



# Embedded Systems Design

**Review**

---

**Guided By : Prof. Selvakumar K**

**Team Members:**

**Ruturaj A. Nanoti - 18BEE0134**

**Amitvikram S. Pujar - 18BEE0135**

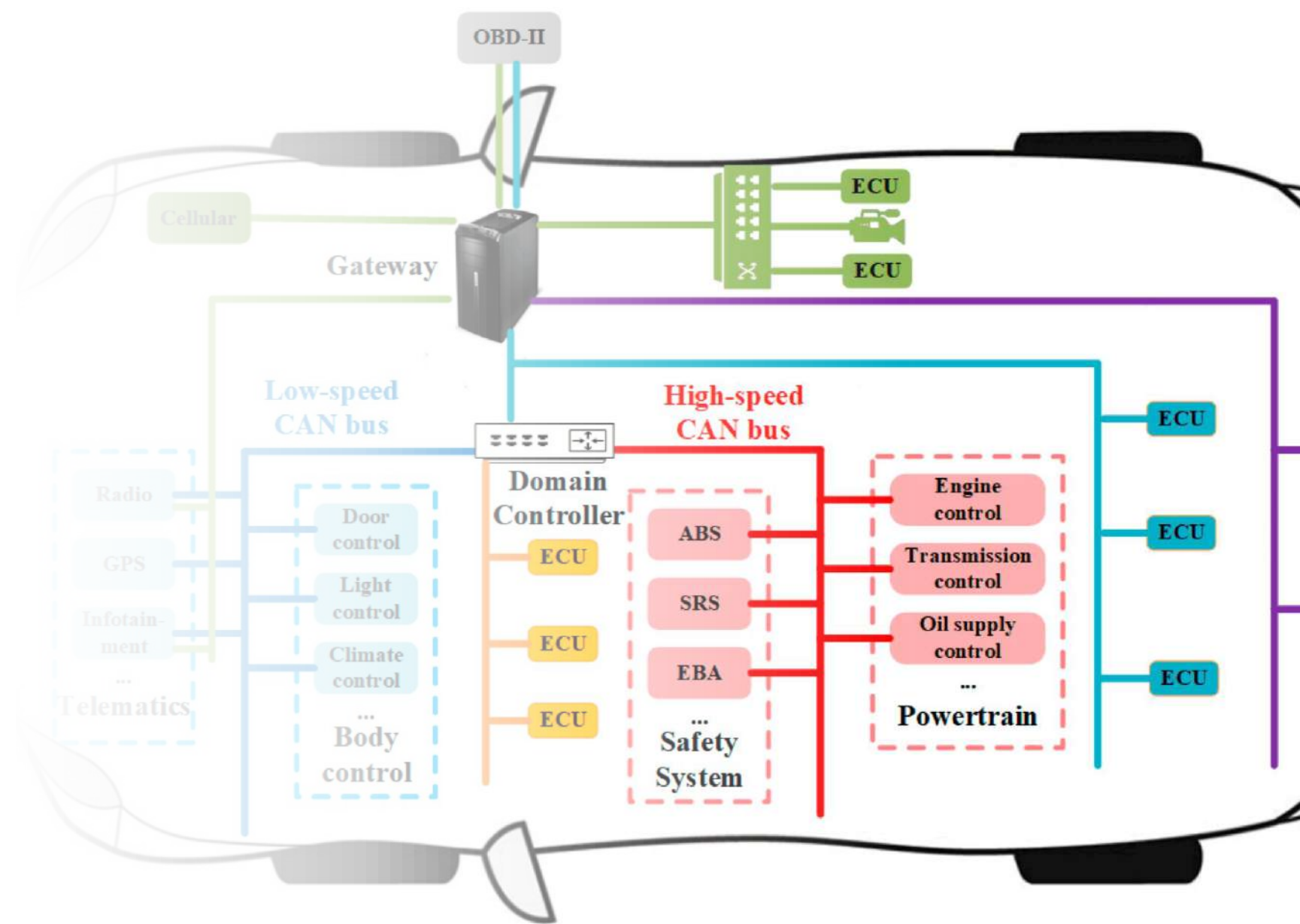


# Controller Area Network

---

# Objective

- Understanding the Implementation of CAN protocol on the Cortex-M4 based Microcontroller.
- Establishing CAN communication between 8-bit and 32-bit MCUs.
- Implementing CAN assisted CAR parking System.



Global Position System  
Second On-board Diagnostic

ABS: Anti-skid Brake System  
SRS: Supplemental Restraint System  
EBA: Emergency Brake Assist

Ethernet  
FlexRay  
CAN

# CAN Protocol

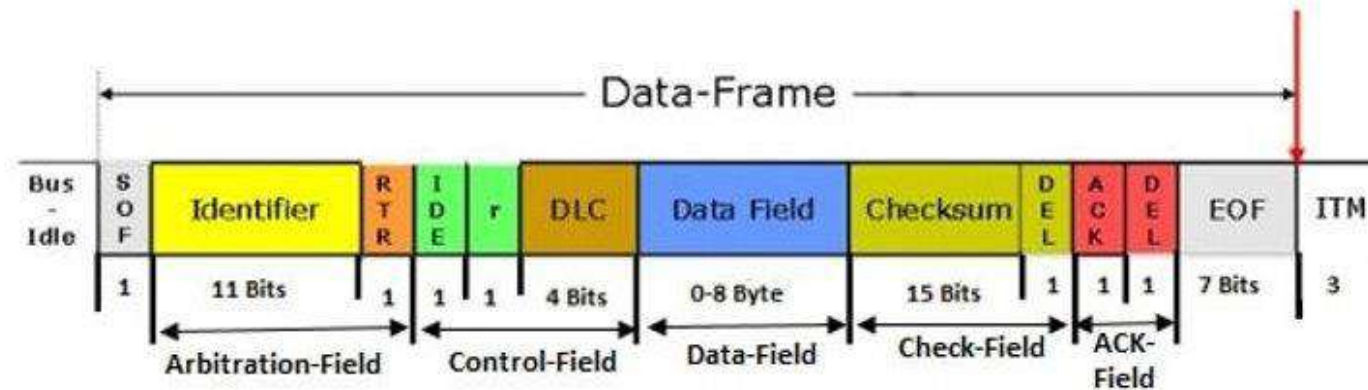
- CAN stands for Controller Area Network. It was first created by the German Automotive Company Robert Bosch in the mid-1980s for automobile applications.
- CAN is a message-based protocol, which means that communication does not happen based on addresses but rather the CAN message itself contains the priority and the data that needs to be transmitted. Every node on the bus receives this message, and then each node decides whether that information is useful or needs to be discarded.
- A single message can be destined for a particular receiver or for multiple nodes that are present on the bus.
- Another important feature of the CAN protocol is its ability to request information from other nodes which is known as "Remote Transmission Request" or "RTR".
- Since CAN is a message-based protocol a node can be added to the system without the need to reprogram the other nodes for acknowledging the new addition.

# Components of CAN Message Frame

- There are four types of CAN frames in the CAN protocol. The most common type is the "Data Frame", which is used when a node transmits information to other nodes. The second one is the "Remote Frame" which is a Data frame with the RTR bit set to signify a Remote Transmission Request. The other two frames are namely the Error Frame and the Overload Frame which are used for handling errors.
- Error Frames are generated by any one of the nodes that detect any one of the many protocol errors defined in CAN.
- Overload Frames are generated by nodes that require more time to process the data that has already been received.

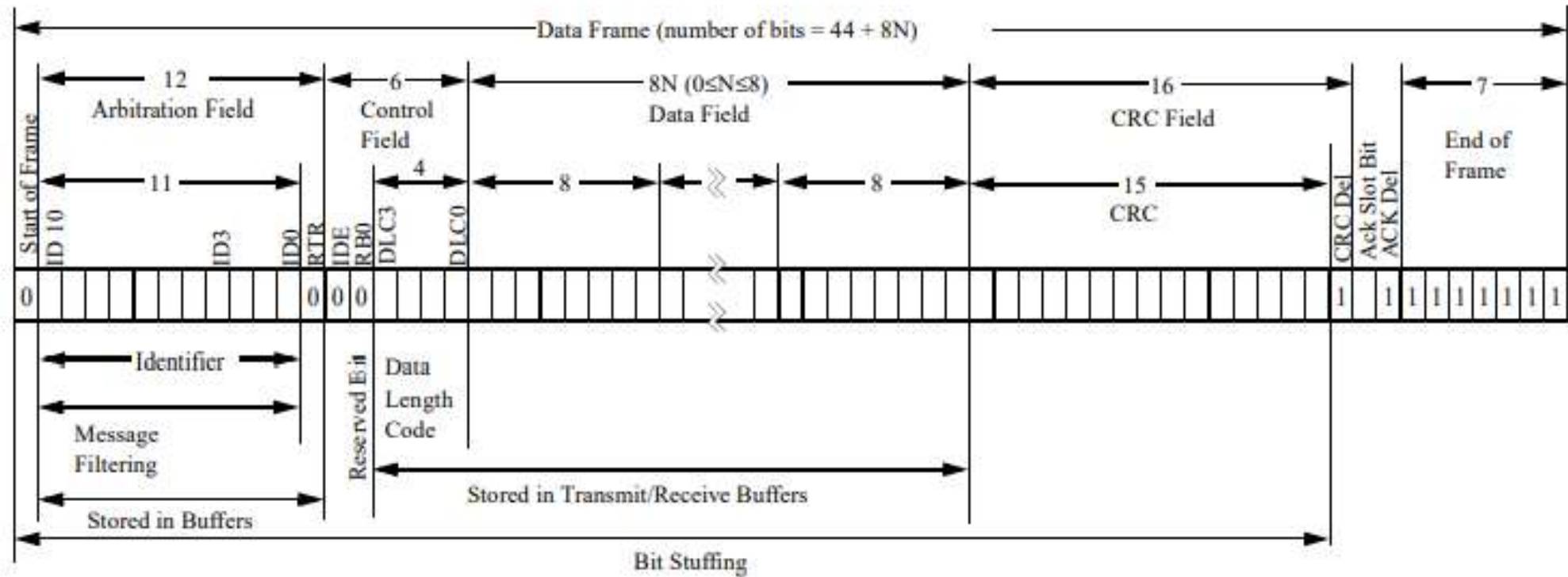


# CAN Data Frame

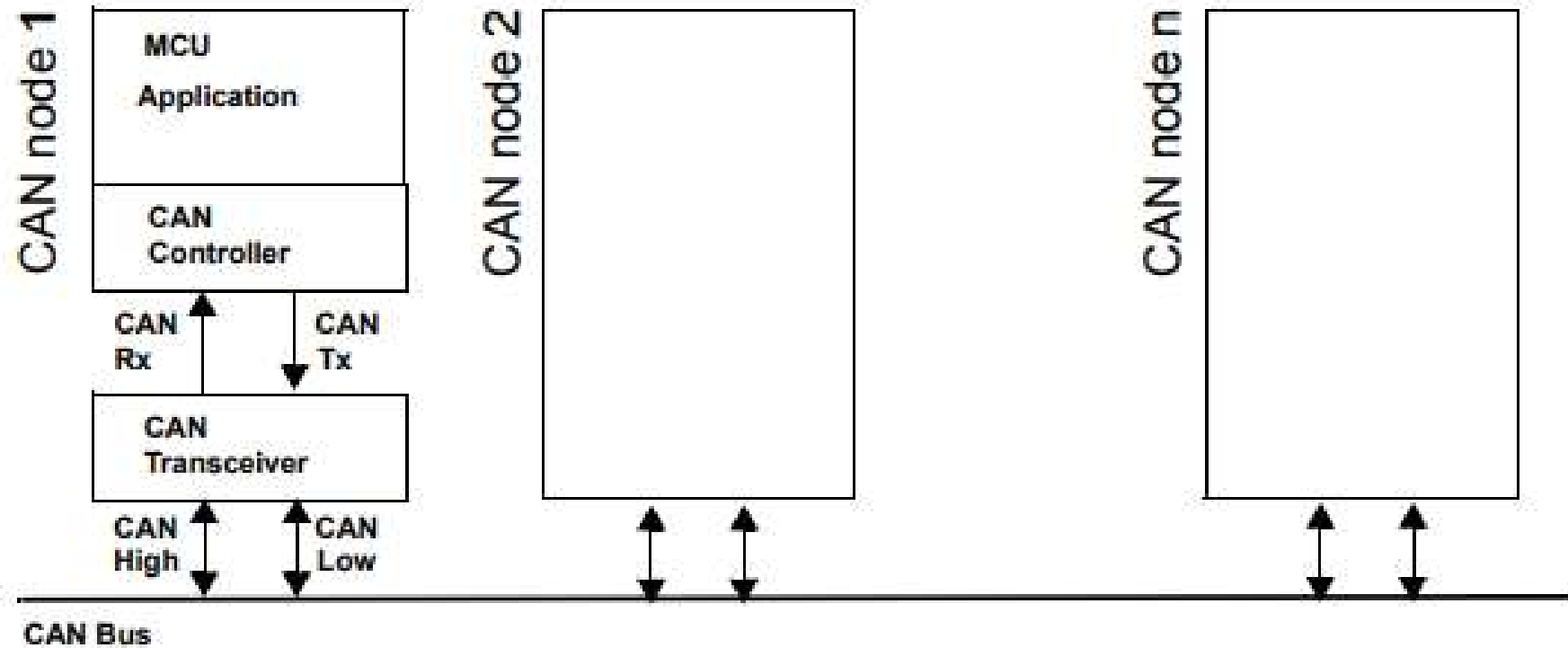


- SOF - Start of Frame
- Identifier - A message Identifier that sets the priority of the Data Frame.
- RTR - Remote Transmission Request, defines the frame type (data frame or remote frame - 1 Bit)
- Control Field - User Defined Functions
- DLC - Data Length Code (4 bits)
- Data Field - User Defined Data (0 to 8 Bytes)
- CRC Filed - Cyclic Redundancy check for Error (Data corruption) detection.
- ACKField - Receivers Acknowledgement

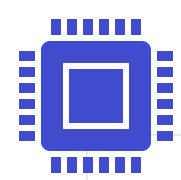
# Extended CAN Data Frame



# CAN Network Topology





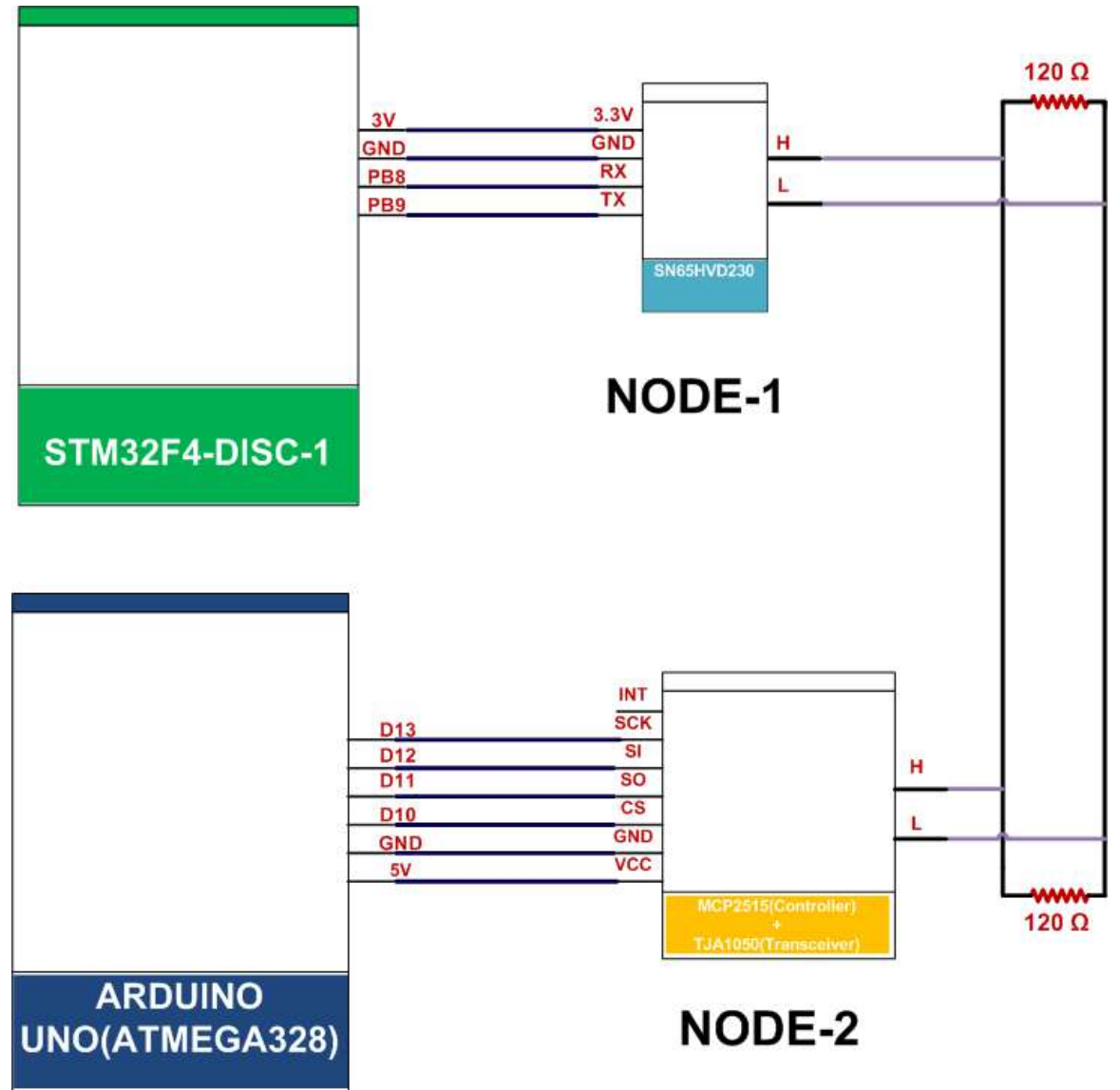


# Components Required

---

- A 32-bit Cortex-M4 based microcontroller (e.g. STM32F4 discovery board)
- An 8-bit microcontroller like ATmega328 (found on Arduino UNO boards)
- CAN controller (MCP2515) & transceiver (SN65HVD230).
- Ultrasonic sensor (e.g. HC-SR04)
- LEDs, jumpers, etc.
- Motor & motor driver IC

# Hardware Schematic



# Methodology

## ***STM32F4 Discovery Board***

- The Cortex-M4 microcontroller has bxCAN 2.0. It supports both the version 2.0A and B. The communication speeds can go up to 1Mbits/s.
- Since the discovery kit has an on-board CAN controller, a CAN transceiver like SN65HVD230 is required for transmitting to & receiving from the CAN bus.
- The SN65 transceiver is terminated by a 120-ohm resistor on one end and on the other end, the CANH & CANL lines are connected to the nodes on the bus

