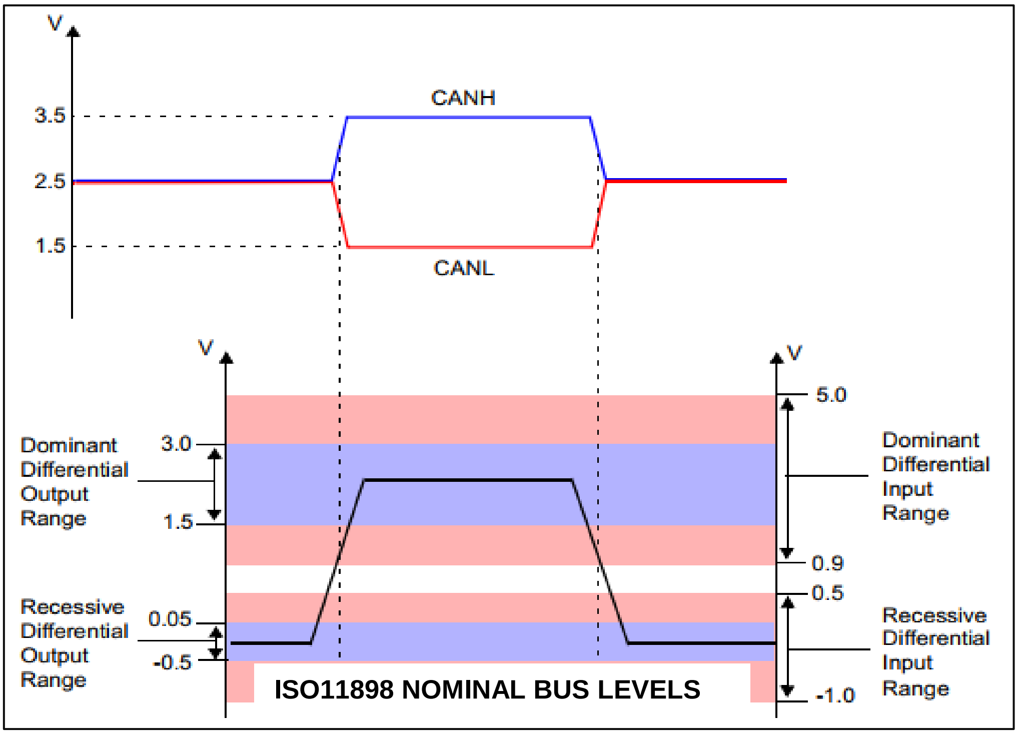
CAN specifies two logical states: recessive and dominant.
ISO-11898 defines a differential voltage to represent recessive and dominant states (or bits), as shown in Figure(4-2).



Figure(4-2)

In the recessive state :
(logic '1' on the MCP2551 TXD input), the differential voltage on CANH and CANL is less than the minimum threshold (<0.5V receiver input or <1.5V transmitter output) .

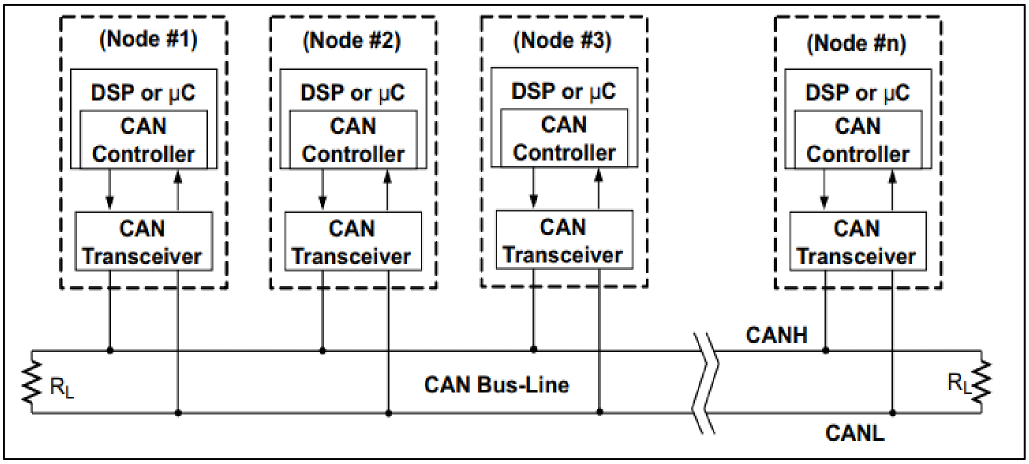
In the dominant state :
(logic ‘0’ on the MCP2551 TXD input), the differential voltage on CANH and CANL is greater than the minimum threshold. A dominant bit overdrives a recessive bit on the bus to achieve nonde structive bitwise arbitration as shown in Figure(4-3).

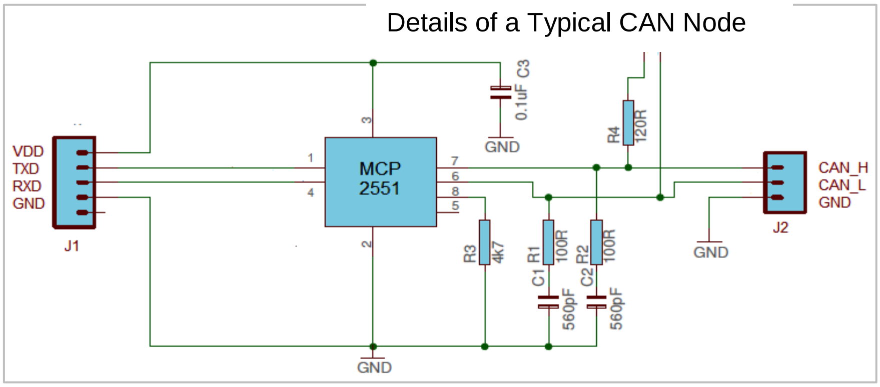

V
3.5
2.5
1.5
CANH
CANL
V
Dominant Differential Output Range
3.0
1.5
Recessive Differential Output Range
0.05
-0.5
ISO11898 NOMINAL BUS LEVELS
V
5.0
Dominant Differential Input Range
0.9
0.5
Recessive Differential Input Range
-1.0

Figure(4-3)

4.1.1.2 Connectors and Wires

ISO-11898-2 does not specify the mechanical wires and connectors. However, the specification does require that the wires and connectors meet the electrical specification.
The specification also requires 120Ω (nominal) terminating resistors at each end of the bus as shown in Figure(4-4).
Transceiver used : MCP2551



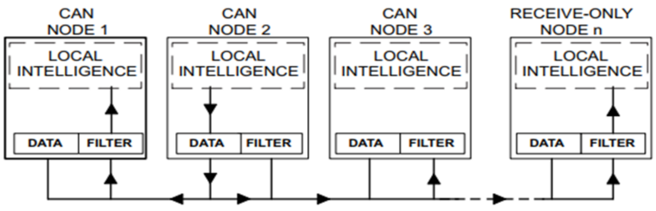


MCP Wiring schematic

Figure(4-4)

4.1.2 Data Link Layer

is responsible for transferring messages from a node to the network without errors. It handles bit stuffing and checksums, and after sending a message, waits for acknowledgment from the receivers. The data flow model is shown in Figure(4-5).



Figure(4-5) **The CAN Data-Flow Model**

since CAN is a broadcast system, a transmitting node places data on the network for all nodes to access. As shown in Figure , only those nodes requiring updated data allow the message to pass through a filter that is set by the network designer messages from certain nodes can pass, and all others are ignored .

4.2 Message Frame Format

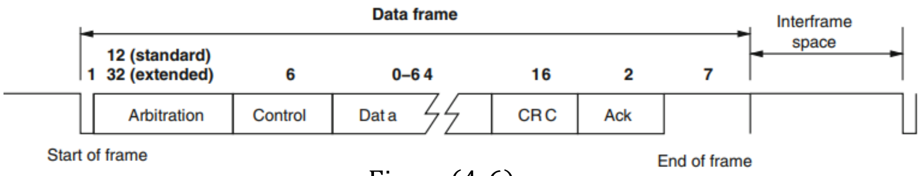
In CAN, there are four different types of frames, defined according to their content and function.

- Data frames: which contain data information from a source to (possibly) multiple receivers.
- Remote frames: which are used to request transmission of a corresponding (same identifier) Data frame.
- Error frame: which are transmitted whenever a node on the network detects an error.

- Overload frames: which are used for flow control to request an additional time delay before the transmission of a Data or Remote frame.

4.2.1 Data Frame Format

Data frames are used to transmit information between a source node and one or more receivers.



