Development of ESP32 CAN Driver

Arduino ESP32 CAN Library

Mohamed Irfanulla MOHAMED ABDULLA

This dissertation is submitted for the degree of
MASTER IN CONTROL AND ROBOTICS
"EMBEDDED REAL TIME SYSTEMS"

Supervisor: **Prof. Pierre Molinaro**,
Maître de Conférences,
École Centrale de Nantes

Reviewer: **Prof. Mikaël Briday**,
Maître de Conférences,
IUT de Nantes

Ecole Centrale de Nantes 28 August 2019

Acknowledgements

Foremost, I would like to express my deep sense of thanks and gratitude to my mentor Prof.Pierre Molinaro, for providing me with this opportunity to pursue my Master degree Thesis. His dedication, guidance, meticulous scrutiny and advice have helped me to a great extent in accomplishment of my work and the report.

I thank profusely all the professors of Control and Robotics, and special thanks to Embedded Real Time System group for their support throughout my Master study.

I would like to thank my parents and friends for their unconditional love and support. Finally, I would like to thank Almighty for keeping me healthy and mentally fit during my work.

Abstract

ESP32 is a low-cost, low-power System on a Chip series of microcontroller with a highly integrated dual core microprocessor, developed by Espressif Systems. The MCU is designed to formulate embedded system and power internet of things usage in professional projects. The ESP32 supports Control Area Network protocol (CAN), the peripheral contains a CAN controller compatible for CAN 2.0B specification. In an effort, this project draw the development of ESP32 CAN driver. The developed driver is featured by the Arduino-ESP32 SDK. The technical specification document of ESP32 does not contain any details of the ESP32 CAN Controller. From discussions it is found that ESP32 CAN controller is a compatible NXP SJA1000 CAN controller. ESP32 CAN registers are reformed from SJA1000 and the registers are documented in this report. The library is written in ACAN style for easy configuration and usage. Efficient CAN-bit settings calculator, customisable driver buffer size and acceptance filter settings are in-built features of the driver. This driver is developed and tested on a ESP32 MiniKit development board integrated with MCP2515 and MCP2517 CAN controller and MCP2562 transceiver. Sequential data flow is successfully achieved by the ESP32 CAN controller using the driver. The operation of the driver is tested using arduino example codes and a demo of the transmitter model is verified using tool UPPAAL.

Keywords : `ESP32`, `Espressif System`, `Controlled Area Network (CAN)`, `ESP32 CAN Driver`, `SJA100 CAN Controller`, `ESP32-MiniKit`.

Table of contents

List of figures vii

List of tables ix

1 CAN Network 2
- 1.1 History . . . 2
- 1.2 CAN Architecture . . . 3
  - 1.2.1 What is CAN? . . . 3
  - 1.2.2 CAN Network Layer . . . 4
  - 1.2.3 CAN Node . . . 5
  - 1.2.4 CAN Bus . . . 5
  - 1.2.5 CAN Signals . . . 6
  - 1.2.6 CAN Types . . . 7
  - 1.2.7 CAN Message . . . 9
- 1.3 Bit Timing Requirements . . . 11
  - 1.3.1 Calculation of Bit Timing Parameters . . . 11
- 1.4 Synchronization . . . 13
- 1.5 CAN Operation . . . 14
  - 1.5.1 CAN Node Transmitting . . . 14
  - 1.5.2 CAN Node Receiving . . . 15

2 ESP32 Overview 16
- 2.1 About ESP32 . . . 16
- 2.2 ESP32 Specification . . . 16
  - 2.2.1 Key Features . . . 17
- 2.3 MH-ET LIVE ESP32 MiniKit . . . 18
- 2.4 Programming Environment . . . 19
  - 2.4.1 Installing the ESP32 Board in Arduino IDE . . . 19
- 2.5 ESP32 CAN Setup . . . 21
- 2.6 ESP32 MiniKit Development Board . . . 22

3 ESP32 CAN Peripheral 23
- 3.1 ESP32 CAN Peripheral Registers . . . 23

3.1.1 Mode Register (CAN_MODE) . . . 24
3.1.2 Command Register (CAN_CMD) . . . 25
3.1.3 Status Register (CAN_STATUS) . . . 26
3.1.4 Interrupt Register (CAN_INTERRUPT) . . . 28
3.1.5 Interrupt Enable Register (CAN_IER) . . . 29
3.1.6 Arbitration Lost Capture Register (CAN_ALC) . . . 29
3.1.7 Error Code Capture Register (CAN_ECC) . . . 30
3.1.8 Error Warning Limit Register (CAN_EWLR) . . . 30
3.1.9 Receive Error Counter Register (CAN_RXERR) . . . 31
3.1.10 Transmit Error Counter Register (CAN_TXERR) . . . 31
3.1.11 Receive Message Counter Register (CAN_RXM_COUNTER) . . . 32
3.1.12 Clock Divider Register (CAN_CLK_DIVIDER) . . . 32
3.2 Transmit and Receive Buffer Register . . . 33
3.2.1 Transmit and Receive Buffer Frame Information (CAN_FRAME_INFO) . . . 33
3.2.2 Transmit and Receive Buffer Identifier (CAN_ID_SFF and CAN_ID_EFF) 34
3.3 CAN Bit Timing Registers . . . 35
3.3.1 Bus Timing Register 0 (CAN_BTR0) . . . 36
3.3.2 Bus Timing 1 Register (CAN_BTR1) . . . 37
3.3.3 ESP32 CAN bit time constraints . . . 37
3.4 Acceptance Filter Register . . . 38

4 ESP32 CAN Driver 40
4.1 Computation of CAN Bit Settings : ESP32ACANSettings class . . . 40
4.1.1 ESP32ACANSettings Constructor . . . 40
4.1.2 ESP32ACANSettings class Properties . . . 41
4.1.3 ESP32ACANSettings Methods . . . 42
4.1.4 Test on Desktop compiler . . . 43
4.2 ESP32 CAN Register Definition . . . 44
4.2.1 ESP32 CAN Registers Test . . . 46
4.3 CAN Driver Initialization . . . 48
4.3.1 Configuration Functions . . . 50
4.4 CAN Communication Process . . . 51
4.4.1 Polling Controlled Process . . . 52
4.4.2 Interrupt Controlled Process . . . 53
4.5 Driver Send and Receive method . . . 56
4.5.1 Transmission Method . . . 56
4.5.2 Receive Method . . . 58
4.5.3 The Driver Error Code . . . 60
4.6 CANMessage class . . . 60
4.7 Acceptance Filter: ESP32ACANFilter class . . . 61
4.8 Driver Model Verification . . . 61

5 ESP32 CAN Driver Examples 63
5.1 LoopBackCheck-Polling . . . 68
5.2 LoopBackCheck - Interrupt . . . 68
5.3 LoopBackCheck - Intensive . . . 70
5.3.1 Intensive check by Polling . . . 70
5.3.2 Intensive check with Interrupt . . . 71
5.4 Normal Operation Mode . . . 71
5.4.1 Test with MCP2515 CAN controller . . . 72
5.5 Acceptance Filter Settings . . . 73

6 Conclusion 75

References 76

Appendix A ESP32 CAN Driver 78
A.1 ESP32 CAN Driver Register Summary . . . 80

Appendix B CAN Handbook 81
B.1 Bit Rate . . . 81
B.2 Remote Data Request . . . 81
B.3 Priorities . . . 81
B.4 CSMA/CD-CR . . . 82
B.5 Arbitration . . . 82
B.6 Bit Stuffing . . . 82
B.7 Error Handling . . . 83

List of figures

1.1 CAN Network Layer 4
1.2 CAN Node 5
1.3 CAN BUS 6
1.4 CAN Signal 6
1.5 CAN 2.0A Standard CAN 8
1.6 CAN 2.0B Standard CAN 8
1.7 CAN 2.0B Extended CAN 9
1.8 CAN Message Frame 9
1.9 Error Frame 10
1.10 CAN Bit Timing Segments by ISO-11898 11
1.11 CAN Node Transmission 15
1.12 CAN Node Receiving 15

2.1 ESP32-D0WDQ6 Pin Layout [1] 17
2.2 ESP32-Function Block Diagram [1] 17
2.3 MH-ET LIVE ESP32-MiniKit 19
2.4 ESP32 CAN Node 21
2.5 ESP32 MiniKit Developement Board 22

3.1 ESP32 CAN Controller Register Map 23
3.2 Mode Register 24
3.3 Command Register 25
3.4 Status Register 27
3.5 Interrupt Register 28
3.6 Interrupt Enable Register 29
3.7 Arbitration Lost Capture Register 29
3.8 Error Code Capture Register 30
3.9 Error Warning Limit Register 30
3.10 Receive Error Counter Register 31
3.11 Transmit Error Counter Register 31
3.12 Receive Message Counter Register 32
3.13 Clock Divider Register 32
3.14 Frame Information Register 34

3.15 Standard Frame Format Identifier . . . 34
3.16 Extended Frame Format Identifier . . . 35
3.17 CAN Bit Time Segments . . . 36
3.18 Bus Timing 0 Register . . . 36
3.19 Bus Timing 1 Register . . . 37
3.20 Single Filter Mode (Standard and Extended Frame) . . . 38
3.21 Dual Filter Mode (Standard and Extended Frame) . . . 39
3.22 Example: Dual Filter Mode (Standard Frame) . . . 39

4.1 Output: Bit Time Settings(test on desktop) . . . 44
4.2 Output: All valid ESP32 CAN bit rate (test on desktop) . . . 44
4.3 Output: ESP32 CAN REGISTER reset value . . . 48
4.4 Driver Initial Configuration . . . 49
4.5 Transmission Polling Controlled . . . 52
4.6 Reception Polling Controlled . . . 53
4.7 Interrupt Handling Error . . . 55
4.8 Transmission Interrupt Controlled . . . 56
4.9 Reception Interrupt Controlled . . . 56
4.10 Driver Transmit Sequence Model . . . 62

5.1 LoopBackCheck Connection . . . 68
5.2 Output: LoopBackCheck-Polling . . . 69
5.3 Output: LoopBackCheck-Interrupt . . . 69
5.4 Output: LoopBackCheck-PollingIntensive . . . 70
5.5 Output: LoopBackCheck-InterruptIntensive . . . 71
5.6 ESP32 CAN and MCP2515 Test Setup . . . 72
5.7 MCP2515 Connection with ESP32 using VSPI pins . . . 72
5.8 Test with MCP2515 bit rate 625 kbit/s . . . 73
5.9 Test with MCP2515 bit rate 25 kbit/s . . . 73
5.10 Output: Acceptance Filter Settings . . . 74

A.1 ESP32 CAN peripheral register summary . . . 80

B.1 CAN Message Priority . . . 81
B.2 Bus Arbitration . . . 82
B.3 CAN Error States . . . 83

List of tables

1.1 CAN History [2] 3

2.1 ESP32-Specification 18

3.1 Transmit and Receive Buffer Layout for SFF and EFF 33
3.2 ESP32 CAN Bit Timing Constraints 38

4.1 Properties of the `ESP32ACANSettings` class 41
4.2 The `ESP32ACANSettings::CANBitSettingConsistency` method error codes 43
4.3 The `ESP32ACANSettings::begin` method error codes 60

5.1 ESP32 CAN Operating Modes 64
5.2 ESP32 CAN Message Processing Method 64

Introduction

The ESP32 module, little beast is an extremely capable wireless programmable microcontroller board developed by Espressif systems. It is preferred by all embedded and IoT developers reason of low cost, its cost is around $4. ESP32 achieves ultra-low power consumption which is a benefit for mobile devices, wearable electronics and IoT applications. The official development frame work for Internet of Things is Espressif IoT Development Framework (ESP-IDF). Another frame work for arduino platform is Arduino-ESP32. There are different versions of ESP32 modules and boards released by Espressif. We use the ESP32-MiniKit board.

Objective of this project is to develop a ESP32 CAN driver. ESP32 is integrated with a CAN peripheral that supports CAN2.0B specification. CAN FD is not supported by the ESP32 CAN controller. The challenge in the project is that the official technical specification document of ESP32 does not contain any details of the CAN peripheral. It does not specify the register map and any features of the CAN peripheral. In forum discussion and other ESP32 sources it is stated that the ESP32 CAN is NXP SJA1000 CAN controller compatible. From the discussion source and unofficial ESP32 CAN driver by Thomas Barth [3] with comparison of arduino ESP32 SDK, the register for ESP32 CAN peripheral are defined in similar way as other available peripheral description in the technical document.

The basic of CAN protocol and specification of CAN network is detailed in the Chapter 1, CAN Network. The structure of CAN frame and types are elaborated. In Chapter 2, ESP32 Overview specifies the key features of ESP32 and configuration of ESP32 in arduino IDE. The register mapping of the ESP32 CAN peripheral and the technical specification are detailed in Chapter 3, ESP32 CAN Peripheral. The driver class and utilization of the driver methods are explained in chapter 4, ESP32 CAN Driver. The driver operation and instantiation of the driver object are documented. In chapter 5, the example codes tested with the driver and output of the ESP32 CAN performance are provided.

The ESP32 is integrated with MCP2515 and MCP2517 CAN controller. The ACAN2515 can be downloaded from https://github.com/pierremolinaro/acan2515 and ACAN2517 from https://github.com/pierremolinaro/acan2517. These drivers are common for Arduino modules and ESP32 (Look the driver documentation for ESP32 adaptability).

Chapter 1

CAN Network

1.1 History

In the earlier stage of Automobile industry, the communication system in the car seemed to be chaotic. The electric world inside the car were not as stable as in the automation world, there were many interfaces from the engine that disturbed the communication between the Electronic Control Units (ECU). The data exchange between the ECU were implemented conventionally, (i.e a physical communication channel was allocated to every signal to be transmitted that led to huge increase of cables and consequently the weight of the car). In the early 80's, a group of people at Robert Bosch were the first to investigate the existing communication systems regarding the possible use on passenger cars.

Finally, to overcome the dilemma of limited data exchange, in 1986, at the SAE congress "*Controlled Area Network*" shortly `CAN` was introduced. In 1987, the first CAN controller chip, the 82526 was presented by Intel and shortly after the Philips Semiconductor presented the 82C200. The first automobile to implement the CAN feature is Mercedes-Benz W140 in the year 1992. Table 1.1 shows the history of CAN Network.

In 1993, the International Organization for Standardization (ISO) released the CAN standard ISO 11898 which was later restructured into two parts; ISO 11898-1 which covers the data link layer, and ISO 11898-2 which covers the CAN physical layer for high-speed CAN. ISO 11898-3 was released later and covers the CAN physical layer for low-speed, fault-tolerant CAN [4]. In 2012, Bosch released CAN FD 1.0 or CAN with Flexible Data-Rate. This specification uses a different frame format that allows a different data length as well as optionally switching to a faster bit rate. CAN FD is compatible with existing CAN 2.0 networks. Since 1994, several higher-level protocols have been standardized on CAN, such as CANopen and DeviceNet. These additional protocols were adopted by other industries, which are now standards for industrial communications.

Year	Event
1983	Bosch started internal projects to develop vehicle network
1986	Official release of CAN protocol
1987	First CAN controller chips from Intel and Philips Semiconductors
1991	Bosch CAN specification 2.0 published
	CAN-based higher-layer protocol introduced by Kvaser
1992	CAN in Automation (CiA) Group was established
	CAN Application Layer (CAL) protocol published by CiA
	First cars from Mercedes-Benz used CAN network
1993	ISO 11898 standard published
1994	1st international CAN Conference (iCC) organized by CiA
1995	ISO 11898 amendment (extended frame format) published
	CiA 301 CANopen protocol was published by CiA
2000	Development of the time-triggered communication protocol for CAN (TTCAN)
2003	Separation of data link and high-speed physical layer (ISO 11898-1 and -2)
2004	Publication of ISO 11898-4 (TTCAN)
2006	Publication of ISO 11898-3 (low-power, low-speed physical layer)
2007	Publication of ISO 11898-5 (low-power, high-speed physical layer)
2011	Start of the CAN FD protocol development
2012	Bosch released CAN FD 1.0
2013	Publication of ISO 11898-6 (physical layer with selective wake-up function)
2015	Publication of the reviewed ISO 11898-1 (Classical CAN and CAN FD)

Table 1.1 CAN History [2]

1.2 CAN Architecture

1.2.1 What is CAN?

CAN (*Controller Area Network*) is a multiplexed serial communication channel through which the data is transferred among distributed electronic module efficiently with a very high level of security. CAN is extremely a robust communication protocol. The CAN standard defines a communication network that establish a link to all the nodes that are connected in a bus to communicate with one another. CAN protocol is message based and not address based i.e, messages are not transmitted from one node to another node based on the address of a CAN node, instead a CAN node will broadcast it's message to all the nodes on the bus, and it is up to the receiving node to determine whether it should act on that message. Single or multiple nodes

may act on the same data. CAN allows distributed control across a network due to the reliability of the data. CAN comprise of the following properties [5].

1. Prioritization of messages.
2. Multicast reception with time synchronization.
3. Multimaster.
4. Guarantee of latency times.
5. Configuration flexibility.
6. System wide data consistency.
7. Error detection and signalling.
8. Automatic re-transmission of error messages as soon as the bus is idle again.

1.2.2 CAN Network Layer

To achieve the design transparency, implementation and flexibility, CAN has been subdivided into three layers; Object Layer, Transport Layer and Physical Layer. In terms of the Open Systems Interconnection model (ISO/OSI), CAN specification; ISO 11898, deals with only the Physical and Data Link Layers for a CAN network, shown in Figure 1.1. The object layer and the transfer layer comprise all services and functions of the data link layer defined by the ISO/OSI model [5].

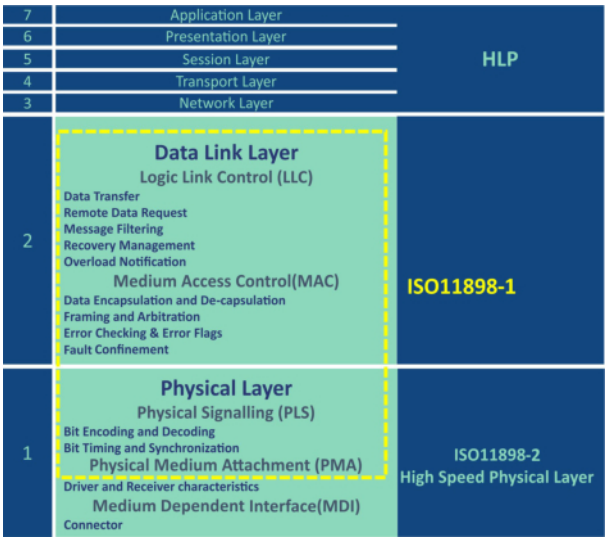


Figure. 1.1 CAN Network Layer

The Logical Link Control (LLC) of Data Link layer defines the Object Layer that is concerned with message filtering, status and message handling. The Transfer Layer represents the kernel of the CAN protocol. The Medium Access Control (MAC) defines the transfer layer that accepts the

message to be transmitted from object layer. The most important features of MAC are Message Framing, Arbitration, Acknowledgement, Error Detection & Signaling, and a management entity called Fault Confinement. The Physical Layer defines how signals are actually transmitted, within this specification the physical layer is not defined so as to allow transmission medium and signal level implementations to be optimized for the application.

1.2.3 CAN Node

The device that needs to establish a communication link are considered to be nodes. These nodes are the Microcontroller (MCU) integrated with the CAN controller, this controller provides two single ended pins (*digital signals*) to communicate (`CAN-TX`, `CAN-RX`). The CAN specification does not allow communication of digital signals, instead uses a *differential signal* (`CAN-High`, `CAN-Low`). The digital signals are short range signals, therefore the differential signals are used that gives more immunity to the noise and data can be transmitted more reliably. The MCU does not produce the differential signal and a CAN transceiver is introduced to convert the digital signals to differential signals. The Figure 1.2 shows the CAN Node with the digital signal (`CAN-TX`, `CAN-RX`) from CAN Controller and differential signal (`CAN-H`, `CAN-L`) from the CAN Transceiver.

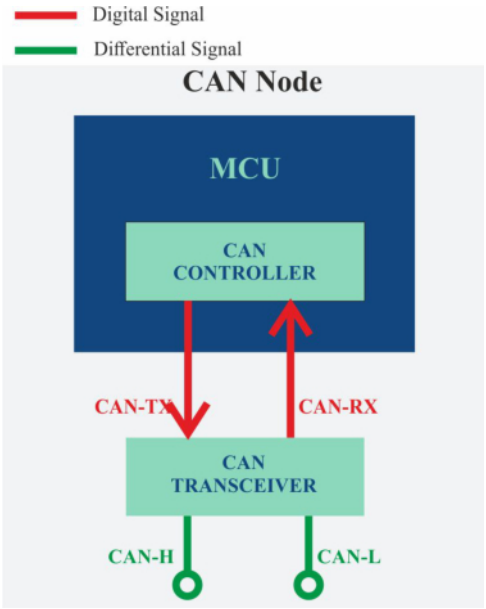


Figure. 1.2 CAN Node

1.2.4 CAN Bus

The physical layer of the CAN bus consists of a two-wire twisted pair connecting all transmitters and receivers, `CAN-Low` (low-speed CAN) and `CAN-High` (high speed CAN). Both lines are terminated by a termination resistance (Rt) of 120Ω to prevent transient phenomena such as reflection [5]. To configure the Node to the CAN bus, connect the `CAN-H` terminal of the node to

CAN-High bus line and CAN-L terminal of the node to CAN-Low bus line. Figure 1.3 shows the pattern of connection between the node to the Bus line. CAN uses broadcast type of bus which allows all the nodes connected in the bus to hear the transmission. There is no possibility to send data specifically to a node with the address. All nodes will pickup the traffic on the bus. The length of the bus wires is of significance and affects the quality of data transmission. According the ISO 11898 standard, it is recommended that the length of the bus not exceed 40 m.

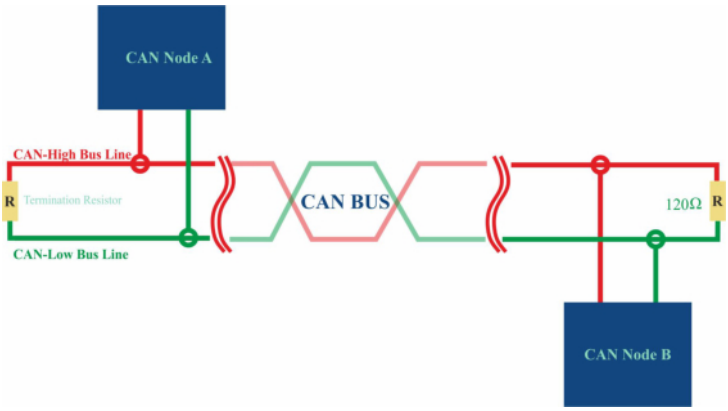


Figure. 1.3 CAN BUS

1.2.5 CAN Signals

Differential signals are used in CAN communication. The CAN transceiver produces differential signals CAN High (CAN-H) and CAN Low (CAN-L) and these signals are complementary signals. CAN-L is actual complimentary signal of CAN-H. To send logic 1, the CAN bus state will be recessive (potential difference between CAN-H and CAN-L is 0 volt) and for logic 0, the state will be dominant (potential difference between CAN-H and CAN-L is around 2 volt) shown inFigure 1.4. The dominant state always overrides the recessive state during the data transfer.

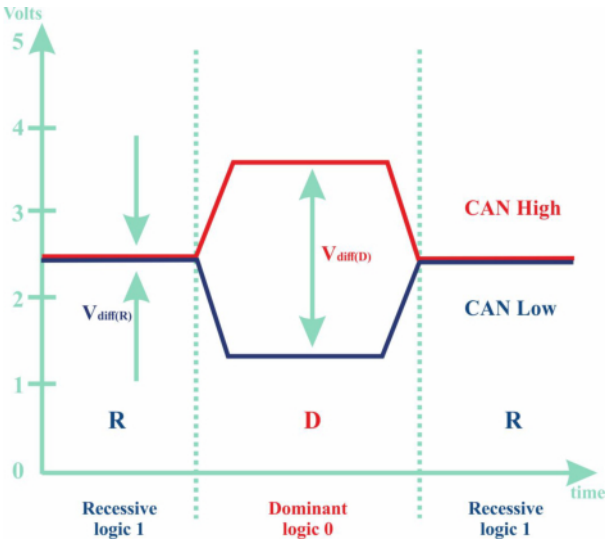


Figure. 1.4 CAN Signal

1.2.6 CAN Types

CAN bus can be categorized into three types based on the identifiers.

- CAN 2.0A - 11-bit message Identifier (Standard CAN).
- CAN 2.0B - 11-bit (Standard CAN) and 29-bit message Identifier (Extended CAN).
- CAN FD - Flexible Data Rate.

The standard 11-bit identifier field provides for 2^{11} (2048), which is 0 to 2047 unique message identifiers. Where as the extended 29-bit identifier provides for 2^{29}, 536870912 unique identifiers. While `CAN 2.0x` has the capacity to hold 8 bytes of data within the CAN-frame, CAN FD can hold up to 64 bytes. CAN FD is not centred in this project as ESP-32 contains only `CAN 2.0x` specifications.

The message frame format of CAN 2.0A is shown in Figure 1.5 for CAN 2.0B Standard Frame Format in Figure 1.6, and Extended Frame Format in Figure 1.7. The difference between a CAN 2.0A and a CAN 2.0B message is that CAN 2.0B supports both 11 bit (standard) and 29 bit (extended) identifiers. Standard and Extended frames may exist on the same bus with numerically equivalent identifiers. In that case, the standard frame prevails.

The message frame is marked with `dominant (logic 0)` and `recessive (logic 1)`. The default condition of each field differentiating the Standard and Extended message frame are given in the Figure 1.8.

The description of the bit fields of Message frame are:

- **Start of Frame (`SOF`) :** This bit indicates the start of a message, and is used to synchronize the nodes on a bus after being idle. It is preceded by at least 11 recessive bits.
- **Identifier (`ID`) :** Logical address and priority of the message. With lower binary value, higher its priority (`0` - highest priority). The Standard CAN Frame has 11-bit identifier and Extended Frame has 29-bit identifier (11-bit base ID and 18-bit extension ID)
- **Remote Transmission Request (`RTR`) :** The RTR Bit is used by the receiver to request a remote transmitter to send its information. If the bit is set to recessive, the frame contains no data field. In that case all the connected nodes can check whether there is a corresponding transmitter defined or not. The request and the possible answer are two completely different frames on the bus. The answer can be delayed due to messages with higher priorities.
- **Substitute Remote Request (`SRR`) :** The bit replaces the RTR bit in the standard message location as a placeholder in the extended format.
- **Identifier Extension (`IDE`) :** This bit used to mark the message as standard or extended. A dominant bit means that a standard CAN identifier with no extension is being transmitted.