# Methodology (Contd.)

## ***ATMega328 (On Arduino UNO)***

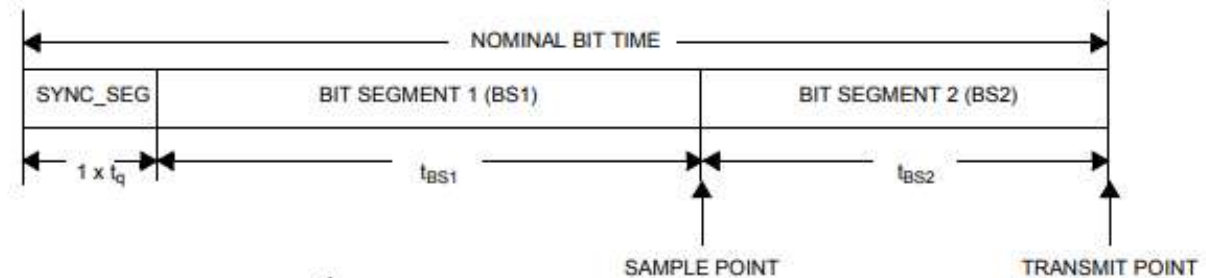
- There's no functionality of CAN controller on the ATMega chip, therefore we use an external CAN controller like MCP2515 which is interfaced through SPI with ATMega chip.
- This CAN controller is further connected to the transceiver via its Tx & Rx pins in order to send the message to & read from the CAN bus.
- The SN65 transceiver is terminated by a 120-ohm resistor on one end and on the other end, the CANH & CANL lines are connected to the nodes on the bus

# CAN Operating Modes

- ***Loopback:*** In this mode, an internal feedback is performed from the Tx output of CAN transceiver to its Rx input
- ***Normal***
- ***Silent / Listen Only***

# Baud Rate Calculation

Figure 346. Bit timing



$$\text{BaudRate} = \frac{1}{\text{NominalBitTime}}$$

$$\text{NominalBitTime} = 1 \times t_q + t_{BS1} + t_{BS2}$$

with:

$$t_{BS1} = t_q \times (\text{TS1}[3:0] + 1),$$

$$t_{BS2} = t_q \times (\text{TS2}[2:0] + 1),$$

$$t_q = (\text{BRP}[9:0] + 1) \times t_{\text{PCLK}}$$

where  $t_q$  refers to the Time quantum

$t_{\text{PCLK}}$  = time period of the APB clock,

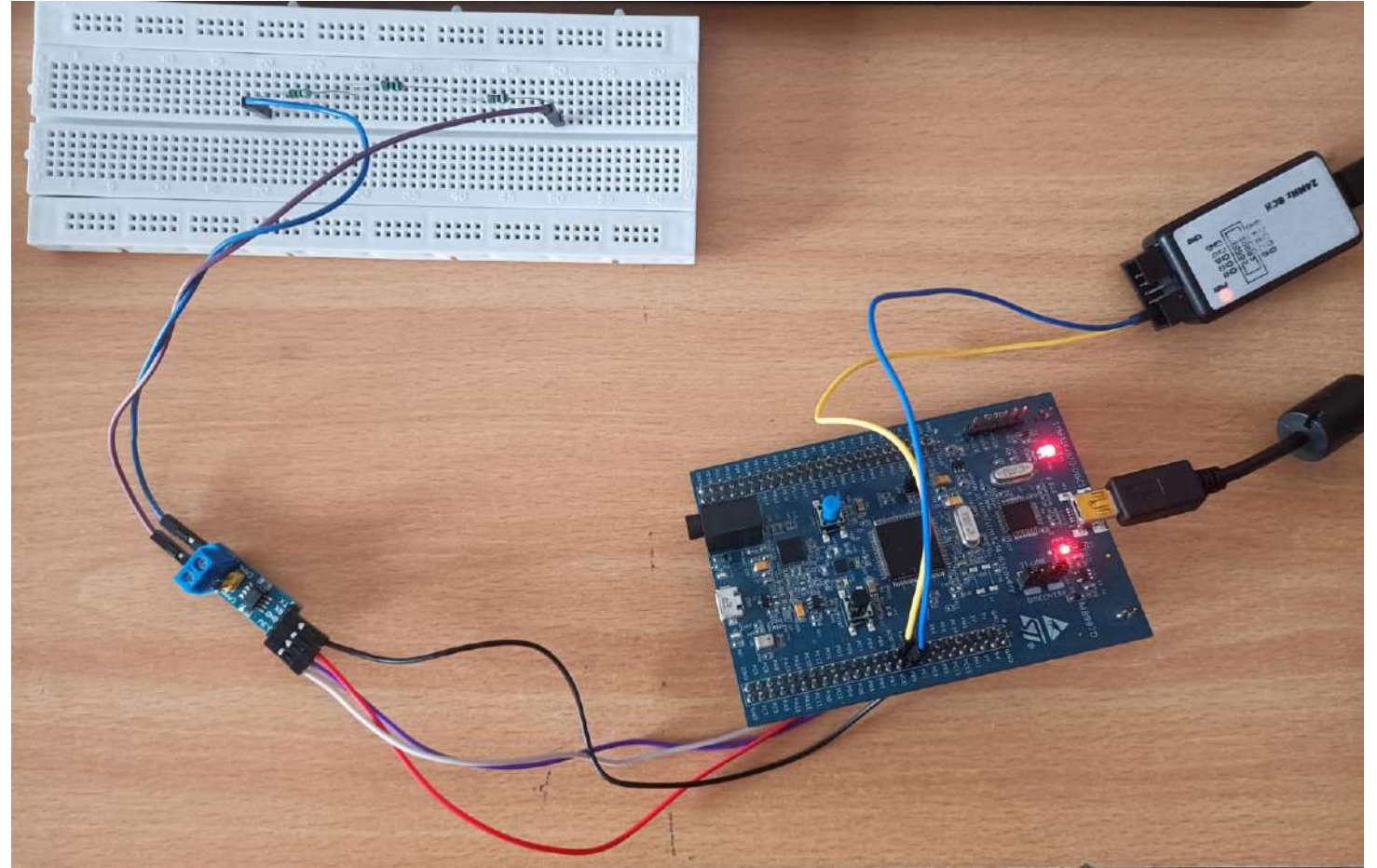
BRP[9:0], TS1[3:0] and TS2[2:0] are defined in the CAN\_BTR register.



# Results

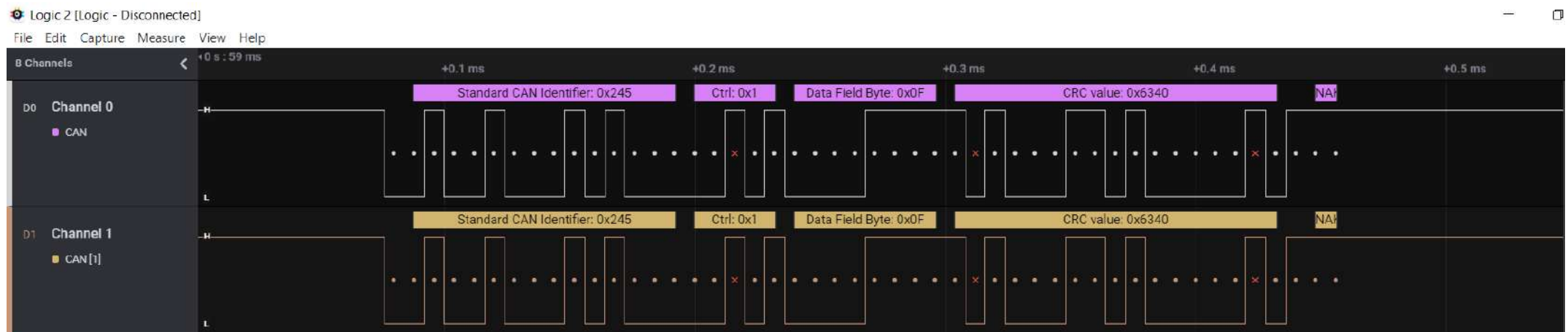
## Loopback Mode

- This is one of Test modes available for the CAN peripheral on the STM32F407 microcontroller.
- It validates the working of the CAN protocol by acting as both the receiver and transmitter node.



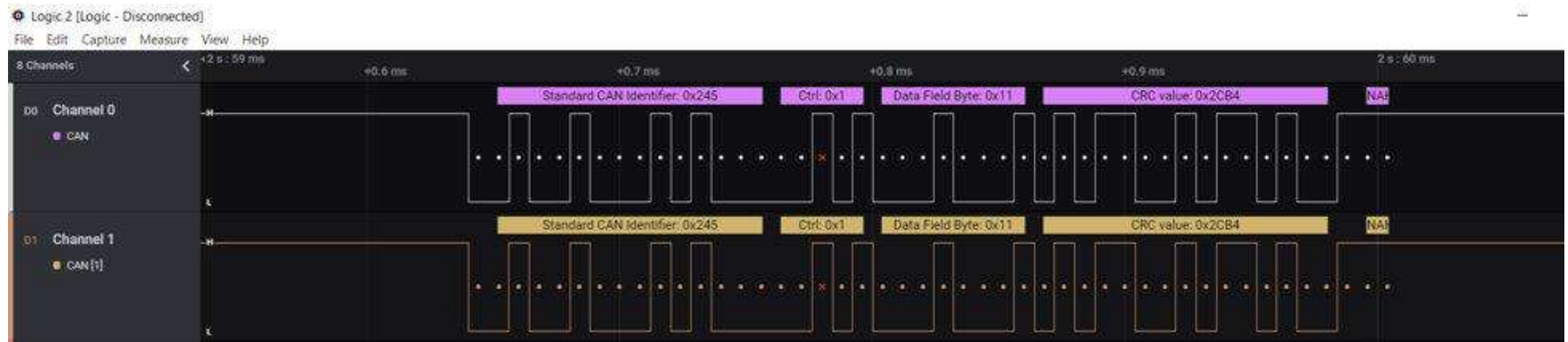
# Results (Contd.)

**Data Transmitted** : Nos. From 1 to 25



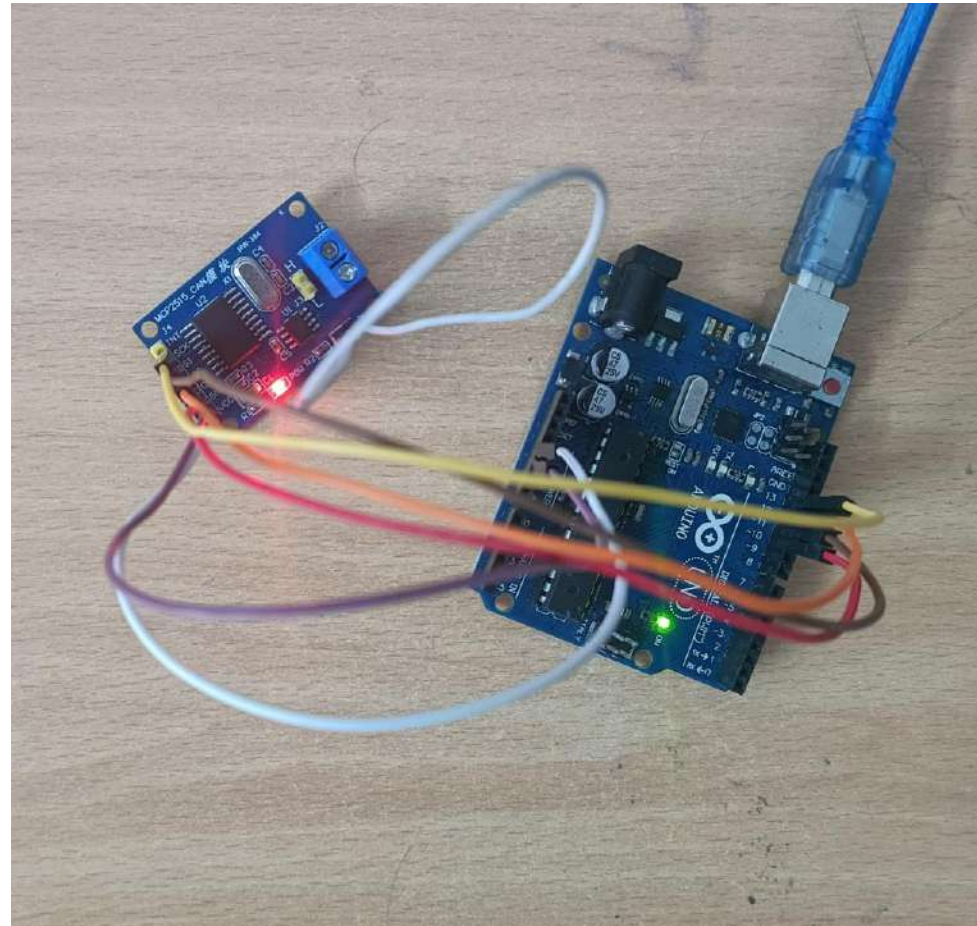
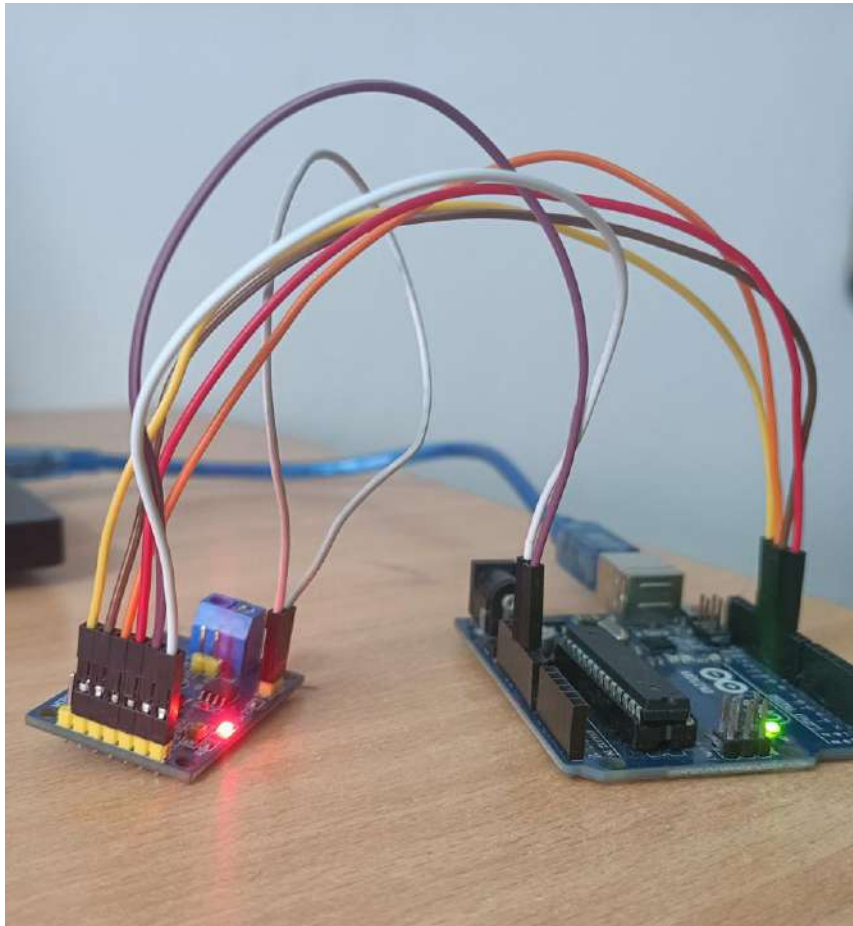
*Message frame corresponding to Data Field Byte = 0x0F (15)*

# Results(Contd.)



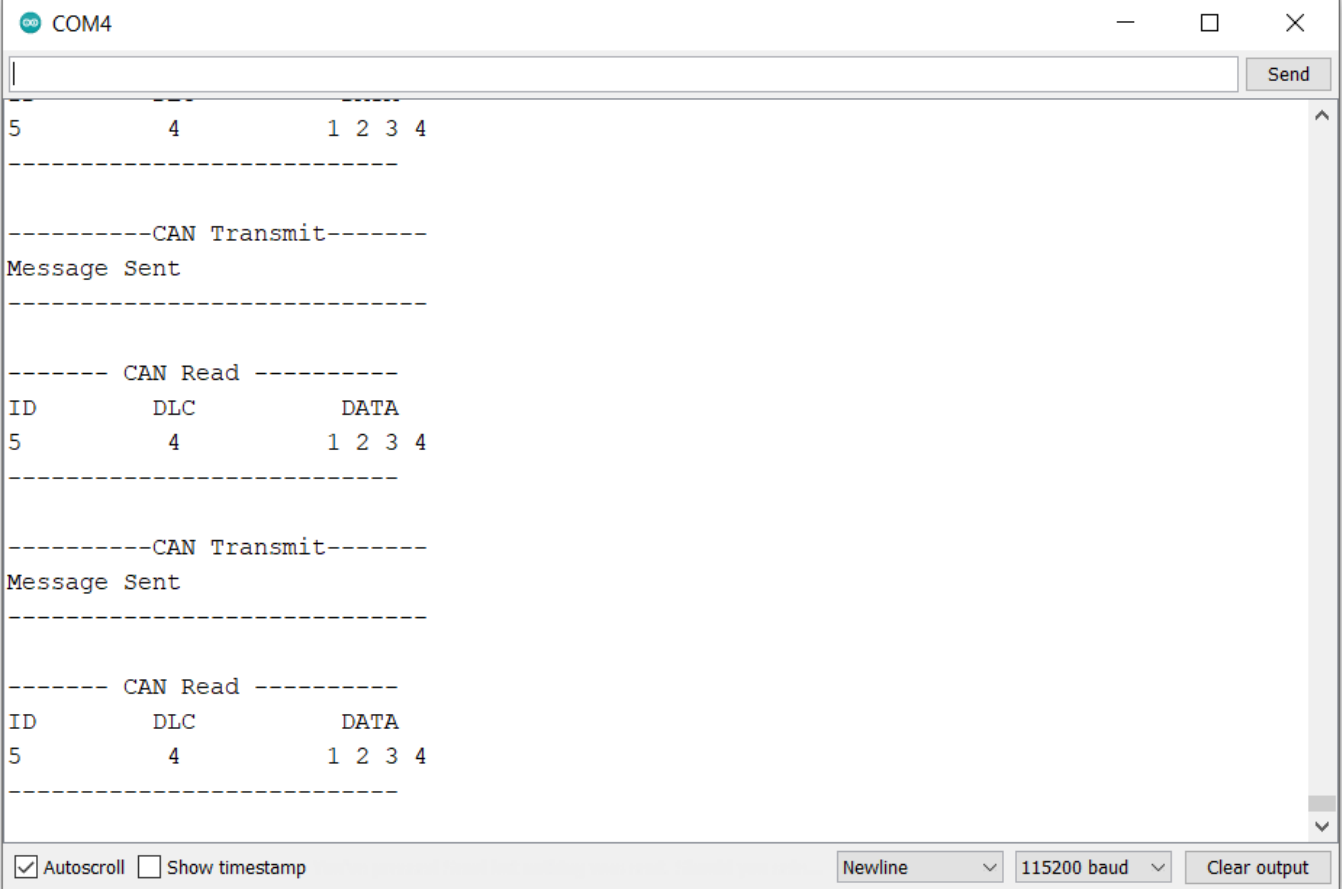
*Message frame corresponding to Data Field Byte = 0x11 (17)*