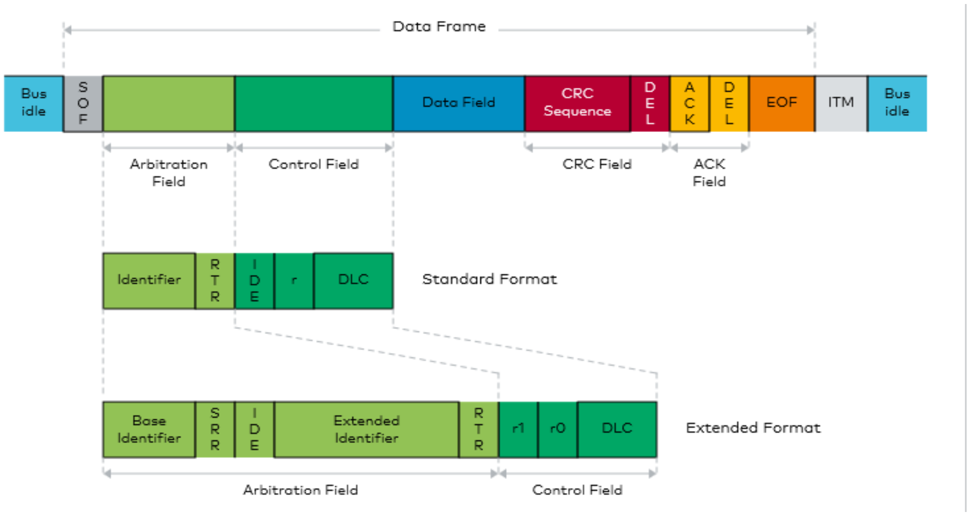
Figure(4-6)

There are two different formats for CAN messages shown in Figure(4-7), according to the type of message identifier that is used by the protocol.:
- Standard frames are frames defined with an 11-bit Identifier field.
- Extended frames have been made available from version 2.0 of the protocol as frames with an 29-bit Identifier field .

Standard and extended frames can be transmitted on the same bus by different nodes or by the same node.

The arbitration part of the protocol works regardless of the identifier version of the transmitted frames, allowing 29 bit identifier messages to be transmitted on the same network together with others with an 11-bit identifier.

In our project we used the both to identify messages IDs .



Figure(4-7) Standard and Extended Frame Format

Segments of the frame :

SOF :

-(start of frame) it's always a dominant bit to mark the start of any CAN frame .

Arbitration Field :

- In standard Frame it's consists of 11 bit identifier so it allows variety of 2048 identifier this 11 bit followed remote transmission request bit (RTR) which is dominant in Data Frame and recessive in Remote Frame .

- In Extends Frame It's consists of 29 bit identifier so it allows 2^{29} identifier the first 11 bits base identifier is followed by substitute Remote Request bit (SRR) it's a recessive bit . Identifier extension bit (IDE) and it's part of arbitration field whichis dominant in Standard Frame and recessive in Extended Frame.

Control Field :

- consists of IDE bit followed by 4 DLC bits Data length code to indicate how many data byte are transmitted and it may take value from 0 to 8.

Data Field :

- it's comprises the actual information to be transmitted . number of data byte may be vary from 0 to 8 byte .

CRC &ACK Field :

- The payload is protected by a checksum using a cyclic redundancy check (CRC) which is ended by a delimiter bit. Based on the results of the CRC, the receivers acknowledge positively or negatively in the ACK slot (acknowledgement) which also is followed by a delimiter bit.

EOF Field :

After this the transmission of a data frame is terminated by seven recessive bits (End Of Frame — EOF).

4.2.2 Error Frame Format

The Error frame is not a real frame, but rather the result of error signaling and recovery action(s). The Error frame given that it consists of 6 dominant or recessive bits in a sequence is of fixed form with no bit stuffing.

The CAN protocol identifies and defines management for five different error types:

• **Bit error :** A Bit error is detected when the bit value read from the bus is different from the bit value that is sent.

• **Stuff error:** A Stuff error is detected at the sixth consecutive occurrence of the same bit in a message field that is subject to bit stuffing.

• **CRC error:** If the CRC computed by the receiver differs from the one stored in the message frame.

• **Form error:** A Form error occurs when a fixed-form bit field contains one or more illegal bits. There 's some fixed form of fields in CAN : CRC , DEL , ACK , EOF , Ifs These bit must be recessive bits ,if receiver find any dominant bits in these fields it's a frame error .

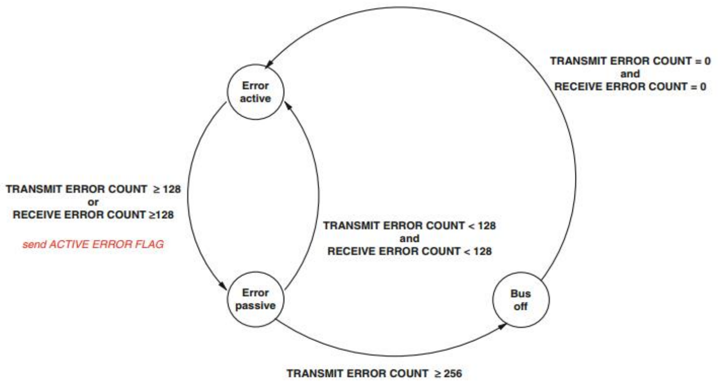
• **Acknowledgement error :** It is detected by a transmitter if a recessive bit is found on the ACK slot.

First we need to know that for each node there's two error counters Transmit error counter & Receive error counter . CAN protocol includes a fault confinement mechanism that detects faulty units and places them in passive or off states, so that they cannot affect the bus state with their outputs. The protocol assigns each node to one of three states:

1- Error Active State: units in this state are assumed to function properly. Units can transmit on the bus and signal errors with an Active Error flag.

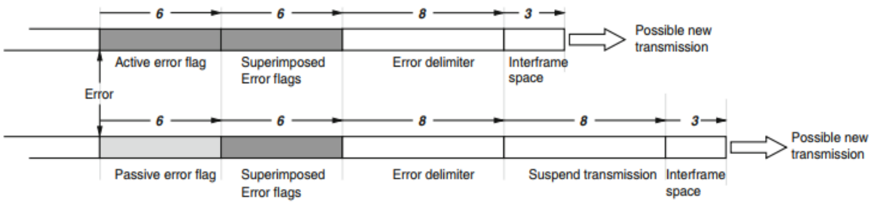
2- Error Passive State : units in this state are suspected of faulty behavior (in transmission or reception). They can still transmit on the bus, but their error signaling capability is restricted to the transmission of a Passive Error flag.

3- Buss Off State : units in this state are very likely corrupted and cannot have any influence on the bus. (their output drivers should be switched off.)


Error active
TRANSMIT ERROR COUNT = 0 and RECEIVE ERROR COUNT = 0
TRANSMIT ERROR COUNT ≥ 128 or RECEIVE ERROR COUNT ≥128
send ACTIVE ERROR FLAG
TRANSMIT ERROR COUNT < 128 and RECEIVE ERROR COUNT < 128
Error passive
Bus off
TRANSMIT ERROR COUNT ≥ 256

Figure(4-8)

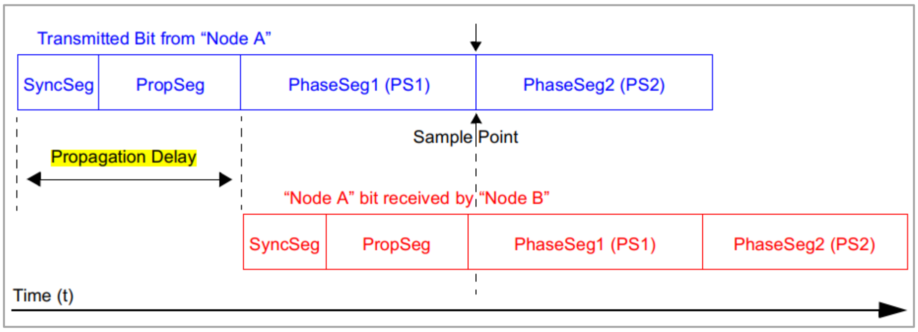
Bit Sequence after detection of an error :



Figure(4-9)

4.3 Bit Timing

The Controller Area Network (CAN) is a serial, asynchronous, multi-master communication protocol for connecting electronic control modules in automotive and industrial applications. A feature of the CAN protocol is that the bit rate, bit sample point and number of samples per bit are programmable. This gives the opportunity to optimize the performance of the network for a given application.



Figure(4-10) One-Way Propagation Delay

CAN Bit Segments as shown in Figure(4-10):

- Synchronization segment (SYNC SEG) :

This is a reference interval, used for synchronization purposes. The leading edge of a bit is expected to lie within this segment.

- Propagation segment (PROP SEG) :

This part of the bit time is used to compensate for the (physical) propagation delays within the network. It is twice the sum of the signal propagation time on the bus line, plus the input comparator delay, and the output driver delay.

- Phase segments (PHASE SEG1 and PHASE SEG2) :

These phase segments are time buffers used to compensate for phase errors in the position of the bit edge.

These segments can be lengthened or shortened to resynchronize the position of SYNCH SEG with respect to the following bit edge.

- Sample point (SAMPLE POINT) :

The sample point is the point of time at which the bus level is read and interpreted as the value of that respective bit. The quantity information processing time is defined as the time required to convert the electrical state of the bus, as read at the SAMPLE POINT into the corresponding bit value.