- **Reserved bit** (`r0,r1`) **:** The reserved bit is for future use. The development of CAN FD utilizes the reserved bit for identification of FD format in the base and extended format. The reserved bit are marked as dominant.
- **Data Length Code** (`DLC`) **:** This bit contains the number of bytes of data being transmitted.
- **Data :** 0 to 64 bits (0 to 8 bytes); contains up to 64 bits of application data to be transmitted.
- **Cyclic Redundancy Check** (`CRC`) **:** Contains the checksum (number of bits transmitted) of the preceding application data for error detection. It can be used only for error detection and not for error correction. The hamming distance of this CRC code is 6. With this it is possible to detect up to 6 single bit errors which are scattered about the message or so called burst errors up to a length of 15 bits.
- **Acknowledge Field** (`ACK`) **:** Every node receiving an accurate message overwrites this recessive bit in the original message with a dominant bit, indicating an error-free message has been sent. If a receiving node detects an error and leaves this bit recessive, it discards the message and re-transmits after re-arbitration. In this way, each node acknowledges the integrity of its data.
- **End of Frame** (`EOF`) **:** The EOF is marked by coding violation. The bit-stuffing rule of the coding technique is violated. When 5 bits of the same logic level occur in succession during normal operation, a bit of the opposite logic level is stuffed into the data. This bit-stuffing is indicated by a dominant bit. This field indicates the end of a CAN frame and disables bit-stuffing.

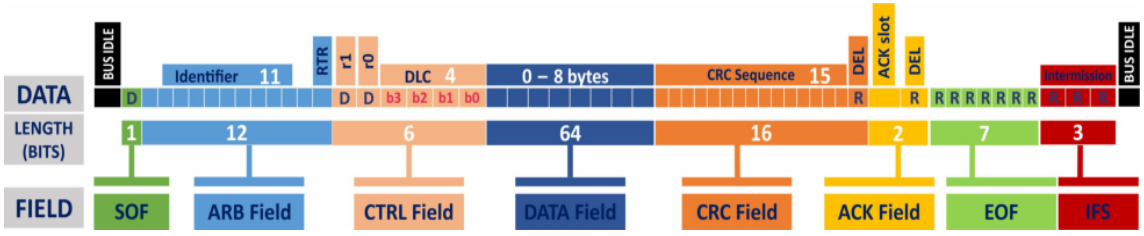


Figure. 1.5 CAN 2.0A Standard CAN

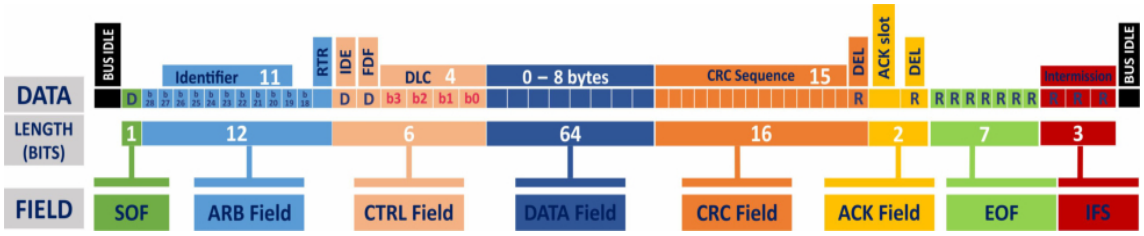


Figure. 1.6 CAN 2.0B Standard CAN

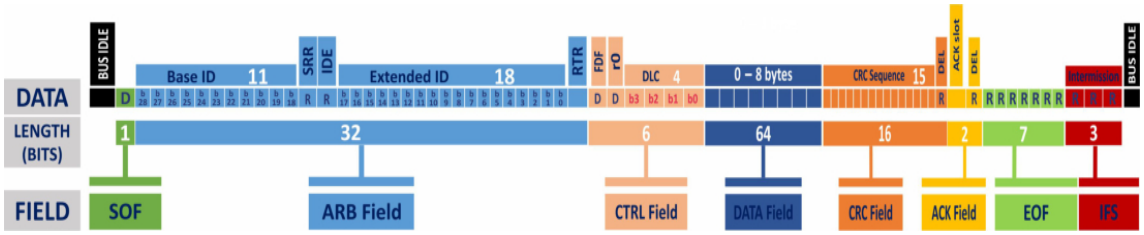


Figure. 1.7 CAN 2.0B Extended CAN

FIELD		CAN 2.0 A	CAN 2.0 B	
		STANDARD FRAME	STANDARD FRAME	EXTENDED FRAME
Start of Frame (SOF)		Must be dominant (D)	Must be dominant (D)	Must be dominant (D)
Arbitration Field (ARB)	Identifier (ID)	11-bit Unique identifier indicates priority	Unique identifier corresponds to 11-bit Base ID in Extended Format	Comprised of 11 bit Base ID and 18 bit Extended ID
	Remote Transmission Request (RTR)	Dominant (D) in Data Frames Recessive (R) in Remote Frames	Dominant (D) in Data Frames Recessive (R) in Remote Frames Followed after the Identifier	Dominant (D) in Data Frames Recessive (R) in Remote Frames
	Substitute Remote Request (SRR)	~	~	Must be recessive. In arbitration between standard and extended frames, recessive SRR guarantees the standard message frame prevails
	Identifier Extention (IDE)	~	~	Recessive (R) for Extended Format
Control Field (CTRL)	Identifier Extention (IDE)	~	Dominant (D) for Standard Format	~
	Reserved (r0,r1)	Must be dominant (D)	Must be dominant (D)	Must be dominant (D)
	Data Length Code (DLC)	Number of data bytes (0–8)	Number of data bytes (0–8)	Number of data bytes (0–8)
Data Field		0–8 bytes Length determined by DLC field	0–8 bytes Length determined by DLC field	0–8 bytes Length determined by DLC field
Cyclic Redundancy Check Field (CRC)	CRC Sequence	15 bit Error detecting field. Receiver compares the message with the Transmitter	15 bit Error detecting field. Receiver compares the message with the Transmitter	15 bit Error detecting field. Receiver compares the message with the Transmitter
	CRC Delimiter	Must be recessive (R)	Must be recessive (R)	Must be recessive (R)
Acknowledge Field (ACK)	ACK Slot	Transmitter sends recessive (R) Receiver asserts dominant (D)	Transmitter sends recessive (R) Receiver asserts dominant (D)	Transmitter sends recessive (R) Receiver asserts dominant (D)
	ACK Delimiter	Must be recessive (R)	Must be recessive (R)	Must be recessive (R)
End of Frame (EOF)		Must be recessive (R)	Must be recessive (R)	Must be recessive (R)

Figure. 1.8 CAN Message Frame

1.2.7 CAN Message

There are four different types of frames that are used for communication over CAN bus.

- Data Frame (Standard and Extended) – send data
- Remote Frame (Standard and Extended) – request data
- Error Frame (Passive and Active) – report error
- Overload Frame – request a delay between two data or remote frames.

Data and Remote Frame

The Data Frame are of two formats Standard and Extended. The difference is the Arbitration field; Standard Frame 11-bit and Extended Frame 29-bit.

The frame format for both the frame type is same with the `RTR` bit as `Dominant bit 0`. Both the frame can co-exists on the same CAN bus. The standard frame will win the arbitration over the extended frame.

Remote frame is similar to data frame with no data payload. It is denoted by the `RTR` bit as `Recessive bit 1`. It is used when one node needs to request data from another node.

Error Frame

When a CAN node detects an error with the data or remote frame, it will immediately abort the transmission and broadcast the error frame. Error Frame is a special message that violates the formatting rules of a CAN message. The transmitting node will know that the message sent was not received properly by all nodes, and will automatically attempt a re-transmission at the next available idle time on the bus [6].

The Error Frame consists of 2 fields, an `Error Flag` field followed by an `Error Delimiter` field. The error flag made up of 6 dominant bits and an error flag delimiter made up of 8 recessive bits, which allows the bus nodes to restart bus communications after an error. Depending on the error count of the node, the error flag are of two mode active and passive.

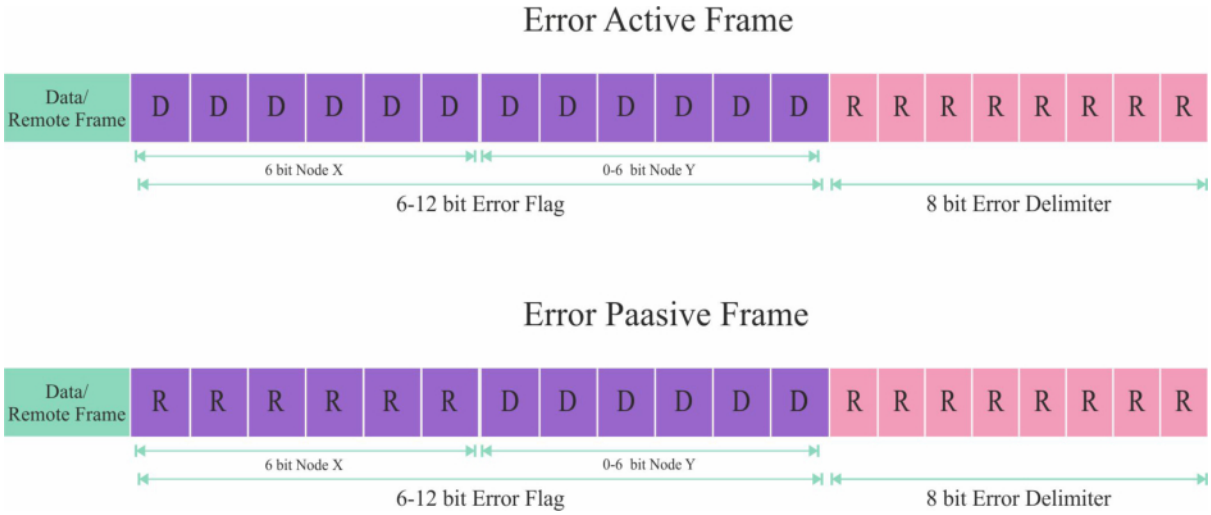


Figure. 1.9 Error Frame

- **Error Active :** Either in transmit or receive buffer of a node the count is $(0 \leq count \leq 128)$. It states that, at least one error has been detected, even when the node is fully functional. The Error Flag field consists of between 6 and 12 consecutive dominant bits (generated by one or more nodes). The Error Delimiter field completes the Error Frame. After completion of the Error Frame bus activity returns to normal and the interrupted node attempts to resend the aborted message.
- **Error Passive :** The error count $(128 \leq 255)$ is considered as error passive. The passive Error Flag consists of 6 consecutive recessive bits, and therefore the Error Frame consists of 14 recessive bits.

Receive errors increments the error count by 1; transmit errors increments the count by 8. Subsequently, error-free messages decrements the error count by 1. If the error count returns to zero, a node will return to normal operating mode.

1.3 Bit Timing Requirements

The performance of CAN Network depends on the CAN bit timing. One advantage of CAN protocol is that, the bit rate, sampling point of the bit, number of samples taken in a bit period can be programmed by the user to optimize the performance of CAN network for a particular application. Bit timing parameters, the reference oscillator tolerance, signal propagation delay are the key features to be considered during the optimization process. Apparently, the larger allowable oscillator tolerance and a long bus length are conflicting goals, that can be met through optimization of bit timing parameters.

The Bit construction of ESP32 CAN; Synchronisation Segment, Propagation Segment, Phase Segment 1, Phase Segment 2 are the non-overlapping CAN Bit Timing Segments, Figure 1.10.

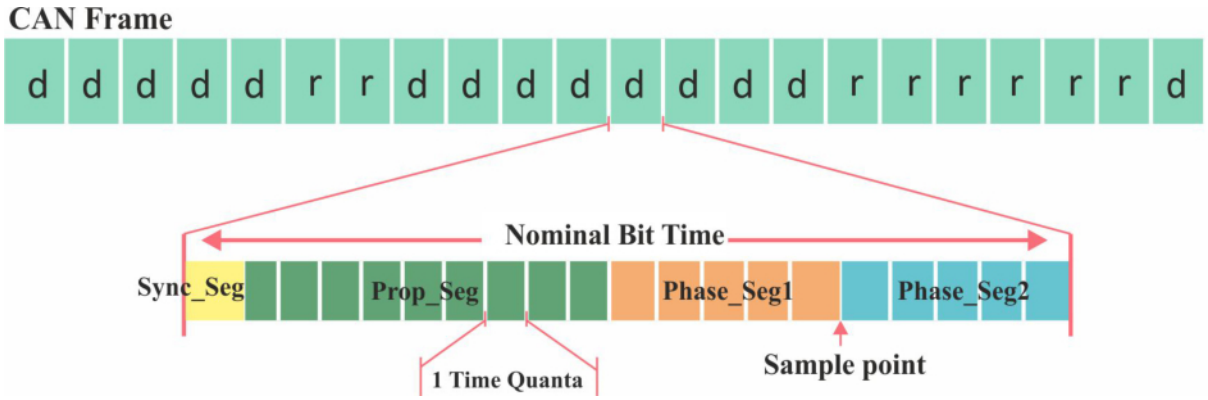


Figure. 1.10 CAN Bit Timing Segments by ISO-11898

1.3.1 Calculation of Bit Timing Parameters

This section details the general procedure for calculation of Bit Timing parameters [7], [8]. The bit timing calculation and register description are different for each CAN Network technologies, we focus on NXP technologies as the ESP32 CAN resembles the features of NXP SJA1000 [9]. Basically the CAN bit period can be subdivided into four time segments Figure 1.10. Each time segment consists of a number of Time Quanta (`TQ`).

The Calculation requires at least three input parameters.

1. Desired Bit Rate, is uniform across the entire bus.
2. CAN Module Clock Frequency, is used to derive the bit-stream sampling clock.

3. Estimation of Propagation Delay between the most distant nodes on the bus.

And the Bit timing calculator must provide at least five outputs.

1. Baud Rate Prescaler(BRP), determines the sampling clock period.
2. Propagation Delay, expressed as a number of time quanta.
3. Phase Segment 1, in time quanta.
4. Phase Segment 2, in time quanta.
5. Synchronization Jump Width(SJW), in time quanta.

Calculation of Bit Timing Parameters [10]

The following example provides the steps for determining the optimum bit timing calculation with the details of the input parameters. The PIC microcontroller is considered for this example.

Bit Rate	1 Mbit/s
Bus Length	20m
Bus propagation delay	5 ns/m
Transceiver loop delay (MCP2562)	125ns
oscillator frequency	8MHz

Step 1: Determine minimum permissible time for the PROP_SEG segment.

$t_{PROP\_SEG} = 2(t_{Bus} + t_{Tx} + t_{Rx})$

Physical bus delay t_{Bus} = Bus length $\times$ Bus propagation delay.

$t_{Bus} = 20 \times 5 \times 10^{-9} = 100ns$

$t_{Tx} + t_{Rx}$ = Transceiver loop delay = $125ns$

$t_{PROP\_SEG} = 2(100ns + 125ns) = 450ns$.

Step 2: Choose CAN System Clock Frequency.

The period of the CAN system clock is equal to the duration of Time Quanta (t_q), which is derived from the MCU system clock (f_{clk}) or oscillator by way of a programmable prescaler, called the Baud Rate Prescaler (BRP).

f_{clk} = 8 MHz BRP = 1 (note. is chosen).

BRP of value 1 gives CAN system clock = 8 MHz.

$t_q = BRP/f_{clk} = 125ns$

Nominal Bit Time (NBT) = 1000/125 = 8TQ

Step 3: Calculate PROP_SEG duration.

$$\texttt{PROP\_SEG} = \texttt{ROUND\_UP}\left(\frac{t_{\texttt{PROP\_SEG}}}{t_q}\right)$$

PROP_SEG = ROUND_UP(500ns/125ns) = 4TQ

Step 4: Determine PHASE_SEG1, PHASE_SEG2.

From NBT subtract the PROP_SEG and SYNC_SEG = 1TQ (default).

$value = 8 - (4 + 1) = 3$

- value < 3 : select higher CAN system clock frequency from Step 2.
- value > 3 and odd number : add 1 to PROP_SEG and recalculate.
- value > and even number : divide by 2 and assign the values to PHASE_SEG1 and PHASE_SEG2.
- value = 3 : PHASE_SEG1 = 1 and PHASE_SEG2 = 2 and only one sample per bit may be chosen.

$value = 3$; So the PHASE_SEG1 = 1 PHASE_SEG2 = 2

Step 5: Determine Synchronization Jump Width (SJW)

SJW is chosen as the smaller of 4 and PHASE_SEG1

$(4 \geq SJW \geq 1)$

this case; SJW = 1

BRP	1
NBT	8
SYN_SEG	1
PROP_SEG	1
PHASE_SEG1	3
PHASE_SEG2	2
SJW	1

1.4 Synchronization

All nodes on the CAN bus must have the same nominal bit rate. Noise, phase shifts, and oscillator drift create situations where the nominal bit rate does not equal the actual bit rate in a real system. The receivers must synchronize to the transmitted data stream to ensure messages are properly decoded. This is achieved by the receivers synchronization on recessive to dominant edges.

There are two methods; Hard Synchronization and Re-synchronization, used for achieving and maintaining synchronization. A Hard Synchronization is done once at the start of a frame;

inside a frame only Re-synchronizations occur at recessive to dominant (1-to-0) edges and adjust the bit clock as necessary. After a hard synchronization, the bit is restarted with the edge of SYNC_SEG, regardless of the edge phase error. Thus it forces the edge which has caused the hard synchronization to lie within the synchronization segment of the restarted bit time. Re-synchronization leads to a shortening or lengthening of the bit time such that the position of the sample point is shifted with regard to the edge.

Synchronization Rules [11]:

1. Only recessive-to-dominant edges will be used for synchronization.
2. Only one synchronization within one bit time is allowed.
3. An edge will be used for synchronization only if the value at the previous sample point differs from the bus value immediately after the edge.
4. A transmitting node will not re-synchronize on a positive and negative phase error. This implies that a transmitter will not re-synchronize due to propagation delays of it's own transmitted message. The receivers will synchronize normally.
5. If the absolute magnitude of the phase error is greater than the SJW, then the appropriate phase segment will be adjusted by an amount equal to the SJW.

1.5 CAN Operation

The transmission and reception of message are are handled by the protocol engine in the CAN module. The CAN module must be in the configuration mode before starting the communication in the CAN network. The below section shows the protocol engine of the ESP32 CAN Node for transmitting and receiving.

1.5.1 CAN Node Transmitting

The CAN module should be initialized with the bit rate settings and all the nodes in the bus must be set with the same bit rate.

After all the configuration and initialization of the CAN module; is set to normal mode. The driver writes the message ID, data length code, data bit into the TXB registers and set the Transmission request bit. The TXB is locked.

At this point the CAN module protocol engine performs the following action on the message. The message is assembled and stuff bits are added where ever necessary. It sense the CAN bus for the idle time. At this time the module will start transmitting the message while checking for error frames. The CAN Transceiver will translate the digital signal from the microcontroller to bus signals on to the bus.

If the CAN message is successfully sent onto the CAN bus, the TXB will be unlocked for the next message.

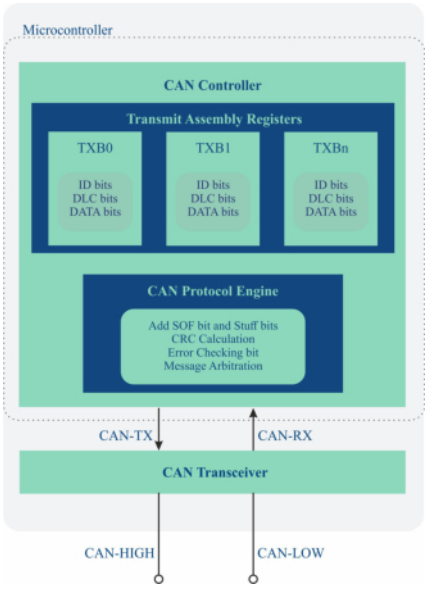


Figure. 1.11 CAN Node Transmission

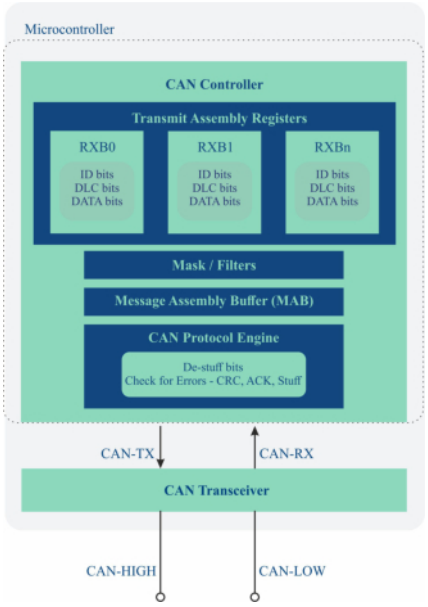


Figure. 1.12 CAN Node Receiving

1.5.2 CAN Node Receiving

When a CAN node transmit a message on the CAN bus, another node on the CAN bus sense the message starting with SOF.

The CAN transceiver will translate the bus signal into digital signal and pass it to the CAN controller. The protocol engine will perform the de-stuffing operation on the stuffed bits and check for the message errors while loading the message into the `Message Assemble Buffer`. If there are no errors; the entire message is loaded in the MAB and the CAN module will check the received message against the `Mask and Filter settings`. If the message ID matches the Acceptance filter settings, then the message will be transferred to the `RX buffer` and the driver will be notified. If the message fails the acceptance filter match, the controller will not be notified. Therefore the Acceptance filter must be set carefully. The message in the RX buffer are removed and processed for the corresponding application.

Chapter 2

ESP32 Overview

2.1 About ESP32

The ESP32 is a low-cost, low-power system-on-chip (SoC) microcontroller series created and developed by Espressif Systems, a Shanghai-based Chinese company, and is manufactured by TSMC ultra-low-power 40 nm technology [12]. Since the end of 2016, the ESP32 has become an improvement to the ESP8266. ESP32 has both Wi-Fi and Bluetooth capabilities, which makes it an all-rounded chip for the development of IoT projects and embedded systems in general. The ESP32 series employs a Tensilica Xtensa LX6 microprocessor in both dual-core and single-core 32-bit microprocessor. ESP32 counts about 19 peripherals beside the WiFi and Bluetooth. The objective of this project is to develop a library for CAN Peripheral. The difficulty is that there are no specification of CAN registers in the ESP32 technical documentation (which is still underdevelopment) [13].

2.2 ESP32 Specification

ESP32 is a highly-integrated solution for Wi-Fi and Bluetooth IoT applications, with around 20 external components. ESP32 achieves ultra-low power consumption, it features all the state-of-the-art characteristics of low-power chips, including fine-grained clock gating, multiple power modes, and dynamic power scaling. ESP32 comprise of in-built antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters and power management modules [1]. ESP32 can perform as a complete standalone system or as a slave device to a host MCU, reducing communication stack overhead on the main application processor. ESP32 can interface with other systems to provide Wi-Fi and Bluetooth functionality through its SPI / SDIO or I2C / UART / CAN interfaces. ESP32 is capable of functioning reliably in industrial environments, with an operating temperature ranging from –40°C to +125°C.

ESP32 is integrated on different modules, in this project the MH-ET LIVE Minikit is operated by ESP32-WROOM-32 module. The module is based on ESP32-D0WDQ6 chip (D - dual core processor; 0 - no embedded flash; WD - WiFi b/g/n + BT/BLE dual mode; Q6 - QNF 6×6), Figure 2.1.

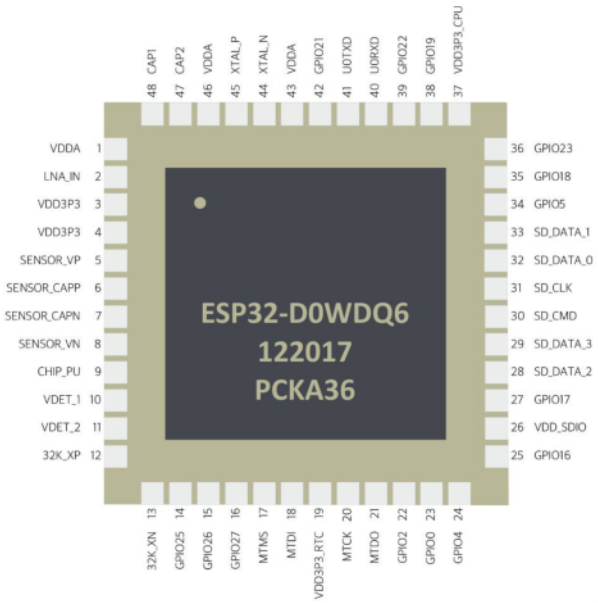


Figure. 2.1 ESP32-D0WDQ6 Pin Layout [1]

2.2.1 Key Features

Rather than using the ESP32 SoC directly, ESP32 boards use an ESP32 module (ESP32-WROOM-32) from Espressif which integrates additionally to the SoC some key components, like SPI flash memory, PSRAM, or crystal oscillator, some of these components are optional. The important features of ESP32 are listed in the Table 2.1.

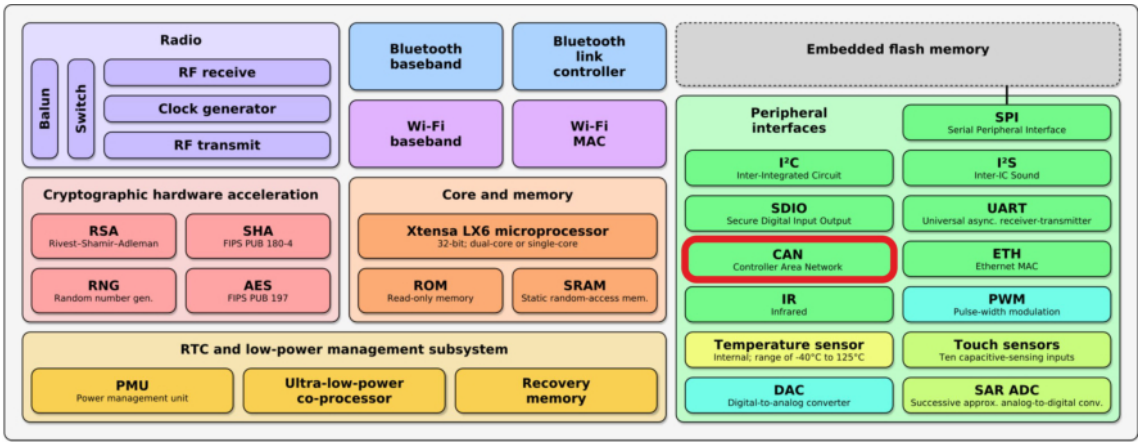


Figure. 2.2 ESP32-Function Block Diagram [1]

Category	**Specification**
MCU	Dual Core
	• CPU: Xtensa dual-core (or single-core) 32-bit LX6 microprocessor.
	• Ultra low power (ULP) co-processor.
Memory	• RAM - 520 KiB SRAM, 16 kByte RTC SRAM
	• ROM – 520 kByte
	• FLASH – 512 kByte/16 Mbyte
Frequency	240 MHz, 160 MHz, 80 MHz
Wi-Fi	802.11 b/g/n - 2.4 GHz up to 150 Mbit/s
Bluetooth	v4.2 BR/EDR and BLE(Bluetooth Low Energy)
Peripheral	• 34 – GPIO (6 only inputs)
	• 2 x SAR-ADC with up to 18 x 12 bit channels
	• 2 x DAC with 8 bit
	• 4 x SPI (Serial Peripheral Interface)
	• 2 x I2C Interface
	• 2 x I2S Interface
	• 3 x UART
	• Ethernet MAC Interface
	• SD/SDIO/MMC host controller
	• SDIO/SPI Slave Controller
	• CAN bus 2.0

Table 2.1 ESP32-Specification

2.3 MH-ET LIVE ESP32 MiniKit

MH-ET LIVE MiniKit is based on ESP32-WROOM-32. The chip embedded is designed to be scalable and adaptive. The two CPU cores can be controlled individually, and the CPU clock frequency is adjustable from 80 MHz to 240 MHz. By default the module has 4MB flash, 40 MHz Integrated crystal oscillator, on-chip hall sensor. The operating current is of average 80mA [14].