## Results (Contd.)



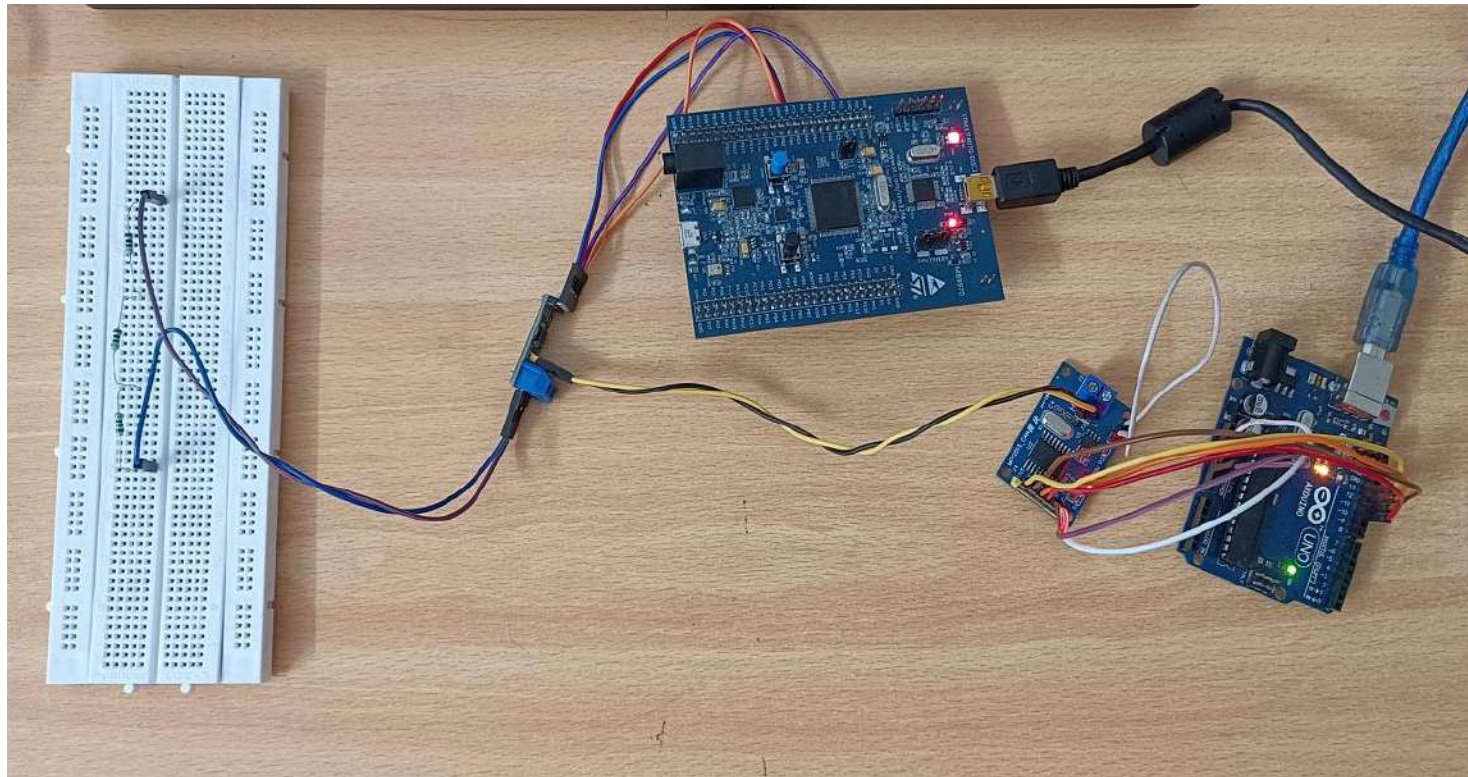


© 2006 The Authors



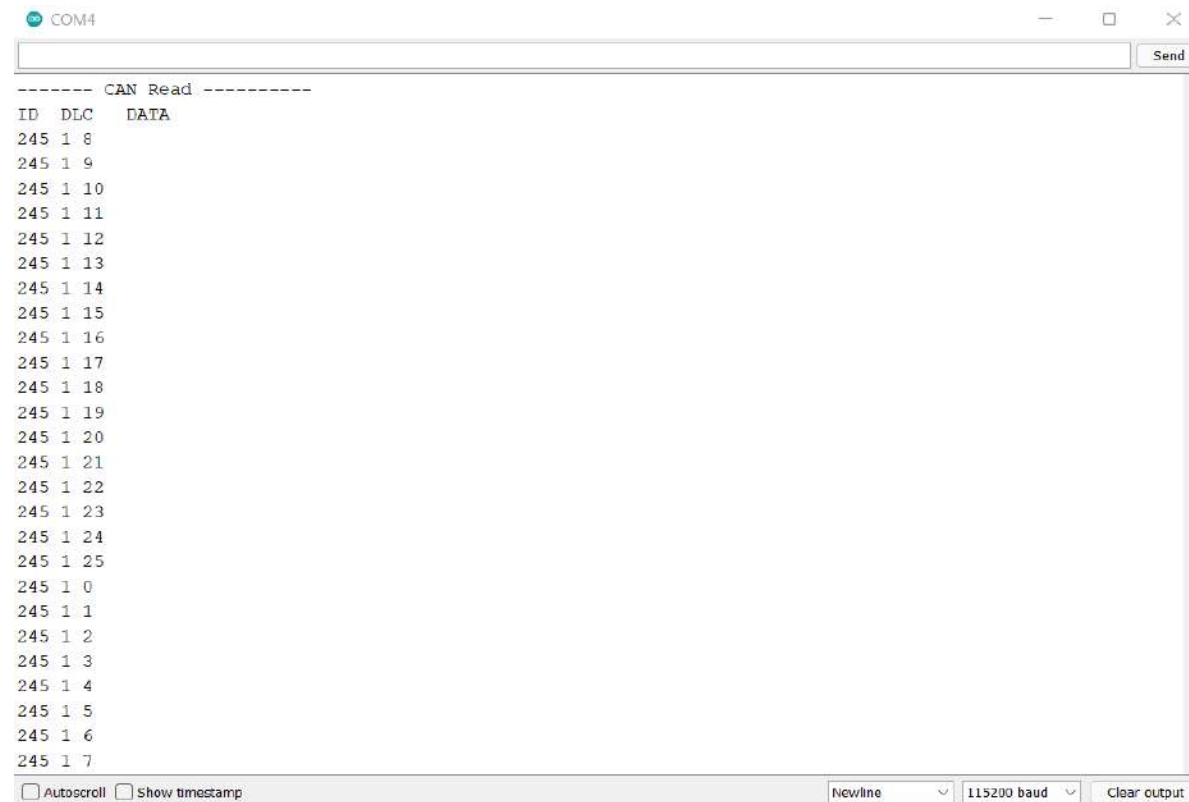
## Arduino Serial Monitor Output for Loopback Mode

# Normal Mode Communication





# Results



The screenshot shows the Arduino Serial Monitor window titled 'COM4'. The main text area displays a series of CAN bus messages. Each message is preceded by a separator line '----- CAN Read -----'. The messages are formatted as follows:

ID	DLC	DATA
245	1	8
245	1	9
245	1	10
245	1	11
245	1	12
245	1	13
245	1	14
245	1	15
245	1	16
245	1	17
245	1	18
245	1	19
245	1	20
245	1	21
245	1	22
245	1	23
245	1	24
245	1	25
245	1	0
245	1	1
245	1	2
245	1	3
245	1	4
245	1	5
245	1	6
245	1	7

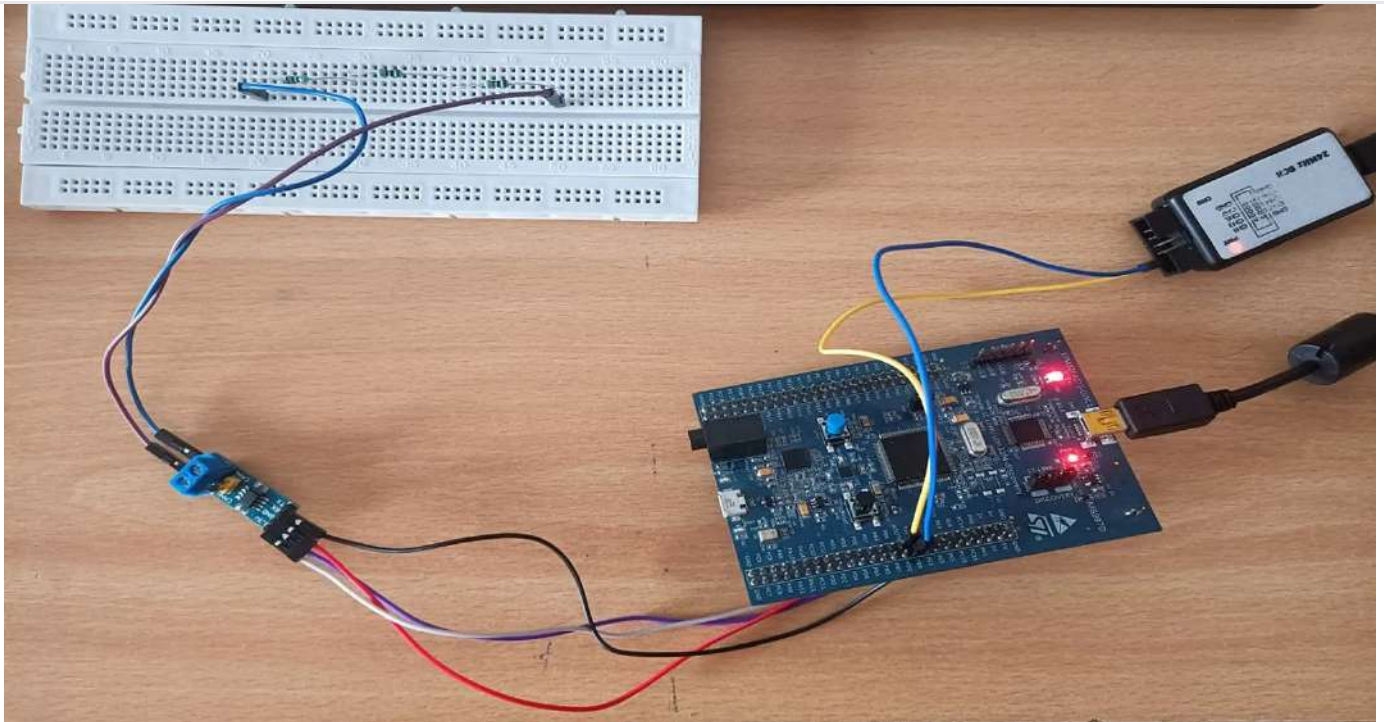
At the bottom of the window, there are control buttons: 'Autoscroll' (unchecked), 'Show timestamp' (unchecked), a 'Newline' dropdown menu, a '115200 baud' dropdown menu, and a 'Clear output' button.

*Arduino Serial Monitor Output  
for Normal Communication*

# References

- <https://www.polytechnichub.com/applications-controller-area-network-can-bus/>
- <https://www.microchip.com/en-us/products/interface-and-connectivity/can/can-2-0-mcus>
- <https://alselectro.wordpress.com/2020/04/29/sensor-data-on-can-bus-arduino-with-can2515/>
- <https://www.ni.com/en-us/innovations/white-papers/06/controller-area-network--can--overview.html>

## Embedded System Projects with STM32F407



# Testing the CAN peripheral on the STM32F407 Discovery Kit in Loopback Mode



**Ruturaj A. Nanoti**

Published on Oct 18, 2021



🕒 14 min read

### Introduction

Hello!! 😊 Today we will explore the CAN peripheral available on the *STM32F407 Discovery Kit*. We will test it using the Loopback Operating mode, which is one of the Test

modes. The purpose of this test is to ensure that the CAN peripheral is working properly, and in our case also acts as a starting point in the journey to learn about the **CAN Communication Protocol**. The main idea of the Loopback test is that there is only one node on the *CAN bus*, and that same node receives the data that it transmits. As the name suggests the data that is transmitted by the node **loops back** towards it.

## Working Principle

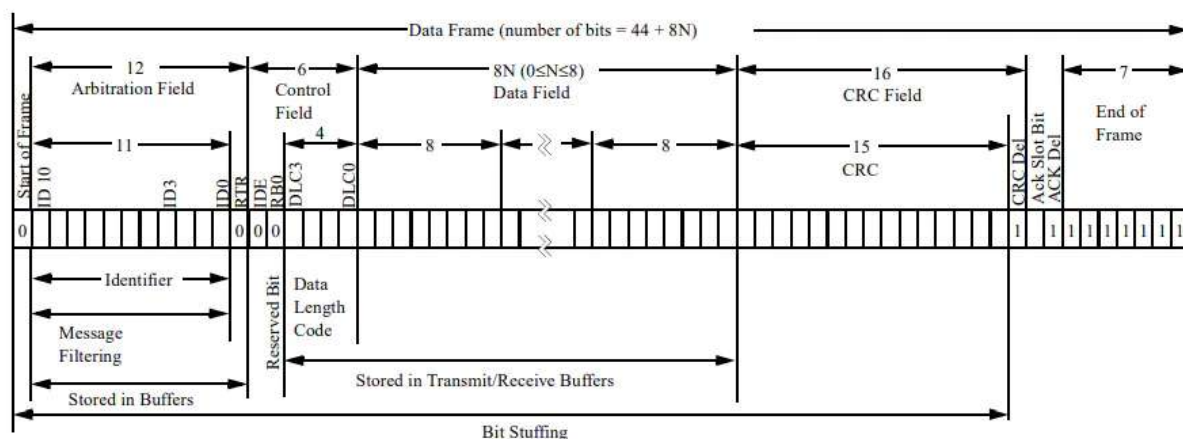
### CAN Protocol

CAN stands for *Controller Area Network*. It was first created by the *German Automotive Company Robert Bosch* in the mid-1980s for automobile applications.

CAN is a **message-based protocol**, which means that communication does not happen based on addresses but rather the CAN message itself contains the priority and the data that needs to be transmitted. Every node on the bus receives this message, and then each node decides whether that information is useful or needs to be discarded. A single message can be destined for a particular receiver or for multiple nodes that are present on the bus.

Another important feature of the CAN protocol is its ability to request information from other nodes which is known as **Remote Transmission Request** or **RTR**. Since CAN is a message-based protocol *a node can be added to the system without the need to reprogram the other nodes for acknowledging the new addition*.

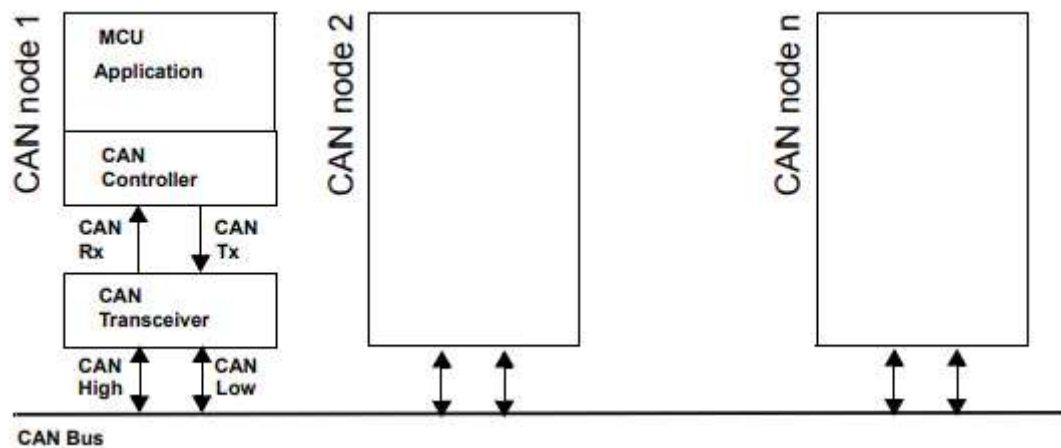
A CAN data frame can be represented as:



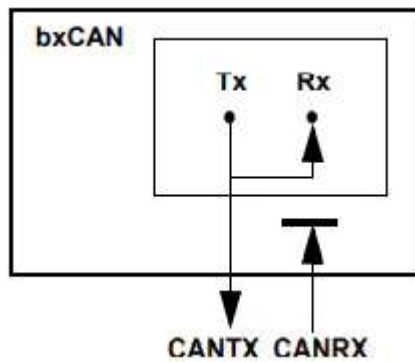
The elements of a CAN Frame are:

- **SOF** – Start of Frame
- **Identifier** – A message Identifier that sets the priority of the Data Frame.
- **RTR** – Remote Transmission Request, defines the frame type (data frame or remote frame – 1 Bit)
- **Control Field** – User-Defined Functions
- **DLC** – Data Length Code (4 bits)
- **Data Field** – User-Defined Data (0 to 8 Bytes)
- **CRC Filed** – Cyclic Redundancy checks for Error (Data corruption) detection.
- **ACK Field** - Receivers Acknowledgement

CAN network topology,



The CAN Loopback mode can be understood by the following figure,



As can be seen in the above figure, the node that is transmitting the data is the same as the one receiving it. The data that the node transmits comes back at it and is received by it, which emulates the behavior of a normal node on the *CAN Bus*.

For implementing the CAN protocol on our MCU, we will need a **CAN transceiver**. The *STM32F407* microcontroller has an in-built **CAN controller**, but the *CAN transceiver* needs to be connected externally. So, we will be using the **SN65HVD230** for that purpose. Now, the *CAN Bus* needs to be terminated by connecting a **120-ohm** resistance between **CAN High** and **CAN Low**. These are basically the two wires that make up the *CAN Bus*, which acts as a medium for communication between the various nodes. If the *CAN Bus* is not terminated by a resistance the transmitted messages might reflect back and corrupt the messages on the bus and lead to a communication failure. The *SN65HVD230* breakout board does not come with a *120-ohm* on-board termination, so we will need to connect it externally.

Now let's talk about how the reception is handled at each node. Since *CAN* is a message-based protocol, each node is identified by the *Identifier Field* that is embedded in the *CAN Frame*. So, each node on the *CAN bus* can be configured to receive messages either from only a specific node or a particular group of nodes based on the identifier of the incoming message. This is achieved by using the filters that are available in the *CAN controllers*. The filters are a part of the hardware so the unwanted messages are discarded before they even reach the CPU of the microcontroller, this helps save time as well as resources that can be used elsewhere for more important tasks, and this is especially helpful since the number of resources in an embedded system is constrained. The filters can be configured in **Identifier List Mode** or **Mask Mode**. In the *Mask Mode* the filters are configured to match some of the bits at a specific position in the identifier of the transmitted message if they match only then the message is accepted by that particular node, otherwise, it is discarded. This mode can be used when a particular node wants to receive messages from a specific group of nodes. While, in the *Identifier List Mode*, all the bits in the *Identifier Field* of the incoming message must match the pre-configured value in the filter



register for the node to accept that message. This mode can be used when a node wants to receive messages only from a particular node on the CAN Bus.