4.4 Developing CAN Interface Driver

Using **Tiva TM4C1234GH6PM** Microcontroller we implement CAN Interface Driver. the discussion of steps presented in **Appendix Code part .**

4.4.1 CAN Interface In Microcontroller

- The protocol controller transfers and receives the serial data from the CAN bus and passes the data on to the message handler.
-The message handler then loads this information into the appropriate message object based on the current filtering and identifiers in the message object memory.
-The message handler is also responsible for generating interrupts based on events on the CAN bus .
- The message object memory is a set of 32 identical memory blocks that hold the current configuration, status, and actual data for each message object. These memory blocks are accessed via either of the CAN message object register interfaces.
- The message memory is not directly accessible in the TM4C123GH6PM memory map, so the TM4C123GH6PM CAN controller provides an interface to communicate with the message memory via two CAN interface register sets for communicating with the message objects. These two interfaces used to read or write to each message object.
- The two message object interfaces allow parallel access to the CAN controller message objects when multiple objects may have new information that must be processed. In general, one interface is used for transmit data and one for receive data.

- TM4c123GH6PM microcontroller includes two CAN units with 32 message Objects with individual identifier Masks we can use them individually or using FIFO Mode which enables Storage of multiple message objects.
- The whole message (including all arbitration bits, data-length

code, and eight data bytes) is stored in the message object. If the Identifier Mask is used, the arbitration bits that are masked to "don't care" may be overwritten in the message object.
- The transmission of any number of message objects may be requested at the same time; they are transmitted according to their internal priority, which is based on the message identifier for the message object, with 1 being the highest priority and 32 being the lowest priority.

4.4.2 CAN Filters

The filtering is done by arbitration identifier of the CAN frame. This technique is also used when monitoring a CAN bus, in order to focus in on messages of importance using an identifier and mask. These allow a range of IDs to be accepted with a single filter rule. When a CAN frame is received, the mask is applied. This determines which bits of the identifier will be used to determine if the frame matches the filter.
-If a mask bit is set to a zero, the corresponding ID bit will automatically be accepted, regardless of the value of the filter bit.
- If a mask bit is set to a one, the corresponding ID bit will be compare with the value of the filter bit; if they match it is accepted otherwise the frame is rejected.

. For Example :

```
identifier = 0x123, mask = 0x7FF -> matches only
frames with identifier 0x123
```

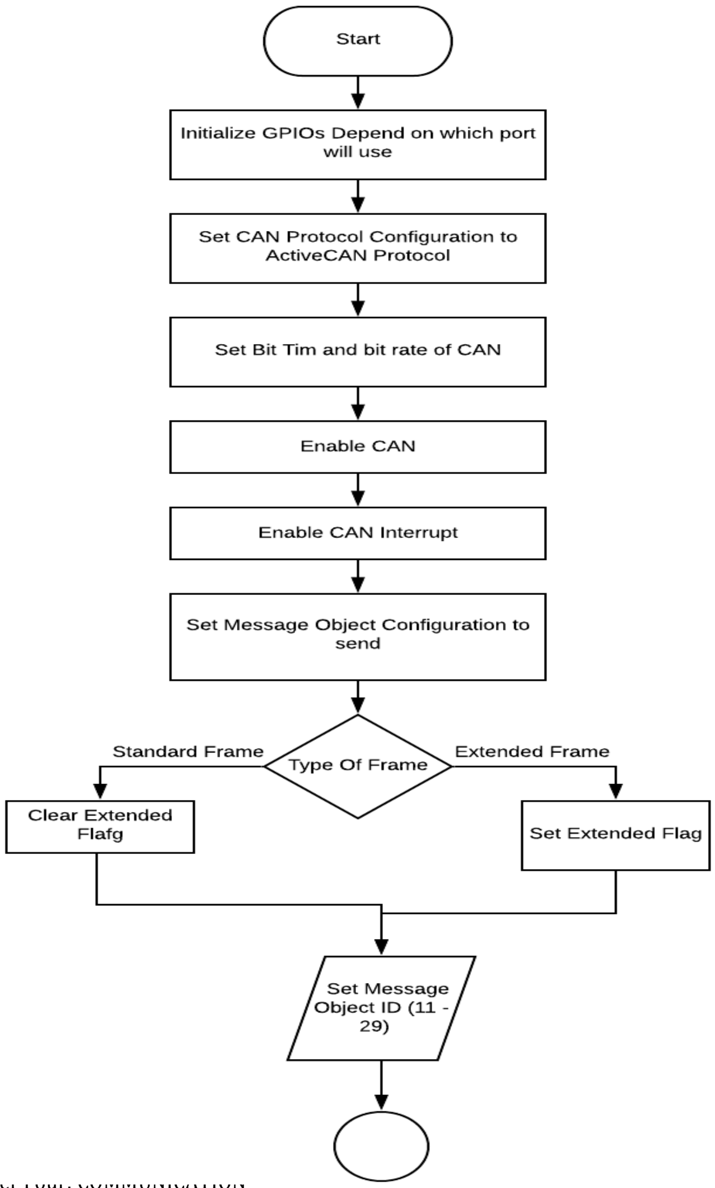
```
identifier = 0x123, mask = 0x7F0 -> matches frames
with identifiers 0x120, 0x121, 0x122, 0x123, ...,
0x12F
```

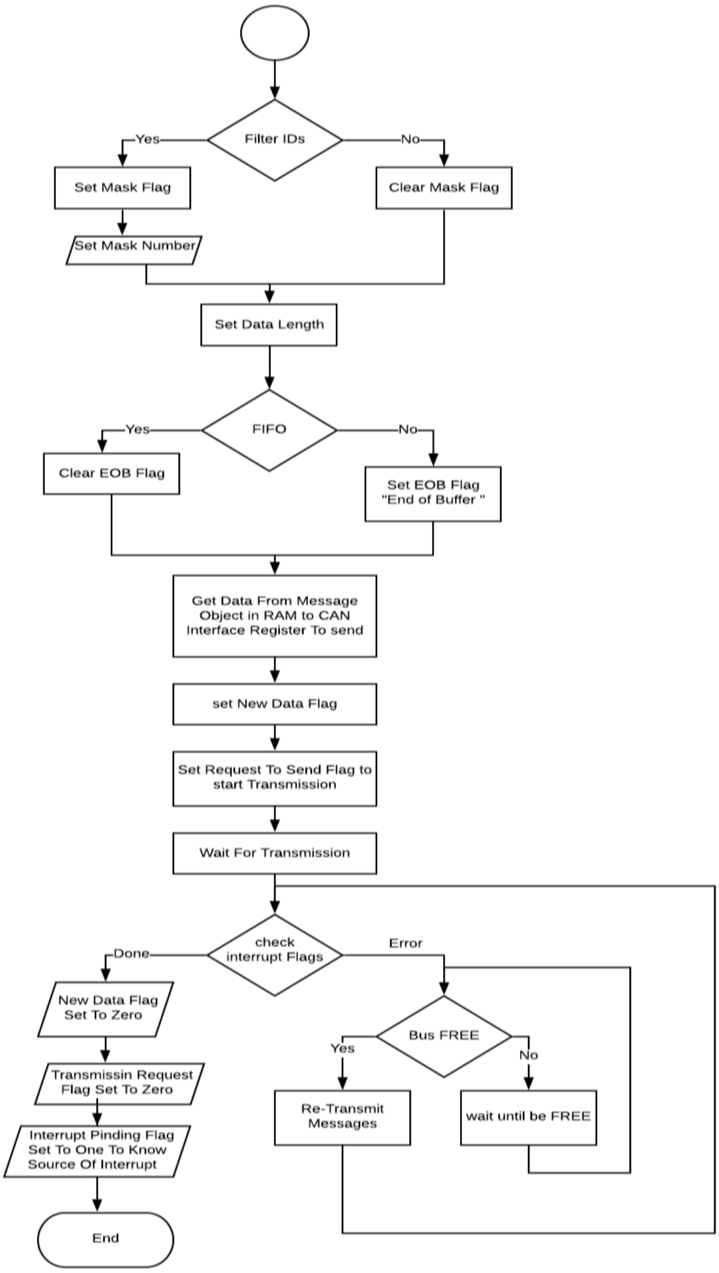
4.4.3 FIFO Buffer :

First in First Out concept using in CAN to Store multiple Messages Received in RAM . as Receiving each message Individually keeps interrupting the CPU for every single Message receives. It will make system overhead especially in multitasking systems .
When the CPU transfers the contents of a message object from a FIFO buffer . To assure the correct function of a FIFO buffer, the CPU should read out the message objects starting with the message object with the lowest message number.
When reading from the FIFO buffer, should be aware that a new received message is placed in the message object with the lowest message number. As a result, the order of the received messages in the FIFO is not guaranteed.