The Figure 2.3 shows the pinout of MH-ET LIVE MiniKit for ESP32 board as defined by the default board configuration compatible to the miniKit development board. The default configuration cannot be used or it may not available, when the optional hardware are used (ADC and DAC GPIOs are mandatory, SPI/UART/I2C/CAN.. are optional). GPIO (`1, 3, 6, 7, 8, 11`) cannot be used, GPIO (`12, 13, 14, 15, 16, 17, 26, 27, 32, 33`) can be used always and other GPIOs can be used only when they are free [15].

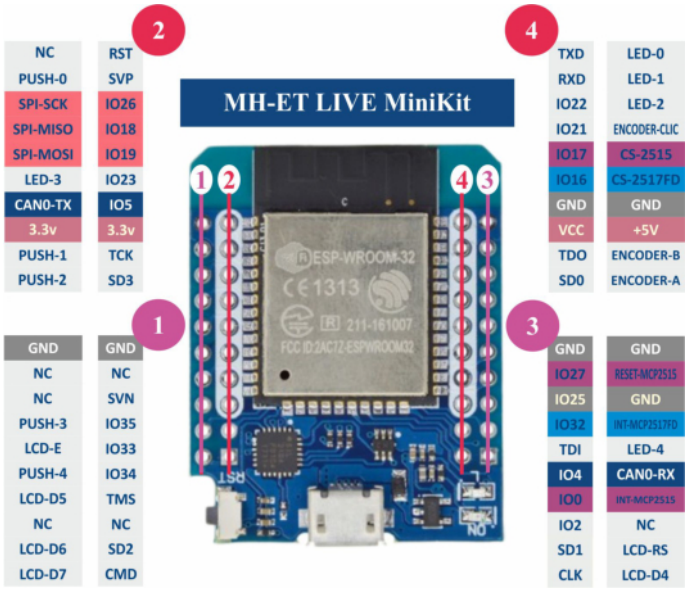


Figure. 2.3 MH-ET LIVE ESP32-MiniKit

2.4 Programming Environment

The ESP32 can be programmed in different programming environments.

1. **Arduino IDE**.
2. Espressif IDF (IoT Development Framework).
3. Micropython.
4. JavaScript.

In this project Arduino IDE is used to program the controller.

2.4.1 Installing the ESP32 Board in Arduino IDE

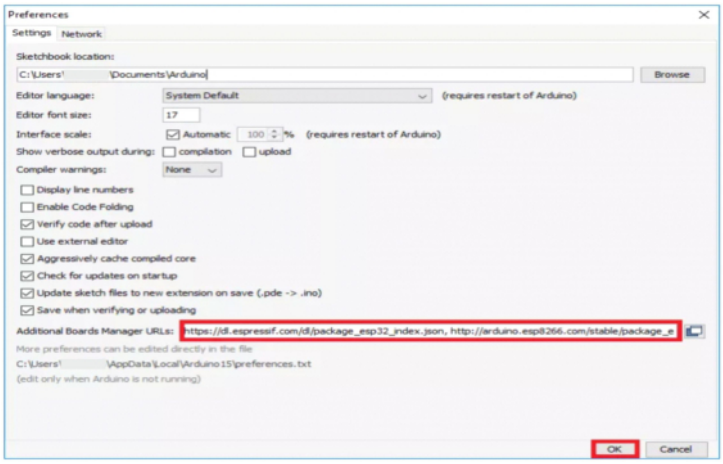
The arduino-esp32 git hub repository can be cloned by following the procedure in, https://github.com/espressif/arduino-esp32/blob/master/docs/arduino-ide/windows.md. The following steps shows the direct clone of ESP32 hardware source on Arduino IDE [16].

Step 1: Install latest version of Arduino IDE (open source IDE) from www.arduino.cc/en/Main/Software.

Step 2: Open the **Preference** window from the Arduino IDE. Go to **File** $->$ **Preferences** or (**Ctrl + comma**).

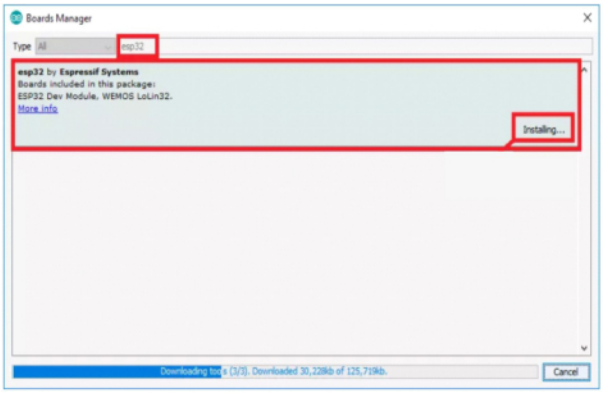
Step 3: Enter **https://dl.espressif.com/dl/package_esp32_index.json** into the **Additional Board Manager URLs** field and click **OK**.

(*Additional URLs can be separated with a comma*)



Step 4: Open boards manager. Go to **Tools** $->$ **Board** $->$ **Boards Manager...**.

Step 5: Search for **ESP32** and click **install** button for the "**ESP32 by Espressif Systems**".



After the installation of ESP32 hardware on the Arduino IDE, the hardware files will be available in the path **C:\Users\xxx \Documents\Arduino\hardware\espressif\esp32**.

2.5 ESP32 CAN Setup

ESP32 has built-in CAN bus peripherals Figure 2.2. The ESP32 CAN controller supports Standard Frame Format (11-bit ID) and Extended Frame Format (29-bit ID) of the CAN2.0B specification, and are not compatible with CAN FD frames (will interpret such frames as errors).

As discussed in Section 1.2.3 on Page 5, a CAN Node is operated by the controller and transceiver. ESP32 CAN controller provides only the data link layer and physical layer signaling sub-layer. ESP32 does not contain an internal transceiver. Therefore, depending on the physical layer requirements, an external transceiver module is required which converts the CAN-RX and CAN-TX signals of the ESP32 into CAN-High and CAN-Low bus signals. There are many transceiver available and are selected depending on the ISO 11898-2 Physical layer compatibility. We make use of the MCP2562 High speed CAN Transceiver [17] for the operation of CAN network.

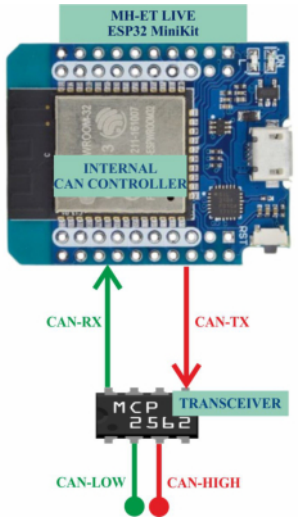


Figure. 2.4 ESP32 CAN Node

The ESP32 internal CAN Controller consists of 4 signal lines.

1. **TX and RX** signal lines are interfaced with the MCP2562 CAN transceiver **TXD** and **RXD** pins. These signal line interprets the recessive bit (3.3v) and dominant bit (0v). The connection should be made careful, as the change in logic level of TX line can be observed on the RX line. Failing the connection causes loss in arbitration or bit error.
2. **BUS-OFF** signal line is optional and is set to a low logic level (0V) whenever the CAN controller reaches a bus-off state. The BUS-OFF signal line is set to a high logic level (3.3V) otherwise.
3. **CLKOUT** signal line is optional and outputs a prescaled version of the CAN controller source clock **APB Clock**.

2.6 ESP32 MiniKit Development Board

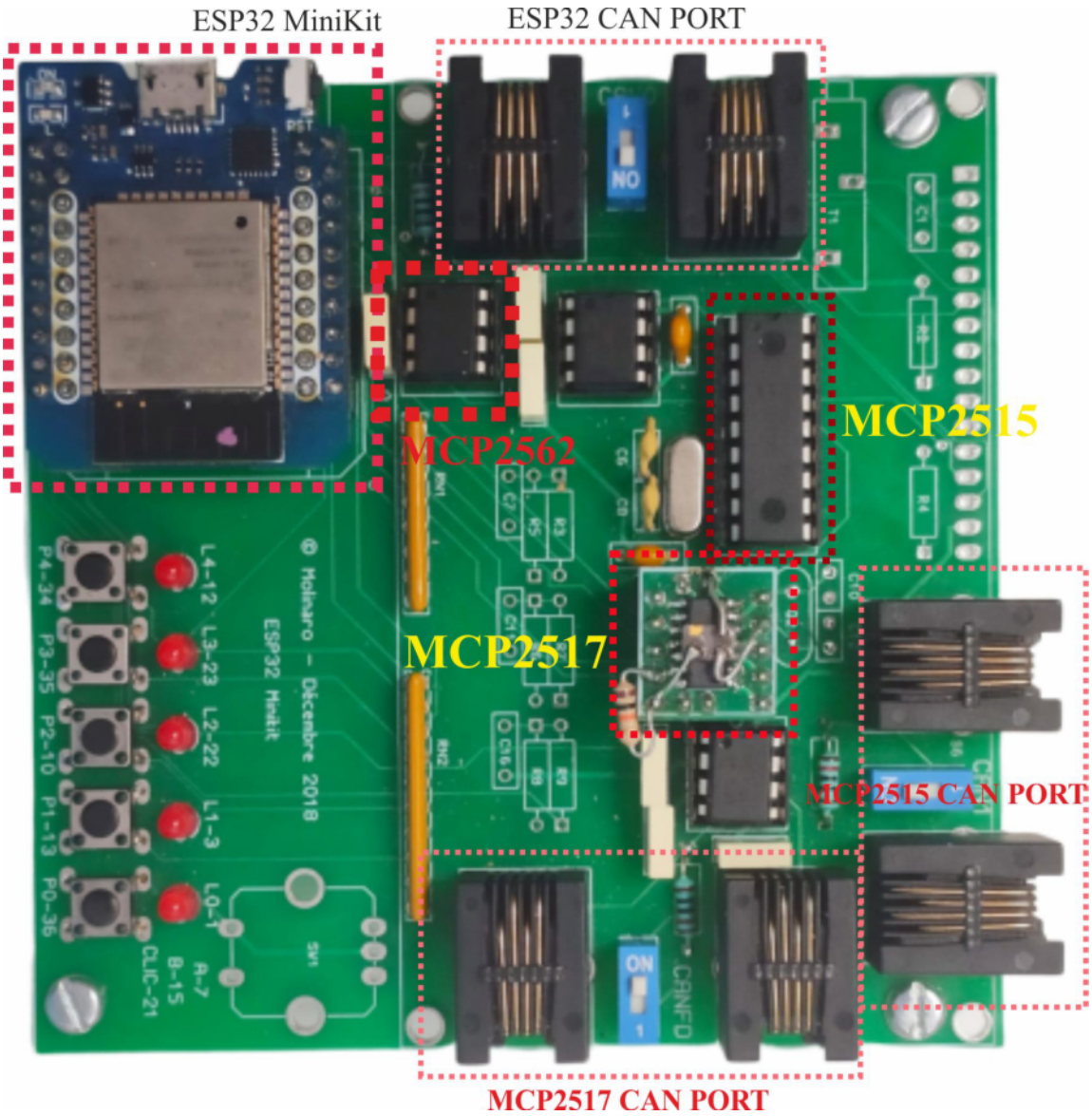


Figure. 2.5 ESP32 MiniKit Developement Board

The development board contains a ESP32 MiniKit MCU. Other components integrated on the board are;

- **CAN Controllers**:
  - MCP2515 [18]
  - MCP2517 [19]
- **CAN Transceiver** [17]; the transceiver is common for all the CAN controllers in the board.

Chapter 3

ESP32 CAN Peripheral

3.1 ESP32 CAN Peripheral Registers

The ESP32 Technical Reference Manual [13] does not contain any detail about the CAN peripheral registers. It is found in the esperssif forum [20] that ESP32 integrates a CAN controller which is compatible with the NXP SJA1000 CAN controller [21].

The CAN Registers are found with the comparison of available ESP32 SDK (developed by espressif) [22], SJA1000 CAN Controller registers, and only one existing unofficial CAN library developed by Thomas Barth [3].

CAN ADDRESS	REGISTERS	HEX ADDRESS	OPERATING MODE				RESET MODE		SJA1000
			READ		WRITE		READ	WRITE	
0	CONFIGURATION AND CONTROL REGISTERS	0x3FF6B000	MODE		MODE		MODE	MODE	
1		0x3FF6B004	#		COMMAND		#	COMMAND	
2		0x3FF6B008	STATUS		#		STATUS	#	
3		0x3FF6B00C	INTERRUPT		#		INTERRUPT	#	
4		0x3FF6B010	INTERRUPT ENABLE		INTERRUPT ENABLE		INTERRUPT ENABLE	INTERRUPT ENABLE	
5		0x3FF6B014	RESERVED		#		RESERVED	#	
6		0x3FF6B018	BUS TIMING 0		#		BUS TIMING 0	BUS TIMING 0	
7		0x3FF6B01C	BUS TIMING 1		#		BUS TIMING 1	BUS TIMING 1	
8		0x3FF6B020	RESERVED		RESERVED		RESERVED	RESERVED	OUTPUT CONTROL
9		0x3FF6B024	RESERVED		RESERVED		RESERVED	RESERVED	TEST
10		0x3FF6B028	RESERVED		#		RESERVED	#	
11	CAPTURE AND COUNTER REGISTERS	0x3FF6B02C	ARBITRATION LOST CAPTURE		#		ARBITRATION LOST CAPTURE	#	
12		0x3FF6B030	ERROR CODE CAPTURE		#		ERROR CODE CAPTURE	#	
13		0x3FF6B034	ERROR WARNING LIMIT		#		ERROR WARNING LIMIT	ERROR WARNING LIMIT	
14		0x3FF6B038	RX ERROR COUNTER		#		RX ERROR COUNTER	RX ERROR COUNTER	
15		0x3FF6B03C	TX ERROR COUNTER		#		TX ERROR COUNTER	TX ERROR COUNTER	
16	SHARED REGISTERS	0x3FF6B040	RX FRAME INFORMATION : SFF	RX FRAME INFORMATION : EFF	TX FRAME INFORMATION : SFF	TX FRAME INFROMATION : EFF	ACCEPTANCE CODE 0	ACCEPTANCE CODE 0	
17		0x3FF6B044	RX IDENTIFIER 0	RX IDENTIFIER 0	TX IDENTIFIER 0	TX IDENTIFIER 0	ACCEPTANCE CODE 1	ACCEPTANCE CODE 1	
18		0x3FF6B048	RX IDENTIFIER 1	RX IDENTIFIER 1	TX IDENTIFIER 1	TX IDENTIFIER 1	ACCEPTANCE CODE 2	ACCEPTANCE CODE 2	
19		0x3FF6B04C	RX DATA 0	RX IDENTIFIER 2	TX DATA 0	TX IDENTIFIER 2	ACCEPTANCE CODE 3	ACCEPTANCE CODE 3	
20		0x3FF6B050	RX DATA 1	RX IDENTIFIER 3	TX DATA 1	TX IDENTIFIER 3	ACCEPTANCE MASK 0	ACCEPTANCE MASK 0	
21		0x3FF6B054	RX DATA 2	RX DATA 0	TX DATA 2	TX DATA 0	ACCEPTANCE MASK 1	ACCEPTANCE MASK 1	
22		0x3FF6B058	RX DATA 3	RX DATA 1	TX DATA 3	TX DATA 1	ACCEPTANCE MASK 2	ACCEPTANCE MASK 2	
23		0x3FF6B05C	RX DATA 4	RX DATA 2	TX DATA 4	TX DATA 2	ACCEPTANCE MASK 3	ACCEPTANCE MASK 3	
24		0x3FF6B060	RX DATA 5	RX DATA 3	TX DATA 5	TX DATA 3	RESERVED	#	
25		0x3FF6B064	RX DATA 6	RX DATA 4	TX DATA 6	TX DATA 4	RESERVED	#	
26		0x3FF6B068	RX DATA 7	RX DATA 5	TX DATA 7	TX DATA 5	RESERVED	#	
27		0x3FF6B06C	FIFO RAM	RX DATA 6	#	TX DATA 6	RESERVED	#	
28		0x3FF6B070	FIFO RAM	RX DATA 7	#	TX DATA 7	RESERVED	#	
29	MISC REGISTERS	0x3FF6B074	RX MESSAGE COUNTER		#		RX MESSAGE COUNTER	#	
30		0x3FF6B078	RESERVED		#		RESERVED	RESERVED	RX BUFFER START ADDRESS
31		0x3FF6B07C	CLOCK DIVIDER		CLOCK DIVIDER		CLOCK DIVIDER	CLOCK DIVIDER	

Figure. 3.1 ESP32 CAN Controller Register Map

The base register for the CAN peripheral is 0x3FF6B000 [20]. The ESP32 can only access peripheral registers every 32bits, but the registers are 8bits, so each CAN register is mapped to the least significant byte of every 32bits. Figure 3.1 details the ESP32 CAN Controller registers.

ESP32 CAN peripheral registers are similar to SJA1000, but some registers and register bits of SJA1000 are not supported on ESP32. The output control register (OCR) that allows to set-up different output driver configuration; Test Register; Receive buffer start address registers are not supported by ESP32 CAN controller. The Sleep mode bit of the mode register is not supported by ESP32. The Wake-Up interrupt bit of the Interrupt register is not supported as the sleep mode is not configurable in the ESP32 CAN controller.

3.1.1 Mode Register (`CAN_MODE`)

The behaviour of the CAN Controller are changed by the bits of the Mode Register. The reserved bits are read as `logic 0`.

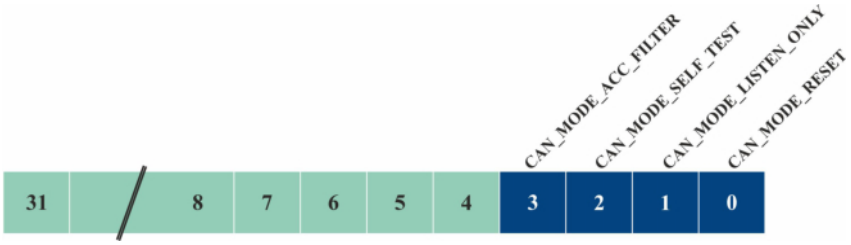


Figure. 3.2 Mode Register

CAN supports four modes of operation.

1. `Reset Mode`.
2. `Normal Mode`.
3. `LoopBack Mode`.
4. `Listen Only Mode`.

The Three modes can be configured by setting the bit field of Mode Register Figure 3.2.

1. `CAN_MODE_RESET` : bit [0]; The bit is set to `logic 1` to enter reset mode. It is obligatory to enter reset mode to attain write access to the configuration registers. By default CAN module enters reset mode during bus-off or hardware reset to prevent the bus activity. When the `CAN_MODE_RESET` bit is set to `logic 0`, the controller will enter the operating mode and wait for the bus states:
    - One occurrence of the bus-free signal i.e., 11 recessive bits in CAN frame.
    - 128 Occurrences of bus free. If the previous reset has been made by controller initiated bus-off, before re-entering the bus-on mode.

2. CAN_MODE_LISTENONLY : bit [1]; In this mode the CAN controller does not acknowledge the can bus even if the message are received successfully. By setting this bit to logic 1 the controller enters Listen Only mode. The special ability of this bit is that it freezes the error counters. This mode of operation also force the CAN controller to be Error Passive (See Section 1.2.7). During the bus-off state the Receive Error Counter is likely to increase, therefore by setting the controller to Listen Only mode prevent the controller from bus-activities; transmission of messages/acknowledgement/error frames are disabled. Bus-monitoring is effective on this mode of operation.

3. CAN_MODE_SELFTEST : bit [2]; Self-Test Mode or LoopBack Mode performs a successful transmission, even without the acknowledgement. This test is performed on the active node by setting the CAN_CMD_SELF_RX_REQ bit. The working condition of the CAN controller can be verified by this test.

4. CAN_MODE_ACCFILTER : bit [3]; The Acceptance Filter can be operated in two modes.

   - Dual Filter Mode.
   - Single Filter Mode.

   By setting the CAN_MODE_ACCFILTER to logic 1 the controller is enabled to operate in Single Acceptance Filter Mode, else default logic 0, operate in Dual Filter Mode.

NOTE

(a) CAN_MODE address in memory : 0x3FF6B000.
(b) The write access to the mode register bits; ListenOnly, SelfTest, AcceptanceFilter are possible only by entering Reset Mode previously.
(c) Register value after hardware reset is 0x21.
(d) Normal Mode can be entered by clearing all the CAN_MODE register bit; necessary acceptance filter mode can be set.

3.1.2 Command Register (CAN_CMD)

The Command Register initiates the action within the transfer layer of the Controller. Internal clock cycle is mandatory between the two command settings.

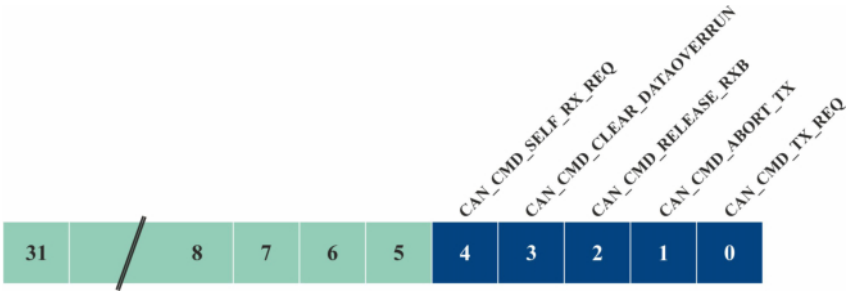


Figure. 3.3 Command Register

1. CAN_CMD_TX_REQ : bit [0]; The message can be transmitted by normally setting this bit. To cancel the transmission, CAN_CMD_ABORT_TX bit is set; reset of CAN_CMD_TX_REQ bit dose not cancel the transmission. The CAN_CMD_TX_REQ bit is cleared automatically either at the moment the CAN_STATUS_TX bit is set; signifying successful transmission of a message is in progress, or when the whole message has been read from the Transmit Buffer register.

   Single Shot Transmission is possible by setting the CAN_CMD_TX_REQ bit and CAN_CMD_ABORT_TX bit simultaneously. This command performs transmission without re-transmission in the event of arbitration lost or error capture.

2. CAN_CMD_ABORT_TX : bit [1]; This bit is set to suspend the previously requested transmission and to transmit a higher priority message.

3. CAN_CMD_RELEASE_RXB : bit [2]; This set to logic 1, after reading the the message from RX FIFO enabling the space for the next available message.

4. CAN_CMD_CLEAR_DATAOVERRUN : bit [3]; The CAN_STATUS_DATAOVERRUN bit is cleared by setting this bit. No further CAN_INTERRUPT_DATAOVERRUN is generated until the CAN_STATUS_DATAOVERRUN bit is set.

5. CAN_CMD_SELF_RX_REQ : bit [4]; This bit is set to transmit and receive message simultaneously. The bit is set during the Self-Test Mode.

NOTE

(a) CAN_CMD register address in memory - 0x3FF6B004.
(b) CAN_CMD is write only register. Reading the register value returns - 0xFF.
(c) Setting CAN_CMD_SELF_RX_REQ bit and CAN_CMD_TX_REQ bit simultaneously will ignore the set CAN_CMD_TX_REQ.
(d) During Self-Test Mode write the CAN_CMD with CAN_CMD_SELF_RX_REQ bit and write CAN_CMD_TX_REQ bit in Normal Mode.
(e) Combination of CAN_CMD register.
   - Single-Shot Transmission - by setting CAN_CMD_TX_REQ bit and CAN_CMD_ABORT_TX bit.
   - Single-Shot Transmission and Reception - by setting CAN_CMD_SELF_RX_REQ bit and CAN_CMD_ABORT_TX bit.

3.1.3 Status Register (CAN_STATUS)

The Status of the CAN Controller is reflected by the content of the Status Register.

1. CAN_STATUS_RXB : bit [0]; This bit reflects the state of Receive buffer; full or empty. When the bit is logic 1 full; one or more complete messages are available in the RX FIFO.

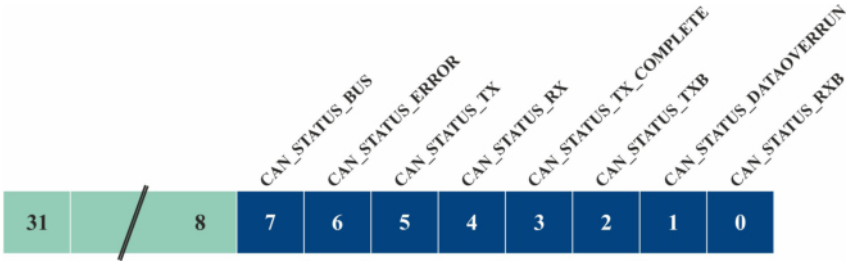


Figure. 3.4 Status Register

2. `CAN_STATUS_DATAOVERRUN` : bit [1]; The `RX FIFO` stores messages that passes successfully through acceptance filter. If the memory is `full`, the message will be dropped and indicated by this bit. The bit is cleared by setting the `CAN_CMD_CLEAR_DATAOVERRUN`.

3. `CAN_STATUS_TXB` : bit [2]; The `Transmit Buffer` register status is either `released` (`logic 1`) or `locked` (`logic 0`) which is indicated by this bit. When the status is `released` the message are written into the `Transmit Buffer`. When `locked` it indicates that there is either a message waiting for transmission or in the process of being transmitted.

4. `CAN_STATUS_TX_COMPLETE` : bit [3]; The status indicates the successful completion of the last transmission. This should be done after the `CAN_STATUS_TXB` bit has been set to `logic 1` or a Transmit Interrupt has been generated. The bit resets automatically whenever Transmission Request bit or Self Reception Request bit is set.

5. `CAN_STATUS_RX` : bit [4]; This bit indicates that the controller is receiving a message.