The Connections that need to be made, can be found in my GitHub repo, which you can access by clicking [here](#).

Enough theory, now let's start the interesting stuff!! 😊

## Components Required

- An STM32F4 based MCU
- A SN65HVD230 Breakout board
- A 120-ohm resistor **OR** any combination of resistors that results in 120-ohms ( I have used two 10-ohm resistors and a 100-ohm resistor and connected them in series to get 120-ohms).
- A Logic Analyzer
- A Few Jumper Wires
- A breadboard

And, that's all. After gathering these components we are ready to start coding.

## Programming our MCU

### 1. Including the necessary Header Files and some comments about the code

```

/*****
* Configurations are made as follows:
* PB8 - CAN1_RX (Need to be set in AF9 for CAN1/CAN2)
* PB9 - CAN1_TX (Need to be set in AF9 for CAN1/CAN2)
*
* The following is a bare metal code written for CAN Loop back test on the
* Arm Cortex-M4 based STM32F407 Discovery Kit.
*
* @File    main.c
* @Author  Ruturaj A. Nanoti
*****/
```

COPY 

```
*****/
```

```
#include "stm32f4xx.h"
#define ARM_MATH_CM4
```

These are the header files included at the top of our `main.c` file. The `stm32f4xx.h` is the one for our microcontroller and the `ARM_MATH_CM4` is for any math operations that we perform in our code.

## 2. Declaring the User-Defined Functions and variables

```
void GPIO_Init(void);
void CAN1_Init(void);
void CAN1_Tx(uint8_t tr);
uint8_t CAN1_Rx(void);
void TIM4_ms_Delay(uint32_t delay);
uint8_t k = 0;
uint8_t rec = 0;
```

COPY 

The `void GPIO_Init(void)` function is used to initialize and configure the `GPIO` pins that we will be needing to connect our SN65HVD230 module and use the CAN Peripheral. The `void CAN1_Init(void)` and the `void CAN1_Tx(uint8_t tr)` functions are used to initialize `CAN1` and transmit a byte of data respectively. The `uint8_t CAN1_Rx(void)` function is used to receive the incoming data. `void TIM4_ms_Delay(uint32_t delay)` acts as our delay function which utilizes the `TIM4` peripheral and provides a delay in *milliseconds*. Finally, we declare the variables `uint8_t k` and `uint8_t rec` which will use to transmit and receive data respectively.

## 3. Defining the User-Defined Functions

- Let's configure the GPIO pins using the `void GPIO_Init()` function,

```
void GPIO_Init(){
    // Enable GPIOA clock signal
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
```

COPY 

```

// Configuring PB8 and PB9 in alternate function mode
GPIOB->MODER |= (GPIO_MODER_MODER8_1 | GPIO_MODER_MODER9_1);

// Selecting AF9 for PB8 and PB9 (See Page 272 of dm00031020)
GPIOB->AFR[1] |= (GPIO_AFRH_AFSEL8_3 | GPIO_AFRH_AFSEL8_0 |
                  GPIO_AFRH_AFSEL9_0 | GPIO_AFRH_AFSEL9_3);
}

```

First, we enable the `GPIOA` clock by writing a `1` to the `GPIOBEN` bit in the `RCC->AHB1ENR` register.

## Reset and clock control for STM32F42xxx and STM32F43xxx (RCC)

RM0090

### 6.3.10 RCC AHB1 peripheral clock register (RCC\_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	OTGHS ULPIEN	OTGHS SEN	ETHMACPT EN	ETHMACRX EN	ETHMACTX EN	ETHMACEN	Res.	DMA2DEN	DMA2EN	DMA1EN	CCMDAT ARAMEN	Res.	BKPSR AMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCE N	Res.	GPIOK EN	GPIOJ EN	GPIOIE N	GPIOH EN	GPIOG EN	GPIOFE N	GPIOEEN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Then, we set `PB8` and `PB9` in *Alternate-Function Mode*. This is done by writing `10` to the `MODER8[1:0]` and `MODER9[1:0]` bits in the `GPIOB->MODER` register.

## 8.4 GPIO registers

This section gives a detailed description of the GPIO registers.

For a summary of register bits, register address offsets and reset values, refer to [Table 39](#).

The GPIO registers can be accessed by byte (8 bits), half-words (16 bits) or words (32 bits).

### 8.4.1 GPIO port mode register (GPIOx\_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

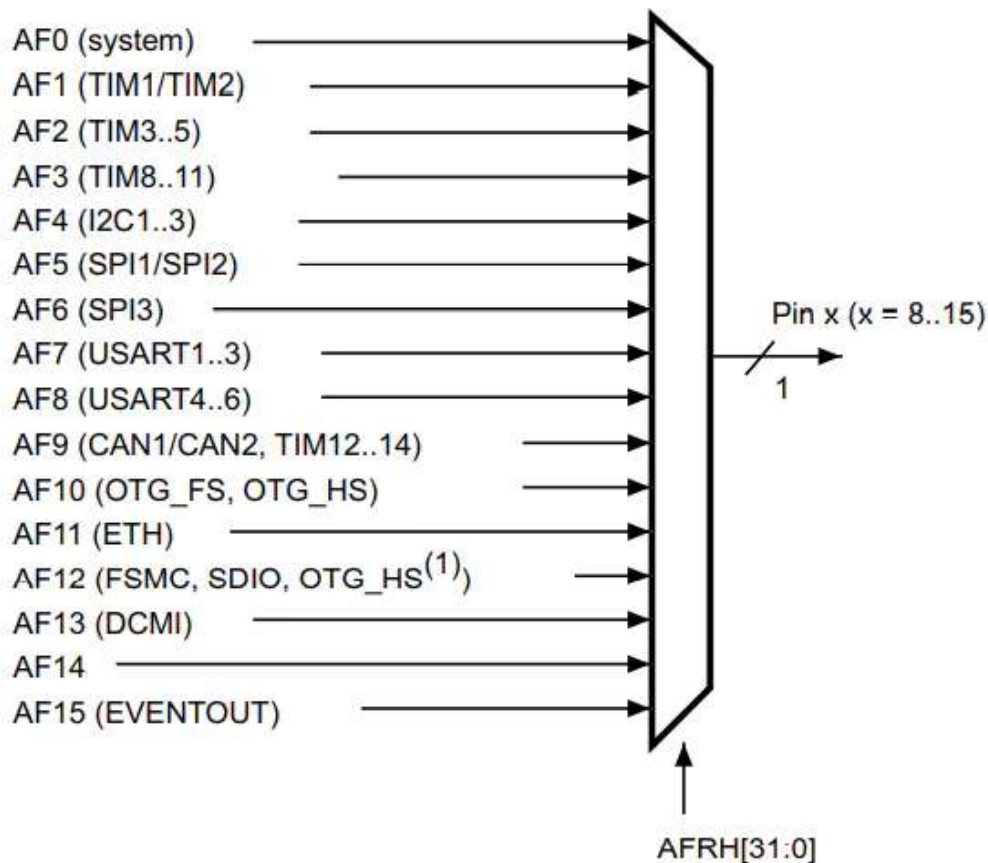
01: General purpose output mode

10: Alternate function mode

11: Analog mode

We need to select AF9 for the CAN1 peripheral. This is done by writing 1001 to the AFRH8[3:0] and the AFRH9[3:0] bits in the GPIOB->AFR[1] register.

For pins 8 to 15, the GPIOx\_AFRH[31:0] register selects the dedicated alternate function



#### 8.4.10 GPIO alternate function high register (GPIOx\_AFRH) (x = A..I/J)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:0 **AFRHy**: Alternate function selection for port x bit y (y = 8..15)

These bits are written by software to configure alternate function I/Os

AFRHy selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

- Then, we come to our `void CAN1_Init()` where we initialize and configure the `CAN1` peripheral,

COPY 

```
void CAN1_Init(){
    /* 1. Setting Up the Baud Rate and Configuring CAN1 in
     * Loop Back Mode -----*/

    // Enable clock for CAN1
    RCC->APB1ENR |= RCC_APB1ENR_CAN1EN;

    // Entering CAN Initialization Mode and wait for acknowledgment
    CAN1->MCR |= CAN_MCR_INRQ;
    while (!(CAN1->MSR & CAN_MSR_INAK)){

    //Set Loop back mode for CAN1
    CAN1->BTR |= CAN_BTR_LBKM;

    //Setting the Re synchronization jump width to 1
    CAN1->BTR &= ~CAN_BTR_SJW;

    //Setting the no. of time quanta for Time segment 2
    // TS2 = 4-1;
    CAN1->BTR &= ~(CAN_BTR_TS2);
    CAN1->BTR |= (CAN_BTR_TS2_1 | CAN_BTR_TS2_0);

    //Setting the no. of time quanta for Time segment 1
    // TS1 = 3-1;
    CAN1->BTR &= ~(CAN_BTR_TS1);
    CAN1->BTR |= (CAN_BTR_TS1_1);

    //Setting the Baud rate Pre-scalar for CAN1
    // BRP[9:0] = 16-1
    CAN1->BTR |= ((16-1)<<0);

    // Exit the Initialization mode for CAN1
    // Wait until the INAK bit is cleared by hardware
    CAN1->MCR &= ~CAN_MCR_INRQ;
    while (CAN1->MSR & CAN_MSR_INAK){}

    //Exit Sleep Mode
    CAN1->MCR &= ~ CAN_MCR_SLEEP;
```



```

while (CAN1->MSR & CAN_MSR_SLAK){}

/* 2. Setting up the Transmission-----*/

CAN1->sTxMailBox[0].TIR = 0;

//Setting up the Std. ID
CAN1->sTxMailBox[0].TIR = (0x245<<21);
CAN1->sTxMailBox[0].TDHR = 0;

// Setting Data Length to 1 Byte.
CAN1->sTxMailBox[0].TDTR = 1;

/* 3. Configuring the Filters-----*/

//Enter Filter Initialization mode to configure the filters
CAN1->FMR |= CAN_FMR_FINIT;

// Configuring the Number of Filters Reserved for CAN1
// and also the start bank for CAN2
CAN1->FMR |= 14<<8;

// Select the single 32-bit scale configuration
CAN1->FS1R |= CAN_FS1R_FSC13;

// Set the receive ID
CAN1->sFilterRegister[13].FR1 = 0x245<<21;

// Select Identifier List mode
CAN1->FM1R |= CAN_FM1R_FBM13;

//Activating filter 14
CAN1->FA1R |= CAN_FA1R_FACT13;

//Exit filter Initialization Mode
CAN1->FMR &= ~CAN_FMR_FINIT;
}

```

Firstly, we enable the clock for `CAN1` by writing a `1` to the `CAN1EN` bit in the `RCC->APB1ENR` register. Then, we enter the *Initialization Mode* for `CAN1` by writing a `1` to the `INRQ` bit in

the `CAN1->MCR` register, and wait for the acknowledgment for entering initialization mode by polling the `INAK` bit in the `CAN1->MSR` register.

### 7.3.13 RCC APB1 peripheral clock enable register (RCC\_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		DAC EN	PWR EN	Reser- ved	CAN2 EN	CAN1 EN	Reser- ved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART 3 EN	USART 2 EN	Reser- ved
		rw	rw			rw		rw		rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw						rw		rw	rw	rw	rw	rw	rw	rw

## 32.9.2 CAN control and status registers

Refer to [Section 2.2 on page 45](#) for a list of abbreviations used in register descriptions.

### CAN master control register (CAN\_MCR)

Address offset: 0x00

Reset value: 0x0001 0002

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															DBF
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESET	Reserved							TTCM	ABOM	AWUM	NART	RFLM	TXFP	SLEEP	INRQ
rs								rw	rw	rw	rw	rw	rw	rw	rw

#### Bit 0 **INRQ**: Initialization request

The software clears this bit to switch the hardware into normal mode. Once 11 consecutive recessive bits have been monitored on the Rx signal the CAN hardware is synchronized and ready for transmission and reception. Hardware signals this event by clearing the `INAK` bit in the `CAN_MSR` register.

Software sets this bit to request the CAN hardware to enter initialization mode. Once software has set the `INRQ` bit, the CAN hardware waits until the current CAN activity (transmission or reception) is completed before entering the initialization mode. Hardware signals this event by setting the `INAK` bit in the `CAN_MSR` register.

## CAN master status register (CAN\_MSR)

Address offset: 0x04

Reset value: 0x0000 0C02

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reserved																
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved.				RX	SAMP	RXM	TXM	Reserved				SLAKI	WKUI	ERRI	SLAK	INAK
				r	r	r	r					rc_w1	rc_w1	rc_w1	r	r

Then, we need to set the *Loopback Mode* as our operating mode. This is done by writing a 1 to the **LBKM** bit in the **CAN1->BTR** register. Then we need to configure the baud rate CAN. For that, I have chosen the baud rate to be **125 Kbps**. CAN supports speeds up to **1 Mbps**. To set up the baud rate we first set the *Re-synchronization* jump width to be 1. This is done by clearing the  **SJW[1:0]** bits in the **CAN1->BTR** register. Since we need to set the value that is one less than the desired value. After that, we set the number of *Time Quanta* for **Time Segment 2** which is 4. This is done by first clearing the  **TS2[2:0]** bits and then writing 011 to them in the **CAN1->BTR** register. After that, we set the **Time Segment 1** to have 3 *Time Quanta* by first clearing the  **TS1[3:0]** bits and then writing 010 to them in the **CAN1->BTR** register. After this is done, we need to set our Prescaler to 16. This is done by writing 15 to the  **BRP[9:0]** bits in the **CAN1->BTR** register.

## CAN bit timing register (CAN\_BTR)

Address offset: 0x1C

Reset value: 0x0123 0000

This register can only be accessed by the software when the CAN hardware is in initialization mode.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	Reserved				SJW[1:0]		Res.	TS2[2:0]			TS1[3:0]			
rw	rw					rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						BRP[9:0]									
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 31 **SILM**: Silent mode (debug)

0: Normal operation

1: Silent Mode

Bit 30 **LBKM**: Loop back mode (debug)

0: Loop Back Mode disabled

1: Loop Back Mode enabled

Bits 29:26 Reserved, must be kept at reset value.

Bits 25:24 **SJW[1:0]**: Resynchronization jump width

These bits define the maximum number of time quanta the CAN hardware is allowed to lengthen or shorten a bit to perform the resynchronization.

$$t_{RJW} = t_q \times (SJW[1:0] + 1)$$

Bit 23 Reserved, must be kept at reset value.

Bits 22:20 **TS2[2:0]**: Time segment 2

These bits define the number of time quanta in Time Segment 2.

$$t_{BS2} = t_q \times (TS2[2:0] + 1)$$

Bits 19:16 **TS1[3:0]**: Time segment 1

These bits define the number of time quanta in Time Segment 1

$$t_{BS1} = t_q \times (TS1[3:0] + 1)$$

For more information on bit timing refer to [Section 32.7.7](#).

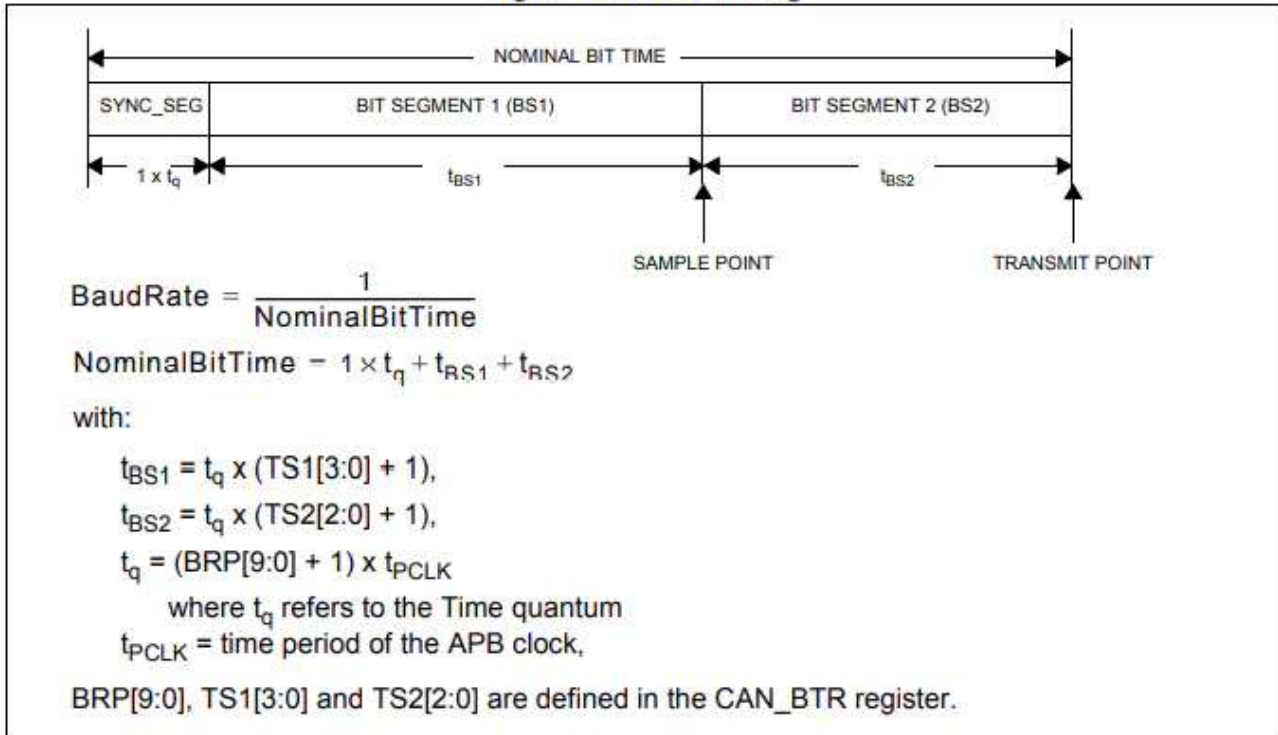
Bits 15:10 Reserved, must be kept at reset value.

Bits 9:0 **BRP[9:0]**: Baud rate prescaler

These bits define the length of a time quanta.

$$t_q = (BRP[9:0] + 1) \times t_{PCLK}$$

**Figure 346. Bit timing**



From the formulas given in the above figure, we will see how our settings result in a baud rate of **125 Kbps**. Or **APB1** clock is running at **16MHz** so,

$$t_{PCLK} = 1/16MHz = 62.5ns = t_p$$

So,

$$t_q = (\text{BRP}[9 : 0] + 1) * (t_p)$$

$$t_q = (15 + 1) * (62.5ns) = 1000ns = t_q$$

And,

$$t_{BS1} = t_q * (\text{TS1}[3 : 0] + 1)$$

$$t_{BS1} = 1000 * 3 = 3000ns$$

$$t_{BS2} = t_q * (\text{TS2}[2 : 0] + 1)$$

$$t_{BS2} = 1000 * 4 = 4000ns$$

Therefore,

$$\text{NominalBitTime} = t_q + t_{BS1} + t_{BS2}$$

$$\text{NominalBitTime} = 1000 + 3000 + 4000 = 8000ns$$

Hence,



$$\text{BaudRate} = 1/\text{NominalBitTime} = 1/8000ns = 125000\text{bits/s}$$

Then, we first exit the *Initialization mode* which is similar to the way we entered initialization mode, the only difference being here, we clear the `INRQ` bit in the `CAN1->MCR` register. Like this, we also exit *sleep mode* to *wake up* CAN. This is done by clearing the `SLEEP` bit in the `CAN1->MCR` register.