4.5 Flow Charts

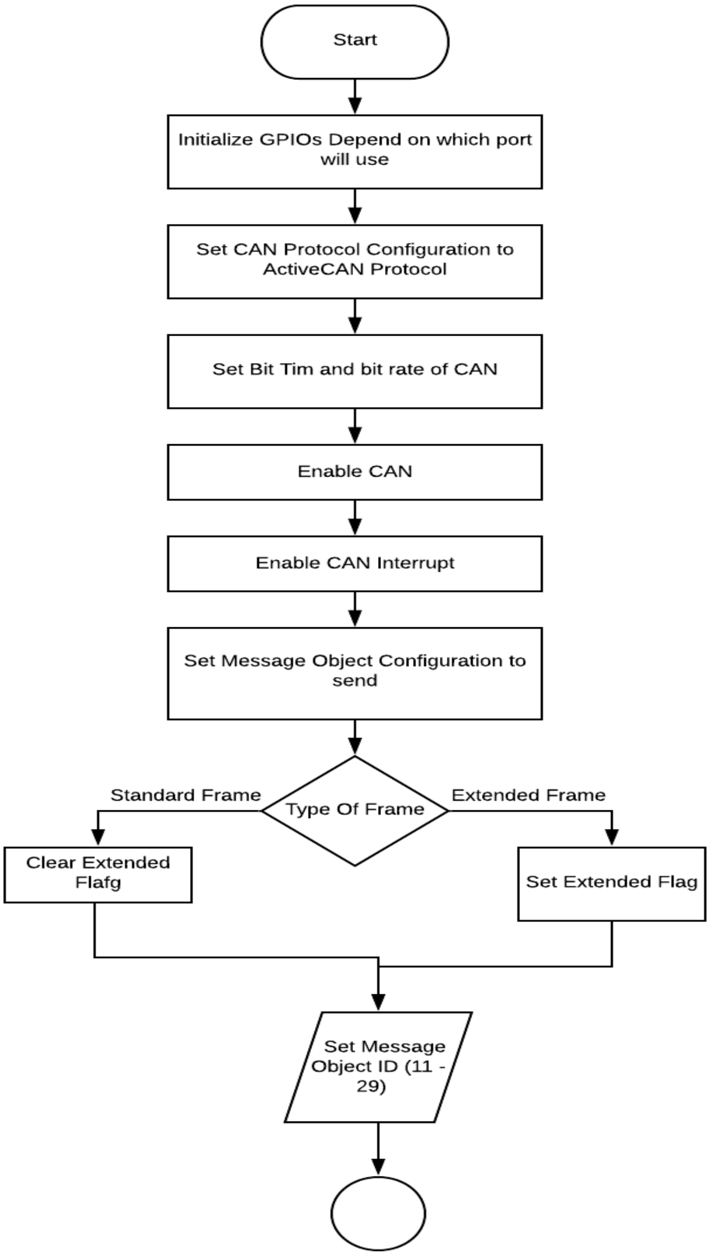
4.5.1 Transmission Flow Chart

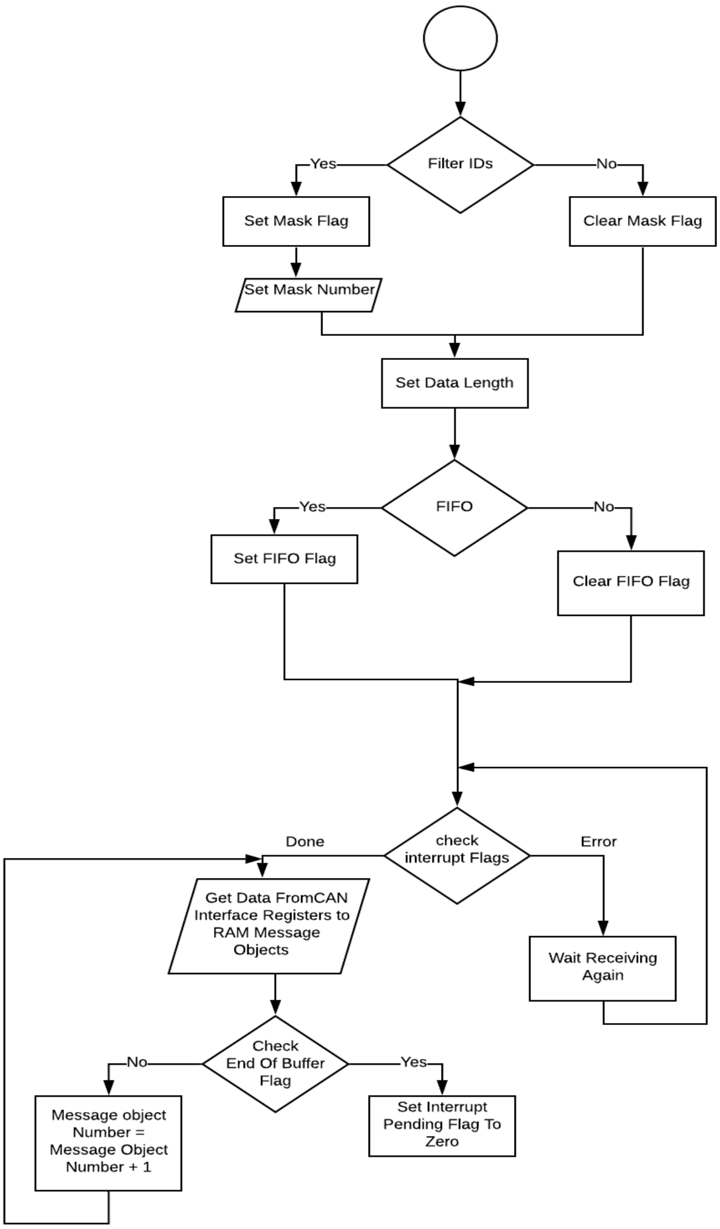

Start
Initialize GPIOs Depend on which port will use
Set CAN Protocol Configuration to ActiveCAN Protocol
Set Bit Tim and bit rate of CAN
Enable CAN
Enable CAN Interrupt
Set Message Object Configuration to send
Type Of Frame
Standard Frame
Extended Frame
Clear Extended Flafg
Set Extended Flag
Set Message Object ID (11 - 29)


Filter IDs
Yes
No
Set Mask Flag
Clear Mask Flag
Set Mask Number
Set Data Length
FIFO
Yes
No
Clear EOB Flag
Set EOB Flag "End of Buffer "
Get Data From Message Object in RAM to CAN Interface Register To send
set New Data Flag
Set Request To Send Flag to start Transmission
Wait For Transmission
check interrupt Flags
Done
Error
New Data Flag Set To Zero
Transmissin Request Flag Set To Zero
Interrupt Pinding Flag Set To One To Know Source Of Interrupt
End
Bus FREE
Yes
No
Re-Transmit Messages
wait until be FREE

Figure(4-11)

4.5.2 Receiving Flow Chart


Start
Initialize GPIOs Depend on which port will use
Set CAN Protocol Configuration to ActiveCAN Protocol
Set Bit Tim and bit rate of CAN
Enable CAN
Enable CAN Interrupt
Set Message Object Configuration to send
Type Of Frame
Standard Frame
Extended Frame
Clear Extended Flafg
Set Extended Flag
Set Message Object ID (11 - 29)


Filter IDs
Yes
No
Set Mask Flag
Clear Mask Flag
Set Mask Number
Set Data Length
FIFO
Yes
No
Set FIFO Flag
Clear FIFO Flag
check interrupt Flags
Done
Error
Get Data FromCAN Interface Registers to RAM Message Objects
Wait Receiving Again
Check End Of Buffer Flag
No
Yes
Message object Number = Message Object Number + 1
Set Interrupt Pending Flag To Zero

Figure(4-12)

CHAPTER FIVE FIRMWARE AND BOOTLOADER

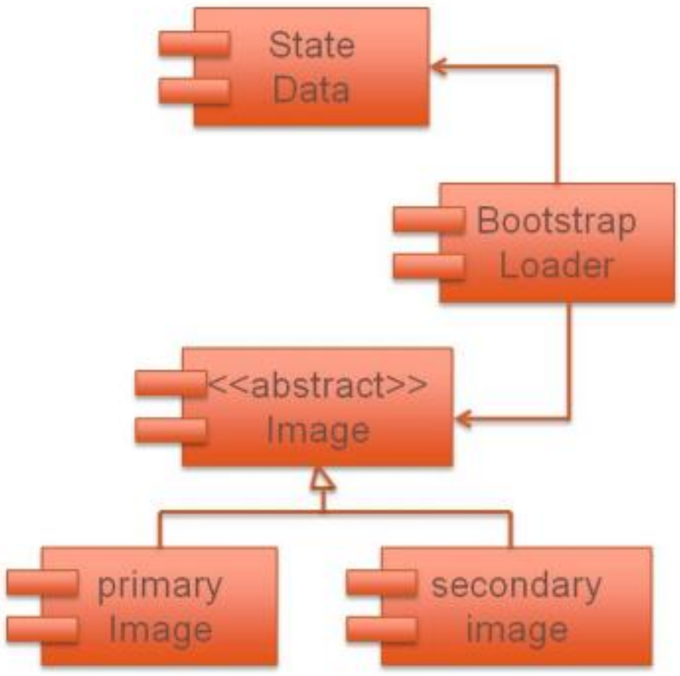
Firmware can be mainly referred to as being a fixed, rather small program that controls hardware in a system. Firmware is generally responsible for very basic low-level operations without which a device would be completely non-functional.

