Requirements and Design Document

Team Name: ACE Team

Date: March 27th, 2019

Stage: 1

Version	Description of Change	Author	Date
1.0	Filled document template	ACE Team	03/27/19

CONTENTS

1 INTRODUCTION 4
1.1 Scope 4
1.2 References 4
1.5 Assumptions and Constraints 4
2 METHODOLOGY 4
3 FUNCTIONAL REQUIREMENTS 5
4.1 Context 5
4.2 User Requirements 5
4.3 Data Flow Diagrams 5
4.4 Sequence Diagrams 6
4.5 Functional Requirements 6
5 OTHER REQUIREMENTS 8
5.1 Interface Requirements 8
5.2 Hardware/Software Requirements 8
5.3 Operational Requirements 8
6 CODE 9
7 FUNCTIONAL EVIDENCE 39
APPENDIX A - GLOSSARY 40

1 INTRODUCTION

CAN is currently the standard protocol for the automotive industry therefore knowledge of it is necessary to understand the current automotive trends and develop the required skills to understand this area of electronics, for this reasons a CAN driver was developed using a NXP S32K144 MCU capable of sending and receiving data in a loopback mode, with some parametrizable data fields.

1.1 Scope

In the following the document the project development will be shown, including everything necessary for this project to run correctly on any compatible device and the necessary constraints this project has.

1.2 References

- NXP Semiconductors, 2017, S32K14x Series Cookbook
- NXP Semiconductors, S32K144EVB Quick start guide
- NXP Semiconductors, 2018, S32K1xx Data Sheet
- NXP Semiconductors, 2018, S32K1xx Series Reference Manual

1.5 Assumptions and Constraints

1.5.1 Assumptions

- Availability of development tools.
- Previous knowledge of the CAN protocol.
- Fixed deadline.
- Team availability.
- Previous general knowledge of bare-metal programming.

1.5.2 Constraints

- Only one S32K144 MCU was given to fully test the CAN protocol.
- How robust the driver should be.
- Uncertainty in finishing a presentable project in time.

2 METHODOLOGY

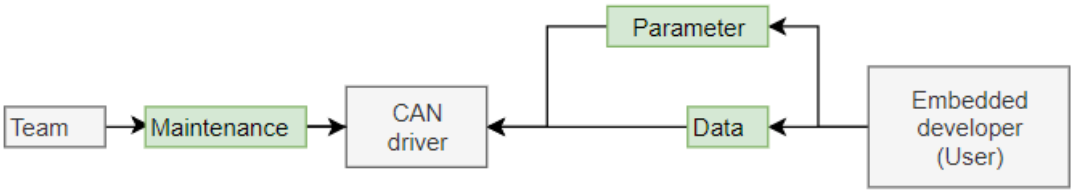
A division of the biggest tasks were made and distributed between the team members after that the team members integrated everything and worked together to create something functional out of everything and with the knowledge of all the individual research done.

3 FUNCTIONAL REQUIREMENTS

4.1 Context

For this driver which is the system in this case, the necessary inputs from its user are the parameters and data and the other possible entity is the team that could give maintenance to the system.

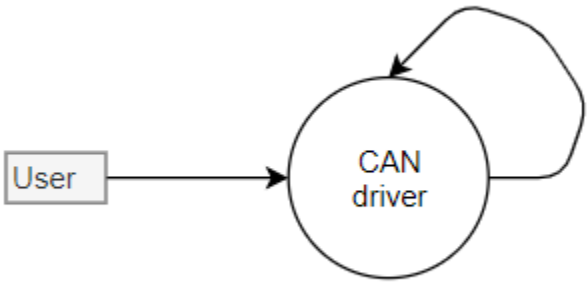
Exhibit 2 - Generic Context Diagram



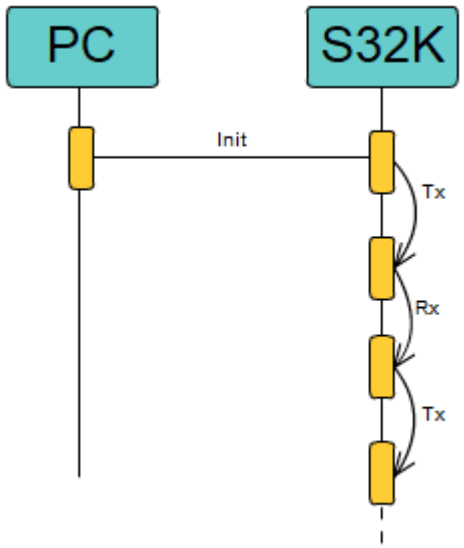
4.2 User Requirements

U-1	User must be able to send data through the driver
U-2	User must be able to send data and use the driver parameters
U-3	User must be able to send data, use the driver parameters and modify the driver to extend usability.