Now, we need to set up the Transmission Mailbox registers. Here, we chose the mailbox zero, and set its `TIR` register to `0`. This is done by the line `CAN1->sTxMailBox[0].TIR = 0`. Then we configure our identifier, I have chosen the identifier to be `0x245`. So, we write this value to the `STID[10:0]` bits in the `CAN1->sTxMailbox[0].TIR` register. Then we set the `CAN1->sTxMailBox[0].TDHR` to `0`. We also need to set the number of bytes that need to be transmitted. In our case, we need to transmit one byte, so we write `1` to the `CAN1->sTxMailBox[0].TDTR` register.

#### CAN mailbox data high register (CAN\_TDHxR) (x=0..2)

All bits of this register are write protected when the mailbox is not in empty state.

Address offsets: 0x18C, 0x19C, 0x1AC

Reset value: 0xFFFF XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA7[7:0]								DATA6[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA5[7:0]								DATA4[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

#### CAN mailbox data length control and time stamp register (CAN\_TDTxR) (x=0..2)

All bits of this register are write protected when the mailbox is not in empty state.

Address offsets: 0x184, 0x194, 0x1A4

Reset value: 0xFFFF XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TIME[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							TGT	Reserved				DLC[3:0]			
							rw					rw	rw	rw	

The last step in the CAN initialization process is to configure the filters. For this, we need to enter the *Filter initialization*. This is done by writing a `1` to the `FINIT` bit in the `CAN1->FMR` register. There are a total of 28 filter banks available for CAN. These are split between `CAN1` and `CAN2`. So, we need to set up this split and also specify the bank



number from which the filter banks for CAN2 start. Here, I have assigned 14 filter banks to CAN1 and the remaining 14 to CAN2. Since we are not using CAN2 here, the filter banks associated with it will not matter. The *Start Bank* for CAN2 or the *End bank* for CAN1 is defined by writing 14 to the CAN2SB[5:0] bits in the CAN1->FMR register. After that, out of the 14 available banks I have used filter number 13 here. Then, we select the *32-bit Scale Configuration* for the filters. This is done by writing a 1 to the FSC13 bit (signifying the 13th filter bank) in the CAN1->FS1R register. To set up the filter for discarding and keeping the messages we write a value of 0x245<<21 in the CAN1->sFilterRegister[13].FR1. This is the same as our transmission Identifier. Since we want all our Identifier bits to match while message reception is done, we choose the *Identifier List Mode*. This is set by writing a 1 to the FBM13 bit in the CAN1->FM1R register.

### 32.9.4 CAN filter registers

#### CAN filter master register (CAN\_FMR)

Address offset: 0x200

Reset value: 0x2A1C 0E01

All bits of this register are set and cleared by software.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CAN2SB[5:0]						Reserved						FINIT	
		rw	rw	rw	rw	rw	rw							rw	

Bits 31:14 Reserved, must be kept at reset value.

Bits 13:8 **CAN2SB[5:0]**: CAN2 start bank

These bits are set and cleared by software. They define the start bank for the CAN2 interface (Slave) in the range 0 to 27.

*Note: When CAN2SB[5:0] = 28d, all the filters to CAN1 can be used.*

*When CAN2SB[5:0] is set to 0, no filters are assigned to CAN1.*

Bits 7:1 Reserved, must be kept at reset value.

Bit 0 **FINIT**: Filter init mode

Initialization mode for filter banks

0: Active filters mode.

1: Initialization mode for the filters.

### CAN filter scale register (CAN\_FS1R)

Address offset: 0x20C

Reset value: 0x0000 0000

This register can be written only when the filter initialization mode is set (FINIT=1) in the CAN\_FMR register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
				r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FSC15	FSC14	FSC13	FSC12	FSC11	FSC10	FSC9	FSC8	FSC7	FSC6	FSC5	FSC4	FSC3	FSC2	FSC1	FSC0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FSCx**: Filter scale configuration

These bits define the scale configuration of Filters 13-0.

0: Dual 16-bit scale configuration

1: Single 32-bit scale configuration

### CAN filter mode register (CAN\_FM1R)

Address offset: 0x204

Reset value: 0x0000 0000

This register can be written only when the filter initialization mode is set (FINIT=1) in the CAN\_FMR register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
				r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FBM15	FBM14	FBM13	FBM12	FBM11	FBM10	FBM9	FBM8	FBM7	FBM6	FBM5	FBM4	FBM3	FBM2	FBM1	FBM0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

**Note:** Refer to [Figure 342](#).

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FBMx**: Filter mode

Mode of the registers of Filter x.

0: Two 32-bit registers of filter bank x are in Identifier Mask mode.

1: Two 32-bit registers of filter bank x are in Identifier List mode.

Then, the filter 13 needs to be activated by writing a 1 to the FACT13 bit in the CAN1->FA1R register. After that, we exit the *Filter initialization* by clearing the FINIT bit in the CAN1->FMR register. Since we did not assign any value to the FFA13 bit in the CAN1->FFA1R register it remains at its default value, that is 0, and hence, the received message gets stored in FIFO 0.

## CAN filter activation register (CAN\_FA1R)

Address offset: 0x21C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FACT27	FACT26	FACT25	FACT24	FACT23	FACT22	FACT21	FACT20	FACT19	FACT18	FACT17	FACT16
				rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FACT15	FACT14	FACT13	FACT12	FACT11	FACT10	FACT9	FACT8	FACT7	FACT6	FACT5	FACT4	FACT3	FACT2	FACT1	FACT0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FACTx**: Filter active

The software sets this bit to activate Filter x. To modify the Filter x registers (CAN\_FxR[0:7]), the FACTx bit must be cleared or the FINIT bit of the CAN\_FMR register must be set.

0: Filter x is not active

1: Filter x is active

## CAN filter FIFO assignment register (CAN\_FFA1R)

Address offset: 0x214

Reset value: 0x0000 0000

This register can be written only when the filter initialization mode is set (FINIT=1) in the CAN\_FMR register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
				rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FFA15	FFA14	FFA13	FFA12	FFA11	FFA10	FFA9	FFA8	FFA7	FFA6	FFA5	FFA4	FFA3	FFA2	FFA1	FFA0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FFAx**: Filter FIFO assignment for filter x

The message passing through this filter will be stored in the specified FIFO.

0: Filter assigned to FIFO 0

1: Filter assigned to FIFO 1

- Message Transmission is handled by the `void CAN1_Tx(uint8_t tr)` function,

COPY 

```
void CAN1_Tx(uint8_t tr){
    // Put the Data to be transmitted into Mailbox Data Low Register
    CAN1->sTxMailBox[0].TDLR = tr;

    // Request for Transmission
    CAN1->sTxMailBox[0].TIR |= 1;
}
```

For this, we put the data provided by the user into the `CAN1->sTxMailBox[0].TDLR` register. Then, a transmission request is raised by writing a `1` to the `TXRQ` bit in the `CAN1->sTxMailBox[0].TIR` register.

### CAN TX mailbox identifier register (CAN\_TlR) (x=0..2)

Address offsets: 0x180, 0x190, 0x1A0

Reset value: 0xFFFF XXXX (except bit 0, TXRQ = 0)

All TX registers are write protected when the mailbox is pending transmission (TMEx reset).

This register also implements the TX request control (bit 0) - reset value 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
STID[10:0]/EXID[28:18]											EXID[17:13]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXID[12:0]													IDE	RTR	TXRQ
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:21 **STID[10:0]/EXID[28:18]**: Standard identifier or extended identifier

The standard identifier or the MSBs of the extended identifier (depending on the IDE bit value).

Bits 20:3 **EXID[17:0]**: Extended identifier

The LSBs of the extended identifier.

Bit 2 **IDE**: Identifier extension

This bit defines the identifier type of message in the mailbox.

0: Standard identifier.

1: Extended identifier.

Bit 1 **RTR**: Remote transmission request

0: Data frame

1: Remote frame

Bit 0 **TXRQ**: Transmit mailbox request

Set by software to request the transmission for the corresponding mailbox.

Cleared by hardware when the mailbox becomes empty.

- For message reception, the `uint8_t CAN1_Rx()` function is used,

```
uint8_t CAN1_Rx(){
    // Monitoring FIFO 0 message pending bits FMP0[1:0]
    while(!(CAN1->RF0R & 3)){

        // Read the data from the FIFO 0 mailbox from Mailbox data low register
        uint8_t RxD = (CAN1->sFIFOMailBox[0].RDLR) & 0xFF;
        rec = RxD;

        // Releasing FIFO 0 output mailbox
        CAN1->RF0R |= 1<<5;
```

COPY 



```

return RxD;
}

```

Here, we first check if there is a message pending in `FIFO 0` by polling the `FMP0[1:0]` bits in the `CAN1->RF0R` register. After this, we declare a local variable `uint8_t RxD` that reads the received value from the `CAN1->sFIFOMailBox[0].RDLR` register. A Bitwise AND operation of the received value and `0xFF` is performed to get the final value. Then, we release the `FIFO 0` mailbox by writing a `1` to the `RF0M0` bit in the `CAN1->RF0R` register, and we return the received value.

### CAN receive FIFO 0 register (CAN\_RF0R)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										RF0M0	FOVR0	FULL0	Res.	FMP0[1:0]	
										rs	rc_w1	rc_w1		r	r

### CAN receive FIFO mailbox data low register (CAN\_RDLxR) (x=0..1)

All bits of this register are write protected when the mailbox is not in empty state.

Address offsets: 0x1B8, 0x1C8

Reset value: 0xFFFF XXXX

All RX registers are write protected.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]								DATA2[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA1[7:0]								DATA0[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:24 **DATA3[7:0]**: Data Byte 3

Data byte 3 of the message.

Bits 23:16 **DATA2[7:0]**: Data Byte 2

Data byte 2 of the message.

Bits 15:8 **DATA1[7:0]**: Data Byte 1

Data byte 1 of the message.

Bits 7:0 **DATA0[7:0]**: Data Byte 0

Data byte 0 of the message.

A message can contain from 0 to 8 data bytes and starts with byte 0.

- The delay is provided by the `void TIM4_ms_Delay(uint32_t delay)` function,

```
void TIM4_ms_Delay(uint32_t delay){
    RCC->APB1ENR |= 1<<2; //Start the clock for the timer peripheral
    TIM4->PSC = 16000-1; //Setting the clock frequency to 1kHz.
    TIM4->ARR = (delay); // Total period of the timer
    TIM4->CNT = 0;
    TIM4->CR1 |= 1; //Start the Timer
    while(!(TIM4->SR & TIM_SR_UIF)){ } //Polling the update interrupt flag
    TIM4->SR &= ~(0x0001); //Reset the update interrupt flag
}
```

Here, we enable the clock for the *Timer 4*, and set the *Pre-scalar* to `16000-1` to reduce the clock speed to *1 KHz*, which means one clock *tick* corresponds to *1ms*. After that, we assign the desired delay value to the `TIM2->ARR` register, initialize the count to zero and start the timer. Then we poll the `UIF` or the *Update Interrupt Flag*, which is set after the desired amount of clock ticks provided in the `TIM2->ARR` register are done, or in other words when the timer overflows. After the `UIF` flag is set we clear it from the `TIM2->SR` register and exit the function.

### 7.3.13 RCC APB1 peripheral clock enable register (RCC\_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

[illegible]

#### 18.4.5 TIMx status register (TIMx\_SR)

Address offset: 0x10

Reset value: 0x0000

[illegible]



#### Bit 0 UIF: Update interrupt flag

- " This bit is set by hardware on an update event. It is cleared by software.  
0: No update occurred.  
1: Update interrupt pending. This bit is set by hardware when the registers are updated:
- " At overflow or underflow (for TIM2 to TIM5) and if UDIS=0 in the TIMx\_CR1 register.
- " When CNT is reinitialized by software using the UG bit in TIMx\_EGR register, if URS=0 and UDIS=0 in the TIMx\_CR1 register.  
When CNT is reinitialized by a trigger event (refer to the synchro control register description), if URS=0 and UDIS=0 in the TIMx\_CR1 register.

### 18.4.1 TIMx control register 1 (TIMx\_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

## 4. Writing the main Function

```
int main(void){
    GPIO_Init();
    CAN1_Init();
    while(1){
        CAN1_Tx(k);
        rec= CAN1_Rx();
        k += 1;
        if (k>25)
            k = 0;
        TIM4_ms_Delay(1000);
    }
}
```

COPY 

In the `main` function, we first configure our `GPIO` pins by calling the `GPIO_Init()` function. Then we initialize the `CAN1` peripheral by calling the `CAN1_Init()` function. After that in a `while` loop, we continuously transmit a range of numbers from `1` to `25`. The variable `k` that we declared earlier is passed as a parameter to the `CAN1_Tx(k)` function, and the variable `rec` is used to store the received value when `CAN1_Rx()` is called. In each iteration, we increment `k` by `1` and if it becomes larger than `25`, we set it back to `0`.

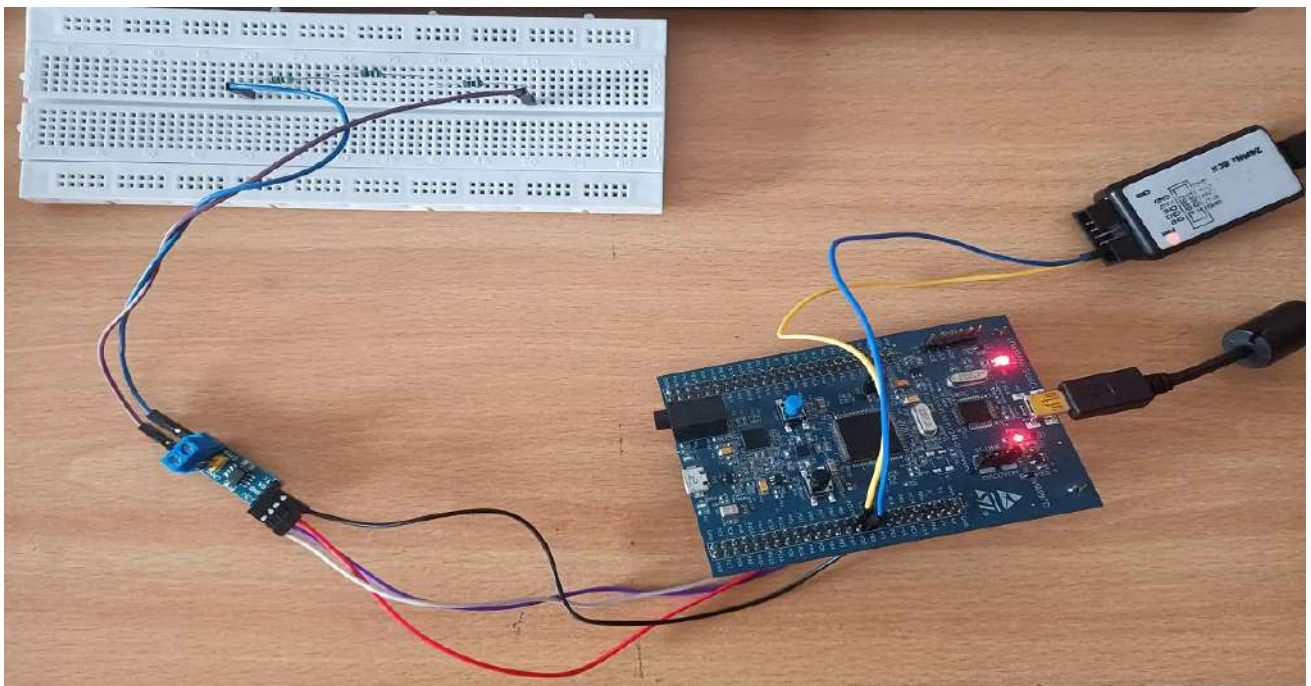
After that, a delay of 1 second or 1000ms is provided using the `TIM4_ms_Delay(1000)` function.

This process repeats indefinitely. This completes the *Loopback* test on the CAN peripheral.