The bootloader is the first code that is executed after a system reset. Its goal is to bring the system to a state in which it can perform its main function. This requires hardware initialization and choosing the correct image to load from flash. Because of its key role, the bootloader is usually placed in a part of the Flash that is protected from accidental erasure or corruption.

5.1 Updating a firmware: When? Why?

Even though firmware is not designed to be changed, updates in a bootloader or firmware are usually needed to correct bugs or to add new functionalities. Indeed, there are firmware that are not correctly designed and so contain bugs which can sometimes be critical. Sometimes, the firmware is just too old and does not comply with the client desires any more. In all these situations, a new firmware is needed.
Using Over-The-Air update: the new firmware is download thanks to wireless communication using a very specific protocol just made for the update operation.

So, updating a firmware is sometimes necessary. However, the update operation is critical because. If an unexpected problem occurs during the transmission and the updating process has not been designed accordingly, all the system can be blocked. This can be the case, for example if connection is lost or if the processor crashes. The first embedded systems were totally blocked when this type of event occurred. Nowadays, there are some methods which have been imagined to realize safe updates on embedded systems. The most famous and logical one can be described by Figure(5-1).



Firmware redundancy

Figure(5-1)

The idea is to keep several firmware images in memory. So if one crashes when uploaded, the bootloader chooses another one.

5.1.1 Problems

As mentioned, when performing an update on an embedded application several problems may appear. If ignored they lead to a non-functional device.
The main problems identified are:

- Power failure during upgrade: the consequence of this is that new firmware is partially written.
- Bad firmware: the consequence of this is that the device may stop functioning (e.g. bad firmware update for cameras)
- flash corruption: NAND flash have a certain ratio error which may prove to be significant while updates are written.
- Communication errors: The consequence of this is partially written firmware.

5.1.2 Solutions

The most common solution is to always keep in Flash the firmware that was written before the device was sold or put in function. Whenever a new firmware is written ensure that the bootloader can switch to the initial firmware in case the new one fails.

5.2 Boot-loader Requirements

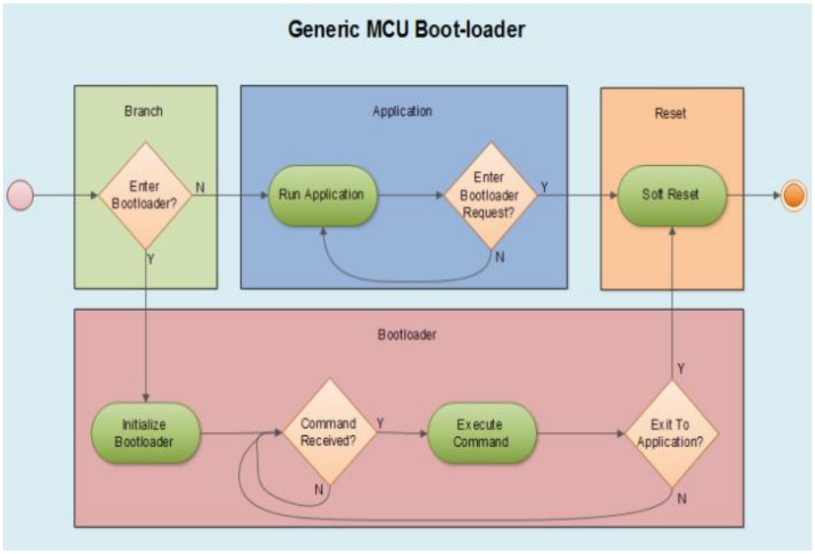
Each boot-loader will have its own unique set of requirements based on the type of application; however, there are still a few general requirements that are common to all bootloaders. They are

1) Ability to switch or select the operating mode (Application or boot-loader).
2) Communication interface requirements (USB, CAN, I2C, USART, etc.)

3) Flash system requirements (erase, write, read, location).
4) Application checksum (verifying the app is not corrupt).
5) Code Security (Protecting the boot-loader and the application).

5.3 The Boot-loader System

Boot-loaders can come in many different sizes and in many different flavors but in general the operation of a system with a boot-loader is relatively standard. There are three major components to these systems that can be seen in the Figure. They are the branching code (green), the application code (blue) and the boot-loader code (red). For most systems we prefer them to be executing the application path the majority of their operational life. Figure(5-2) highlights the execution path to get to the application by the dashed red line. The orange block is a common block used by both the bootloader and the application to reset the system.



Figure(5-2)

5.3.1 Boot-loader Behavior

A boot-loader in itself is not that different from a standard application; in fact, it is a standard application. What makes a boot-loader special is that it is sharing flash space with another application and has the capability to erase and program a new application in its place. Like every other application, one of the first priorities of the boot-loader is to initialize the processor and the minimum number of peripherals required to carry out boot-loading functions. It's best to keep the use of peripherals to as few as possible in the boot-loader in order to attempt to maximize the flash space that will be available for the application code.
The typical sequence for programming a device can be:

1) Start the boot-loader
2) Erase the flash
3) Send binary file information to the boot-loader
4) Generate Checksum
5) Quit the boot-loader and enter the application

5.3.2 Application Behavior

The behavior of the application image is of for the most part not of any interest to the bootloader designer except in one aspect; the application needs to be capable of receiving a command to enter the boot-loader. This means that the application needs to have two boot-loader like capabilities:

1) set a piece of information that the boot-loader can detect to enter boot-load mode.
2) Reset the system to initiate a branching decision.

5.3.3 Start-Up Branching

When the system starts up, there are at least two different software images that can be loaded and executed by the micro-controller, the boot-loader, the application and possibly a backup application image. It is therefore necessary that as part of the boot-loader image code a branching algorithm be included that can handle the decision making process of which image to load.

There are many different methods that can be used to decide which image to load. The simplest method that can be used and is most often used in example code from chip manufacturers is the use of a GPIO line to make the decision. For example, if the GPIO signal is high, load the application; if it is low then load the boot-loader.

A single I/O line being used to branch to the boot-loader is not a very robust solution. A user could accidentally push the boot-load button or a noise event could cause the system to enter the boot-loader. Instead, a common method used to detect a request to enter the boot-loader is to change an EEPROM value.

The robust solution to the branching code is that the code will calculate the application image checksum. When it is completed, the branching code will check "Does the application reset vector exist?"

5.3.4 Memory Partitioning

Every microcontroller has some form of non-volatile memory that is used to store the program. The most commonly used type of memory is flash. Flash is broken up into divisible sections. The smallest section of flash is often referred to as a page. Pages are organized into larger structures known as sectors. Sectors are in turn organized into larger structures known as blocks.
Each microprocessor is different as to how these sections of flash

can be manipulated. Most will allow you to write a single byte to flash at a time. Others may require that 8 bytes or 256 bytes be written at one time. In most cases, the smallest section of flash that can be erased at a time is a single sector that often consists of 4kB. It is important that before a designer gets too far in their boot-loader design that they pull out the microcontroller datasheet and read through the flash and memory organization chapters.

The primary purpose for detailed examination of the memory map is to determine what sections of flash are available and best used for the boot-loader.

There are a number of factors that should be taken into account when selecting where to locate the boot-loader.