6. `CAN_STATUS_TX` : bit [5]; This bit indicates that the controller is transmitting a message.

7. `CAN_STATUS_ERROR` : bit [6]; When any of the error counters exceeds the warning limit is reflected by this bit.

8. `CAN_STATUS_BUS` : bit [7]; `bus-off`; `bus-on`; The bus activities either on or off are indicated by this bit. When `Transmission Error Counter > 255` the bit is set to `logic 1`; `bus-off` and simultaneously, the `CAN_MODE_RESET` bit is set and an `CAN_INTERRUPT_ERR_WARNING` is generated. In reset mode the `Transmit Error Counter` is set to 127 and the `Receive Error Counter` is cleared. The controller remains in the same state until the reset mode is cleared. The controller then waits for 128 occurrences of the bus-free signal counting down the `Transmit Error Counter`. The Error Counter bits are also cleared when `CAN_STATUS_BUS` bit is cleared.

NOTE

(a) The `CAN_STATUS` register memory address - `0x3FF6B008`
(b) The Status register is `Read-Only` memory.
(c) Register value after reset - `0x0C`.

(d) The `Receive buffer` status is cleared after reading the message in `RX FIFO` and setting the `CAN_CMD_RELEASE_RXB` bit.

(e) Trying to write message to the Transmit Buffer when `locked`, the message will not be accepted and lost without any indication.

(f) If bit the `CAN_STATUS_TX` and `CAN_STATUS_RX` are `logic 0` the bus state is idle. On a hardware reset, 11 consecutive recessive bit has to be detected to reach the idle state.

3.1.4 Interrupt Register (`CAN_INTERRUPT`)

The Interrupt register allows the identification of an Interrupt source. When one or more bits of the Interrupt register are set, an output interrupt will be indicated.

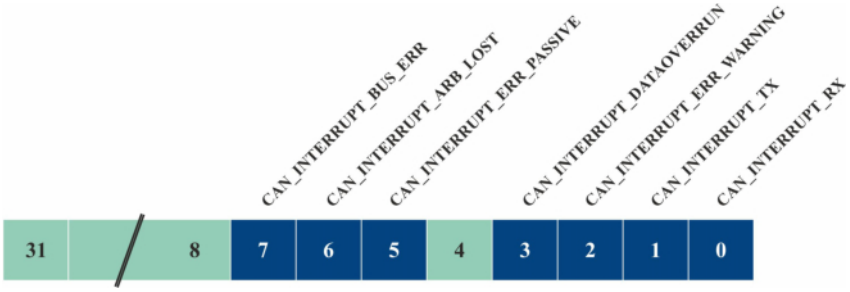


Figure. 3.5 Interrupt Register

1. `CAN_INTERRUPT_RECEIVE` : bit [0]; This bit is set when a message is present in `RX FIFO` and reset when `RX FIFO` is empty. If there is another message available within the `RX FIFO` after the `CAN_CMD_RELEASE_RXB` bit is set, the `CAN_INTERRUPT_RECEIVE` bit is set again. Otherwise it remains cleared.

2. `CAN_INTERRUPT_TRANSMIT` : bit [1]; Whenever the Transmit Buffer status changes from 0 `locked` to 1 `release`; (edge controlled) the `CAN_INTERRUPT_TRANSMIT` bit is set.

3. `CAN_INTERRUPT_ERR_WARNING` : bit[2]; The change (set and reset) in `CAN_STATUS_ERROR` and `CAN_STATUS_BUS` bit are reflected in this bit.

4. `CAN_INTERRUPT_DATAOVERRUN` : bit [3]; The bit is set when `CAN_STATUS_DATAOVERRUN` bit is active.

5. `CAN_INTERRUPT_ERR_PASSIVE` : bit [5]; This bit is set when Error Counter > 127 (Error Passive) or change from Error Passive to Error Active. (See Section 1.2.7)

6. `CAN_INTERRUPT_ARB_LOST` : bit [6]; `set`, when the controller arbitration is lost and becomes a receiver. Bit position where arbitration has been lost is located within `CAN_ALC` register.

7. `CAN_INTERRUPT_BUS_ERR` : bit [7]; `set`, when a bus error occurs (bit, stuff, crc, ack or form error).

NOTE

(a) `CAN_INTERRUPT` register memory address - `0x3FF6B00C`.
(b) Register value after reset - `0xE0`.
(c) Except for the `CAN_INTERRUPT_RECEIVE` bit, all bits are cleared after reading.
(d) `CAN_INTERRUPT_RECEIVE` bit is cleared temporarily when `CAN_CMD_RELEASE_RXB` bit is present.
(e) The transmit interrupt is generated even if the message was aborted because the `CAN_STATUS_TXB` bit changes to `released`.
(f) A new Bus error interrupt is not possible until the `Error Code Capture` register is read out once.

3.1.5 Interrupt Enable Register (`CAN_IER`)

The Interrupt Register are set by enabling the corresponding Interrupt Enable register bits. The register is read and write memory.

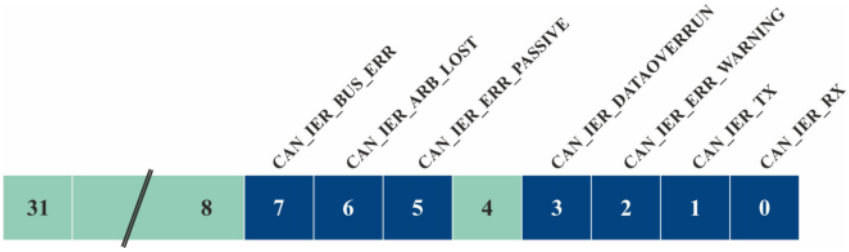


Figure. 3.6 Interrupt Enable Register

3.1.6 Arbitration Lost Capture Register (`CAN_ALC`)

The bit position of the lost arbitration is reflected by the `CAN_ALC` register. On arbitration lost, the corresponding arbitration lost interrupt is enabled.

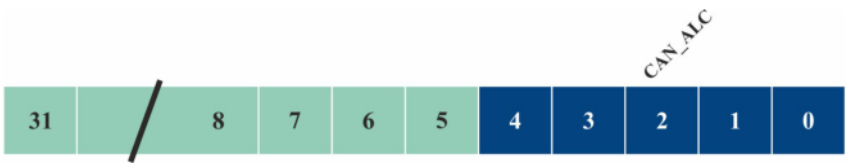


Figure. 3.7 Arbitration Lost Capture Register

3.1.7 Error Code Capture Register (CAN_ECC)

The type and location of bus error occurred during transmission and reception are indicated on this register. This register memory is read only.

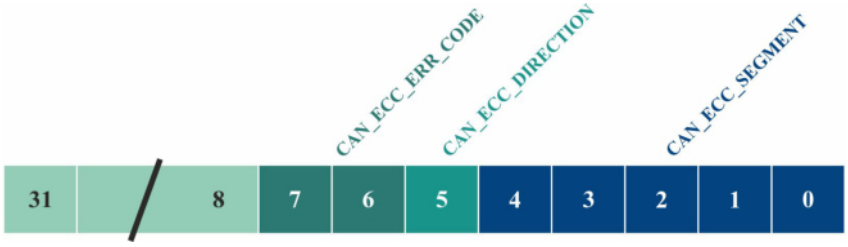


Figure. 3.8 Error Code Capture Register

1. CAN_ECC_SEGMENT : bit [4 - 0]; the bit settings reflect the error on the corresponding segment of the current message frame.
2. CAN_ECC_DIRECTION : bit [5]; The bit tokens the error either during transmission; logic 0 or reception; logic 1.
3. CAN_ECC_ERR_CODE : bit [7 - 6]; The type of error on the bus is indicated by this bit. 0 - bit error, 1 - form error, 2 - stuff error and 3 - other type of error.

NOTE

(a) The Error Code Capture register does not indicate a error until the previous error content in the register is read.
(b) Bus error interrupt is not possible until the Error Code Capture register is cleared by reading.

3.1.8 Error Warning Limit Register (CAN_EWLR)

The Error Warning Limit can be defined within this register. The default value of the register is 96. The register memory is read/write only in reset mode.

Figure. 3.9 Error Warning Limit Register

CAN_EWLR : bit [7 - 0]; value that determines the Error Warning Limit. When one of the Error Counters reaches the Error Warning Limit and the CAN_IER_ERR_WARNING bit of Interrupt Enable register is set, the CAN_INTERRUPT_ERR_WARNING will be generated. The content of the Error Warning Limit register can only be changed in Reset mode. A change in the error state

and the corresponding Error Warning Interrupt that is generated, will occur only in operating mode.

3.1.9 Receive Error Counter Register (CAN_RXERR)

The Receive Error count is indicated on the Receive Error Counter. The register memory is read/write only in Reset mode. Maximum value of the counter is 0x7F.

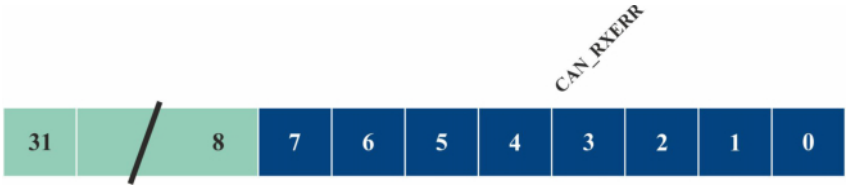


Figure. 3.10 Receive Error Counter Register

CAN_RXERR : bit [7 - 0]; value that determines the output of the Receive Error Counter. A change in the error state and the corresponding Error Warning Interrupt or Error Passive Interrupt forced by the new register content, will not occur until the mode changed from Reset to operating.

3.1.10 Transmit Error Counter Register (CAN_TXERR)

The Transmit Error Counter register indicates the Transmission Error Count. The register memory is read only in operation mode.

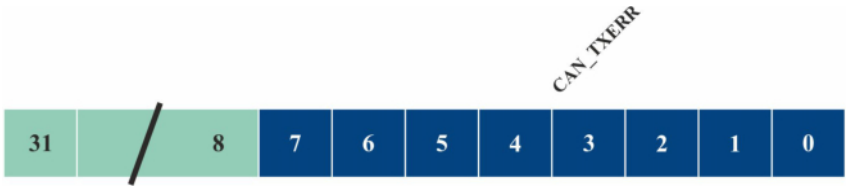


Figure. 3.11 Transmit Error Counter Register

CAN_TXERR : bit [7 - 0]; value that determines the output of the Transmit Error Counter. If a Bus-Off event occurs, the Transmit Error Counter is initialized to 127 to count the minimum protocol-defined time (128 occurrences of the bus-free signal). Reading the counter during this time gives information about the status of the Bus-Off recovery.

The content of the Transmit Error Counter can only be changed in Reset mode. A change in the error state and the corresponding Error Warning Interrupt or Error Passive Interrupt forced by the new register content, will not occur until Reset mode has been cleared.

3.1.11 Receive Message Counter Register (CAN_RXM_COUNTER)

The register indicates the number of message in the RX FIFO. The register value increases each time a new message is added to the RX FIFO.

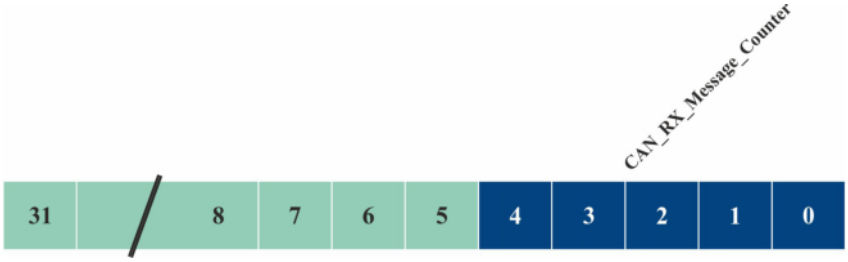


Figure. 3.12 Receive Message Counter Register

Note

(a) The register memory is read only.
(b) During the read-out the register value returned as 0.
(c) Register bit [7 - 5]; always in low logic state,logic 0.
(d) CAN_CMD_RELEASE_RXB command decrements the register value.

3.1.12 Clock Divider Register (CAN_CLK_DIVIDER)

The Register value defines the type of the operation mode; PeliCAN or BasciCAN mode and configures the CLKOUT frequency. CLKOUT is a pre-scaled version of System Clock (APB CLK).

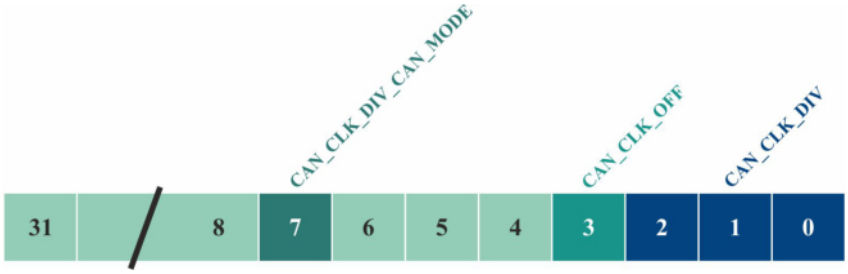


Figure. 3.13 Clock Divider Register

CAN_CLK_DIV : bit [2 - 0]; the value defines clock divider for generation of CLKOUT frequency at the CLKOUT pin.

CAN_CLK_OFF : bit [3]; setting this pin disables the CLKOUT pin.

CAN_CLK_DIV_CAN_MODE : bit [7];

- logic 1 : PeliCAN mode; additional registers are exposed. Transmit CAN2.0B CAN message frame. Both Standard and Extended Frame format are supported in PeliCAN mode.

- `logic 0` : `BasicCAN` mode. Only Standard Frame Format are supported, if extended frame are detected on the CAN bus they are tolerated and an acknowledgement is given if the message were correct, but no receive interrupt is generated.

NOTE

(a) `CAN_CLK_DIV` bits are accessible during both Reset and operating modes.
(b) `CAN_CLK_DIV_CAN_MODE` bit write access is possible only in `Reset mode`.

3.2 Transmit and Receive Buffer Register

Both the Transmit and Receive buffer share the same CAN memory address.

- `Frame Information` : `CAN_FRAME_INFO`
- `Frame Identifier` : `CAN_ID_SFF` and `CAN_ID_EFF`
- `Frame Data` : `CAN_DATA_SFF` and `CAN_DATA_EFF`.

Register Address	Standard Frame Format	Extended Frame Format
0x3FF6B040	CAN_FRAME_INFO	CAN_FRAME_INFO
0x3FF6B044	CAN_ID_SFF (0)	CAN_ID_EFF (0)
0x3FF6B048	CAN_ID_SFF (1)	CAN_ID_EFF (1)
0x3FF6B04C	CAN_DATA_SFF (0)	CAN_ID_EFF (2)
0x3FF6B050	CAN_DATA_SFF (1)	CAN_ID_EFF (3)
0x3FF6B054	CAN_DATA_SFF (2)	CAN_DATA_EFF (0)
0x3FF6B058	CAN_DATA_SFF (3)	CAN_DATA_EFF (1)
0x3FF6B05C	CAN_DATA_SFF (4)	CAN_DATA_EFF (2)
0x3FF6B060	CAN_DATA_SFF (5)	CAN_DATA_EFF (3)
0x3FF6B064	CAN_DATA_SFF (6)	CAN_DATA_EFF (4)
0x3FF6B068	CAN_DATA_SFF (7)	CAN_DATA_EFF (5)
0x3FF6B06C	-	CAN_DATA_EFF (6)
0x3FF6B070	-	CAN_DATA_EFF (7)

Table 3.1 Transmit and Receive Buffer Layout for SFF and EFF

3.2.1 Transmit and Receive Buffer Frame Information (`CAN_FRAME_INFO`)

Transmit and Receive Frame Information register is same for both Standard Frame Format and Extended Frame Format.

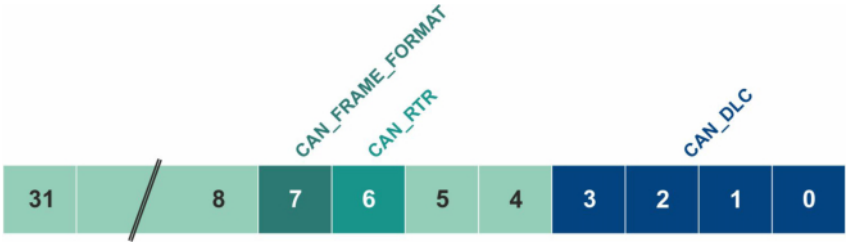


Figure. 3.14 Frame Information Register

`CAN_DLC` : bit [3 : 0]; Data Length Code. The number of bytes in the data field of a CAN message. In Remote frame the data length code is not considered as absence of data field. For compatibility reasons data length code should be less than 8.

`CAN_RTR` : bit [6]; Remote Transmission Request.

1. `logic 1` : `CAN_RTR`. Remote frame will be transmitted by the CAN controller.
2. `logic 0` : data; data frame will be transmitted by the CAN controller.

`CAN_FRAME_FORMAT` : bit [7]; Frame Format.

1. `logic 1` : `CAN_FRAME_FORMAT_EFF`. Extended Frame Format.
2. `logic 0` : `CAN_FRAME_FORMAT_SFF`. Standard Frame Format.

3.2.2 Transmit and Receive Buffer Identifier (`CAN_ID_SFF` and `CAN_ID_EFF`)

The Identifier acts as the message name. Standard Frame identifier consists of 11 bits and Extended Frame consists of 29 bits. ID 28 the most significant bit of the identifier frame is transmitted first on the bus during the arbitration process. It is subsequently used in a receiver for acceptance filtering. It also determines the bus access priority during the arbitration process. The lower the binary value of the identifier, the higher the message priority. This is due to the larger number of leading dominant bits during arbitration.

31 / 8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
CAN_ID_SFF (0)	ID 28	ID 27	ID 26	ID 25	ID 24	ID 23	ID 22	ID 21
CAN_ID_SFF (1)	ID 20	ID 19	ID 18	X / RTR	X	X	X	X

Figure. 3.15 Standard Frame Format Identifier

Figure 3.15 denotes the bit field of Transmit and Receive Frame Identifier for standard frame format. The `X` are don't cares. For the receive frame identifier the bit[4] of `CAN_ID_SFF (1)` is a RTR bit which is a direct copy of the RTR bit from Frame Information register, and for transmit frame identifier it is a don't care bit.

Figure. 3.16 Extended Frame Format Identifier

Figure 3.16 denotes the bit field of Transmit and Receive Frame Identifier for extended frame format. The `X` are don't cares. For the receive frame identifier the bit[2] of `CAN_ID_EFF (3)` is a RTR bit which is a direct copy of the RTR bit from Frame Information register, and for transmit frame identifier it is a don't care bit.

3.3 CAN Bit Timing Registers

The CAN bit timing registers are one of the important registers to be initialized during configuration of the CAN controller to work with the desired bit rate. The register value defines the bit rate of the CAN controller in the CAN bus. The non-overlapping segments CAN bit in Figure 3.17 are slightly different from the Bosch CAN bit segment in Figure 1.10. Each CAN family specification has unique CAN bit segment specification, but the mechanism remains the same.

The ESP32 CAN controller bit segments are defined according to the NXP manufacturers (SJA1000 CAN controller). The bit timing segments are defined as,

- `SYN_SEG` : Synchronize all nodes on the bus.
- `TSEG1` : It is the sum of the `PROP_SEG` and the `PHASE_SEG1`. This facilitates the programming of bit timing parameters.
- `TSEG2` : It is `PHASE_SEG2`.
- `Sample point` : Sampling point is the point of time at which the bus level is read and interpreted as the value at that respective time.
- `Time Quanta (TQ)` : The Time Quanta is a fixed unit of time derived from the oscillator period.
- `Baud Rate Prescaler (BRP)` : The Time Quanta is equal to the period of the CAN system clock, which is derived from the system clock or oscillator by dividing the system clock or oscillator by the programmable pre-scalar, called `BRP`.

- `Nominal Bit Time (NBT)` : This is the sum of all the CAN bit time segments.

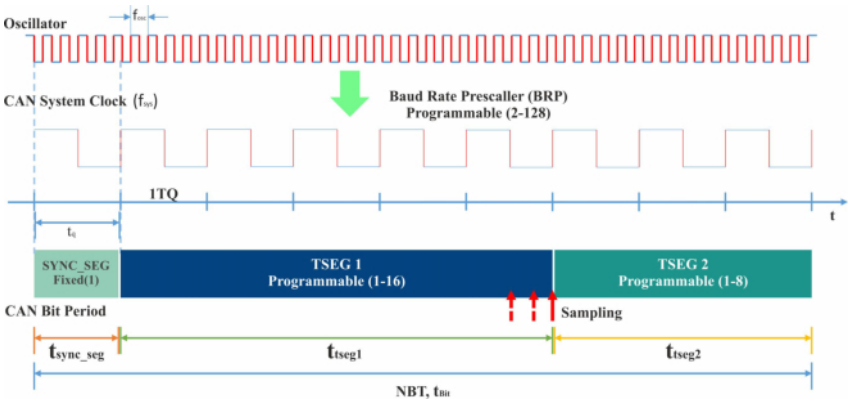


Figure. 3.17 CAN Bit Time Segments

In Figure 3.17, f_{osc} is the oscillator period. The length of time quanta t_q; t_q = BRP / f_{sys}. f_{sys} is the CAN controller system clock; $f_{sys} = f_{osc}$ or $f_{sys} = f_{osc} / 2$.

The bit timing configuration `BRP`, `SJW`, `TSEG1`, `TSEG2` have to be programmed in two register bytes; `CAN_BTR0` and `CAN_BTR1`.

3.3.1 Bus Timing Register 0 (`CAN_BTR0`)

`CAN_BTR0` register controls the Baud Rate Prescaler (`BRP`) and Synchronization Jump Width (`SJW`). The register memory is read-only in operation mode.

Figure. 3.18 Bus Timing 0 Register

1. `CAN_BTR0_BRP` : bit [5 : 0]; write the value of Baud Rate Prescaler.

   $t_q = \frac{2 \times BRP}{fosc}$

   BRP = 32 × bit[5] + 16 × bit[4] + 8 × bit[3] + 4 × bit[2] + 2 × bit[1] + bit[0] + 1.

2. `CAN_BTR0_SJW` : bit [7 : 6]; `SJW` defines the maximum number of time quanta a bit period may be shortened or lengthened by one re-synchronization, to compensate for phase shifts between clock oscillators of different bus controllers and propagation delays between CAN-bus nodes.

   $t_{SJW} = \frac{SJW}{t_q}$

   SJW = 2 × bit[7] + bit[6] + 1

NOTE

The CAN_BTR0 register memory write access is granted only in Reset mode.

3.3.2 Bus Timing 1 Register (CAN_BTR1)

The length of bit period, location of sample point and the number of samples taken at each sample point are written into the CAN_BTR1 register. In operating mode the register memory is read only.

Figure. 3.19 Bus Timing 1 Register

Time Segment 1 and Time Segment 2 determine the number of time quanta per bit period and the location of the sample point.

- CAN_BTR1_TSEG1 : bit [3 : 0];

  $t_{TSEG1} = TSEG1 \times t_q$

  $TSEG1 = 8 \times bit[3] + 4 \times bit[2] + 2 \times bit[1] + bit[0] + 1$

- CAN_BTR1_TSEG2 : bit [6 : 4];

  $t_{TSEG2} = TSEG2 \times t_q$

  $TSEG2 = 4 \times bit[6] + 2 \times bit[5] + bit[4] + 1$

CAN_SAMPLING : bit [7]; Two sampling modes are supported.

- logic 1 : Triple Sampling; the bus is sampled three times. Recommended for low / medium speed buses.
- logic 0 : Single Sampling; recommended for high speed buses

3.3.3 ESP32 CAN bit time constraints

Every CAN module has its unique bit timing constraints. It is mandatory to respect the bit timing constraints to attain the desired bit rate on the CAN Network.

Segments	Min TQ	Max TQ
BRP	2	128
SYNC_SEG	1 (fixed)	
TSEG1	1	16
TSEG2	1	8
SJW	1	4
TQ	3	25

Table 3.2 ESP32 CAN Bit Timing Constraints

3.4 Acceptance Filter Register

Acceptance Filter allow an automatic check on the identifier and data bytes of the CAN message to be accepted by the CAN node receiver to store the message in the Receive buffer. A CAN node that filters out a message will not receive the message, but still it will acknowledge the message.

The acceptance filter is controlled by two registers `Acceptance Code` and `Acceptance Mask`. In PeliCAN mode these registers are expanded to 4; 8-bit wide registers for a versatile filtering of the messages.

- `Acceptance Code:` The bit sequence of the message: `ID`, `RTR`, `Data bytes`, to be received by the controller is defined with in the `Code` register.
- `Acceptance Mask:` The bit positions of the `acceptance code` that can be ignored are defined in this registers.

For accepting a CAN message, all the received bits have to match the respective bits of the Acceptance code register. The acceptance filter can be used in two modes by the `CAN_MODE_ACC_FILTER` bit of `CAN_MODE` register (See Item 4 Page 25).

- `Single Filter Mode`.
- `Dual Filter Mode`.

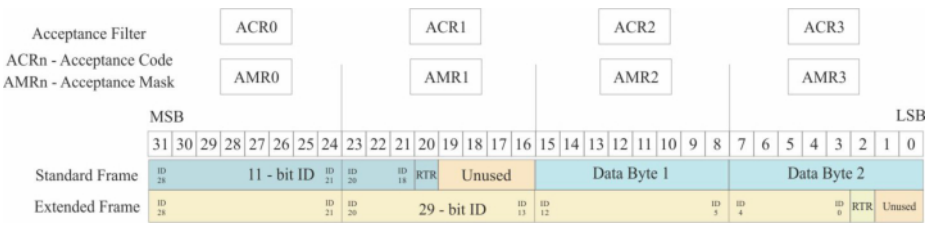


Figure. 3.20 Single Filter Mode (Standard and Extended Frame)

The acceptance filter registers configuration are different for the standard and extended frame format in single and dual filter mode. The Figure 3.20 shows the single filter mode configuration

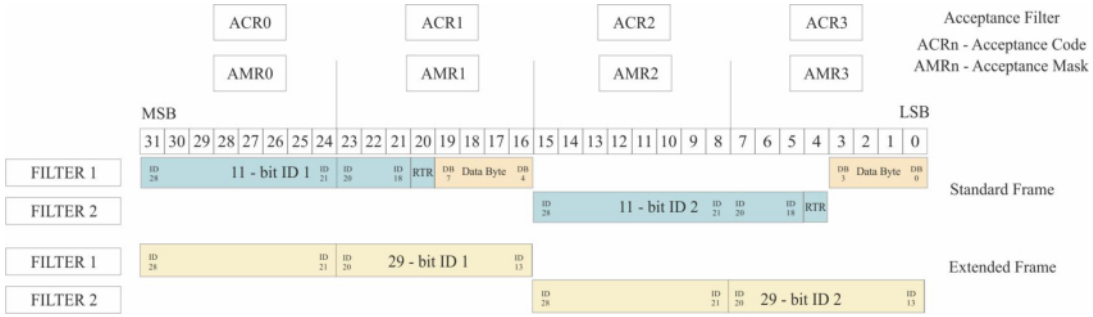


Figure. 3.21 Dual Filter Mode (Standard and Extended Frame)

of registers for the standard and extended frame. The ACRn is the acceptance code register and AMRn is the acceptance mask register. For standard frame the 11-bit ID, RTR, Data byte1 and Data byte2 are used for acceptance. The 8-bit of the register ACR0, ACR2, ACR3 and upper 4bits of ACR1 are used. The unused bits of AMR1 register should be set to 1. For extended frame 29-bit ID and RTR bits used for acceptance. In ACR3 and AMR3 register only the upper 6 bits are used. The unused bits of AMR3 register must be set to 1.