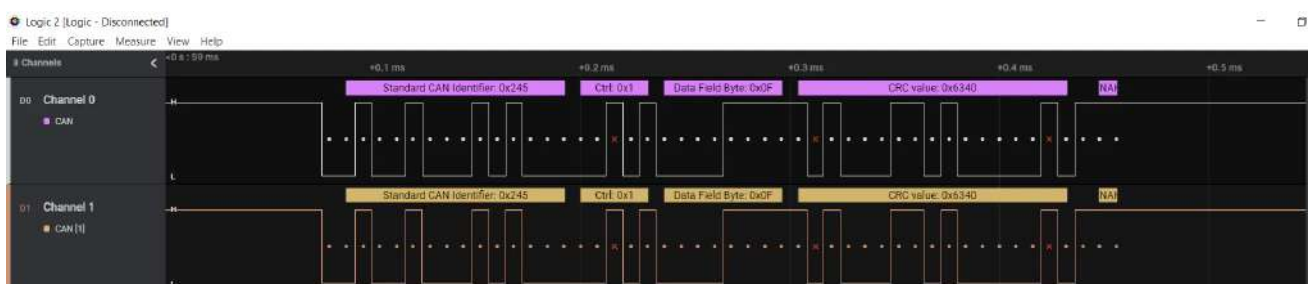
## Conclusion

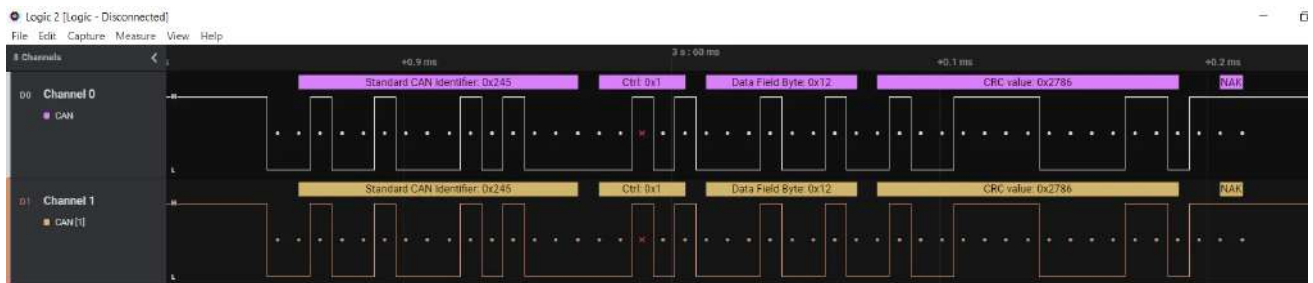
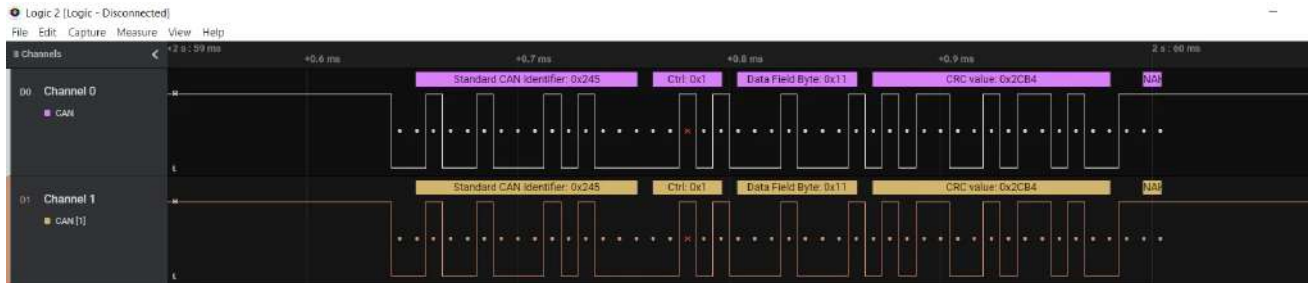
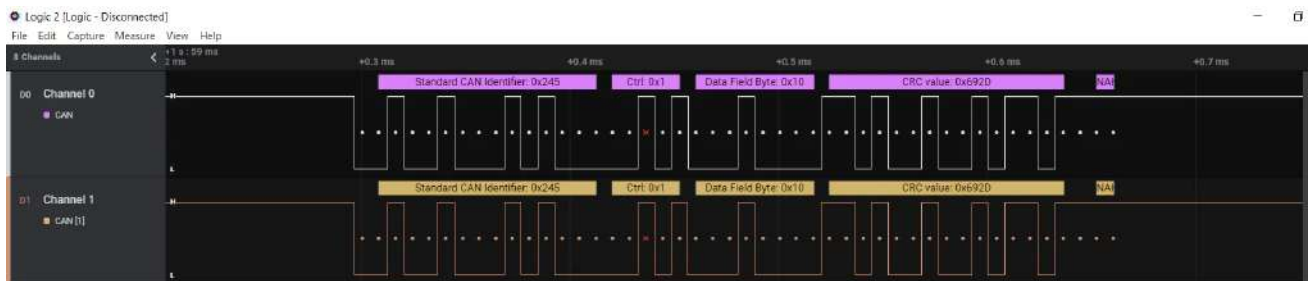
This article provided an overview of the process that is to be used to perform the *Loopback* test on the CAN peripheral on the *STM32F407 Discovery Kit*.

I hope you found this article useful and easy to follow along 😊. All the codes used here are present in my GitHub repository, which you can access by clicking [here](#).



The following pictures show the various parts of the **CAN Frame** captured through a *Logic Analyzer*. The *Channel 0* is connected to *CAN Tx* and *Channel 1* is connected to *CAN Rx*.





#embedded

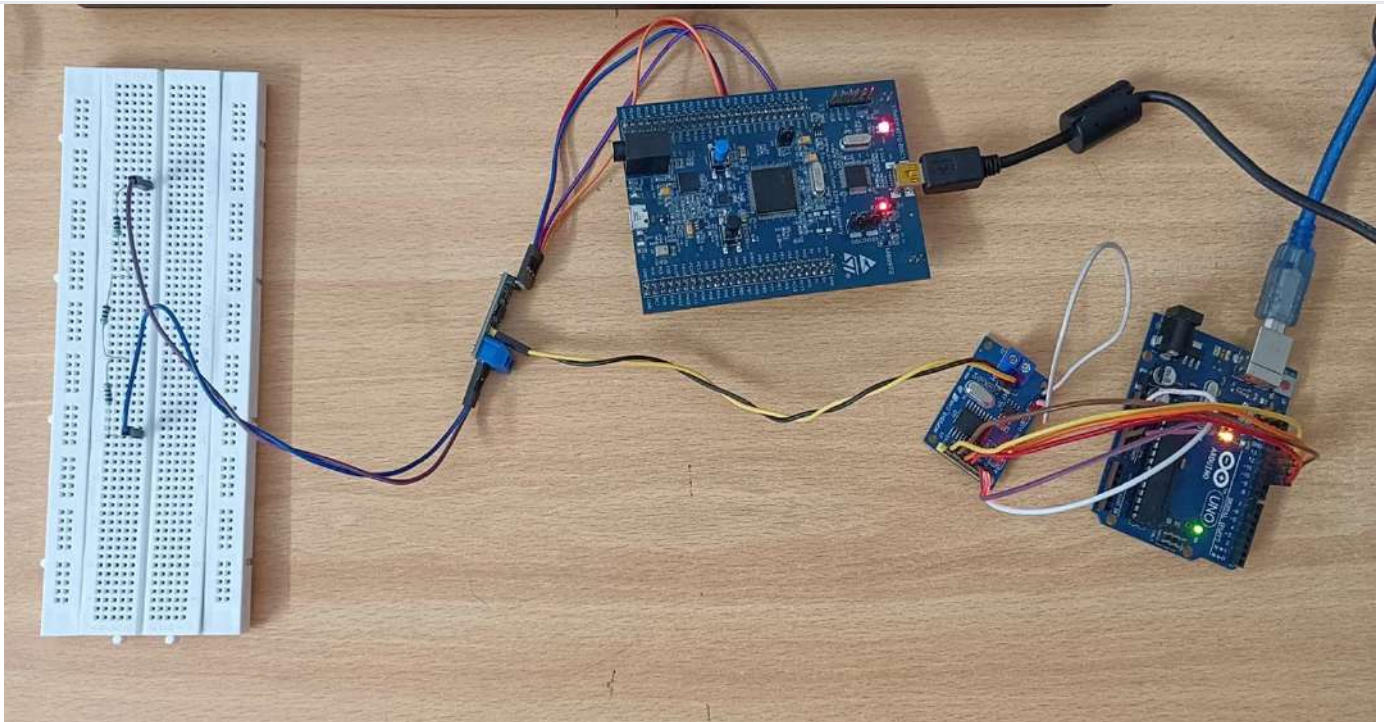


😊 Like

READ NEXT



## Embedded System Projects with STM32F407



# Establishing Communication Between an Arduino UNO and the STM32F407 Discovery Kit using the CAN Protocol



**Ruturaj A. Nanoti**

Published on Nov 2, 2021



16 min read

### Introduction

Hi!! 😊 In this article we will continue our discussion on the **CAN (Controller Area Network) Protocol**. In this blog, I will be demonstrating the working of the CAN peripheral on the *STM32F407 Discovery Kit* in the **Normal Mode**. For the purposes of this project, I have used an *Arduino UNO* as the second node on the *CAN Bus*. We will be trying to get these two boards to transmit and receive information from one another. Here, I have set the *STM32F407 Discovery Kit* to be the transmitting node and the *Arduino UNO* to receive the messages sent by the former. Although each node can transmit and receive at the same time, both the nodes have been restricted to a single function for simplicity. The simultaneous reception and transmission on a node can be implemented in a very similar way, as is described in the further sections.

## Working Principle

- **CAN Protocol**

In my last blog, I focused on the **Loopback Mode** using the CAN peripheral, in which I have explained the basics of the CAN protocol, various message frames, the elements of the message frames, and the filter configurations. So, I would recommend you to go through it, before continuing with this article, if the concept of CAN is new to you. You can read it by clicking [here](#).

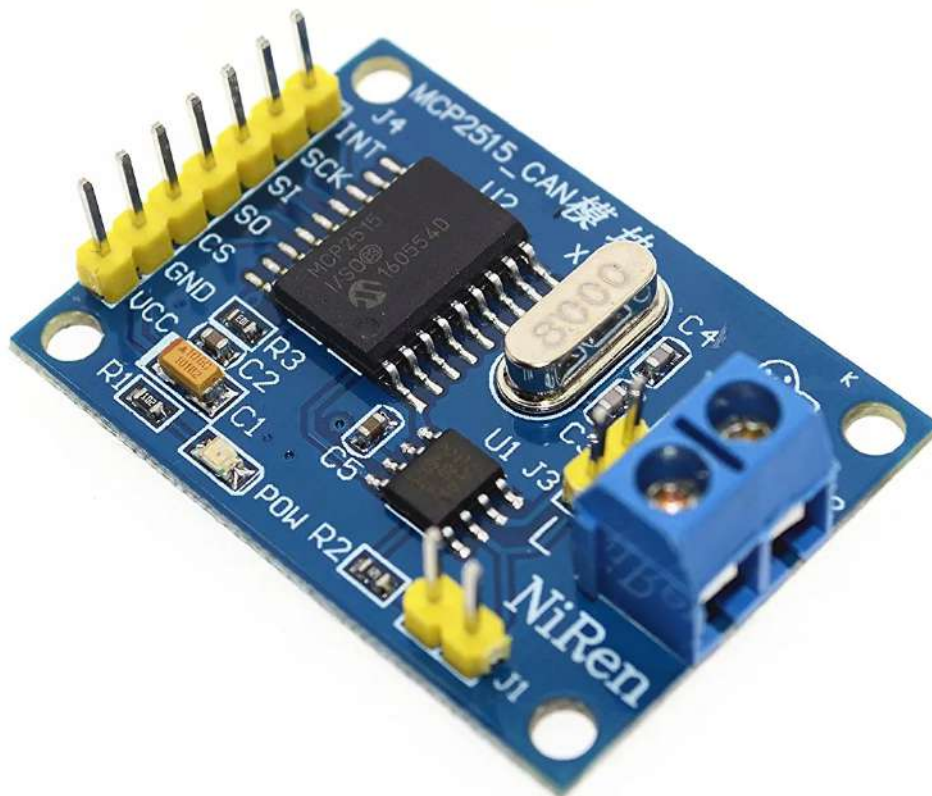
Now, let's understand how the CAN Bus moderates the various messages sent by the different nodes. This process is known as **Arbitration**. It happens based on the **Identifier Field** present in the *CAN Message*. By doing this, all the nodes agree on which message needs to be transmitted first, or which node will be utilizing the bus. During this, all the other nodes must remain silent or idle, except the node that has won the arbitration. Here, *smaller the message identifier, higher the message priority*. A **dominant** bit (logic 0) overrides a **recessive** bit (logic 1), that is if the bus is in a dominant state (i.e at logic 0), a single node cannot pull the bus to a recessive state. This can only be done if all the nodes pull the bus up to the recessive state simultaneously. Hence, the name *dominant bit*.

A CAN node monitors the bus while transmitting a message, if it detects a **dominant** bit when it has transmitted a **recessive bit**, it will immediately stop transmission and become a receiver. The node that wins the transmission at the end is the one that transmits and detects the same bit while monitoring the bus. This node continues to transmit its message, while the other nodes have to wait for the bus to get idle again for re-transmitting. This is how transmission conflicts are resolved during CAN transmission. This helps maintain error-free and efficient communication between the nodes.

- **MCP2515 and TJA1050: CAN Controller and Transceiver**

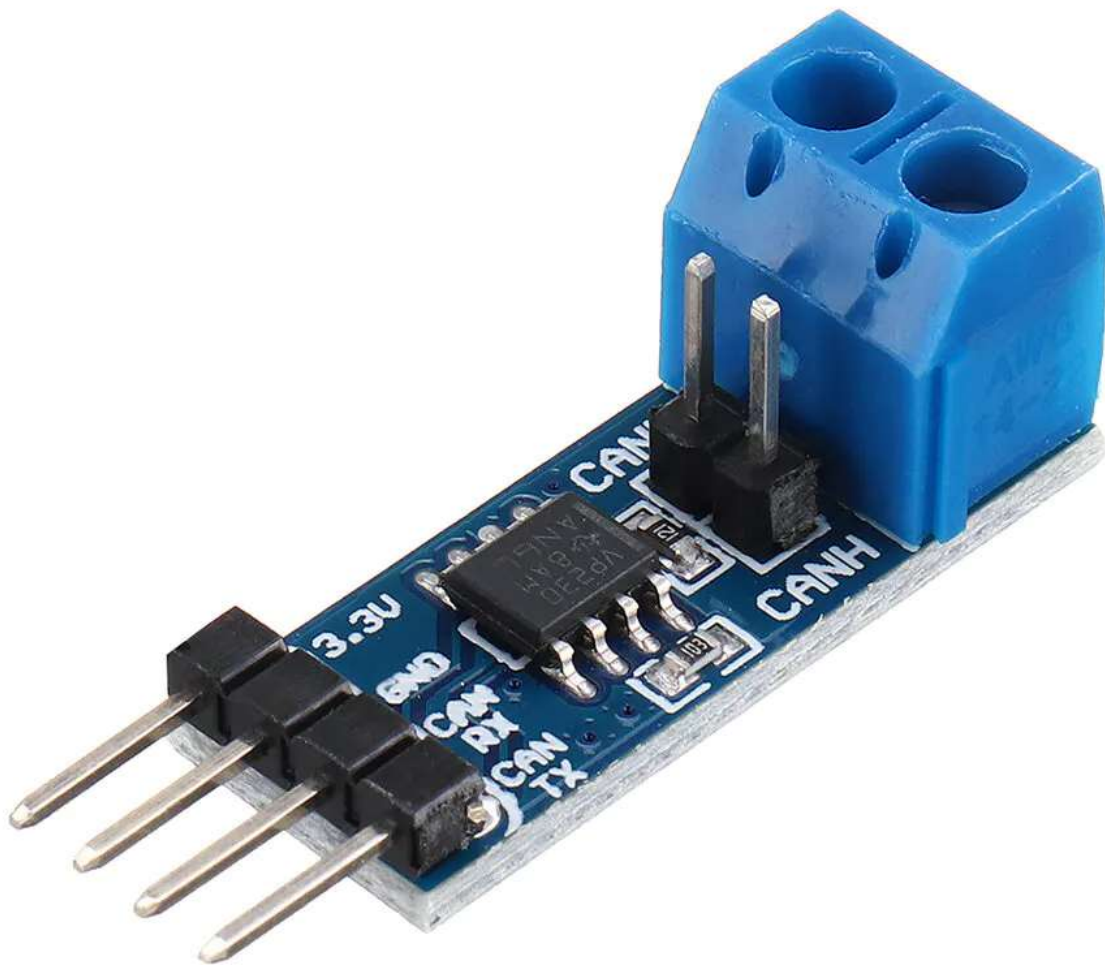
The *ATMega 328* microcontroller present on the *Arduino UNO* does not have a CAN controller present on-board. So, we need to connect the CAN controller as well as the *CAN Transceiver* externally. We interface the *MCP2515 (CAN Controller)* with the *ATMega 328* using the *SPI Protocol*. I will be discussing the *SPI Protocol* in detail, in a later blog. For the purposes of this project, *SPI* is just another synchronous serial communication protocol that can be configured to work as a 4-wire or a 3-wire protocol. It is a *Single Master Multi Slave* full-duplex type of protocol. The various lines used are, **MISO (Master In Slave Out)**, **MOSI (Master Out Slave In)**, **SCK (Serial Data Clock)**, and **CS (Chip Select)**. Each slave is accessed by the CS line.

In this project, I have used a breakout board containing both the *MCP2515* and *TJA1050*. Also, this CAN module has an onboard 120-ohm bus termination, which can be enabled by shorting the terminals *J1* terminals using a jumper wire.



Since the *STM32F407 Discovery Kit* already has a CAN controller, we just need to connect a transceiver externally. Here, I have used the *SN65HVD230 CAN Transceiver*.





The connections that need to be made are given in my GitHub repository, which can be accessed by clicking [here](#).

## Components Required

- An STM32F4 based MCU
- An Arduino UNO
- CAN Module containing CAN Transceiver and CAN Controller
- A SN65HVD230 Breakout board


- A 120-ohm resistor OR any combination of resistors that results in 120-ohms ( I have used two 10-ohm resistors and a 100-ohm resistor and connected them in series to get 120-ohms).
- A Logic Analyzer
- A Few Jumper Wires
- A breadboard

These are the components required to get started with this project. Let's dive into the code now.

## Programming the STM32F407

The code used here is very similar to the one I have used in my last blog on the **Loopback Test** on the CAN Peripheral. There are very few changes, but I will be explaining the code here also. Although we are only transmitting from this node, I have written the function for receiving data and the filters also have been configured. Here, I am trying to transmit the numbers from 1-25 from the STM32F407 to the Arduino UNO.

### 1. Including the necessary Header Files and some comments about the code

COPY 

```

/*****
* Configurations are made as follows:
* PB8 - CAN1_RX (Need to be set in AF9 for CAN1/CAN2)
* PB9 - CAN1_TX (Need to be set in AF9 for CAN1/CAN2)
*
* The following is a bare-metal code written for CAN Normal Communication test
* on the Arm Cortex-M4 based STM32F407 Discovery Kit.
*
* @File    main.c
* @Author  Ruturaj A. Nanoti
* @Date    4 October, 2021
*****/

#include "stm32f4xx.h"
#define ARM_MATH_CM4

```

These are the header files included at the top of our `main.c` file. The `stm32f4xx.h` is the one for our microcontroller and the `ARM_MATH_CM4` is for any math operations that we perform in our code.

## 2. Declaring the User-Defined Functions and variables

```
void GPIO_Init(void);
void CAN1_Init(void);
void CAN1_Tx(uint8_t tr);
uint8_t CAN1_Rx(void);
void TIM4_ms_Delay(uint32_t delay);
uint8_t k = 0;
uint8_t rec = 0;
```

COPY 

The `void GPIO_Init(void)` function is used to initialize and configure the `GPIO` pins that we will be needing to connect our SN65HVD230 module and use the CAN Peripheral. The `void CAN1_Init(void)` and the `void CAN1_Tx(uint8_t tr)` functions are used to initialize `CAN1` and transmit a byte of data respectively. The `uint8_t CAN1_Rx(void)` function is used to receive the incoming data. `void TIM4_ms_Delay(uint32_t delay)` acts as our delay function which utilizes the `TIM4` peripheral and provides a delay in *milliseconds*. Finally, we declare the variables `uint8_t k` and `uint8_t rec` which will use to transmit and receive data respectively.

## 3. Defining the User-Defined Functions

- Let's configure the GPIO pins using the `void GPIO_Init()` function,

```
void GPIO_Init(){
    // Enable GPIOA clock signal
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;

    // Configuring PB8 and PB9 in alternate function mode
    GPIOB->MODER |= (GPIO_MODER_MODER8_1 | GPIO_MODER_MODER9_1);

    // Selecting AF9 for PB8 and PB9 (See Page 272 of dm00031020)
    GPIOB->AFR[1] |= (GPIO_AFRH_AFSEL8_3 | GPIO_AFRH_AFSEL8_0 |
```

COPY 

```
GPIO_AFRH_AFSEL9_0 | GPIO_AFRH_AFSEL9_3);
}
```

First, we enable the `GPIOA` clock by writing a `1` to the `GPIOBEN` bit in the `RCC->AHB1ENR` register.