4.3 Data Flow Diagrams



4.4 Sequence Diagrams



4.5 Functional Requirements

- Initialize driver
- Send message
- Receive message

4.5.1 Functional Requirements Group 1

Initialize driver

Requirements Group 1

Section/ Requirement ID	Requirement Definition
FR1.0.	The system shall initialize driver
FR1.1	The system shall enable the clock
FR1.2	The system shall disable the CAN module before selecting clock
FR1.2.1	The system shall choose the Oscillator Clock
FR1.2.2	The system shall choose the Peripheral Clock
FR1.3	The system shall enable the CAN module
FR1.4	The system shall wait for FRZACK to be frozen
FR1.5	The system shall change the register in CTRL1
FR1.6	The system shall enable module configuration
FR1.7	The system shall check all ID bits
FR1.8	The system shall enable for reception

Section/ Requirement ID	Requirement Definition
FR1.9	The system shall write Standard ID
FR1.10	The system shall enable for reception
FR1.11	The system shall set CANFD as not used
FR1.12	The system shall wait for FRZACK to be unfrozen
FR1.13	The system shall wait for CAN Module to be ready

4.5.2 Functional Requirements Group 2

Send message

Sample Requirements Group 2

Section/ Requirement ID	Requirement Definition
FR1.0.	The system shall send a message.
FR1.1.	The system shall clean the MB0 interrupt flag.
FR1.2.	The system shall load the payload
FR1.3.	The system shall load an ID.
FR1.4	The system shall load a valid TX configuration.

4.5.3 Functional Requirements Group 3

Receive message

Sample Requirements Group 3

Section/ Requirement ID	Requirement Definition
FR1.0.	The system shall receive a message.
FR1.1.	The system shall receive the code.
FR1.2.	The system shall receive the ID.
FR1.3.	The system shall receive the length.
FR1.4	The system shall receive the payload.
FR1.4.1	The system shall store the payload.
FR1.5.	The system shall save the timestamp.
FR 1.6.	The system shall save the timer.
FR1.6.1	The system shall unlock the bus.

5 OTHER REQUIREMENTS

- Scalable
- Maintainable
- Usable
- Standard compliable

5.1 Interface Requirements

A personal computer

5.1.1 Hardware Interfaces

NXP S32K144 MCU

5.1.2 Software Interfaces

S32 design studio for arm 2018.R1.

5.1.3 Communications Interfaces

CAN is a standard used to communicate devices with each other or with applications, it was developed by Robert Bosch for automotive applications.

There are seven fields that compose the CAN data frame: Start of frame, arbitration, control, data, cyclical redundancy check, acknowledge and end of frame.

Every node has access to read and write data on the CAN bus, all network nodes waiting to transmit synchronize with the start of frame and begin transmitting at the same time. To determine which node will take control of the bus, an arbitration scheme is used.

5.2 Hardware/Software Requirements

- NXP S32K144 MCU, the driver was made specifically for this MCU.
- A personal computer capable of running S32 design studio for arm 2018.R1.
- S32 design studio for arm 2018.R1.
- PEmicro USB debugging drivers.

5.3 Operational Requirements

- Standalone CAN driver.
- Standard memory consumption
- The driver only has been tested in the loopback setting.

5.4 Error Handling

Delimitation of parameters.

CODE

GPIO.h

```
/**
 *      \file   GPIO.h
 *      \brief
 *                      This is the source of GPIO in baremetal environment
 *                      where the watch dog and ports are disabled and enabled
 *                      respectively.
 *      \author ACE TEAM
 *                      Andres Hernandez
 *                      Carem Bernabe
 *                      Eric Guedea
 *      \date   27/03/2019
 */
#ifndef GPIO_H_
#define GPIO_H_

#include "CAN.h"

typedef enum{PORT_A, PORT_B, PORT_C, PORT_D, PORT_E} Port_t;

/*********************************************************************
**********************************/
/*********************************************************************
**********************************/
/*********************************************************************
**********************************/
/*!
        \brief          Disable the watchdog
        \param[in] Void
        \return         Void
 */
void WDOG_disable (void);

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

```
/*****************************************************************
*********************************/
/*!
        \brief      Configure the needed clock
        \param[in] Port of CAN and peripheral port
        \return     Void
 */
void PORT_init (PortCAN_t portCAN, Port_t port);