Figure 3.21 shows the register configuration for the dual filter mode. For standard frame only one data byte is used for acceptance in Filter 1. For extended frame only the 16 most significant bit of extended frame ID is used.

For example two message ID have to be accepted by the CAN node. Data byte and RTR bit are not considered. The Code and Mask register configuration are:

Message ID 1 : 0x205 : 0100 0000 101
Message ID 2 : 0x2A5 : 0101 0100 101

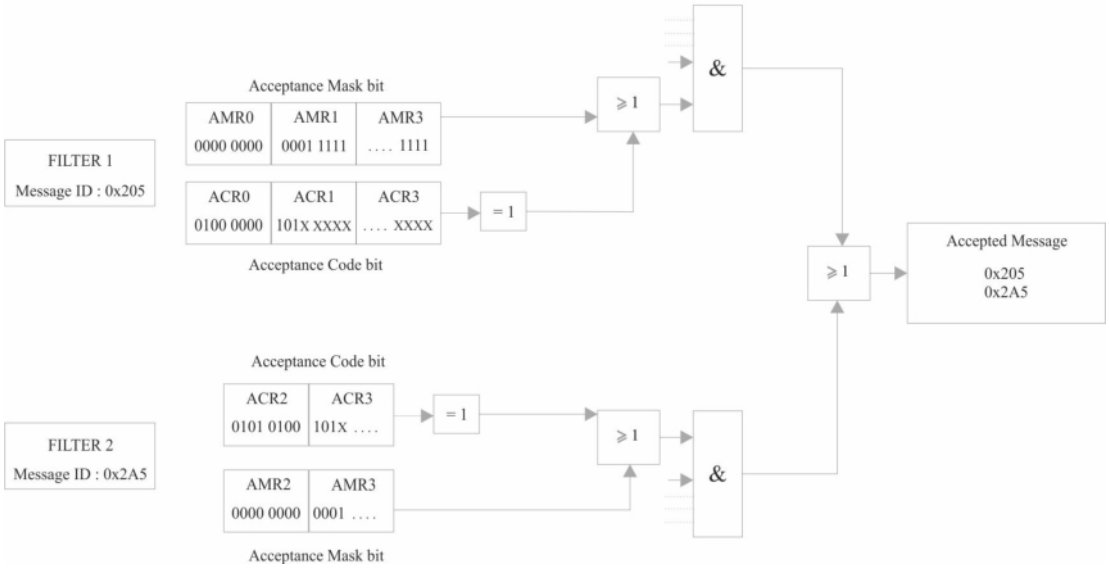


Figure. 3.22 Example: Dual Filter Mode (Standard Frame)

Chapter 4

ESP32 CAN Driver

4.1 Computation of CAN Bit Settings : `ESP32ACANSettings` class

The Initial phase of developing a CAN driver is to define the Bit Timing Calculator. CAN bus operates in different data rates. Each data rate defines different bus timing configuration. This section details the driver CAN bus timing calculation.

`ESP32ACANSettings` class computes the value of the bit timing segments for the desired bit rate. Other features of the class is that it explores all CAN bit rates from 1 bit/s to 1 Mbit/s for the ESP32 CAN controller source clock (See Figure 4.1). It is found that the CAN controller operates with the APB clock in ESP32 module [23]. It also checks that all the computed CAN bit decomposition are consistent, even if they are too far from the desired bit rate.

4.1.1 `ESP32ACANSettings` Constructor

The constructor of the `ESP32ACANSettings` has one mandatory argument: the Desired Bit Rate. The constructor computes the CAN bit settings for the desired bit rate. On a successful computation, the class property `mBitRateClosedToDesiredRate` is set to `true`, else it is set to `false` and other bit timing segments are set on respecting the constraints.

The computation process of the Bit timing parameters:

Step 1: Baud Rate Prescaler `BRP` and Nominal Bit Time `NBT` are determined with the dependency of the desired bit rate (input argument) and clock frequency (APB clock 80 MHz).

$$\mathtt{BRP}_{\mathtt{min}} = \frac{\mathtt{APB\_CLOCK\_FREQUENCY}}{(\mathtt{DESIRED\_BIT\_RATE} * \mathtt{NBT}_{\mathtt{min}})}$$

Step 2: Time Segment 2 `TSEG2` is calculated with a sampling rate of 80%. Sampling point can be in the range from 50% to 90%. The value is fixed respecting the constrains.

Step 3: Time Segment 1 TSEG1 is set with the remaining value of NBT obtained by subtracting the TSEG2 and Sync Segment (fixed 1). If the TSEG1 value exceeds the maximum value, TSEG2 is updated to satisfy the maximum value.

Step 4: SJW segment value is set 3; if the value of TSEG2 < 4, else SJW is set to 4.

Step 5: Sampling : Single or Triple sampling type is defined on the required condition.

Step 6: Finally the configuration is compared with the Tolerance PPM(part-per-million) and the bit settings closed to desired bit rate property is returned true or false.

The constraints of the Bit Timing property are:

$$2 \leq \texttt{mBitRatePrecaler} \leq 128$$
$$1 \leq \texttt{mSJW} \leq 4$$
$$1 \leq \texttt{mTimeSegment1} \leq 16$$
$$1 \leq \texttt{mTimeSegment2} \leq 8$$

4.1.2 ESP32ACANSettings class Properties

The mandatory parameters for the CAN bit timing calculation are the CAN Module Clock Source and Desired bit rate. The CAN controller operates with the APB clock in ESP32 module. The Peripheral input clock macros are defined in soc.h file of the ESP32 hardware.

```
#define  APB_CLK_FREQ ( 80*1000000 )     //unit: Hz
```

The Clock source does not change in any circumstances, therefore it is defined as a static constant property kSourceClockAPB.

The properties of the ESP32ACANSettings class :

Property	Type	Initial value
mDesiredBitRate	uint32_t	constructor argument
mBitRatePrescaler	uint8_t	0
mTimeSegment1	uint8_t	0
mTimeSegment2	uint8_t	0
mSJW	uint8_t	0
mTQcount	uint8_t	0
mTripleSampling	bool	false
mBitRateClosedToDesiredRate	bool	false
mRequestedCANMode	CANMode	NormalMode
mControlMessagebyMethod	CANProcess	InterruptControlled
mDriverReceiveBufferSize	uint16_t	32
mDriverTransmitBufferSize	uint16_t	16

Table 4.1 Properties of the ESP32ACANSettings class

The ESP32ACANSettings::mRequestedCANMode property defines the mode requested at the end of the configuration. By default the mode is set to NormalMode, other modes LoopBackMode and ListenOnlyMode.

The ESP32ACANSettings::mControlMessagebyMethod property defines the message control method: InterruptControlled (default) and PollingControlled.

4.1.3 ESP32ACANSettings Methods

These methods verify the consistency of the CAN bit time settings. The return value of the methods are irrelevant, if the CAN bit settings are not consistent.

actualBitRate method

The actualBitRate method returns the actual bit rate computed from the property values mBitRatePrescaler, mTimeSegment1, mTimeSegment2, mSJW.

$$\text{Actual bit rate} = \frac{\texttt{kSourceClockAPB}}{\texttt{mBitRatePrescaler}\cdot(\texttt{SYNC\_SEG} + \texttt{mTimeSegment1} + \texttt{mTimeSegment2})}$$

ppmFromDesiredBitRate method

The ppmFromDesiredBitRate method returns the distance from the actual bit rate to the desired bit rate (in ppm): 1 ppm = 10^{-6}. In other words, 10,000 ppm = 1%

samplePointFromBitStart method

This method returns the distance of sample point from the start of the CAN bit, expressed in part-per-cent (ppc): 1 ppc = 10^{-2} = 1%. If triple sampling is selected, the returned value is the distance of the first sample point from the start of the CAN bit. It is good practice to get sample point from 65% to 80%.

$$\text{Sampling point (}\texttt{single sampling}\text{)} = 100 \cdot \frac{\texttt{SYNC\_SEG} + \texttt{mTimeSegment1}}{\texttt{SYNC\_SEG} + \texttt{mTimeSegment1} + \texttt{mTimeSegment2}}$$

$$\text{Sampling point (}\texttt{triple sampling}\text{)} = 100 \cdot \frac{\texttt{mTimeSegment1}}{\texttt{SYNC\_SEG} + \texttt{mTimeSegment1} + \texttt{mTimeSegment2}}$$

exactBitRate method

The method returns true if the actual bit rate matches the desired bit rate, and false otherwise.

`CANBitSettingConsistency` **method**

This method checks the consistency of the CAN bit segment property values given by: `mBitRate Prescaler`, `mTimeSegment1`, `mTimeSegment2`, `mSJW`. It returns 0, if the CAN bit timing values are consistent, else the returned value is a bit field of error properties.

The static constant properties of the error bit field are:

Bit	Error Name	Error
0	`kBitRatePrescalerIs LowerThan2`	`mBitRatePrescaler` < 2
1	`kBitRatePrescalerIsGreaterThan128`	`mBitRatePrescaler` > 128
2	`kTimeSegment1IsZero`	`mTimeSegment1` == 0
3	`kTimeSegment1IsGreaterThan16`	`mTimeSegment1` > 16
4	`kTimeSegment2IsZero`	`mTimeSegment2` == 0
5	`kTimeSegment2IsGreaterThan8`	`mTimeSegment2` > 8
6	`kTimeSegment1Is1AndTripleSampling`	(`mTimeSegment1` == 1) && `mTripleSampling`
7	`kSJWIsZero`	`mSJW` == 0
8	`kSJWIsGreaterThan4`	`mSJW` > 4
9	`kTimeSegment2IsGreaterTimeSegment1`	`mTimeSegment2` > `mTimeSegment1`

Table 4.2 The `ESP32ACANSettings::CANBitSettingConsistency` method error codes

4.1.4 Test on Desktop compiler

The `ESP32ACANSettings` class can be compiled on any desktop C++ compiler. The ESP32 CAN bit settings computation always succeeds for the classical bit rates: 125 kbit/s, 250 kbit/s, 500 kbit/s, 1 Mbit/s.

In Figure 4.1, Result : 1,2,3,4 shows the computed bit timing segment values for classical CAN bit rates. Result : 5 and 6, are computed for the desired bit rate of 20 kbit/s and 10 kbit/s; the bit settings are not close to desired bit rate and the property `mBitRateClosedToDesiredRate` returns `false` which results in (`Settings OK: no`). The actual bit rate computed does not match with the desired bit rate. The distance from the actual bit rate to the desired bit rate are printed in `ppm` field (`ppm:250000`) and the sample point is printed in `Sample Point` field.

All CAN bit rate settings are explored from 1 bit/s to 1 Mbit/s: 311088 bit rates are valid, and 25 are exact bit rates. An exhaustive search is performed to print all the valid exact CAN bit rate (Figure 4.2).

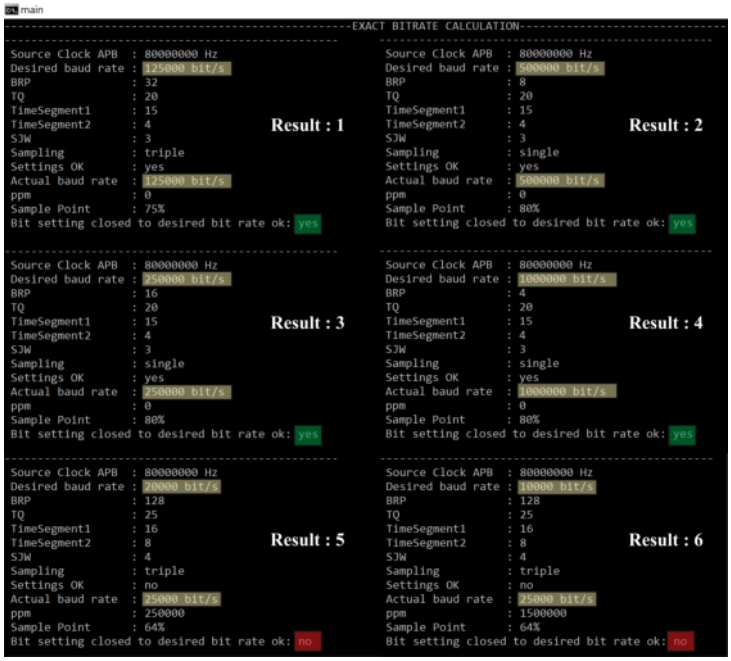


Figure. 4.1 Output: Bit Time Settings(test on desktop)

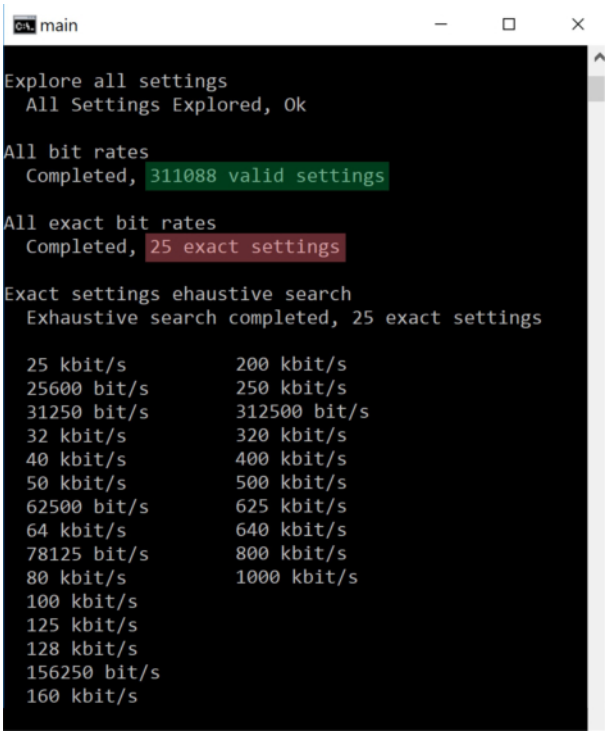


Figure. 4.2 Output: All valid ESP32 CAN bit rate (test on desktop)

4.2 ESP32 CAN Register Definition

ESP32 CAN Peripheral registers are not defined in the ESP32 Technical Reference Manual with the forum and available source the SJA1000 CAN compatible register are defined according to the other ESP32 Peripheral registers (See Section 3.1 Page 23). The ESP32 CAN Peripheral registers

are 8bits, however the ESP32 can only access peripheral registers every 32bits. Therefore each ESP32 CAN register are mapped to the least significant byte of every 32bits.

ESP32 CAN Register Base Address **0x3FF6B000** .

The Peripheral registers of a device can be accessed using different approaches like pointer access, struct access., The performance between the approaches remains the same. For better code readability and organization we prefer to use the pointer access.

Mapping of Register address with pointer

A pointer to the register can be created using either a pointer to a `volatile const uint32_t` or by a macro definition.

The example of a pointer to a volatile const uint32_t:

```
volatile uint32_t* const CAN_REG = (uint32_t*) REG_ADDRESS;
```

To understand better use of this definition one should know the keywords `const` and `volatile`. The `const` keyword is a compiler-enforced, i.e., it makes the object non-modifiable type. The `volatile` is a type qualifier that prevents the object from the compiler optimization. If an object defined by `volatile` type, the compiler reloads the value from the memory each time it is accessed by the program; it prevents from the cache of a variable into a register. The use of `volatile` and `const` in the above object definition is very important as the current value of the register can be read without any assumption and change in the value by the code.

The workaround with the pointers may be complex, to make a simple implementation we use the macro definition of the register for the driver.

The example using macros to define the register:

```
#define REG  (*((volatile uint32_t*) REG_ADDRESS))
```

In the macro definition the left $(*)$ pointer dereferences `REG_ADDRESS`, after first casting it to type pointer to `volatile uint32_t` ; `REG` can be used as a plain variable. In simple words the current value of register memory are read by the `REG` variable.

The peripheral CAN register definition of `MODE` register.

```
static const uint32_t ESP32CAN_BASE = 0x3FF6B000;

typedef volatile uint32_t vuint32_t;

/*--- Configuration and Control Registers

  #define CAN_MODE        (*((vuint32_t *)(ESP32CAN_BASE)))
```

```
    /* Bit definitions and macros for CAN_MODE */
    static const uint32_t CAN_MODE_RESET      = 0x01 ;
    static const uint32_t CAN_MODE_LISTENONLY = 0x02 ;
    static const uint32_t CAN_MODE_SELFTEST   = 0x04 ;
    static const uint32_t CAN_MODE_ACCFILTER  = 0x08 ;
```

The macro definition of the Mode register with variable `CAN_MODE`. The bit of the CAN Mode register is defined with the constant bit address. Accessing specific bits can be done using bit shifting and bit masking.

For example the Mode register can be written and read by,

```
CAN_MODE = CAN_MODE_SELFTEST;
const unit32_t mode_value = CAN_MODE;
```

All other register definition is similar, writing the read-only register as `const volatile` type the write can be prevented. The header file `ESP32CANRegisters.h` Appendix contains the definition of all the ESP32 CAN Registers.

4.2.1 ESP32 CAN Registers Test

An arduino example code `ESP32CANRegisterTest.ino` is written to test the ESP32 CAN registers read and write access.

The ESP32 CAN Peripheral register mapping is defined by macros in the header file

```
#include <ESP32CANRegisters.h>
```

```
void setup() {
//--- Switch on builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
  digitalWrite (LED_BUILTIN, HIGH) ;
//--- Start serial
  Serial.begin (115200) ;
//--- Wait for serial (blink led at 10 Hz during waiting)
  while (!Serial) {
    delay (50) ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
  }
```

Builtin Led is used for signaling. It blinks the Led at 10 Hz during until serial monitor is ready.

```
//--- Enable the ESP32 CAN Peripheral
  periph_module_enable(PERIPH_CAN_MODULE);
```

For instance to work with the ESP32 CAN, the peripheral module must be enabled. This function is defined in the *espertools/sdk/include/driver/driver/periph_ctrl.h*

```
#include "driver/periph_ctrl.h"
```

This line includes the peripheral control functions.

```
void periph_module_enable(periph_module_t periph) {
    portENTER_CRITICAL(&periph_spinlock);
    DPORT_SET_PERI_REG_MASK(get_clk_en_reg(periph),
    get_clk_en_mask(periph));
    DPORT_CLEAR_PERI_REG_MASK(get_rst_en_reg(periph),
    get_rst_en_mask(periph, true));
    portEXIT_CRITICAL(&periph_spinlock);
}
```

The `periph_module_enable` function ungates the clock for the module, and reset is de-asserted, it returns NULL.

```
 Serial.println ("ESP32 CAN REGISTERS") ;
  for (uint32_t idx=0; idx<=31 ; idx++) {
    uint32_t value = REGALL(idx);
    uint32_t Reg_no = 0x3FF6B000+0x000+4*idx;
    Serial.printf("REG : %X --- %X \n",Reg_no,value);
  }
}
```

The current value of the CAN register is printed with the register address and register access is tested using the example code `ESP32CANRegisterTest.ino`.

```
void loop() {
....
 //--- Write Access to bus timming registers
 for (uint16_t i = 0; i <= 255; i++) {
   CAN_BTR0 = (byte)i;
   CAN_BTR1 = (byte)i;
   const uint32_t value1 = CAN_BTR0;
   const uint32_t value2 = CAN_BTR1;
//Serial.printf ("Value : %d---Reg BTR0 : %X \n",i,value1);
//Serial.printf ("Value : %d---Reg BTR1 : %X \n",i,value2);
   if ((i != value1) && (i != value2)) {
     Serial.println("Reg Error");
   }
 }
}
```

ESP32 CAN Registers are tested by writing byte value to the bus timing registers `BTR0` and `BTR1`. Register access returns error if the read value is not same as the write value.

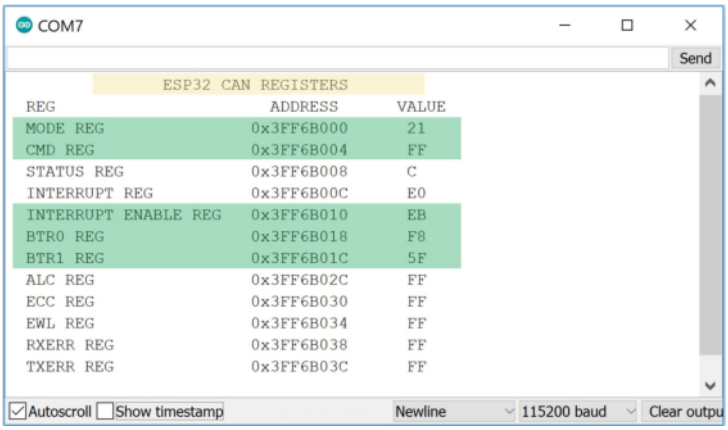


Figure. 4.3 Output: ESP32 CAN REGISTER reset value

Figure 4.3, shows the ESP32 CAN register value after reset. Some register memory are read only. The highlighted registers are read/write registers.

4.3 CAN Driver Initialization

The ESP32 CAN driver is written in ACAN style. `ESP32ACAN.h` defines the driver class and methods used to control the CAN Message.

The main part of the driver is to initialize the correct configuration setting needed for the operation of the CAN controller. The `ESP32ACAN::begin` method, setup the CAN driver initial configuration.

The access rights to CAN peripheral registers of the ESP32 module is possible only by enabling the CAN peripheral module. The `periph_module_enable(PERIPH_CAN_MODULE);` function enables the ESP32 CAN peripheral module. Some registers of the module can be written only by entering Reset mode. The CAN module supports both CAN message format which is defined in the `CAN_CLK_DRIVER` register (See Section 3.1.12 Page 32), `BasicCAN` mode and `PeliCAN` mode. By configuring the CAN mode to `PeliCAN` mode the complete CAN 2.0B functionality is supported. Note. The Driver register are exposed to `PeliCAN` mode, therefore it is advised to enter `PeliCAN` mode. The Bus timing registers has to be configured with the CAN bus timing settings for the desired bit rate. In order to accept all messages in the CAN bus set the default acceptance filter. Allocate the `GPIO` pins to the `Output RXD` pin and `Input TXD` pin respectively (See Section 2.5 Page 21). After writing all the configuration information to the control segments the CAN module is switched into requested mode by clearing the reset mode. The transmission process can be controlled by either Interrupt request or by Polling status (See Section 4.4 Page 51). In the case of Interrupt controlled the `Interrupt Service Routine` is installed to handle the interrupts triggered.

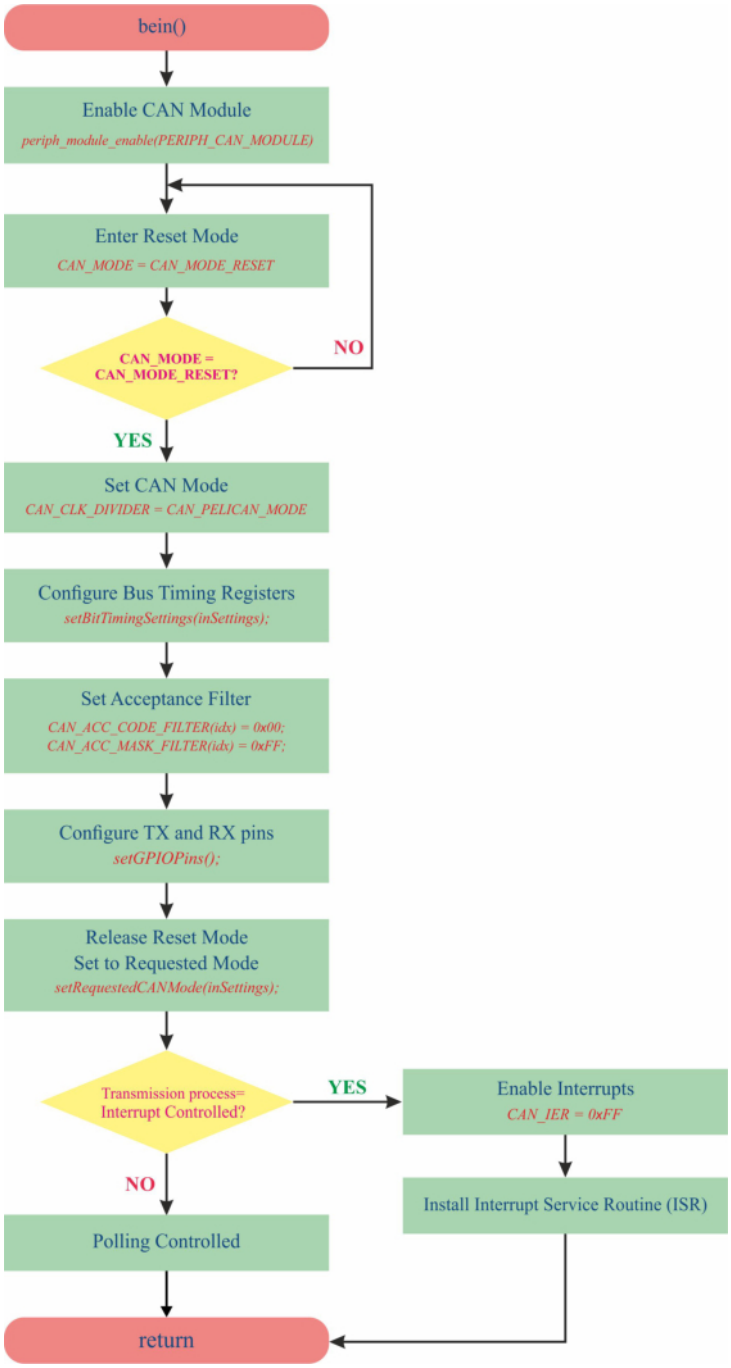


Figure. 4.4 Driver Initial Configuration

```
uint32_t ESP32ACAN::begin (const ESP32ACANSettings &inSettings)
```

The `ESP32ACAN::begin()` method takes the configuration setting parameters and returns the error code as value 0; if all the configuration are Ok.