## Reset and clock control for STM32F42xxx and STM32F43xxx (RCC)

RM0090

### 6.3.10 RCC AHB1 peripheral clock register (RCC\_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	OTGHS ULPIEN	OTGHS SEN	ETHMACPT EN	ETHMACRX EN	ETHMACTX EN	ETHMACEN	Res.	DMA2DEN	DMA2EN	DMA1EN	CCMDAT ARAMEN	Res.	BKPSRAM EN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCE N	Res.	GPIOKEN	GPIOJ EN	GPIOIE N	GPIOH EN	GPIOG EN	GPIOFE N	GPIOEEN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Then, we set `PB8` and `PB9` in *Alternate-Function Mode*. This is done by writing `10` to the `MODER8[1:0]` and `MODER9[1:0]` bits in the `GPIOB->MODER` register.

## 8.4 GPIO registers

This section gives a detailed description of the GPIO registers.

For a summary of register bits, register address offsets and reset values, refer to [Table 39](#).

The GPIO registers can be accessed by byte (8 bits), half-words (16 bits) or words (32 bits).

### 8.4.1 GPIO port mode register (GPIOx\_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

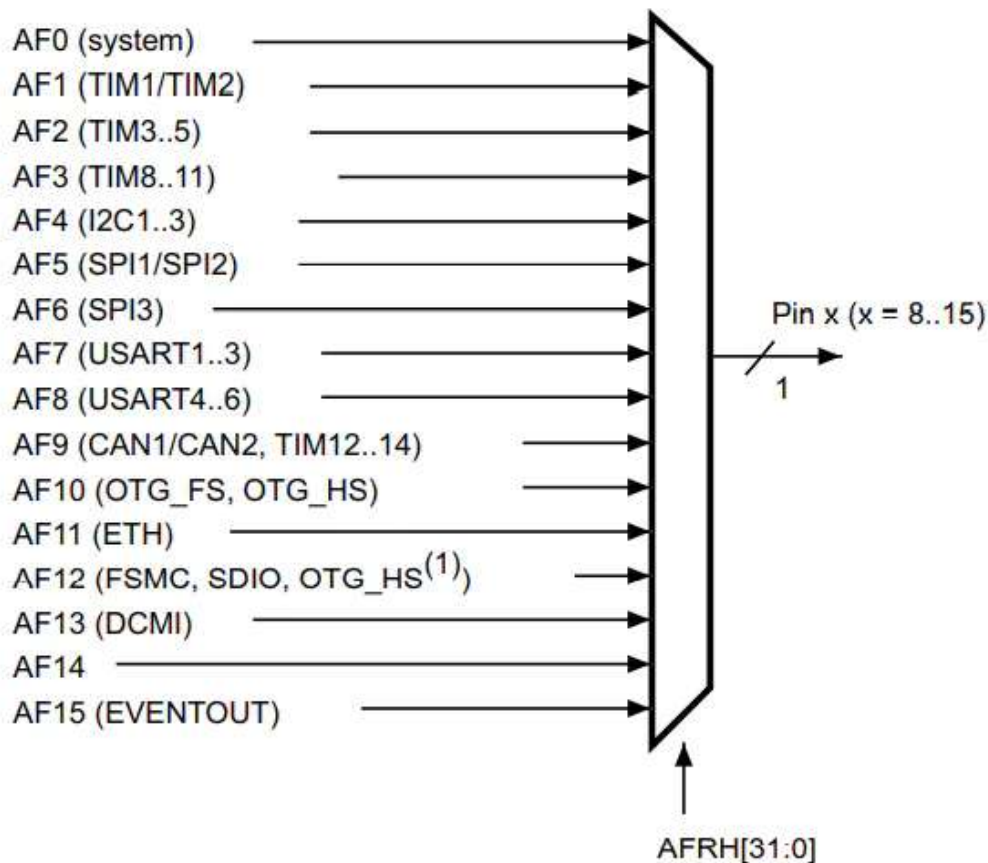
01: General purpose output mode

10: Alternate function mode

11: Analog mode

We need to select AF9 for the CAN1 peripheral. This is done by writing 1001 to the AFRH8[3:0] and the AFRH9[3:0] bits in the GPIOB->AFR[1] register.

For pins 8 to 15, the GPIOx\_AFRH[31:0] register selects the dedicated alternate function



#### 8.4.10 GPIO alternate function high register (GPIOx\_AFRH) (x = A..I/J)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:0 **AFRHy**: Alternate function selection for port x bit y (y = 8..15)

These bits are written by software to configure alternate function I/Os

AFRHy selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15



- Then, we come to our `void CAN1_Init()` where we initialize and configure the `CAN1` peripheral,

COPY 

```
void CAN1_Init(){
    /* 1. Setting Up the Baud Rate and Configuring CAN1 in
     * Normal Mode -----*/

    // Enable clock for CAN1
    RCC->APB1ENR |= RCC_APB1ENR_CAN1EN;

    // Entering CAN Initialization Mode and wait for acknowledgment
    CAN1->MCR |= CAN_MCR_INRQ;
    while (!(CAN1->MSR & CAN_MSR_INAK)){

    //Exit Sleep Mode
    CAN1->MCR &= ~CAN_MCR_SLEEP;
    while (CAN1->MSR & CAN_MSR_SLAK){}

    //Set Loop back mode for CAN1
    //CAN1->BTR |= CAN_BTR_LBKM;

    //Setting the Re synchronization jump width to 1
    CAN1->BTR &= ~CAN_BTR_SJW;

    //Setting the no. of time quanta for Time segment 2
    // TS2 = 4-1;
    CAN1->BTR &= ~(CAN_BTR_TS2);
    CAN1->BTR |= (CAN_BTR_TS2_1 | CAN_BTR_TS2_0);

    //Setting the no. of time quanta for Time segment 1
    // TS1 = 3-1;
    CAN1->BTR &= ~(CAN_BTR_TS1);
    CAN1->BTR |= (CAN_BTR_TS1_1);

    //Setting the Baud rate Pre-scalar for CAN1
    // BRP[9:0] = 16-1
    CAN1->BTR |= ((16-1)<<0);

    /* 2. Configuring the Filters-----*/
    //Enter Filter Initialization mode to configure the filters
    CAN1->FMR |= CAN_FMR_FINIT;
```

```

// Configuring the Number of Filters Reserved for CAN1
// and also the start bank for CAN2
CAN1->FMR |= 14<<8;

// Select the single 32-bit scale configuration
CAN1->FS1R |= CAN_FS1R_FSC13;

// Set the receive ID
CAN1->sFilterRegister[13].FR1 = 0x103<<21;

// Select Identifier List mode
CAN1->FM1R |= CAN_FM1R_FBM13;

//Activating filter 13
CAN1->FA1R |= CAN_FA1R_FACT13;

//Exit filter Initialization Mode
CAN1->FMR &= ~CAN_FMR_FINIT;

/* 3. Setting up the Transmission-----*/

CAN1->sTxMailBox[0].TIR = 0;

//Setting up the Std. ID
CAN1->sTxMailBox[0].TIR = (0x245<<21);
CAN1->sTxMailBox[0].TDHR = 0;

// Setting Data Length to 1 Byte.
CAN1->sTxMailBox[0].TDTR = 1;

// Exit the Initialization mode for CAN1
// Wait until the INAK bit is cleared by hardware
CAN1->MCR &= ~CAN_MCR_INRQ;
while (CAN1->MSR & CAN_MSR_INAK){}

}

```

Firstly, we enable the clock for `CAN1` by writing a `1` to the `CAN1EN` bit in the `RCC->APB1ENR` register. Then, we enter the *Initialization Mode* for `CAN1` by writing a `1` to the `INRQ` bit in

the `CAN1->MCR` register, and wait for the acknowledgment for entering initialization mode by polling the `INAK` bit in the `CAN1->MSR` register.

### 7.3.13 RCC APB1 peripheral clock enable register (RCC\_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		DAC EN	PWR EN	Reser- ved	CAN2 EN	CAN1 EN	Reser- ved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART 3 EN	USART 2 EN	Reser- ved
		rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

## 32.9.2 CAN control and status registers

Refer to [Section 2.2 on page 45](#) for a list of abbreviations used in register descriptions.

### CAN master control register (CAN\_MCR)

Address offset: 0x00

Reset value: 0x0001 0002

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															DBF
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESET	Reserved							TTCM	ABOM	AWUM	NART	RFLM	TXFP	SLEEP	INRQ
rs								rw	rw	rw	rw	rw	rw	rw	rw

#### Bit 0 **INRQ**: Initialization request

The software clears this bit to switch the hardware into normal mode. Once 11 consecutive recessive bits have been monitored on the Rx signal the CAN hardware is synchronized and ready for transmission and reception. Hardware signals this event by clearing the `INAK` bit in the `CAN_MSR` register.

Software sets this bit to request the CAN hardware to enter initialization mode. Once software has set the `INRQ` bit, the CAN hardware waits until the current CAN activity (transmission or reception) is completed before entering the initialization mode. Hardware signals this event by setting the `INAK` bit in the `CAN_MSR` register.

## CAN master status register (CAN\_MSR)

Address offset: 0x04

Reset value: 0x0000 0C02

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reserved																
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved.				RX	SAMP	RXM	TXM	Reserved				SLAKI	WKUI	ERRI	SLAK	INAK
				r	r	r	r					rc_w1	rc_w1	rc_w1	r	r

Then, we need to set our operating mode as **Normal**, which is automatically done after we exit the sleep mode of the CAN peripheral. We exit *sleep mode* to *wake up* CAN. This is done by clearing the **SLEEP** bit in the **CAN1->MCR** register. **CAN Loopback Mode** can be set by the following step. This is done by writing a **1** to the **LBKM** bit in the **CAN1->BTR** register. Then we need to configure the baud rate CAN. For that, I have chosen the baud rate to be **125 Kbps**. CAN supports speeds up to **1 Mbps**. To set up the baud rate we first set the *Re-synchronization* jump width to be **1**. This is done by clearing the **SJW[1:0]** bits in the **CAN1->BTR** register. Since we need to set the value that is one less than the desired value. After that, we set the number of *Time Quanta* for **Time Segment 2** which is **4**. This is done by first clearing the **TS2[2:0]** bits and then writing **011** to them in the **CAN1->BTR** register. After that, we set the **Time Segment 1** to have **3** *Time Quanta* by first clearing the **TS1[3:0]** bits and then writing **010** to them in the **CAN1->BTR** register. After this is done, we need to set our Prescaler to **16**. This is done by writing **15** to the **BRP[9:0]** bits in the **CAN1->BTR** register.

## CAN bit timing register (CAN\_BTR)

Address offset: 0x1C

Reset value: 0x0123 0000

This register can only be accessed by the software when the CAN hardware is in initialization mode.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	Reserved				SJW[1:0]		Res.	TS2[2:0]			TS1[3:0]			
rw	rw					rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						BRP[9:0]									
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 31 **SILM**: Silent mode (debug)

0: Normal operation

1: Silent Mode

Bit 30 **LBKM**: Loop back mode (debug)

0: Loop Back Mode disabled

1: Loop Back Mode enabled

Bits 29:26 Reserved, must be kept at reset value.

Bits 25:24 **SJW[1:0]**: Resynchronization jump width

These bits define the maximum number of time quanta the CAN hardware is allowed to lengthen or shorten a bit to perform the resynchronization.

$$t_{RJW} = t_q \times (SJW[1:0] + 1)$$

Bit 23 Reserved, must be kept at reset value.

Bits 22:20 **TS2[2:0]**: Time segment 2

These bits define the number of time quanta in Time Segment 2.

$$t_{BS2} = t_q \times (TS2[2:0] + 1)$$

Bits 19:16 **TS1[3:0]**: Time segment 1

These bits define the number of time quanta in Time Segment 1

$$t_{BS1} = t_q \times (TS1[3:0] + 1)$$

For more information on bit timing refer to [Section 32.7.7](#).

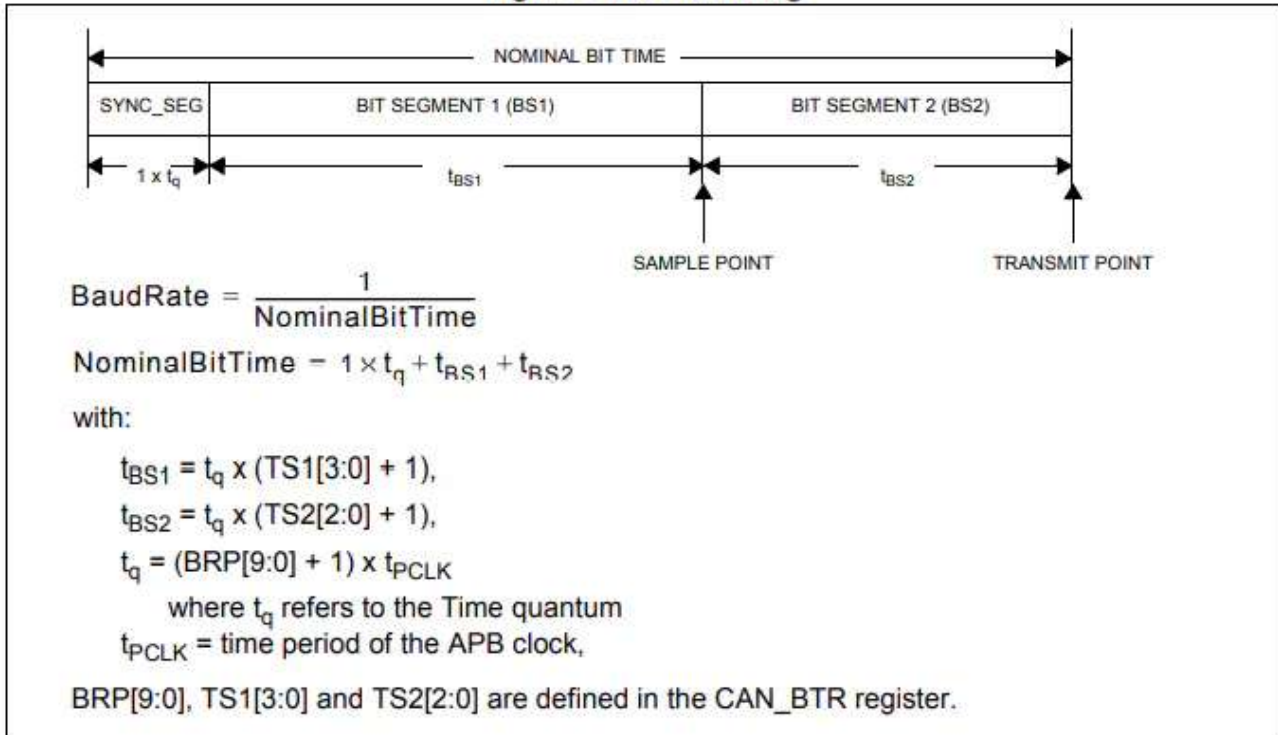
Bits 15:10 Reserved, must be kept at reset value.

Bits 9:0 **BRP[9:0]**: Baud rate prescaler

These bits define the length of a time quanta.

$$t_q = (BRP[9:0] + 1) \times t_{PCLK}$$

**Figure 346. Bit timing**



From the formulas given in the above figure, we will see how our settings result in a baud rate of **125 Kbps**. Or **APB1** clock is running at **16MHz** so,

$$t_{PCLK} = 1/16MHz = 62.5ns = t_p$$

So,

$$t_q = (\text{BRP}[9 : 0] + 1) * (t_p)$$

$$t_q = (15 + 1) * (62.5ns) = 1000ns = t_q$$

And,

$$t_{BS1} = t_q * (\text{TS1}[3 : 0] + 1)$$

$$t_{BS1} = 1000 * 3 = 3000ns$$

$$t_{BS2} = t_q * (\text{TS2}[2 : 0] + 1)$$

$$t_{BS2} = 1000 * 4 = 4000ns$$

Therefore,

$$\text{NominalBitTime} = t_q + t_{BS1} + t_{BS2}$$

$$\text{NominalBitTime} = 1000 + 3000 + 4000 = 8000ns$$

Hence,



$$BaudRate = 1/NominalBitTime = 1/8000ns = 125000bits/s$$

The next step in the CAN initialization process is to configure the filters. For this, we need to enter the *Filter initialization*. This is done by writing a 1 to the **FINIT** bit in the **CAN1->FMR** register. There are a total of 28 filter banks available for CAN. These are split between **CAN1** and **CAN2**. So, we need to set up this split and also specify the bank number from which the filter banks for **CAN2** start. Here, I have assigned 14 filter banks to **CAN1** and the remaining 14 to **CAN2**. Since we are not using **CAN2** here, the filter banks associated with it will not matter. The *Start Bank* for **CAN2** or the *End bank* for **CAN1** is defined by writing 14 to the **CAN2SB[5:0]** bits in the **CAN1->FMR** register. After that, out of the 14 available banks I have used filter number 13 here. Then, we select the *32-bit Scale Configuration* for the filters. This is done by writing a 1 to the **FSC13** bit (signifying the 13th filter bank) in the **CAN1->FS1R** register. To set up the filter for discarding and keeping the messages we write a value of **0x103<<21** in the **CAN1->sFilterRegister[13].FR1**. This can be the identifier for the *Arduino UNO* if in case it is configured to transmit something. Here, we are not doing that, this is just to show how that configuration can be achieved. Since we want all our Identifier bits to match while message reception is done, we choose the *Identifier List Mode*. This is set by writing a 1 to the **FBM13** bit in the **CAN1->FM1R** register.

### 32.9.4 CAN filter registers

#### CAN filter master register (CAN\_FMR)

Address offset: 0x200

Reset value: 0x2A1C 0E01

All bits of this register are set and cleared by software.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CAN2SB[5:0]						Reserved						FINIT	
		rw	rw	rw	rw	rw	rw							rw	

Bits 31:14 Reserved, must be kept at reset value.

Bits 13:8 **CAN2SB[5:0]**: CAN2 start bank

These bits are set and cleared by software. They define the start bank for the CAN2 interface (Slave) in the range 0 to 27.

*Note: When CAN2SB[5:0] = 28d, all the filters to CAN1 can be used.*

*When CAN2SB[5:0] is set to 0, no filters are assigned to CAN1.*

Bits 7:1 Reserved, must be kept at reset value.

Bit 0 **FINIT**: Filter init mode

Initialization mode for filter banks

0: Active filters mode.

1: Initialization mode for the filters.