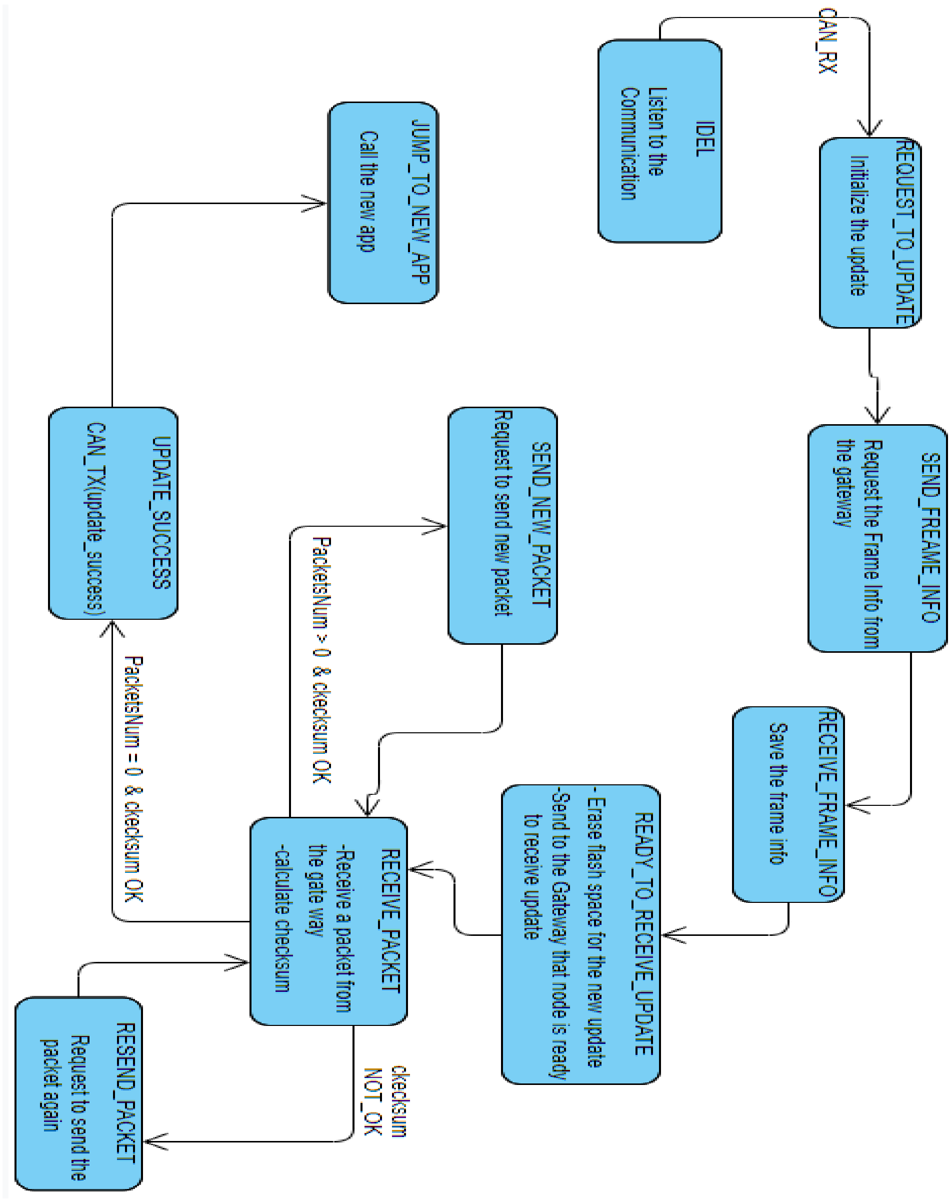
1) Size of the boot-loader
2) Location of the vector tables
3) Write protection and code security flash sections

5.3.5 Embedded Application Setup

In order for the embedded application to function properly there is an important change that needs to be made so that it plays nice with the boot-loader. The linker file needs to be updated so that it does not try to locate code or data within the boot-loader memory space.

The linker file is used by the compiler to decide where to place functions in RAM and flash. While the boot-loader memory space would be protected within flash, if the application tries to locate itself in the flash space the boot-loader would ignore those memory locations. The application would never get written correctly to flash. This could lead to a unit that is completely non-functional. It is therefore imperative that the linker be modified so that it does not occupy the same space. Figure(5-3) shows this sequence.

5.4 Update State Machine Diagram


IDEL
Listen to the Communication
CAN_RX
REQUEST_TO_UPDATE
Initialize the update
SEND_FREAME_INFO
Request the Frame Info from the gateway
RECEIVE_FRAME_INFO
Save the frame info
READY_TO_RECEIVE_UPDATE
- Erase flash space for the new update
-Send to the Gateway that node is ready to receive update
RECEIVE_PACKET
-Receive a packet from the gate way
-calculate checksum
ckecksum NOT_OK
RESEND_PACKET
Request to send the packet again
SEND_NEW_PACKET
Request to send new packet
PacketsNum > 0 & ckecksum OK
PacketsNum = 0 & ckecksum OK
UPDATE_SUCCESS
CAN_TX(update_success)
JUMP_TO_NEW_APP
Call the new app

Figure(5-3)

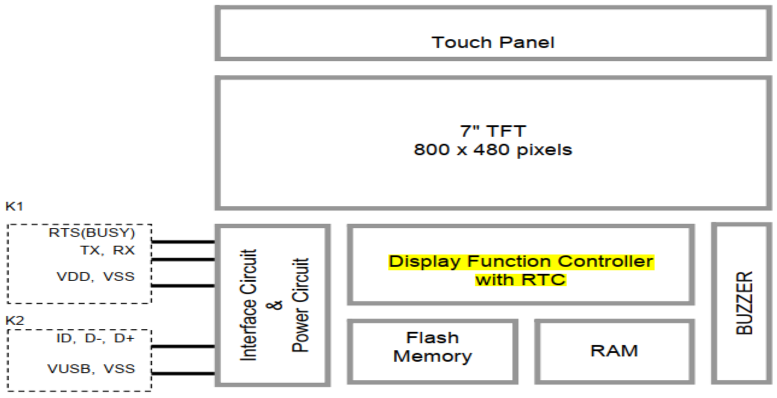
CHAPTER SIX SMART LCD

TOPWAY is a Smart TFT Module with 32bit MCU on board. Its graphics engine provides numbers of outstanding features. It supports TOPWAY TML 3.0 for preload and pre-design display interface that simplify the host operation and development time.

6.1 Smart LCD Highlight

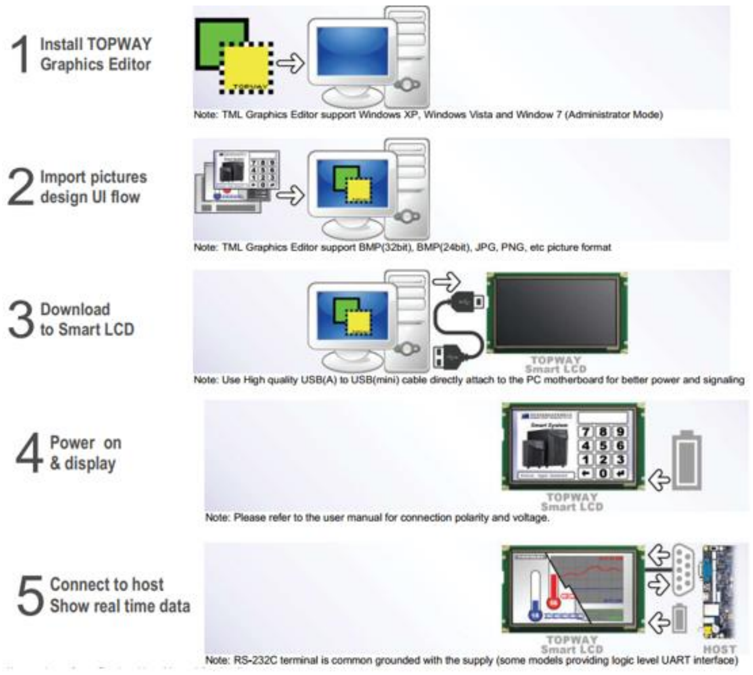
- Standard RS232-C communication interface
    - Reliable packet protocol ensure reliable communication
    - Host data could be accepted at any moment
    - Free the host form interface response and handling
    - Direct connect the USB terminal to PC for development
    - USB thumb drive with OTG cable can be used for data preload in production stage
    - 256Mbyte Flash (vary by model) for interface pictures preload (more than 300 pictures (800x480))

6.2 Block Diagram



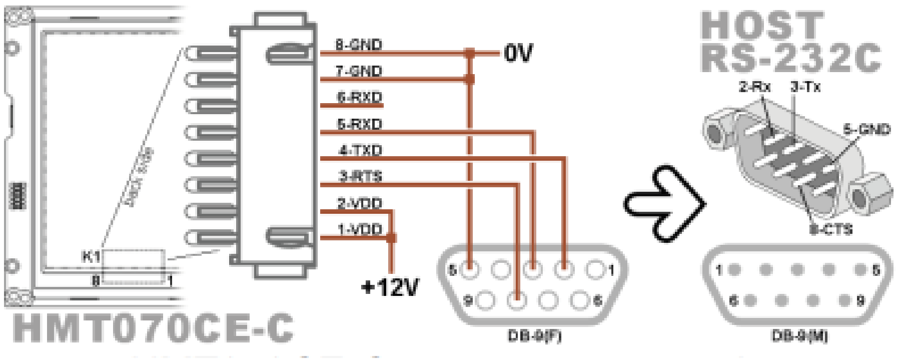
Figure(6-1)

6.3 How to Use a Smart LCD


1 Install TOPWAY Graphics Editor
Note: TML Graphics Editor support Windows XP, Windows Vista and Window 7 (Administrator Mode)
2 Import pictures design UI flow
Note: TML Graphics Editor support BMP(32bit), BMP(24bit), JPG, PNG, etc picture format
3 Download to Smart LCD
TOPWAY Smart LCD
Note: Use High quality USB(A) to USB(mini) cable directly attach to the PC motherboard for better power and signaling
4 Power on & display
TOPWAY Smart LCD
Note: Please refer to the user manual for connection polarity and voltage.
5 Connect to host Show real time data
TOPWAY Smart LCD
HOST
Note: RS-232C terminal is common grounded with the supply (some models providing logic level UART interface)