4.3.1 Configuration Functions

GPIO Settings : `setGPIOPins()`

The CAN controller's signal lines has to be routed to the `GPIO` pins through `GPIO` matrix. We can select the alternative pins for connecting the `Tx` and `Rx` pins. The other two signal lines `Bus-off` and `CLKOUT` are optional. Since the project hardware setup is defined the `Tx` and `Rx` are fixed. It can be changed according to the user hardware setup. In this project setup, `GPIO 5` is assigned as `Tx` and `GPIO 4` is assigned as `Rx` pins. The `setGPIOPins()` function is called to configure the `Tx` and `Rx` pins.

```
void ESP32ACAN::setGPIOPins(void) {
//TX Pin - Default set to IO5.
//RX Pin - Default set to IO4.
  gpio_num_t TXPin = GPIO_NUM_5;
  gpio_num_t RXPin = GPIO_NUM_4;

  //Set TX pin
    gpio_set_pull_mode(TXPin, GPIO_FLOATING);
    gpio_matrix_out(TXPin, CAN_TX_IDX, false, false);
    gpio_pad_select_gpio(TXPin);

  //Set RX pin
    gpio_set_pull_mode(RXPin, GPIO_FLOATING);
    gpio_matrix_in(RXPin, CAN_RX_IDX, false);
    gpio_pad_select_gpio(RXPin);
    gpio_set_direction(RXPin, GPIO_MODE_INPUT);
}
```

Bus Timing Register Configuration : `setBitTimingSettings(settings)`

The ESP32 CAN works with the at most speed of 1 Mbit/s. The CAN Bus can be set to desired bit rates. Bus timing values computed for the desired bit rate by the `ESP32ACANSettings` class is written to the Bus Timing Registers `BTR0` and `BTR1` by `setBitTimingSettings(settings)` function. The value for the respective bit of the register is written by bit shift method.

```
void ESP32ACAN::setBitTimingSettings(const
                ESP32ACANSettings&inSettings)
{
/* SJW | BRP */
  CAN_BTR0 = ((inSettings.mSJW - 1) << 6) |
             ((((inSettings.mBitRatePrescaler) / 2) - 1) << 0)
      ;
/* Sampling | Tseg2 | Tseg1 */
```

```
  CAN_BTR1 = ((inSettings.mTripleSampling) << 7)    |
             ((inSettings.mTimeSegment2 - 1) << 4) |
             ((inSettings.mTimeSegment1 - 1) << 0)
      ;
}
```

Requested Mode Configuration : : `setRequestedCANMode(settings)`

The ESP32 CAN driver supports three modes of operation. `NormalMode`, `LoopBackMode` and `ListenOnlyMode`. These modes can be selected according to the desired application. The mode register bits are set in accordance with the requested mode. The Mode register macros defined in the `ESP32CANRegisters.h` file are used. By default the driver mode is set to `NormalMode` in the `ESP32ACANSettings` class.

```
void ESP32ACAN::setRequestedCANMode(const
                ESP32ACANSettings &inSettings) {

  uint8_t requestedMode = 0 ;
  switch (inSettings.mRequestedCANMode) {
    case ESP32ACANSettings::Normal :
      break ;
    case ESP32ACANSettings::ListenOnly :
       requestedMode = CAN_MODE_LISTENONLY ;
      break ;
    case ESP32ACANSettings::NoACK :
       requestedMode = CAN_MODE_SELFTEST ;
      break ;
  }

  CAN_MODE = requestedMode | CAN_MODE_RESET ;

  do{
    CAN_MODE = requestedMode ;
  }while ((CAN_MODE & CAN_MODE_RESET) != 0) ;
}
```

4.4 CAN Communication Process

The transmission and reception of message by the CAN controller is done according to the CAN protocol. The process can be controlled either by an Interrupt request or by Polling Status flags.

The driver is compatible to be operated by either of the control method. Driver Internal transmit and receive buffer are added to the driver for the Interrupt controlled process.

4.4.1 Polling Controlled Process

The transmission and reception of the message are controlled by the Status flag of the `CAN_STATUS` register.

Sending Frames

The flow of the polling controlled transmission is shown in the Figure 4.5. The `Transmit Buffer` is `locked` while writing the message to be transmitted. By checking the `CAN_STATUS_TXB` bit of the `CAN_STATUS`, the `lock` and `release` state of the `Transmit Buffer` can be monitored.

The `tryToSendbyPolling` method of the driver is called for sending the message in the CAN Network. When `Transmit Buffer` status is released a new message can be written into the `Transmit Buffer` by the `internalSendMessage` function. The function transfers the message to the `Transmit Buffer` registers and set the command `CAN_CMD_TX_REQ`; in `NormalMode`, `CAN_CMD_SELF_RX_REQ` command; in `LoopBackMode` to start the transmission.

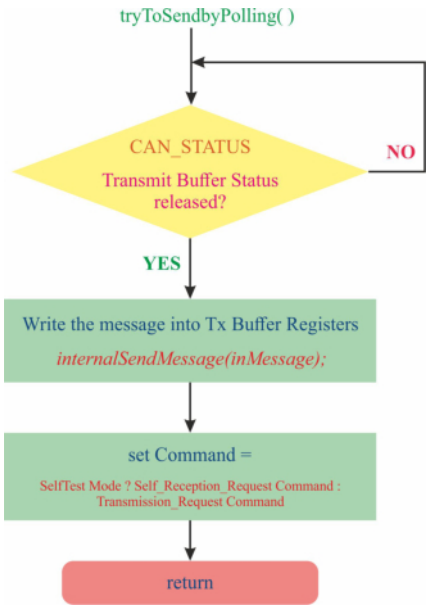


Figure. 4.5 Transmission Polling Controlled

Receiving Frames

The `CAN_STATUS_RXB` bit of the `CAN_STATUS` register reflects the `Receive Buffer` state either `full` or `empty`. When the `receivebyPolling` method is called and the `Receive Buffer`

status is `full`; the message are read and handled by the `handleMessages` function. After reading the message from the `Receive Buffer`; the `Receive Buffer` is set free by the `Release Receive Buffer Command`.

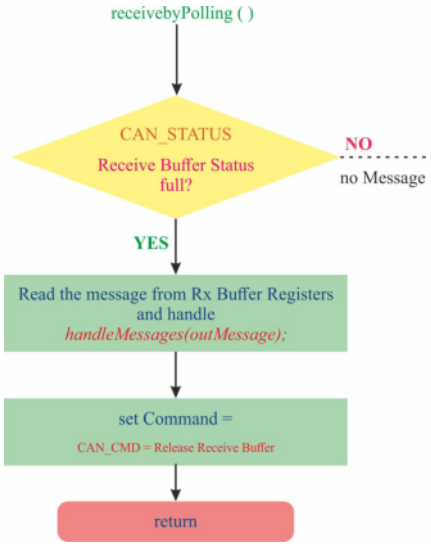


Figure. 4.6 Reception Polling Controlled

4.4.2 Interrupt Controlled Process

The message are controlled by handling the interrupt. The Interrupt has to be enabled and cleared before starting the process.

Interrupt Allocation

Interrupt can be allocated for certain peripheral in ESP32 by setting `esp_intr_alloc`. An applicable interrupt for the peripheral is found and allocated by the interrupt multiplexer. Interrupt handler and Interrupt Service Routine are installed to the required peripheral. There are two different kinds of interrupt that the function can handle; `shared Interrupt` and `non-shared Interrupt`. The CAN peripheral utilizes the non-shared interrupt; with a separate interrupt allocated with only one ISR being called.

Interrupts generated by CAN peripheral are external interrupts; as CAN peripheral is outside the CPU core. The external peripheral source are defined in `soc.h`; `ETS_CAN_INTR_SOURCE` is defined for CAN peripheral and is hardware specific that cannot be changed. As ESP32 comprise of dual core, the external interrupt slots on both the core are wired to an interrupt multiplexer, that can route the interrupt to any slot.

- Allocating an external interrupt will always allocate it on the core that does the allocation.
- Freeing an external interrupt must always happen on the same core it was allocated on.
- Disabling and enabling external interrupts from another core is allowed.

- Multiple external interrupt sources can share an interrupt slot by passing ESP_INTR_FLAG _SHARED as a flag to esp_intr_alloc().

Interrupt handling by our CAN driver utilizes single core. On further implementation of multiple task handling in the driver, interrupt can be handle by dual core by allocating the task by using xTaskCreatePinnedToCore() with specific core ID.

The function esp_intr_alloc() is defined in esp_intr_alloc.h header [24].

```
esp_err_t esp_intr_alloc(int source, int flags, intr_handler_t
handler, void *arg, intr_handle_t *ret_handle)
```

The interrupt will always be allocated on the core that runs this function. This finds an interrupt that matches the restrictions as given in the flags parameter, maps the given interrupt source to it and hooks up the given interrupt handler (with optional argument) as well. If needed, it can return a handle for the interrupt as well.

esp_intr_alloc calling in the CAN driver.

```
esp_intr_alloc(ETS_CAN_INTR_SOURCE, 0, isr, this, NULL);
```

- source : ETS_CAN_INTR_SOURCE, it is the interrupt source.
- flags : 0, it will default to allocating a non-shared interrupt of level 1, 2 or 3. If this is ESP_INTR_FLAG_SHARED, it will allocate a shared interrupt of level 1. Setting ESP_INTR_FLAG_INTRDISABLED will return from this function with the interrupt disabled.
- handler : isr, the interrupt handler defined in the driver. Must be NULL when an interrupt of level >3 is requested, because these types of interrupts are not C-callable.
- arg : this, when argument passed to the interrupt handler, NULL when no arguments handled.
- ret_handle : NULL, no handle is required; pointer to an intr_handle_t to store a handle that can later be used to request details or free the interrupt.

The interrupt handler of the CAN driver is declared as the static isr() method. The method handles the Transmit interrupt and Receive interrupt triggered in the interrupt register.

In ESP32 the interrupt has to be handled quickly [25], otherwise it results in Guru Meditation Error Figure 4.7. Error occurs even when the interrupts are unhandled. The interrupts are handled in a short loop to handle it fast. Interrupt handling is protected by the entering critical section. As ESP32 derives the Free-RTOS, taskENTER_CRITICAL() of FREE-RTOS is deprecated as portENTER_CRITICAL() in ESP32 [26]. To enter the critical section, portENTER_CRITICAL() and to exit critical section portEXIT_CRITICAL() are declared.

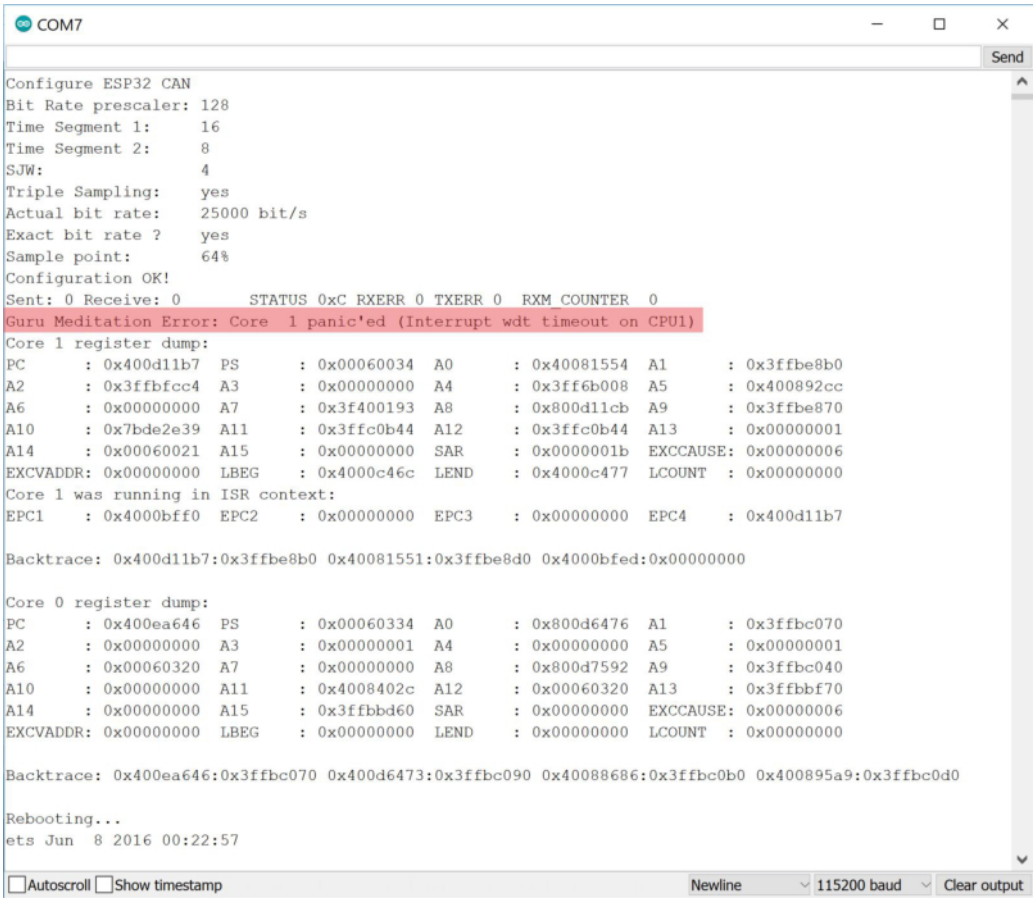


Figure. 4.7 Interrupt Handling Error

Transmission Interrupt Controlled

The `tryToSend` method is called for sending the message in the CAN network. The `Transmit Buffer` state is locked while writing a message. Writing a new message into the `Transmit Buffer` is not possible and the message will be lost without any indication. `Driver Transmit Buffer`, a temporary storage memory is created to store the message when `Transmit Buffer` is locked. The `DriverSending` flag is set when the `Transmit Buffer` is locked and the message are written in to `Driver Transmit Buffer`. The message in the `Driver Transmit Buffer` will be handled by the Interrupt Service Routine which is initiated at the end of current transmission.

Upon the reception of `Transmission Interrupt`, the interrupt handler `handleTXInterrupt` checks the message in `Driver Transmit Buffer` and removes the message to write in to the `Transmit Buffer`.

Receive Interrupt Controlled

When `Receive Interrupt` is generated the `handleRXInterrupt` interrupt handler reads the message from the `RX FIFO` and appends in to the `Driver Receive Buffer`.

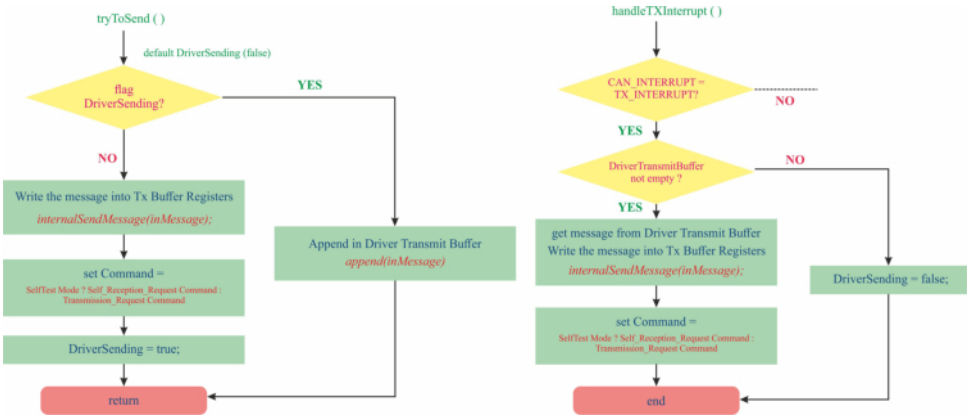


Figure. 4.8 Transmission Interrupt Controlled

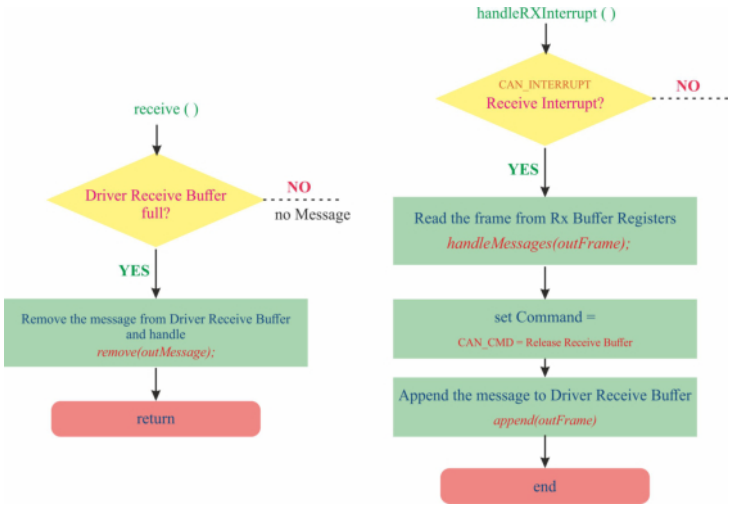


Figure. 4.9 Reception Interrupt Controlled

4.5 Driver Send and Receive method

This section details the `ESP32ACAN` Driver Sending and Receiving CAN frame methods.

4.5.1 Transmission Method

The driver contains two methods `tryToSendbyPolling` and `tryToSend` for controlling the transmission process.

The `tryToSendbyPolling` method is called when the sequential process is controlled by polling. The Transmit Buffer status is checked; if `released` the message frame is sent to the Transmit Buffer through `internalSendMessage` function.

The `tryToSendbyPolling` method returns:

- `true` if the message has been successfully transmitted to the transmit buffer; note. the message is not actually sent.
- `false` if the Transmit buffer is locked; the message is not successfully transmitted to the transmit buffer.

The `tryToSend` method is called for transmitting the message frame by handling the Interrupt and storing the message in Driver Transmit Buffer when the Transmit Buffer status is locked. `mDriverSending` flag is checked; if true append the message into the Driver Transmit Buffer, else write message frame into the Transmit Buffer. By default, `mDriverSending` flag is `false`.

The `tryToSend` method returns:

- `true` if the message has been successfully transmitted to the driver transmit buffer; note. the message is not actually sent.
- `false` if the message is not successfully transmitted to the driver transmit buffer.

Note

In Last driver update the `tryToSendbyPolling` method is called within the `tryToSend` method for ease of driver implementation. `mSendbyPoll` flag in driver is set `true` by `mControllMessageByMethod` property in the driver initialization `begin` method. The `tryToSend` method checks the state of the `mSendbyPoll` flag; if true : `tryToSendbyPolling` is called, else send by interrupt handling.

The `internalSendMessage` function writes the message frame into the `Transmit Buffer` register.

The properties of CAN message; `dlc`, `rtr`, `frame format` are written into the `CAN_FRAME_INFO` register. The CAN frame length is constrained to 8 byte; if length is greater than 8, data length code is fixed to 8. RTR bit is set if the frame is Remote Frame. The frame format is set by the `ext` property of CAN Message; if `true`, Extended Frame Format, else Standard Frame Format.

Next identifier register are written with the 32bit CAN message ID. The number of Identifier register for Standard Frame Format is 2 and Extended Frame Format is 4. The register are written by bit shifting.

```
//--- Set ID - Standard Frame Format
CAN_ID_SFF(0)  = (uint8_t)((inFrame.id) >> 3) ;
CAN_ID_SFF(1)  = (uint8_t)((inFrame.id) << 5) ;
```

```
//--- Set ID - Extended Frame Format
CAN_ID_EFF(0) = (uint8_t)((inFrame.id) >> 21);
CAN_ID_EFF(1) = (uint8_t)((inFrame.id) >> 13);
CAN_ID_EFF(2) = (uint8_t)((inFrame.id) >> 5);
CAN_ID_EFF(3) = (uint8_t)((inFrame.id) << 3);
```

The CAN message data field is written into the corresponding Data field registers: `CAN_DATA_SFF` -standard frame format and `CAN_DATA_EFF`-extended frame format.

After writing the Transmit Buffer register the command register is set with `CAN_CMD_SELF_RX_REQ` bit; in `LoopBackMode` and `CAN_CMD_TX_REQ` bit in Normal operating mode

```
CAN_CMD = ((CAN_MODE & CAN_MODE_SELFTEST) !=0)?
   CAN_CMD_SELF_RX_REQ : CAN_CMD_TX_REQ;
```

Driver Transmit Buffer

The Driver Transmit Buffer size is defined with in the `ESP32ACANSettings` class. By default the size is 16. The value can be changed by setting the `mDriverTransmitBufferSize` properties of the settings variable; for example:

```
ESP32ACANSettings settings (DESIRED_BIT_RATE) ;
settings.mDriverTransmitBufferSize = 32 ;
const uint16_t errorcode = can.begin (settings) ;
```

`driverTransmitBufferSize` **method** The method returns the allocated size of the driver transmit buffer.

```
const uint16_t buffersize = can.driverTransmitBufferSize ;
```

`driverTransmitBufferCount` **method** The method returns the current number of message in the driver transmit buffer.

```
const uint16_t buffermsgcount = can.driverTransmitBufferCount ;
```

`driverTransmitBufferPeakCount` **method** The method returns the peak value of message count in the driver transmit buffer.

```
const uint16_t msgpcount = can.driverTransmitBufferPeakCount ;
```

4.5.2 Receive Method

The `receivebyPollinging` and `receive` method are used for receiving CAN message from the Receive Buffer.

The `receivebyPolling` method returns:

- `true` if the message has been removed from the Receive buffer and the Receive buffer is released.
- `false` if the Receive buffer is empty; No message.

The `receive` method returns:

- `true` : if a message has been removed from the `Driver Receive Buffer`, and the message argument is assigned.
- `false` : if the `Driver Receive Buffer` is empty, message argument is not modified.

Note

The final driver update uses `receive` method for both polling and interrupt control process. The `receivebyPolling` method is called within the `receive` method, if `mSendbyPoll` flag is set `true`. The `mSendbyPoll` flag is set in the initialization `begin` method with the `mControllMessageByMethod` property.

The message in the Receive Buffer are handled by the `handleMessages` function. The CAN message frame in the Receive Buffer is retrieved.

After handling the message the Receive Buffer is released by setting the Release Receive Buffer bit with command register.

Driver Receive Buffer

The default size of the driver receive buffer is 32, defined in the `ESP32ACANSettings` class. This value can be changed by setting the `mDriverTransmitBufferSize` property before calling the begin method:

```
ESP32ACANSettings settings (DESIRED_BIT_RATE) ;
settings.mDriverReceiveBufferSize = 100 ;
const uint16_t errorcode = can.begin (settings) ;
```

As the size of CANMessage class is 16 bytes, the actual size of the driver receive buffer is the value of `settings.mReceiveBufferSize * 16`.

`driverReceiveBufferSize` **method**

The method returns the allocated size of the driver receive buffer.

```
const uint16_t buffersize = can.driverReceiveBufferSize ;
```

`driverReceiveBufferCount` **method**

The method returns the current number of message in the driver receive buffer.

```
const uint16_t buffermsgcount = can.driverReceiveBufferCount ;
```

`driverReceiveBufferPeakCount` **method** The method returns the peak value of message count in the driver receive buffer.

```
const uint16_t msgpcount = can.driverReceiveBufferPeakCount ;
```

4.5.3 The Driver Error Code

The `ESP32ACAN::begin` method return an error code. The returned value is 0, if no error. The `ESP32ACAN` class defines the static constant error properties.

Bit	Error Name	Error
0	`kNotInRestModeInConfiguration`	CAN mode is not in Reset Mode during the CAN controller configuration.
1	`kCANRegistersError`	Error writing the Bus timing register. CAN register not accessible
2	`kTooFarFromDesiredBitRate`	When `mBitRateClosedToDesiredRate` property of the settings object is `false`
3	`kInconsistentBitRateSettings`	When CAN bit properties values are inconsistent
4	`kCannotAllocateDriverReceiveBuffer`	When not enough RAM left to allocate the receive buffer
5	`kCannotAllocateDriverTransmitBuffer`	When not enough RAM left to allocate the transmit buffer

Table 4.3 The `ESP32ACANSettings::begin` method error codes

4.6 CANMessage class

CAN Message frame contains an identifier, frame information, frame length, data. The CAN-Message class defines the CAN frame for the driver to work with the CAN Message. The class object contains all the CAN frame user information. By default the frame is standard data frame with identifier equal to 0 and without any data.

Note the class declaration is protected by include guard that causes the macro `GENERIC_CAN_MESSAGE_DEFINED` to be defined. The `CANMessage.h` is identical for the `ACAN` driver, `ACAN2515` driver, `ACAN2517` driver (idx member is not used by the ESP32 CAN driver, as Compatibility with the ACAN2515 and ACAN2517 library it is necessary to have the field).

```
#ifndef GENERIC_CAN_MESSAGE_DEFINED
#define GENERIC_CAN_MESSAGE_DEFINED

class CANMessage {
  public : uint32_t id = 0;  // Frame identifier
  // false -> standard frame, true -> extended frame
  public : bool ext = false ;
  // false -> data frame, true -> remote frame
  public : bool rtr = false ;
  public : uint8_t idx = 0 ;
  public : uint8_t len = 0 ;  // data length code (0 ... 8)
```

```
  public : union {
    uint64_t data64     ; // Caution: subject to endianness
    uint32_t data32 [2] ; // Caution: subject to endianness
    uint16_t data16 [4] ; // Caution: subject to endianness
    uint8_t  data   [8] = {0, 0, 0, 0, 0, 0, 0, 0} ;
  } ;
} ;
```

4.7 Acceptance Filter: ESP32ACANFilter class

ESP32ACANFilter class is defined in ESP32AcceptanceFilter.h file. For the ESP32 acceptance filter registers and operation (See Section 3.4).

The CAN_ACC_CODE_FILTER(idx) for Code and CAN_ACC_MASK_FILTER(idx) for Mask register are defined in the ESP32CANRegisters.h file.

By default the can.begin(settings) receive all message by default filter settings.

Four functions are defined to accept standard or extended frame in single or dual filter mode.

1. acceptSingleFilterStandard and acceptSingleFilterExtended for single filter mode.
2. acceptDualFilterStandard and acceptDualFilterExtended for dual filter mode.

acceptSingleFilterStandard function takes the standard code and mask id with the data byte1 and data byte2. For instance, the RTR bit is not handled by the functions.

The filter settings are written into the registers by setAcceptanceFilter method called inside the configuration method.

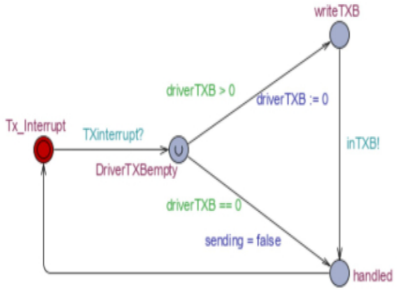
For Single filter mode, CAN_MODE_ACCFILTER bit must be set to logic 1. This action is performed by the true state of ESP32ACANFilter class member mAMFSingle. By default mAMFSingle property is false (dual acceptance filter mode).

4.8 Driver Model Verification

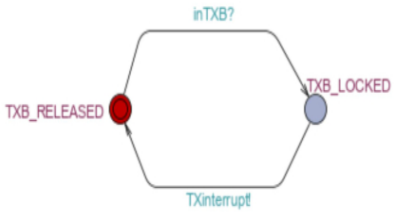
This section exhibits a demo test of the driver transmit sequence by transmit interrupt.