### CAN filter scale register (CAN\_FS1R)

Address offset: 0x20C

Reset value: 0x0000 0000

This register can be written only when the filter initialization mode is set (FINIT=1) in the CAN\_FMR register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FSC27	FSC26	FSC25	FSC24	FSC23	FSC22	FSC21	FSC20	FSC19	FSC18	FSC17	FSC16
				r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FSC15	FSC14	FSC13	FSC12	FSC11	FSC10	FSC9	FSC8	FSC7	FSC6	FSC5	FSC4	FSC3	FSC2	FSC1	FSC0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FSCx**: Filter scale configuration

These bits define the scale configuration of Filters 13-0.

0: Dual 16-bit scale configuration

1: Single 32-bit scale configuration

### CAN filter mode register (CAN\_FM1R)

Address offset: 0x204

Reset value: 0x0000 0000

This register can be written only when the filter initialization mode is set (FINIT=1) in the CAN\_FMR register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FBM27	FBM26	FBM25	FBM24	FBM23	FBM22	FBM21	FBM20	FBM19	FBM18	FBM17	FBM16
				r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FBM15	FBM14	FBM13	FBM12	FBM11	FBM10	FBM9	FBM8	FBM7	FBM6	FBM5	FBM4	FBM3	FBM2	FBM1	FBM0
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

**Note:** Refer to [Figure 342](#).

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FBMx**: Filter mode

Mode of the registers of Filter x.

0: Two 32-bit registers of filter bank x are in Identifier Mask mode.

1: Two 32-bit registers of filter bank x are in Identifier List mode.

Then, the filter 13 needs to be activated by writing a 1 to the FACT13 bit in the CAN1->FA1R register. After that, we exit the *Filter initialization* by clearing the FINIT bit in the CAN1->FMR register. Since we did not assign any value to the FFA13 bit in the CAN1->FFA1R register it remains at its default value, that is 0, and hence, the received message gets stored in FIFO 0.

### CAN filter activation register (CAN\_FA1R)

Address offset: 0x21C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FACT27	FACT26	FACT25	FACT24	FACT23	FACT22	FACT21	FACT20	FACT19	FACT18	FACT17	FACT16
				rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FACT15	FACT14	FACT13	FACT12	FACT11	FACT10	FACT9	FACT8	FACT7	FACT6	FACT5	FACT4	FACT3	FACT2	FACT1	FACT0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FACTx**: Filter active

The software sets this bit to activate Filter x. To modify the Filter x registers (CAN\_FxR[0:7]), the FACTx bit must be cleared or the FINIT bit of the CAN\_FMR register must be set.

0: Filter x is not active

1: Filter x is active

### CAN filter FIFO assignment register (CAN\_FFA1R)

Address offset: 0x214

Reset value: 0x0000 0000

This register can be written only when the filter initialization mode is set (FINIT=1) in the CAN\_FMR register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				FFA27	FFA26	FFA25	FFA24	FFA23	FFA22	FFA21	FFA20	FFA19	FFA18	FFA17	FFA16
				rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FFA15	FFA14	FFA13	FFA12	FFA11	FFA10	FFA9	FFA8	FFA7	FFA6	FFA5	FFA4	FFA3	FFA2	FFA1	FFA0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 31:28 Reserved, must be kept at reset value.

Bits 27:0 **FFAx**: Filter FIFO assignment for filter x

The message passing through this filter will be stored in the specified FIFO.

0: Filter assigned to FIFO 0

1: Filter assigned to FIFO 1

Now, we need to set up the Transmission Mailbox registers. Here, we chose the mailbox zero, and set its `TIR` register to `0`. This is done by the line `CAN1->sTxMailBox[0].TIR = 0`. Then we configure our identifier, I have chosen the identifier to be `0x245`. So, we write this value to the `STID[10:0]` bits in the `CAN1->sTxMailbox[0].TIR` register. Then we set the `CAN1->sTxMailBox[0].TDHR` to `0`. We also need to set the number of bytes that need to be transmitted. In our case, we need to transmit one byte, so we write `1` to the `CAN1->sTxMailBox[0].TDTR` register.

### CAN mailbox data high register (CAN\_TDHxR) (x=0..2)

All bits of this register are write protected when the mailbox is not in empty state.

Address offsets: 0x18C, 0x19C, 0x1AC

Reset value: 0xFFFF XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA7[7:0]								DATA6[7:0]							
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA5[7:0]								DATA4[7:0]							
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

### CAN mailbox data length control and time stamp register (CAN\_TDTxR) (x=0..2)

All bits of this register are write protected when the mailbox is not in empty state.

Address offsets: 0x184, 0x194, 0x1A4

Reset value: 0xFFFF XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TIME[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							TGT	Reserved				DLC[3:0]			
							rw					rw	rw	rw	

Then, we need to exit the *Initialization mode* which is similar to the way we entered initialization mode, the only difference being here, we clear the `INRQ` bit in the `CAN1->MCR` register.

- Message Transmission is handled by the `void CAN1_Tx(uint8_t tr)` function,

```
void CAN1_Tx(uint8_t tr){
    // Put the Data to be transmitted into Mailbox Data Low Register
    CAN1->sTxMailBox[0].TDLR = tr;

    // Request for Transmission
    CAN1->sTxMailBox[0].TIR |= 1;
}
```

COPY 

For this, we put the data provided by the user into the `CAN1->sTxMailBox[0].TDLR` register. Then, a transmission request is raised by writing a `1` to the `TXRQ` bit in the `CAN1->sTxMailBox[0].TIR` register.



## CAN TX mailbox identifier register (CAN\_TlRxR) (x=0..2)

Address offsets: 0x180, 0x190, 0x1A0

Reset value: 0xFFFF XXXX (except bit 0, TXRQ = 0)

All TX registers are write protected when the mailbox is pending transmission (TMEx reset).

This register also implements the TX request control (bit 0) - reset value 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
STID[10:0]/EXID[28:18]											EXID[17:13]				
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXID[12:0]													IDE	RTR	TXRQ
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Bits 31:21 **STID[10:0]/EXID[28:18]**: Standard identifier or extended identifier

The standard identifier or the MSBs of the extended identifier (depending on the IDE bit value).

Bits 20:3 **EXID[17:0]**: Extended identifier

The LSBs of the extended identifier.

Bit 2 **IDE**: Identifier extension

This bit defines the identifier type of message in the mailbox.

0: Standard identifier.

1: Extended identifier.

Bit 1 **RTR**: Remote transmission request

0: Data frame

1: Remote frame

Bit 0 **TXRQ**: Transmit mailbox request

Set by software to request the transmission for the corresponding mailbox.

Cleared by hardware when the mailbox becomes empty.

- For message reception, the `uint8_t CAN1_Rx()` function is used. I have written this function to showcase how simultaneous message reception and transmission can be done.

COPY 

```
uint8_t CAN1_Rx(){
    // Monitoring FIFO 0 message pending bits FMP0[1:0]
    while(!(CAN1->RF0R & 3)){

        // Read the data from the FIFO 0 mailbox from Mailbox data low register
        uint8_t RxD = (CAN1->sFIFOMailBox[0].RDLR) & 0xFF;
        rec = RxD;

        // Releasing FIFO 0 output mailbox
        CAN1->RF0R |= 1<<5;

        return RxD;
    }
}
```

Here, we first check if there is a message pending in `FIFO 0` by polling the `FMP0[1:0]` bits in the `CAN1->RF0R` register. After this, we declare a local variable `uint8_t RxD` that reads the received value from the `CAN1->sFIFOMailBox[0].RDLR` register. A Bitwise AND operation of the received value and `0xFF` is performed to get the final value. Then, we release the `FIFO 0` mailbox by writing a `1` to the `RF0M0` bit in the `CAN1->RF0R` register, and we return the received value.

### CAN receive FIFO 0 register (CAN\_RF0R)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										RF0M0	FOVR0	FULL0	Res.	FMP0[1:0]	
										rs	rc_w1	rc_w1		r	r

### CAN receive FIFO mailbox data low register (CAN\_RDLxR) (x=0..1)

All bits of this register are write protected when the mailbox is not in empty state.

Address offsets: 0x1B8, 0x1C8

Reset value: 0xFFFF XXXX

All RX registers are write protected.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA3[7:0]								DATA2[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA1[7:0]								DATA0[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:24 **DATA3[7:0]**: Data Byte 3

Data byte 3 of the message.

Bits 23:16 **DATA2[7:0]**: Data Byte 2

Data byte 2 of the message.

Bits 15:8 **DATA1[7:0]**: Data Byte 1

Data byte 1 of the message.

Bits 7:0 **DATA0[7:0]**: Data Byte 0

Data byte 0 of the message.

A message can contain from 0 to 8 data bytes and starts with byte 0.

- The delay is provided by the `void TIM4_ms_Delay(uint32_t delay)` function,

COPY 

```
void TIM4_ms_Delay(uint32_t delay){
    RCC->APB1ENR |= 1<<2; //Start the clock for the timer peripheral
    TIM4->PSC = 16000-1; //Setting the clock frequency to 1kHz.
```



```

TIM4->ARR = (delay); // Total period of the timer
TIM4->CNT = 0;
TIM4->CR1 |= 1; //Start the Timer
while(!(TIM4->SR & TIM_SR_UIF)){ } //Polling the update interrupt flag
TIM4->SR &= ~(0x0001); //Reset the update interrupt flag
}

```

Here, we enable the clock for the *Timer 4*, and set the *Pre-scalar* to `16000-1` to reduce the clock speed to *1 KHz*, which means one clock *tick* corresponds to *1ms*. After that, we assign the desired delay value to the `TIM2->ARR` register, initialize the count to zero and start the timer. Then we poll the `UIF` or the *Update Interrupt Flag*, which is set after the desired amount of clock ticks provided in the `TIM2->ARR` register are done, or in other words when the timer overflows. After the `UIF` flag is set we clear it from the `TIM2->SR` register and exit the function.

### 7.3.13 RCC APB1 peripheral clock enable register (RCC\_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		DAC EN	PWR EN	Reserved	CAN2 EN	CAN1 EN	Reserved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART3 EN	USART2 EN	Reserved
		rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

### 18.4.5 TIMx status register (TIMx\_SR)

Address offset: 0x10

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF	
		rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	

#### Bit 0 UIF: Update interrupt flag

- " This bit is set by hardware on an update event. It is cleared by software.  
0: No update occurred.  
1: Update interrupt pending. This bit is set by hardware when the registers are updated:
- " At overflow or underflow (for TIM2 to TIM5) and if UDIS=0 in the TIMx\_CR1 register.
- " When CNT is reinitialized by software using the UG bit in TIMx\_EGR register, if URS=0 and UDIS=0 in the TIMx\_CR1 register.  
When CNT is reinitialized by a trigger event (refer to the synchro control register description), if URS=0 and UDIS=0 in the TIMx\_CR1 register.

### 18.4.1 TIMx control register 1 (TIMx\_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

## 4. Writing the main Function

```
int main(void){
    GPIO_Init();
    CAN1_Init();
    while(1){
        CAN1_Tx(k);
        //rec= CAN1_Rx();
        k += 1;
        if (k>25)
            k = 0;
        TIM4_ms_Delay(1000);
    }
}
```

COPY 

In the `main` function, we first configure our `GPIO` pins by calling the `GPIO_Init()` function. Then we initialize the `CAN1` peripheral by calling the `CAN1_Init()` function. After that in a `while` loop, we continuously transmit a range of numbers from `1` to `25`. The variable `k` that we declared earlier is passed as a parameter to the `CAN1_Tx(k)` function, and the variable `rec` is used to store the received value when `CAN1_Rx()` is called. The reception line is commented since it is not used in this particular project. In each iteration, we

increment `k` by `1` and if it becomes larger than `25`, we set it back to `0`. After that, a delay of `1` second or `1000ms` is provided using the `TIM4_ms_Delay(1000)` function.

This process repeats indefinitely. This completes the working of the **Normal Mode** on the CAN peripheral, in the *STM32F407*.

Now let's have a look at the Arduino Uno code.

## Programming the Arduino UNO

For this, I have used a CAN library provided [here](#), for configuring the MCP2515. The code is taken majorly from the example code provided in the above repository.

```

/*****
 * The connection are as follows:
 * CS    - D10
 * MOSI  - D11
 * MISO  - D12
 * SCK   - D13
 */

#include <SPI.h>
#include<mcp2515.h>

struct can_frame canMsg;
MCP2515 mcp2515(10);

void setup() {
    Serial.begin(115200);
    SPI.begin();
    mcp2515.reset();
    mcp2515.setBtrrate(CAN_125KBPS,MCP_8MHZ);
    mcp2515.setNormalMode();
    Serial.println("----- CAN Read -----");
    Serial.println("ID  DLC  DATA");
}

void loop() {
    if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {
        Serial.print(canMsg.can_id, HEX); // print ID
    }
}

```

COPY 

```

Serial.print(" ");
Serial.print(canMsg.can_dlc, DEC); // print DLC
Serial.print(" ");

for (int i = 0; i<canMsg.can_dlc; i++) { // print the data
    Serial.print(canMsg.data[i],DEC);
    Serial.print(" ");
}
Serial.println();
}
}

```

Firstly, we include the necessary header files like the `SPI.h` and `mcp2515.h`. Then we initialize the CAN frame as a struct. The MCP2515 is initialized by calling the `mcp2515(10)` function, where the pin number connected to `CS` is passed as its parameter.

Then in the `void setup()` function, we set the baud rate to `115200`, for the communication between the host and the Arduino. Then, we begin the `SPI` communication by calling the `SPI.begin()` function. After, that we reset the MCP2515 by calling the `mcp2515.reset()` function. Then the **Baud Rate** for the CAN communication and the **clock speed** for the CAN Controller is configured by passing `CAN_125KBPS` and `MCP_8MHZ` as parameters to the `mcp2515.setBaudrate(CAN_125KBPS,MCP_8MHZ)` function. Finally, we set the `Normal Mode` for communication by calling the `mcp2515.setNormalMode()` function. Then, we print some information on the Serial Monitor for our reference.

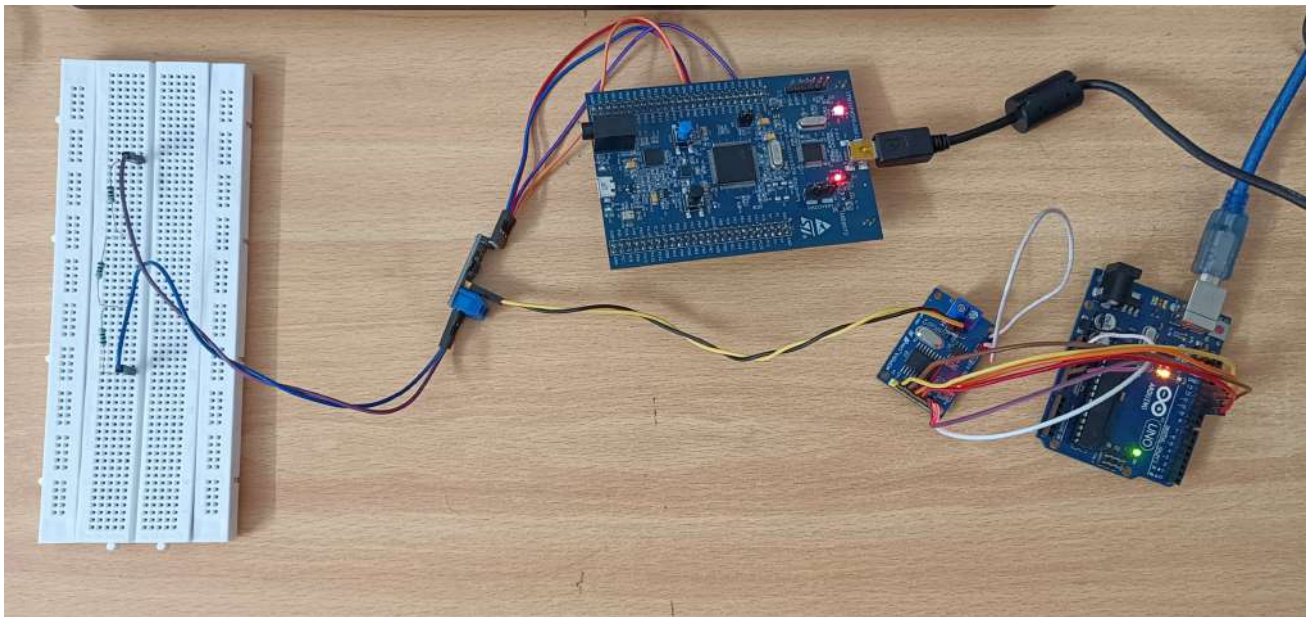
In the `void loop()` we first check if there was an error while receiving the CAN frame by the line `if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK)`. After that is done, and no error is detected we print the **CAN message ID**, in `Hex` format, and the **Data Length** in the `Decimal` format. After that, we print the received data one by one in the `Decimal` format using a `for` loop which runs up to the **Data Length**. Then, we go to a new line before leaving the current iteration of the infinite loop.

This is done infinitely, and the received data is continuously printed on the *Serial Monitor* on the *Arduino IDE*.

## Conclusion

This article was aimed at providing an idea about the basic functioning of the CAN communication Protocol, and how two nodes can send and receive data using it.

I hope you found this article useful and easy to follow along 😊. All the codes used here are present in my GitHub repository, which you can access by clicking [here](#).



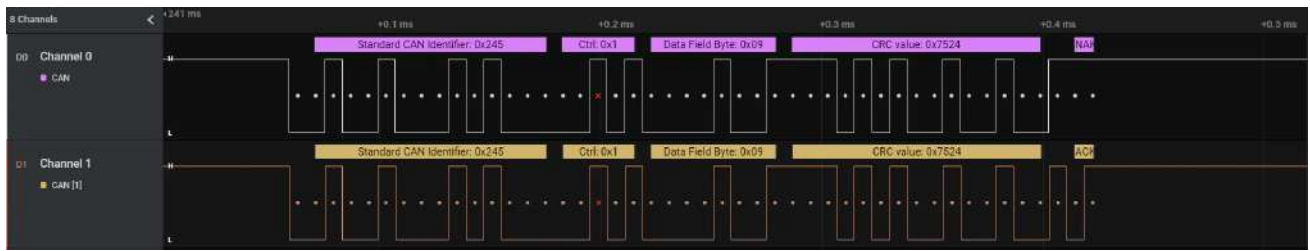
The received messages can be seen in the *Serial Monitor* in the *Arduino IDE*, as shown below.

```
COM4
----- CAN Read -----
ID  DLC  DATA
245  1  8
245  1  9
245  1 10
245  1 11
245  1 12
245  1 13
245  1 14
245  1 15
245  1 16
245  1 17
245  1 18
245  1 19
245  1 20
245  1 21
245  1 22
245  1 23
245  1 24
245  1 25
245  1  0
245  1  1
245  1  2
245  1  3
245  1  4
245  1  5
245  1  6
245  1  7
```

☐ Autoscroll ☐ Show timestamp Newline 115200 baud Clear output



The following pictures show the various parts of the **CAN Frame** captured through a *Logic Analyzer*. *Channel 0* is connected to **CAN Tx** and *Channel 1* is connected to **CAN Rx**.



#embedded



😊 Like