/*****************************************************************
*********************************/
/*****************************************************************
*********************************/
/*****************************************************************
*********************************/
/*!
        \brief      Configure a delay
        \param[in] Count
        \return     Void
 */
void delay(uint32_t count);

#endif /* GPIO_H_ */
```

GPIO.c

```
void PORT_init ( int can_N, int port )
{
  switch(){
        case CAN0:
                if(port = PORT_E){
                  PCC->PCCn[PCC_PORTE_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock
for PORTE */
                  PORTE->PCR[4] |= PORT_PCR_MUX(5);                   /* Port
E4: MUX = ALT5, CAN0_RX */
                  PORTE->PCR[5] |= PORT_PCR_MUX(5);                   /* Port
E5: MUX = ALT5, CAN0_TX */
                }
                else if(port = PORT_C){
                  PCC->PCCn[PCC_PORTC_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock
for PORTC */
                  PORTC->PCR[3] |= PORT_PCR_MUX(3);                   /* Port
C3: MUX = ALT3, CAN0_RX */
                  PORTC->PCR[2] |= PORT_PCR_MUX(3);                   /* Port
C2: MUX = ALT3, CAN0_TX */
                }
                else if(port = PORT_B){
```

```
                    PCC->PCCn[PCC_PORTB_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock
for PORTB */
                    PORTB->PCR[1] |= PORT_PCR_MUX(5);                   /* Port
B1: MUX = ALT5, CAN0_RX */
                    PORTB->PCR[0] |= PORT_PCR_MUX(5);                   /* Port
B0: MUX = ALT5, CAN0_TX */
                }
                else{
                        PRINTF("Invalid");
                }
        break;
        case CAN1:
                if(port = PORT_C){
                    PCC->PCCn[PCC_PORTC_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock
for PORTC */
                    PORTC->PCR[7] |= PORT_PCR_MUX(3);                   /* Port
C7: MUX = ALT3, CAN1_RX */
                    PORTC->PCR[6] |= PORT_PCR_MUX(3);                   /* Port
C6: MUX = ALT3, CAN1_TX */
                }
                else if(port = PORT_A){
                    PCC->PCCn[PCC_PORTA_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock
for PORTA */
                    PORTA->PCR[13] |= PORT_PCR_MUX(3);                  /* Port
A13: MUX = ALT3, CAN0_RX */
                    PORTA->PCR[12] |= PORT_PCR_MUX(3);                  /* Port
A12: MUX = ALT3, CAN0_TX */
                }
                else{
                        PRINTF("Invalid");
                }
        break;
        case CAN2:
                if(port = PORT_C){
                    PCC->PCCn[PCC_PORTC_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock
for PORTC */
                    PORTC->PCR[17] |= PORT_PCR_MUX(3);                  /* Port
C17: MUX = ALT3, CAN1_RX */
                    PORTC->PCR[16] |= PORT_PCR_MUX(3);                  /* Port
C16: MUX = ALT3, CAN1_TX */
                }
                else if(port = PORT_B){
                    PCC->PCCn[PCC_PORTB_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock
for PORTB */
                    PORTB->PCR[13] |= PORT_PCR_MUX(4);                  /* Port
B1: MUX = ALT4, CAN0_RX */
                    PORTB->PCR[12] |= PORT_PCR_MUX(4);                  /* Port
B0: MUX = ALT4, CAN0_TX */
                }
                else{
                        PRINTF("Invalid");
                }
        break;
        default:
                PRINTF("Invalid");
        break;
```

```
  }

#ifdef SBC_MC33903 /* If board has MC33904, SPI pin config. is required */
  PCC->PCCn[PCC_PORTB_INDEX] |= PCC_PCCn_CGC_MASK;    /* Enable clock for PORTB */
  PORTB->PCR[14] |= PORT_PCR_MUX(3);                  /* Port B14: MUX =
ALT3, LPSPI1_SCK */
  PORTB->PCR[15] |= PORT_PCR_MUX(3);                  /* Port B15: MUX =
ALT3, LPSPI1_SIN */
  PORTB->PCR[16] |= PORT_PCR_MUX(3);                  /* Port B16: MUX =
ALT3, LPSPI1_SOUT */
  PORTB->PCR[17] |= PORT_PCR_MUX(3);                  /* Port B17: MUX =
ALT3, LPSPI1_PCS3 */
#endif
}
```

LPSPI.h

```
#ifndef LPSPI_H_
#define LPSPI_H_

/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*!
        \brief          Configure the master with SPI
        \param[in] Void
        \return  Void
 */
void LPSPI1_init_master (void);

/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*!
        \brief          Configure the transceiver
        \param[in] Void
        \return  Void
 */
void LPSPI1_init_MC33903 (void);