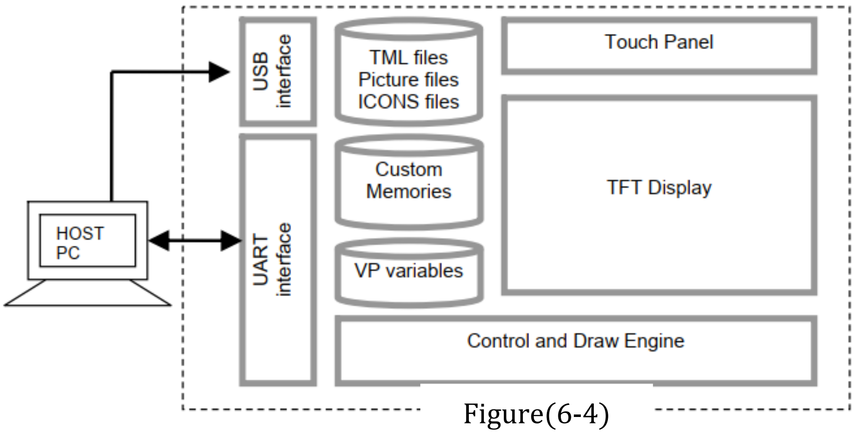
Figure(6-2)

-Smart LCD Connection



Figure(6-3)

6.4 Operation Function Descriptions

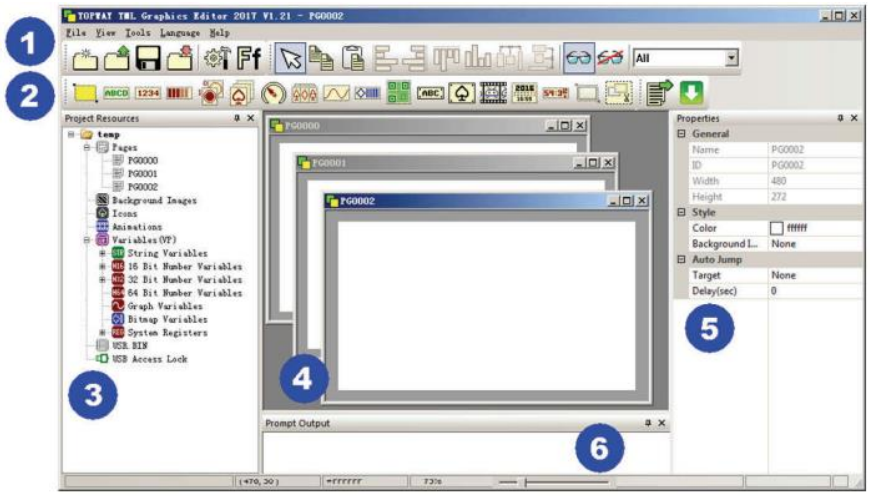


Figure(6-4)

- TML files, Picture files, ICON files are stored inside FLASH memory area. They are preloaded to LCD for stand alone interface use.
- Those files are preloaded via USB interface as an USB drive.
- All the interface flow and the touch response are based on the preloaded TML files
- VP variables memory is inside RAM area, it provides real time access via UART by the HOST or display onto the TFT by TML file.

- Custom Memories are inside FLASH memory area It can be accessed via UART interface by the HOST.
- Control and Draw Engine executes HOST commands and response respectively
- It also reports the real time Touch Key number to the HOST

6.5 Graphics Editor



Name	PG0002
ID	PG0002
Width	480
Height	272
Color	ffffff
Background I...	None
Target	None
Delay(sec)	0

Figure(6-5)

1- Menu	Provide basic software operation, View options, tools options, etc.
2- Tools bar	There are four type of tools - file tools for project open, save, compile output, etc. - alignment and display filter tools - display elements tools - compile and download tools
3- Project Resources window	Resource windows (right click on the resources) - built new page, - import pictures (IMG_BKG, IMG_ICO, IMG_ANI) - allocated VP variable (VP_N16, VP_N32, VP_N64) - user file, etc...
4- Working Area	The working area for composing the display page.

	User could build element onto the page.
5-Properties Window	Display the selected element properties or Page properties
6-Prompt Output Window	Prompt output window show the compiling information, warning and error information

6.6 Serial Communication

Smart LCD serial command is for real-time control and access. Host machine get the data which input through the Smart LCD interface or provide the data for display.

6.6.1 Hardware connection

Smart LCDs serial UART interface are based on RS232-C standard by default config as 8N1 115200bps.

6.6.2 Communication Packet Structure

Commands and Response Packet should be format as follow (host → module):

Seq	Code	Code type	Description
1	0xAA	Packet header	1byte
2	Cmd-code	Command code	1byte
3	Par-data	Parameter or Data	
:	:	-	-
:	:	-	-
:	:	-	-
N-3 th	0xCC		
N-2 th	0x33		
N-1 th	0xC3		

N th	0x3C		

6.6.3 Packet Acknowledgment

Packet Acknowledgment is two byte in ASCII (module → host):

Response	code	Description
Command (in packet) executed and wait for next Command	":>"	In ASCII (0x3a, 0x3e)
Command (in packet) error and wait for next Command	"!>"	In ASCII (0x21,0x3e)
Invalid Packet	null	No response

CHAPTER SEVEN GATEWAY

7.1 Abstract

Increased consumer demand for greater vehicle functionality is spurring more complex electronics in cars with an increased number of computers called Electronic Control Units (ECUs) with different network interfaces. Modern vehicles can integrate over 100 ECUs connected over multiple networks such as CAN (Control Area Network), LIN (Local Interconnect Network), FlexRay, and Ethernet.

A gateway is a central hub that securely and reliably interconnects and processes data across these heterogeneous vehicle networks. It provides physical isolation and protocol translation to route data between functional domains (powertrain, chassis and safety, body control, infotainment, telematics, ADAS) that share data to enable new features. Gateways allow engineers to design more robust and functional vehicle networks that can enhance the driving experience.

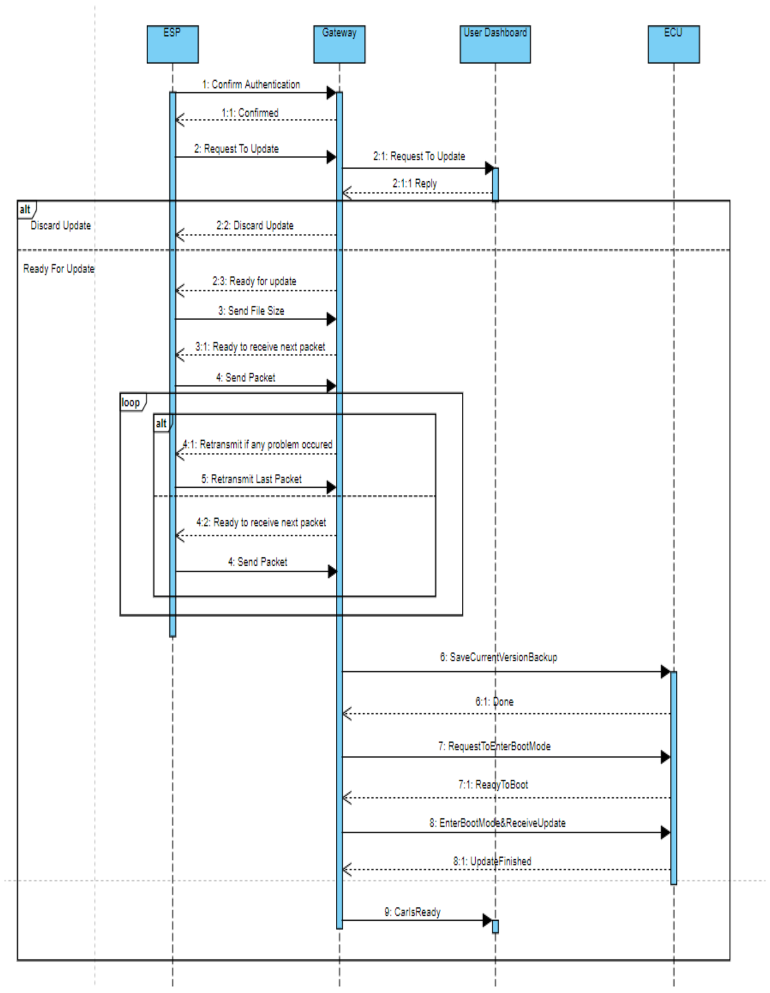
Vehicle manufacturers (OEMs) are highly motivated to create new features to differentiate themselves from competition. A gateway is essential for enabling autonomous driving which requires secure connectivity and high bandwidth communications across functional domain ECUs. Being central to the vehicle networks, the gateway is also ideal to support vehicle-wide applications such as Over-the-Air (OTA) updates and vehicle analytics with secure communications to OEM servers (cloud).

7.2 Our Gateway

Our gateway is responsible for communicating with telematics unit (ESP) and receive update packets, decrypt them, store in memory and then transmit each piece of update to its appropriate node through CAN bus.

Figure(7-1) shows the sequence diagram of the gateway.