Three models are created in UPPAAL tool, to depict the transmit function, transmit interrupt handler and transmit buffer of the driver shown in Figure 4.10.

(a) Transmit Model



(b) Transmit Interrupt handler



(c) Transmit Buffer Model

Figure. 4.10 Driver Transmit Sequence Model

`Tx interrupt` is generated when the `Transmit Buffer` status changes from `locked` state to `release` (i.e., 0 to 1), which is an edge action. The `TXinterrupt` is declared as a channel to synchronize the edge action. Other flags are declared; `sending` flag for driver sending state, by default its false. A driver transmit buffer of size 16 this can be change in the `declaration`.

The properties that are verified with this model:

- Is the system deadlock free?
  `A[] not deadlock`
  Result : `Satisfied` no deadlock.
- Is there any situation in which the `WriteTXB` is possible when the Transmit Buffer is locked (`TXB_LOCKED`)
  `E<> tryTosend.writeTXB and TX_Buffer.TXB_LOCKED`
  Result : `Not satisfied`, There is no possibility to write TXB when in locked state.

Furthermore test can be conducted on the model. It is concluded with the satisfying result. Bus model with some CAN nodes can be created for verification of the sequence process in the CAN network for detailed understanding of the CAN protocol.

Chapter 5

ESP32 CAN Driver Examples

The configuration and operation of ESP32 CAN driver are demonstrated with the example code.

File Name	Function
LoopBackCheck-Polling.ino	ESP32 CAN Self-Test by Polling Controlled
LoopBackCheck-Interrupt.ino	ESP32 CAN Self-Test by Interrupt Controlled
LoopBackCheck-PollingIntensive.ino	Self-Test by Polling Intensive Test
LoopBackCheck-InterruptIntensive.ino	Self-Test by Interrupt Intensive Test
ESP32CANTestWithACAN2515.ino	ESP32 CAN test with ACAN215 in Normal mode.

The Example codes are included in the ESP32ACAN library. The codes are compiled using Arduino IDE platform on ESP32 microcontroller board.

The common and mandatory declaration of the driver instance for all the example code are detailed below[1].

```
static const uint32_t DESIRED_BIT_RATE = 125UL * 1000UL ;
// 125 kb/s
```

Declaration of the desired CAN bit rate. The static constant variable can be initialized with all the valid CAN bit rate settings computed from the TestOnDesktop.

```
ESP32ACAN can ;
```

Definition of the ESP32ACAN library object instance; can.

The configuration of the ESP32ACAN driver are paced into five steps within the setup function.

```
void setup() {
....
  // CAN bit rate settings
  ESP32ACANSettings settings (DESIRED_BIT_RATE);
```

[1]Subject to change settings for respective examples. It is described in the respective example section

Step 1. Instantiation of the ESP32ACANSettings class object; settings. The constructor returns a settings object with all the CAN bit settings for the DESIRED_BIT_RATE parameter passed, and default value for other driver configuration properties.

```
  // Select operating mode
  settings.mRequestedCANMode = ESP32ACANSettings::LoopBackMode;
```

Step 2. The configuration of the CAN operating mode is set by the mRequestedCANMode property of the settings object. By default the CAN operating mode is Normal mode. This property can be overridden with the required operation mode.

Enum Variable	Function
NormalMode	ESP32 CAN connection with the physcial CAN network.
ListenOnlyMode	ESP32 CAN only receive messages.
LoopBackMode	ESP32 CAN Self Test.

Table 5.1 ESP32 CAN Operating Modes

```
    // Select Message Control Process
    settings.mControlMessageByMethod =
    ESP32ACANSettings::InterruptControlled;
```

Step 3. The message processing control in the CAN Driver is defined by the mControlMessage ByMethod property. The process can be either by Polling controlled method or by Interrupt controlled method.

Method	Operation
PollingControlled	Transmission and Reception by checking the status flag
InterruptControlled	Transmission and Reception by Interrupt Service Routine

Table 5.2 ESP32 CAN Message Processing Method

```
  // Driver Configuration
  const uint16_t errorCode = can.begin (settings) ;
}
```

Step 4. Configuration of the ESP32ACAN driver object can with the settings value. The begin method sets the initial configuration of the ESP32 CAN controller and installs the ISR; if the control process is by Interrupt.

```
   if (errorCode == 0) {
    Serial.print ("Bit Rate prescaler: ") ;
    Serial.println (settings.mBitRatePrescaler) ;
    Serial.print ("Time Segment 1:     ") ;
    Serial.println (settings.mTimeSegment1) ;
```

```
    Serial.print ("Time Segment 2:      ") ;
    Serial.println (settings.mTimeSegment2) ;
    Serial.print ("SJW:                 ") ;
    Serial.println (settings.mSJW) ;
    Serial.print ("Triple Sampling:     ") ;
    Serial.println (settings.mTripleSampling ? "yes" : "no") ;
    Serial.print ("Actual bit rate:     ") ;
    Serial.print (settings.actualBitRate ()) ;
    Serial.println (" bit/s") ;
    Serial.print ("Exact bit rate ?     ") ;
    Serial.println (settings.exactBitRate () ? "yes" : "no") ;
    Serial.print ("Sample point:        ") ;
    Serial.print (settings.samplePointFromBitStart ()) ;
    Serial.println ("%") ;
    Serial.println ("Configuration OK!");
  }else {
    Serial.print ("Configuration error 0x") ;
    Serial.println (errorCode, HEX) ;
  }
}
}
```

Step 5. Final step the `ESP32ACAN` driver configuration returns error code; stored in the `errorCode` constant. It returns 0; on successful configuration. The values of the CAN bit timing segments are printed in the serial monitor. If returned with a configuration error; the error `HEX` value is printed.

The sequential operation of the CAN message by the CAN controller are succeeded by `tryToSend` and `receive` method of the `ESP32ACAN` driver.

```
static uint32_t gReceivedFrameCount = 0 ;
static uint32_t gSentFrameCount = 0 ;
```

The global variables are declared to note a successful transmission of message as the driver method returns before the message begin actually sent. `gSentFrameCount` counts the number of sent messages and `gReceivedFrameCount` counts the number of received messages.

The CAN message send and receive operation are sequential. Therefore the driver Transmission method and Receiving method are declared inside the `loop` function.

```
void loop() {
  CANMessage frame ;
```

The `CANMessage` class instance `frame` object is fully initialized by the default constructor; Standard Data Frame with identifier equal to 0, and without any data.

```
    const bool ok = can.tryToSend (frame) ;
```

```
    if (ok) {
      gSentFrameCount += 1 ;
      Serial.print ("Sent: ") ;
      Serial.println (gSentFrameCount) ;
    }else{
      Serial.println ("Send failure");
    }
```

The `tryToSend` method returns `true` or `false`; on successful transmission of the CAN message to the Transmit Buffer. It does not tell that the CAN message is actually sent on the CAN network. On each successful transmission the `gSentFrameCount` is incremented and printed in the serial monitor. If the transmission fail, the `gSentFrameCount` is not changed and `Send failure` string is printed on the serial monitor.

```
  if (can.receive(frame)) {
    gReceivedFrameCount += 1 ;
    Serial.print ("Received: ") ;
    Serial.println (gReceivedFrameCount) ;
  }
```

The `receive` method returns `true` if the message in the receive buffer is removed successfully and returns `false` if no message in the receive buffer. If a message has been received, the `gReceivedFrameCount` is incremented and printed on the monitor.

For transmitting and receiving Extended Frame `ext` property of the CANMessage class is set to true. By default the driver transmit Standard Frame.

```
void loop {
    CANMessage message;
    message.ext = true;
}
```

The general example code structure is;

```
/*---------- Board Check -----------*/
#ifndef ARDUINO_ARCH_ESP32
  #error "Select an ESP32 board"
#endif

/*---------- Include files ---------*/
#include "ESP32ACAN.h"

//----  ESP32 Desired Bit Rate definition
....

//----  ESP32 CAN Driver object instance
```

```
....

void setup() {
 //--- Switch on builtin led
  pinMode (LED_BUILTIN, OUTPUT) ;
  digitalWrite (LED_BUILTIN, HIGH) ;
//--- Start serial
  Serial.begin (115200) ;
//--- Wait for serial (blink led at 10 Hz during waiting)
  while (!Serial) {
    delay (50) ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
  }
//--- Configure ESP32 CAN
  ....
}

//--------------global Variable -------------------------
static uint32_t gBlinkLedDate = 0;
static uint32_t gReceivedFrameCount = 0;
static uint32_t gSentFrameCount = 0;
//-----------------------------------------------------------

void loop() {
//---CANMessage Class instance
CANMessage Frame;
  if (gBlinkLedDate < millis ()) {
    gBlinkLedDate += 2000 ;
    digitalWrite (LED_BUILTIN, !digitalRead (LED_BUILTIN)) ;
    //--- Call tryToSend function
    ....
  }
 //--- Call receive function
 ....
}
```

The example codes are compatible for ESP32 microcontroller; Check board before compilation. The ESP32 built-in Led is switched on to monitor the serial wait period and blink on CAN message sequence every 2 s; By calling the `tryToSend` method inside `gBlinkLedDate` loop function. The `receive` method is called outside the loop, as the Receive message has higher priority; message in the Receive buffer is removed first.

5.1 LoopBackCheck-Polling

The Figure 5.1 shows the connection of the ESP32 microcontroller pin setup. It is mandatory to connect the `Tx` pin with the `Rx` pin. In this case the default pins settings are used. Note. Only transmission occurs and reception fails if the pins are unconnected.

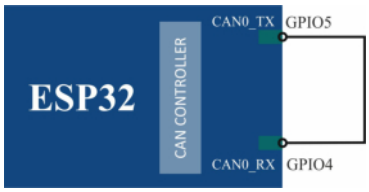


Figure. 5.1 LoopBackCheck Connection

`LoopBackCheck-Polling.ino` file performs a Self-Test of the ESP32 CAN Controller by message polling process. It checks the status of the `Transmit Buffer` and `Receive Buffer` flag in the `Status Register`.

```
  settings.mRequestedCANMode = ESP32ACANSettings::LoopBackMode;
        settings.mControllMessageByMethod = ESP32ACANSettings
::PollingControlled;
```

Setting the CAN operation to `LoopBackMode` is mandatory as the driver default operating mode is defined as `NormalMode`. The message control method by default is set to `PollingControlled`.

The test is performed for all the valid CAN bit rate settings. Figure 5.2 shows the output for CAN bit rate of 1 Mbit/s. On every flash of the ESP32 board; the Boot factors are displayed, if error.

The `Sent` count and `Received` count are incremented on each successful return of the corresponding sequential action. Note. the resulting does not mean that the CAN message are successful in the CAN network.

5.2 LoopBackCheck - Interrupt

`LoopBackCheck-Interrupt.ino` file implements transmission and reception using Interrupt Service Routine.

```
settings.mControlMessageByMethod = ESP32ACANSettings::
   InterruptControlled;
```

By setting the `mControlMessageByMethod` property to `InterruptControlled` installs the `ISR` in the Driver configuration. The `tryToSend` method uses the driver transmit buffer size of 16 and `receive` method uses the driver receive buffer size of 32. The edge controlled Interrupts are handled in the ISR.

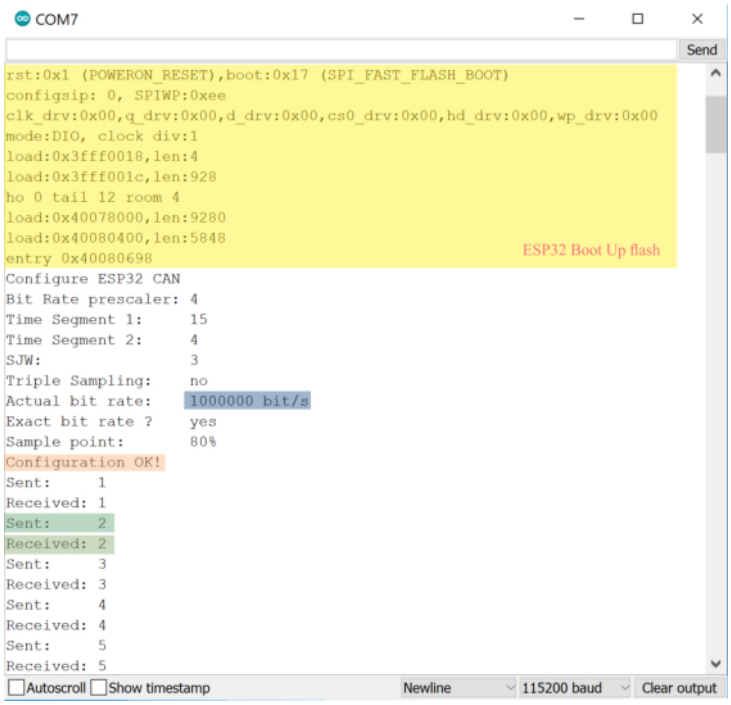
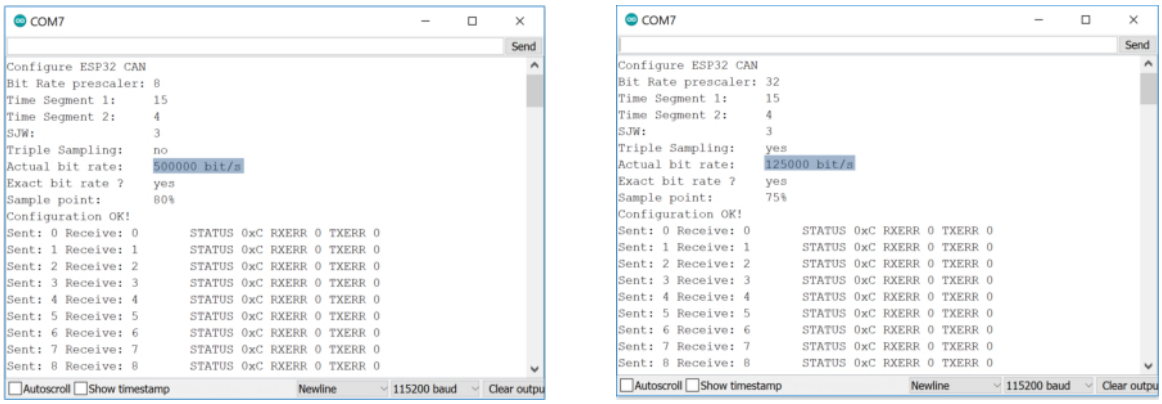


Figure. 5.2 Output: LoopBackCheck-Polling



(a) CAN bit rate 500 kbit/s (b) CAN bit rate 125 kbit/s

Figure. 5.3 Output: LoopBackCheck-Interrupt

Figure 5.3 shows the test result for the CAN bit rate 500 kbit/s and 125 kbit/s. The value of the `Status` register and the `Error counter` register value are printed to check the cause of error during sequential process. The Status register value read is `0xC`; bit `CAN_STATUS_TX_COMPLETE`, `CAN_STATUS_TXB`. The previous message transmission is completed and the transmit buffer status is released to write a new message. If any error occurs during the Transmission and Receive sequence the value are indicated in the Error Counters. If successful, the error counter value is 0.

5.3 LoopBackCheck - Intensive

The previous example sequence sends the message and receive message before transmission of next message. The Intensive example uses a message count variable with the number of message (in 1000s) to be transmitted. By the intensive check performance of the CAN controller at the desired bit rate can be checked effectively. The total message count sent must be received, failing results in loss of message.

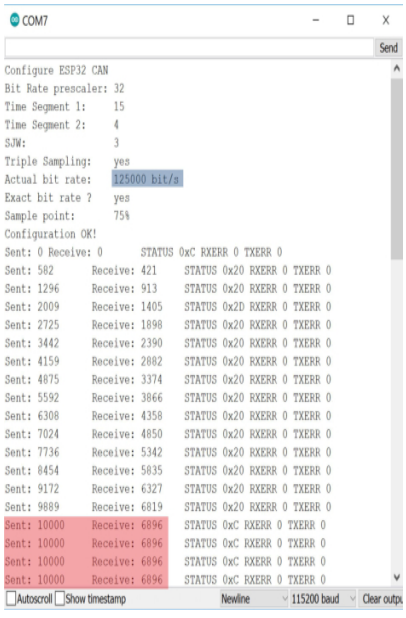
```
static const uint32_t MESSAGE_COUNT = 10 * 1000;
```

The `tryToSend` method is called until the `gSentFrameCount` does not exceed the `MESSAGE_COUNT`.

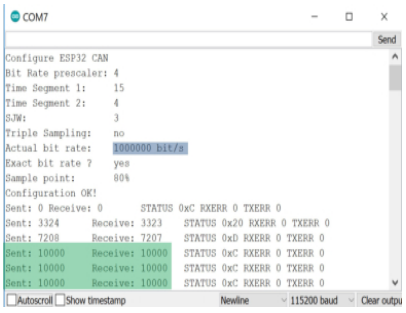
```
if (gSentFrameCount < MESSAGE_COUNT) {
//--- call tryToSend method
....
}
```

5.3.1 Intensive check by Polling

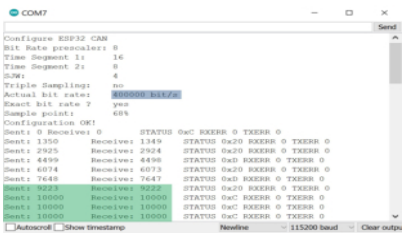
`LoopBackCheck-PollingIntensive.ino` file test the process by the Polling of Status register flag. The 10,000 messages are tested by the example code. The message count value can be changed as required.



(a) CAN bit rate 125 kbit/s



(b) CAN bit rate 1 Mbit/s



(c) CAN bit rate 400 kbit/s

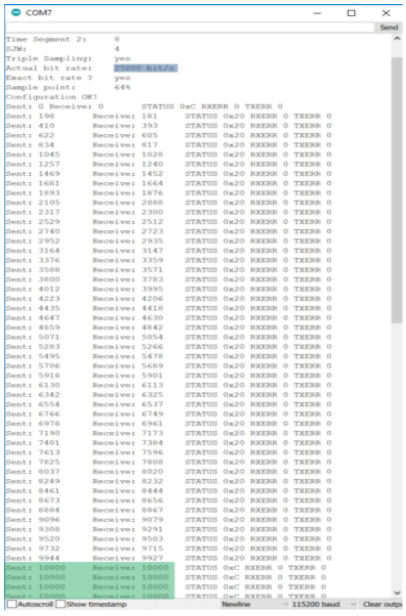
Figure. 5.4 Output: LoopBackCheck-PollingIntensive

This demo works well for the Higher CAN bit rates. When operated with slow CAN bit rates below 125 kbit/s, the controller does not receive the complete message count sent. In Figure 5.4

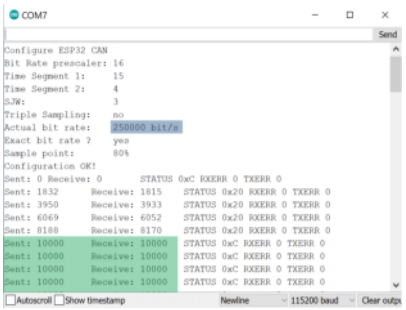
(a) the sent message count completes but the receive count stops at 6896 message count. This failure is unpredictable as slow CAN bit rate works well in all other conditions.

5.3.2 Intensive check with Interrupt

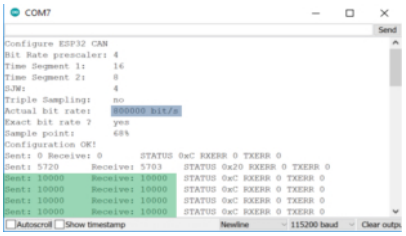
`LoopBackCheck-InterruptIntensive.ino` code runs intensive check in Self-Test mode by handling the interrupt. The interrupt source is allocated and handled by the `ISR` static member function of the `ESP32ACAN` driver. It handles the receive interrupt and transmit interrupt.



(a) CAN bit rate 25 kbit/s



(b) CAN bit rate 250 kbit/s



(c) CAN bit rate 800 kbit/s

Figure. 5.5 Output: LoopBackCheck-InterruptIntensive

All the valid CAN bit rates have been tested with this example and have returned a successful result. The total message count 10,000, is successfully transmitted and received by the ESP32 CAN module. The count values are printed in serial monitor every 5 ms, to verify the sequence of the control. `CAN_STATUS` register flags the `TX_STATUS` bit indicating the controller transmission state. After completion of transmission and reception of the 10,000 messages, `CAN_STATUS` register flags the `TX_COMPLETE_STATUS` and `TX_BUFFER_STATUS` released. There are no errors countered in the phase transmission and reception for all valid CAN bit rate.

5.4 Normal Operation Mode

Previous tests were performed in Self-Test mode which are concluded to be working efficient for the cases except slow CAN bit rate in polling method. Next test in controller normal mode is performed with `MCP2515` and `MCP2517` can controller in the ESP32 MiniKit development board.

5.4.1 Test with MCP2515 CAN controller

Both ESP32 CAN and MCP2515 CAN controller are connected with MCP2562 CAN Transceiver. The development board is built with two CAN port for each CAN controller. Each CAN port lines; `CAN-HIGH` and `CAN-LOW` lines are terminated with a 120Ω resistor by turning on the switch. For ESP32 CAN test with MCP2515, any one of the corresponding CAN ports are linked. The ACAN2515 library created by Prof. Pierre Molinaro can be downloaded from https://github.com/pierremolinaro/acan2515.

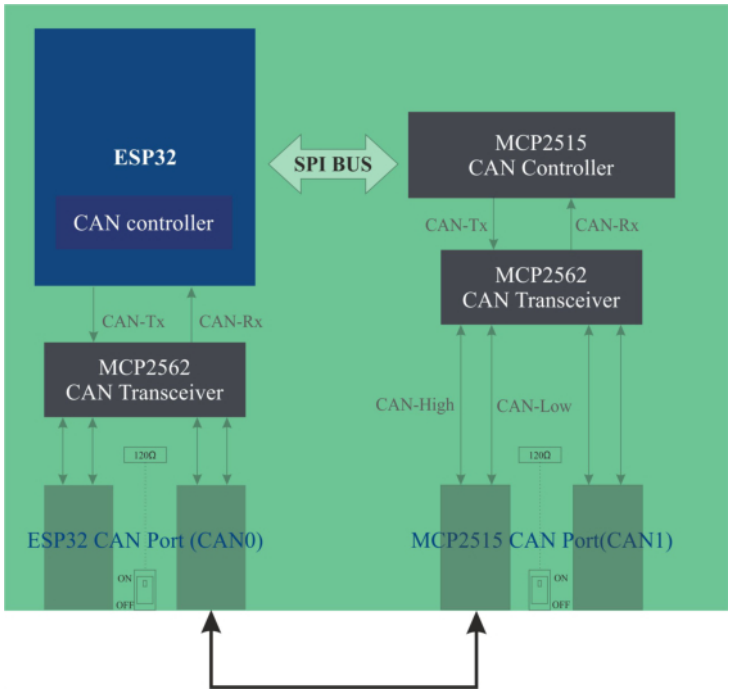


Figure. 5.6 ESP32 CAN and MCP2515 Test Setup

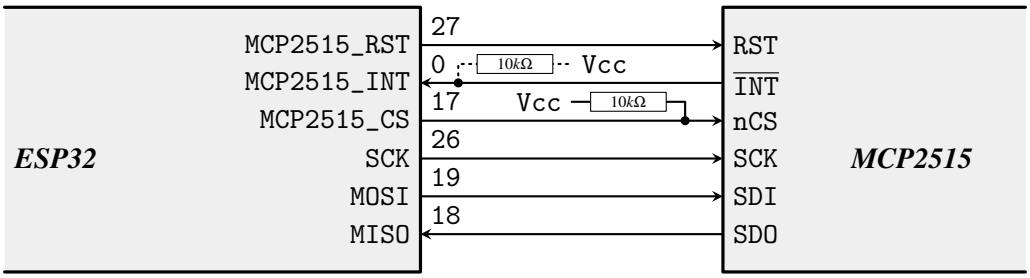


Figure. 5.7 MCP2515 Connection with ESP32 using VSPI pins

The MCP2515 CAN controller is connected to the ESPS32 through SPI bus. The pins are configured by `SPI.begin`

```
SPI.begin (MCP_SCK, MCP_SDO, MCP_SDI) ;
```

For the object declaration and property settings of the ACAN2515 driver see the document https://github.com/pierremolinaro/acan2515/blob/master/extras/acan2515.pdf

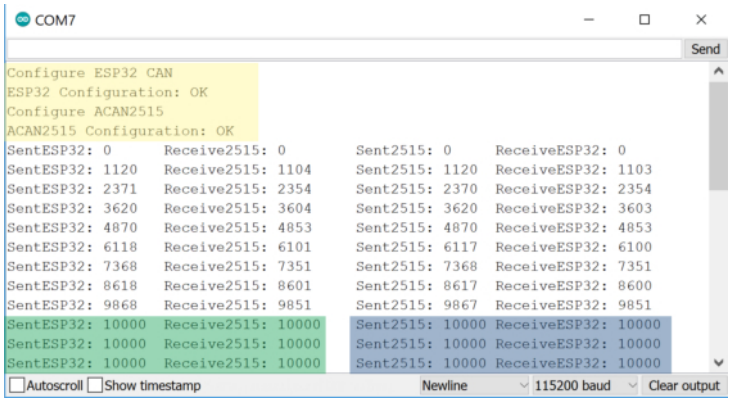


Figure. 5.8 Test with MCP2515 bit rate 625 kbit/s

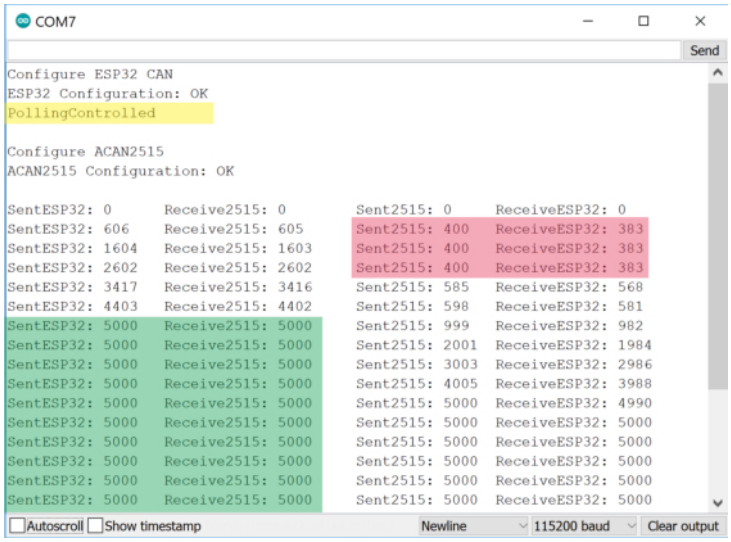


Figure. 5.9 Test with MCP2515 bit rate 25 kbit/s

Figure 5.8, shows the test result for CAN bit rate 625 kbit/s. All the message sent by the ESP32 CAN are received by MCP2515 and vice versa. The transmission and reception are successful. Figure 5.9, a message count of 5000 is transmitted at CAN bit rate 25 kbit/s. The red highlighted column shows the MCP2515 stops sending message at 400 count. This is caused because of the arbitration of message sent by ESP32 to MCP2515. After a period the MCP2515 continues sending the message and finally all the sent message are received.

5.5 Acceptance Filter Settings

Acceptance Filter settings can be set for the receiver to accept the particular message with the ID required. For details of the ESP32 CAN acceptance filter See section 3.4.

`ESP32CANFilterSettings.ino` example source code details working with the acceptance filter function.

The acceptance filter settings function parameters:

- `acceptSingleFilterStandard (Code ID, Code byte1, Code byte2, Mask ID, Mask byte1, Mask byte2)`
- `acceptSingleFilterExtended (Code ID, Mask ID)`
- `acceptDualFilterStandard (Code ID1, Code ID2, Code byte1, Mask ID1, Mask ID2, Mask byte1)`
- `acceptDualFilterExtended (Code ID1, Code ID2, Mask ID1, Mask ID2)`

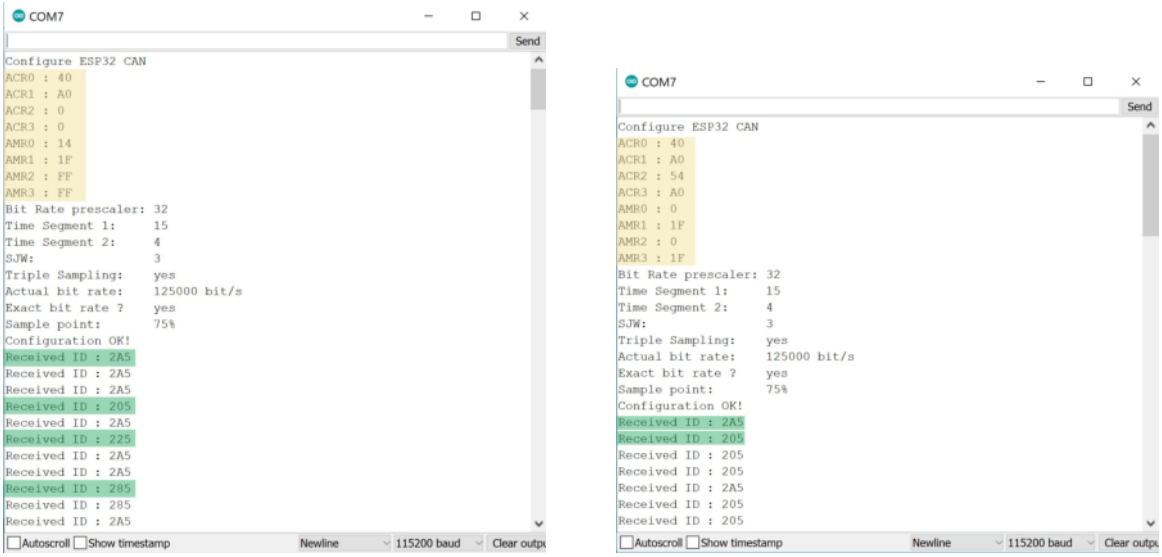
```
const ESP32ACANFilter filter = acceptSingleFilterStandard(0
    x205, 0, 0, 0x0A0, 0xFF, 0xFF); // Single Filter

  //const ESP32ACANFilter filter = acceptDualFilterStandard(0
     x205,0x2A5,0,0x000,0x000,0); // Dual Filter

  const uint16_t errorCode = can.begin(settings, filter);
```

The `ESP32ACANFilter` class instance object `filter` passed with the function containing the Code and Mask register parameters.

This example code uses the two message ID used in the Section 3.4. Message ID1 : 0x205 and Message ID2 : 0x2A5. The message ID are standard frame format, and they are tested for both dual and single filter settings with the respective function.



(a) Single Filter Standard Format Frame

(b) Dual Filter Standard Format Frame

Figure. 5.10 Output: Acceptance Filter Settings

In Figure 5.10: (a), Acceptance filter settings using Single Filter Mode. The Code and Mask register value are displayed for verification purpose. The message ID accepted by the receiver are `0x205, 0x2A5, 0x285, 0x225`. In single filter mode the required two message IDs are not decoded properly, therefore for better result Dual Filter mode is used. The output for dual filter mode in Figure 5.10: (b); the accepted message IDs are *0x205, 0x2A5*.

Chapter 6

Conclusion

The basics of CAN protocol is well documented in this report. The ESP32 CAN peripheral registers are studied from NXP SJA1000 CAN controller and reproduced for the ESP32 compatibility, and this is also documented.

The project has covered the implementation of the Driver Development from beginning till end. The driver features are easy to adapt and configured for the corresponding applications.

The driver features that improvise the CAN Controller are:

1. efficient CAN-bit settings computation from user bit rate.
2. easy definition of acceptance filter settings.
3. Any mode selection; Normal, Self-Test, ListenOnly.
4. Both Standard and Extended CAN message frame can be handled.

These features are well organized in the driver and are tested with the sample codes of the library.

From the test output, it is concluded that the ESP32 CAN controller exhibits satisfying operation with the driver. Only the slow CAN bit rate test by Polling demo failed and the cause for the failure is unpredictable. But it is obvious that the ESP32 CAN controller works well in slow CAN bit rate settings from the Interrupt control. Complete successful performance of the ESP32 CAN Controller cannot be justified as there is one case that the controller does not satisfy.

Improvements of the driver can be done on the handling of other interrupts, multiple task operation by utilizing the two cores in ESP32. This driver CAN be adapted and modified for any ESP32 projects.

References

[1] Espressif Inc. ESP32 series datasheet, Version 3.1 2019. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[2] CAN in Automation (CiA). History of CAN technology — Cia, the international users' and manufacturers' group for the CAN network, Online [2019]. https://www.can-cia.org/de/can-knowledge/can/can-history/.

[3] Thomas Barth. ESP32-CAN-Driver. https://github.com/ThomasBarth.

[4] Wikipedia,. CAN BUS, Online [2019]. https://en.wikipedia.org/wiki/CAN_bus.

[5] CAN Specification. Bosch. *Robert Bosch GmbH, Postfach*, Version 2.0, 1991. http://esd.cs.ucr.edu/webres/can20.pdf.

[6] Keith Pazul. CAN Basics Microchip Technology. *AN713*, 2005. http://ww1.microchip.com/downloads/en/AppNotes/00713a.pdf.

[7] Florian Hartwich and Armin Bassemir. The configuration of the CAN bit timing. In *6th International CAN Conference*, pages 2–4, 1999.

[8] Meenanath Taralkar. Computation of CAN bit timing parameters simplified. In *CAN in Automation, iCC*, 2012.

[9] Heinz-Jürgen Oertel. Bit timing calculator. http://www.bittiming.can-wiki.info/#NXP.

[10] Stuart Robb, East Kilbride. CAN bit timing requirements, AN1798. https://www.nxp.com/docs/en/application-note/AN1798.pdf.

[11] Pat Richards. Understanding Microchip's CAN Module Bit Timing, AN754. http://ww1.microchip.com/downloads/en/appnotes/00754.pdf.

[12] Wikipedia. ESP32, 2016. https://en.wikipedia.org/wiki/ESP32.

[13] Espressif Inc. ESP32 Technical Reference Manual, 2018. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[14] Espressif Inc. ESP32-WROOM-32 Datasheet, Version 2.8 2019. https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[15] Gunar Schorcht. MH-ET LIVE MiniKit, RIOT - open source. https://riot-os.org/api/group__boards__esp32__mh-et-live-minikit.html.

[16] Random Tutorials. ESP32 set-up on Arduino IDE. https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/.

[17] Microchip Technology. MCP2562 High Speed CAN Transceiver, 2006. http://ww1.microchip.com/downloads/en/DeviceDoc/20005167C.pdf.

[18] Microchip Technology. MCP2515. http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf.

[19] Microchip Technology. MCP2517. http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD-External-CAN-FD-Controller-with-SPI-Interface-20005688B.pdf.

[20] Espressif Inc. About the CAN controller. https://esp32.com/viewtopic.php?t=380.

[21] Philips Semiconductors. SJA1000 Stand-alone CAN controller. https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf.

[22] Espressif Systems. ESP-IDF - CAN_STRUCT. https://github.com/espressif/esp-idf/blob/master/components/soc/esp32/include/soc/can_struct.h.

[23] Espressif Systems Forum. Peripheral Clock. https://esp32.com/viewtopic.php?t=991.

[24] Espressif Systems. Interrupt Allocation. https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/system/intr_alloc.html.

[25] Stack Overflow. Guru Meditation Error. https://stackoverflow.com/a/51751214.

[26] Espressif Systems Forum. Critical code section, 2017. https://esp32.com/viewtopic.php?t=1703.

Appendix A

ESP32 CAN Driver

The ESP32 CAN Driver is available in **GitHub**. It can be downloaded or cloned from:

https://github.com/irfanafa/ESP32ACAN.

The Downloaded Zip can be added to the Arduino IDE:

ARDUINO IDE → Sketch → Include Library → Add .Zip Library... → ESP32ACAN.zip

CAN-DRIVER v1.0

Initial Release - ESP32 CAN REGISTER DEFINITION

test-ESP32CANSettings-on-desktop - CAN Bit Timing Calculator test on desktop compiler (This can be tested on any C++ compiler)

src/ESP32CANRegisters - Defines the ESP32 CAN Registers Address.
src/ESP32ACANSettings.h - CAN Bit time calculator and Settings class.
src/ESP32ACANSettings.cpp - CAN Bit time calculator and Settings class.

examples : ARDUINO FILES

examples/ESP32CANBitTimingSettings - Check the bit timing for desired bit rate.
examples/ESP32CANRegisterTest - Checks the CAN register access.

CAN-Driver v1.1

src/ESP32ACAN.h - driver functions.
src/ESP32ACAN.cpp
src/CANMessage.h - CAN Message format properties.

Self testing the ESP32 CAN Controller. Handling both extended and standard frame formats. The sequence of message controlled by Polling method (using Buffer Status Flags). The CAN Controller reponds well for higher CAN bit rates (250 kbit/s and above), but fails for Slow bit rates (125 kbit/s and below). The loss of frames can be seen with the Intensive Check.

examples : ARDUINO FILES

examples/LoopBackCheck-Polling
examples/LoopBackCheck-IntensivePolling - Sends 10000 messages.

CAN-Driver v1.2

src/ACANBuffer16.h

Message control method by handling Interrupts. Driver buffers are added. The size of the Driver buffers can be changed in src/ESP32ACANSettings.h. The Driver works in NormalMode operation.

NormalMode Test with - MCP2515 and MCP2517

examples : ARDUINO FILES

examples/LoopBackCheck-Interrupt
examples/LoopBackCheck-IntensiveInterrupt - Sends 10000 messages.
examples/ESP32CANTestWith-ACAN2515 - test with MCP2515 Normal Mode operation.

CAN-Driver v2.0

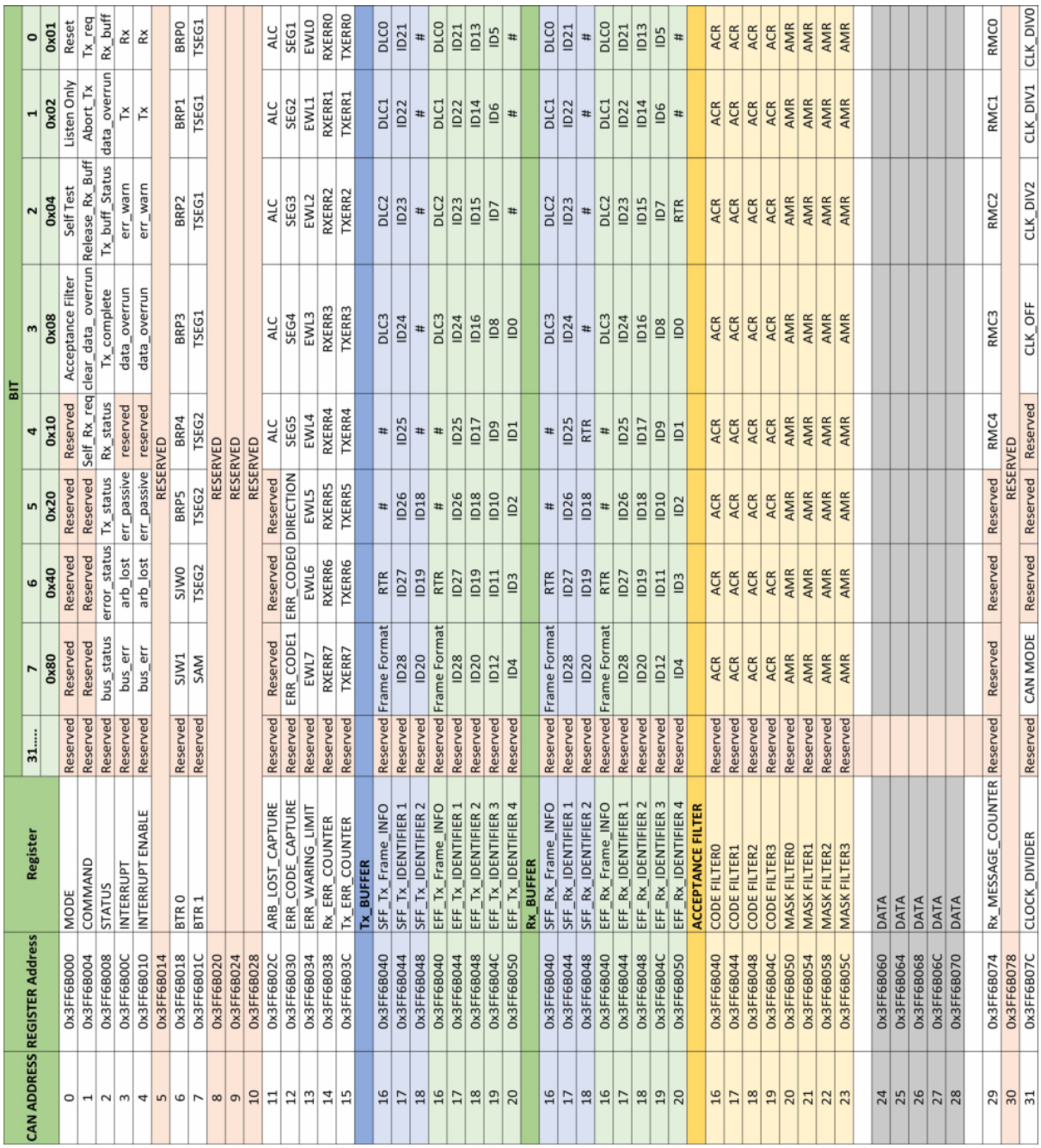
src/ESP32AcceptanceFilters.h

Driver work with reception filters. Single and Dual Filter settings for Standard and Extended Frame can be set. The only message ID defined are accepted and handled by the receiver.

examples : ARDUINO FILES

example/ESP32CANFilterSettings - works with filter settings.

A.1 ESP32 CAN Driver Register Summary



CAN ADDRESS	REGISTER Address	Register	BIT 31.....	7	6	5	4	3	2	1	0
				0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01
0	0x3FF6B000	MODE	Reserved	Reserved	Reserved	Reserved	Reserved	Acceptance Filter	Self Test	Listen Only	Reset
1	0x3FF6B004	COMMAND	Reserved	Reserved	Reserved	Reserved	Self_Rx_req	clear_data_overrun	Release_Rx_Buff	Abort_Tx	Tx_req
2	0x3FF6B008	STATUS	Reserved	bus_status	error_status	Tx_status	Rx_status	Tx_complete	Tx_buff_Status	data_overrun	Rx_buff
3	0x3FF6B00C	INTERRUPT	Reserved	bus_err	arb_lost	err_passive	reserved	data_overrun	err_warn	Tx	Rx
4	0x3FF6B010	INTERRUPT ENABLE	Reserved	bus_err	arb_lost	err_passive	reserved	data_overrun	err_warn	Tx	Rx
5	0x3FF6B014		RESERVED								
6	0x3FF6B018	BTR 0	Reserved	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
7	0x3FF6B01C	BTR 1	Reserved	SAM	TSEG2	TSEG2	TSEG2	TSEG1	TSEG1	TSEG1	TSEG1
8	0x3FF6B020		RESERVED								
9	0x3FF6B024		RESERVED								
10	0x3FF6B028		RESERVED								
11	0x3FF6B02C	ARB_LOST_CAPTURE	Reserved	Reserved	Reserved	Reserved	ALC	ALC	ALC	ALC	ALC
12	0x3FF6B030	ERR_CODE_CAPTURE	Reserved	ERR_CODE1	ERR_CODE0	DIRECTION	SEG5	SEG4	SEG3	SEG2	SEG1
13	0x3FF6B034	ERR_WARING_LIMIT	Reserved	EWL7	EWL6	EWL5	EWL4	EWL3	EWL2	EWL1	EWL0
14	0x3FF6B038	Rx_ERR_COUNTER	Reserved	RXERR7	RXERR6	RXERR5	RXERR4	RXERR3	RXERR2	RXERR1	RXERR0
15	0x3FF6B03C	Tx_ERR_COUNTER	Reserved	TXERR7	TXERR6	TXERR5	TXERR4	TXERR3	TXERR2	TXERR1	TXERR0
		Tx_BUFFER									
16	0x3FF6B040	SFF_Tx_Frame_INFO	Reserved	Frame Format	RTR	#	#	DLC3	DLC2	DLC1	DLC0
17	0x3FF6B044	SFF_Tx_IDENTIFIER 1	Reserved	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
18	0x3FF6B048	SFF_Tx_IDENTIFIER 2	Reserved	ID20	ID19	ID18	#	#	#	#	#
16	0x3FF6B040	EFF_Tx_Frame_INFO	Reserved	Frame Format	RTR	#	#	DLC3	DLC2	DLC1	DLC0
17	0x3FF6B044	EFF_Tx_IDENTIFIER 1	Reserved	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
18	0x3FF6B048	EFF_Tx_IDENTIFIER 2	Reserved	ID20	ID19	ID18	ID17	ID16	ID15	ID14	ID13
19	0x3FF6B04C	EFF_Tx_IDENTIFIER 3	Reserved	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5
20	0x3FF6B050	EFF_Tx_IDENTIFIER 4	Reserved	ID4	ID3	ID2	ID1	ID0	#	#	#
		Rx_BUFFER									
16	0x3FF6B040	SFF_Rx_Frame_INFO	Reserved	Frame Format	RTR	#	#	DLC3	DLC2	DLC1	DLC0
17	0x3FF6B044	SFF_Rx_IDENTIFIER 1	Reserved	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
18	0x3FF6B048	SFF_Rx_IDENTIFIER 2	Reserved	ID20	ID19	ID18	RTR	#	#	#	#
16	0x3FF6B040	EFF_Rx_Frame_INFO	Reserved	Frame Format	RTR	#	#	DLC3	DLC2	DLC1	DLC0
17	0x3FF6B044	EFF_Rx_IDENTIFIER 1	Reserved	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
18	0x3FF6B048	EFF_Rx_IDENTIFIER 2	Reserved	ID20	ID19	ID18	ID17	ID16	ID15	ID14	ID13
19	0x3FF6B04C	EFF_Rx_IDENTIFIER 3	Reserved	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5
20	0x3FF6B050	EFF_Rx_IDENTIFIER 4	Reserved	ID4	ID3	ID2	ID1	ID0	RTR	#	#
		ACCEPTANCE FILTER									
16	0x3FF6B040	CODE FILTER0	Reserved	ACR	ACR	ACR	ACR	ACR	ACR	ACR	ACR
17	0x3FF6B044	CODE FILTER1	Reserved	ACR	ACR	ACR	ACR	ACR	ACR	ACR	ACR
18	0x3FF6B048	CODE FILTER2	Reserved	ACR	ACR	ACR	ACR	ACR	ACR	ACR	ACR
19	0x3FF6B04C	CODE FILTER3	Reserved	ACR	ACR	ACR	ACR	ACR	ACR	ACR	ACR
20	0x3FF6B050	MASK FILTER0	Reserved	AMR	AMR	AMR	AMR	AMR	AMR	AMR	AMR
21	0x3FF6B054	MASK FILTER1	Reserved	AMR	AMR	AMR	AMR	AMR	AMR	AMR	AMR
22	0x3FF6B058	MASK FILTER2	Reserved	AMR	AMR	AMR	AMR	AMR	AMR	AMR	AMR
23	0x3FF6B05C	MASK FILTER3	Reserved	AMR	AMR	AMR	AMR	AMR	AMR	AMR	AMR
24	0x3FF6B060	DATA									
25	0x3FF6B064	DATA									
26	0x3FF6B068	DATA									
27	0x3FF6B06C	DATA									
28	0x3FF6B070	DATA									
29	0x3FF6B074	Rx_MESSAGE_COUNTER	Reserved	Reserved	Reserved	Reserved	RMC4	RMC3	RMC2	RMC1	RMC0
30	0x3FF6B078		RESERVED								
31	0x3FF6B07C	CLOCK_DIVIDER	Reserved	CAN MODE	Reserved	Reserved	Reserved	CLK_OFF	CLK_DIV2	CLK_DIV1	CLK_DIV0

Figure. A.1 ESP32 CAN peripheral register summary

Appendix B

CAN Handbook

B.1 Bit Rate

The number of bits per second that can be transmitted along a network. CAN Bit rates up to 1 Mbit/s are possible at network lengths below 40 m. Decreasing the bit rate allows longer network distances (e.g.,500 m at 125 kbit/s). The speed of CAN may be different in different systems. However, in a given system the bitrate is uniform and fixed.

B.2 Remote Data Request

A node that requires Data frame from another node on the network can request a transmission by sending a Remote Frame. The Data frame and the corresponding Remote frame have the same Identifier.

B.3 Priorities

The Identifier defines a static message priority during bus access. The highest priority is given to the message with lowest number ID. For example in Figure Figure B.1, in three nodes, node 2 with lowest number is given highest priority.

	NODE	Message ID	Binary Representation of Message ID	
	Node 1	0x2B1	001010110001	
Lowest Number	Node 2	0x101	000100000001	Highest Priority
	Node 3	0x51C	010100011100	

Figure. B.1 CAN Message Priority

B.4 CSMA/CD-CR

Carrier Sense Multiple Access (CSMA) and Collision Detection with Collision Resolution (CD-CR) [6]. The CAN communication protocol is a Carrier Sense (CS), multiple-access protocol with collision detection and arbitration on message priority, each node on a bus must wait for a prescribed period of inactivity (inter-frame) before attempting to send a message. When there is no activity in the bus, every node has an equal opportunity to transmit a message. If two nodes transmit at the same time, a collision occurs. Collision resolution refers to “non destructive bitwise arbitration”, messages remain intact even after collision occurs. To implement CSMA CDCR, the physical layer needs to support Dominant and recessive bit states; in which dominant bits wins arbitration over recessive bits.

B.5 Arbitration

The conflicts of bus access is resolved by the bit-wise arbitration. All arbitration takes place without corruption or delay of the highest priority message, the message that losses the arbitration is re-transmitted at the next available time. The mechanism of arbitration guarantees that neither information nor time is lost. If a Data frame and a Remote frame with the same Identifier are initiated at the same time, the Data frame prevails over the Remote frame Figure Figure B.2. During simultaneous transmission of dominant and recessive bits, the resulting bus value will be dominant. When a recessive level is sent, but a dominant level is monitored, the node has lost arbitration and must withdraw without sending any further bits.

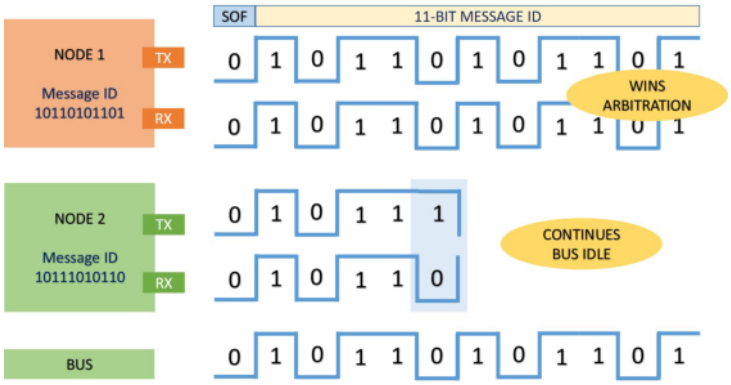


Figure. B.2 Bus Arbitration

B.6 Bit Stuffing

Bit stuffing is introduced at the protocol level to ensure there are enough edges (recessive to dominant) in a CAN frame to maintain Synchronization in the network. It occurs after 5 like bits in a row, either recessive or dominant. The protocol handler should add an additional bit of

opposite polarity to the CAN frame to force an edge. The bits are added by CAN Transmitting node and are removed by Receive node at the protocol level. It cannot be seen in user application.

B.7 Error Handling

The CAN node performs the error handling by detecting the error conditions. To ensure the integrity of the messages these error conditions are defined in the CAN protocol. The error conditions causes the CAN node to transmit error frames, which increments the internal transmit or receive error counters. By using these counters the node is able to detect the source of error, for instance Bus problems and subsequently disconnects from the CAN Bus.

Types of Error Detection:

1. **Bit Error** : When the transmitter node monitors a different signal on the bus from the original sent signal. This error check is not done during the arbitration and acknowledgement field, because the nodes will have equal access to bus in that time.
2. **Bit Stuffing Error** : The stuffing error occurs when any node that detects 6 consecutive bit of the same polarity signal between the `SOF` and the end of `CRC` field.
3. **CRC Error** : When the CRC sequence received not identical to the CRC sequence calculated.
4. **ACK Error** : An ACK error is detected by a transmitter whenever a dominant bit is not monitor during the ACK slot.
5. **Form Error** : When a fixed form bit field (CRC delimiter, ACK delimiter, EOF field) contains one or more irrelevant bits.

If any of these errors detected, the transmitter will send an error thus destroying the current message and will transmit the original message on next idle bus time.

Each CAN controller has 3 error states:

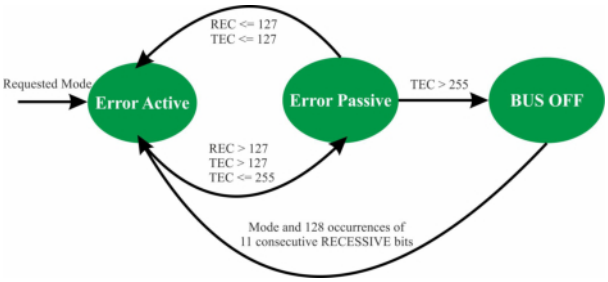


Figure. B.3 CAN Error States

Change between the error states is done automatically by the CAN controller. The different states depends on the value of error counters. `TEC` is Transmit Error Counter and `REC` is Receive Error Counter exists in each CAN controller.