7.2.1 Gateway Sequence diagram



Figure(7-1)

7.2.2 Gateway code architecture

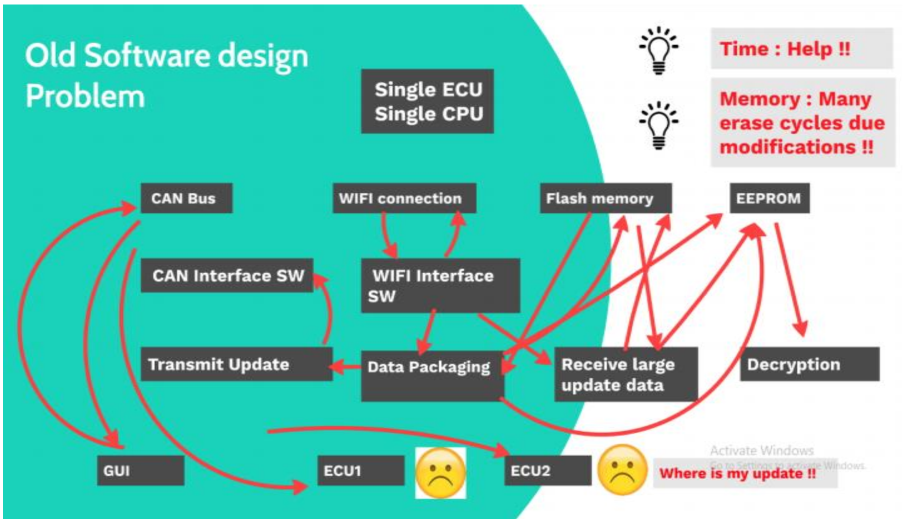
The gateway is responsible for handling a huge amount of data, and communicating with many nodes and with telematics unit, and performing many tasks, so we have emulated the application and RTE layers of AUTOSAR architecture in the gateway design to simplify its work, and also we have used the AUTOSAR memory stack to handle storing and restoring data to/from nonvolatile memory safely and easily.

7.2.2.1 Gateway tasks

Gateway is responsible for 4 main tasks :

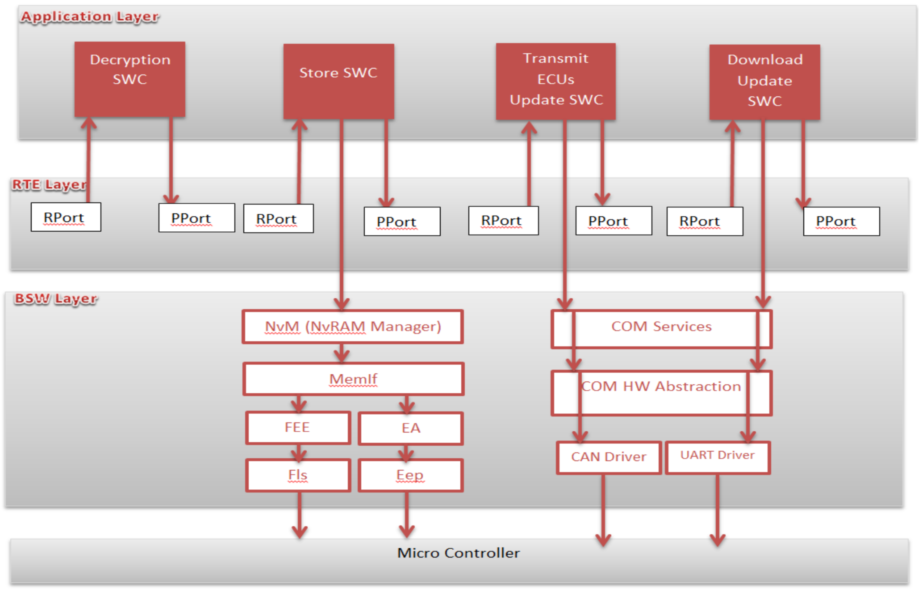
1. Download update from ESP.
2. Decrypt the received packets.
3. Store decrypted packets into nonvolatile memory.
4. Transmit ECUs Update.

Before AUTOSAR:



Figure(7-2)

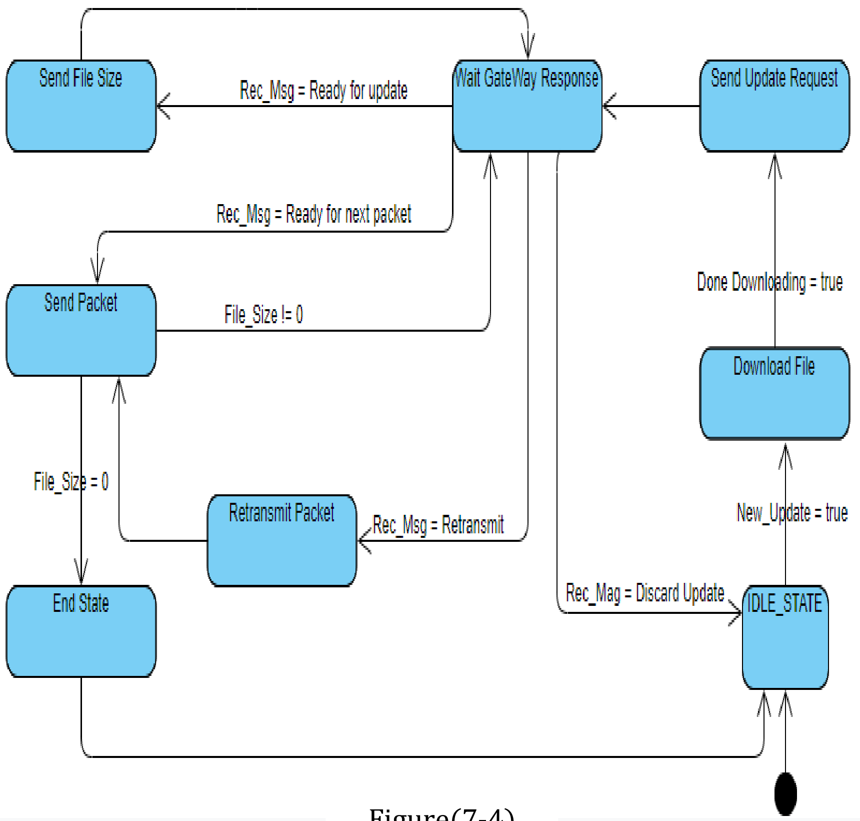
After AUTOSAR :



Figure(7-3)

7.2.2.2 NodeMCU Side

Figure(7-4) shows the State diagram of internal work flow of NodeMCU Application.



Figure(7-4)

Appendix I

Hardware Components & Prices

Serial no.	**Component**	**Number of pieces**	**Price**
1.	UART LCD7”,800x480, Consuming LCD Module-Closed Frame	1	1500 L.E
2.	TM4c123GH6pm MicroController	3	3x600 L.E
3.	Stm32f407VG microcontroller	1	800 L.E
4.	NodeMCU(ESP8266 WiFi Programming & Development Kit)	1	140 L.E
5.	CAN bus Transceiver	4	4x40 L.E
Total row	**---**	**10**	**4400 L.E**

Code

You can find the project full repository here:

https://github.com/SaharElnagar/OTA_GraduationProject2020

Sources and References

1. OTA Technical papers & articles

   - Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned.
   - Making Full Vehicle OTA Updates a Reality BY : Daniel Mckenna , BU Automotive , NXP Semiconductors.
   - Automotive OTA The potential and the challenge BY: Dr. Walter J. Buga, CEO.

2. AUTOSAR Documents

   https://www.autosar.org/standards/classic-platform/

3. TI FEE User Guide

   https://processors.wiki.ti.com/images/8/88/TI_FEE_User_Guide.pdf

4. Tiva TM4c123gh6pm Micro Controller Data Sheet

   https://www.ti.com/lit/ds/spms376e/spms376e.pdf?ts=1595585569401&ref_url=https%253A%252F%252Fwww.google.com%252F

5. practical microcontroller engineering with arm technology BY: Dr. Ying Bai

6. CAN technical papers

Understanding and Using the Controller Area Network Communication Protocol by A. Ghosal, Haibo Zeng, and Marco Di Natale.

Computation of CAN Bit Timing Parameters Simplified Meenanath Taralkar, OTIS ISRC PVT LTD, Pune, India.

The configuration of CAN Bit Timing BY: 6th International CAN Conference 2nd to 4th November, Turin (Italy).

- BOSCH CAN Specification.

7. BootLoader technical papers

- Firmware and bootloader Otilia Anton, Brice Gelineau and J´er´emy Sauget 16 mars 2012
- Bootloader Design for Microcontrollers in Embedded Systems By Jacob Beningo
- Cortex™-M4 Devices Generic User Guide
- AN2606 Application note STM32 microcontroller system memory boot mode
- AN3155 Application note USART protocol used in the STM32 bootloader

8. Bootloader & CAN

- RM0090 Reference manual STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs.
- Mastering STM32 BY Carmine Noviello.
- The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors.

9. LCD

- HKT070DTA-1C LCD Module User Manual
- User_handbook_TOPWAY_SmartLCD(Editor2017)(en)Rev103