/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*!
        \brief          Transmit one date
        \param[in] Data
        \return  Void
```

```
 */
void LPSPI1_transmit_16bits (uint16_t);

/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*****************************************************************************************
*******/
/*!
        \brief          Receive one date
        \param[in] Void
        \return   Data
 */
uint16_t LPSPI1_receive_16bits (void);

#endif /* LPSPI_H_ */
```

LPSPI.c

```
/* LPSPI.c            (c) 2016 NXP
 * Descriptions: S32K144 FlexCAN example functions.
 * May 31 2016 S. Mihalik: Initial version.
 * Sep 15 2016 SM: Added MC33904 initialization for CAN communication.
 * Oct 31 2016 SM: Adjusted PRESCALE for 40 MHz SPLLDIV2_CLK
 * Nov 02 2016 SM - cleared flags in transmit, receive functions
 */

#include "S32K144.h"          /* include peripheral declarations S32K144 */
#include "LPSPI.h"
#include "CAN.h"          /* FlexCAN.h defines SBC_MC33904 */

#ifdef SBC_MC33903  /* If board has MC33903, SPI pin config. is required */

void LPSPI1_init_master (void)
{
  PCC->PCCn[PCC_LPSPI1_INDEX] = 0;         /* Disable clocks to modify PCS ( default) */
  PCC->PCCn[PCC_LPSPI1_INDEX]    =    0xC6000000;    /*    Enable PCS=SPLL_DIV2 (40 MHz func'l clock) */

  LPSPI1->CR    = 0x00000000;   /* Disable module for configuration */
  LPSPI1->IER   = 0x00000000;   /* Interrupts not used */
  LPSPI1->DER   = 0x00000000;   /* DMA not used */
  LPSPI1->CFGR0 = 0x00000000;   /* Defaults: */
```

```
                    /* RDM0=0: rec'd data to FIFO as normal */
                    /* CIRFIFO=0; Circular FIFO is disabled */
                    /* HRSEL, HRPOL, HREN=0: Host request disabled */
  LPSPI1->CFGR1 = 0x00000001;   /* Configurations: master mode*/
                    /* PCSCFG=0: PCS[3:2] are enabled */
                    /* OUTCFG=0: Output data retains last value when CS
negated */
                    /* PINCFG=0: SIN is input, SOUT is output */
                    /* MATCFG=0: Match disabled */
                    /* PCSPOL=0: PCS is active low */
                    /* NOSTALL=0: Stall if Tx FIFO empty or Rx FIFO
full */
                    /* AUTOPCS=0: does not apply for master mode */
                    /* SAMPLE=0: input data sampled on SCK edge */
                    /* MASTER=1: Master mode */
  LPSPI1->TCR   = 0x5300000F;   /* Transmit cmd: PCS3, 16 bits, prescale
func'l clk by 4, etc*/
                    /* CPOL=0: SCK inactive state is low */
                    /* CPHA=1: Change data on SCK lead'g, capture on
trail'g edge*/
                    /* PRESCALE=2: Functional clock divided by 2**2 = 4
*/
                    /* PCS=3: Transfer using PCS3 */
                    /* LSBF=0: Data is transfered MSB first */
                    /* BYSW=0: Byte swap disabled */
                    /* CONT, CONTC=0: Continuous transfer disabled */
                    /* RXMSK=0: Normal transfer: rx data stored in rx
FIFO */
                    /* TXMSK=0: Normal transfer: data loaded from tx
FIFO */
                    /* WIDTH=0: Single bit transfer */
                    /* FRAMESZ=15: # bits in frame = 15+1=16 */
  LPSPI1->CCR    = 0x04090808;    /* Clock dividers based on prescaled
func'l clk of 100 nsec */
                    /* SCKPCS=4: SCK to PCS delay = 4+1 = 5 (500 nsec)
*/
                    /* PCSSCK=4: PCS to SCK delay = 9+1 = 10 (1 usec)
*/
                    /* DBT=8: Delay between Transfers = 8+2 = 10 (1 usec)
*/
```

```
                    /* SCKDIV=8: SCK divider =8+2 = 10 (1 usec: 1 MHz
baud rate) */
  LPSPI1->FCR   = 0x00000003;   /* RXWATER=0: Rx flags set when Rx
FIFO >0 */
                    /* TXWATER=3: Tx flags set when Tx FIFO <= 3 */
  LPSPI1->CR    = 0x00000009;   /* Enable module for operation */
                    /* DBGEN=1: module enabled in debug mode */
                    /* DOZEN=0: module enabled in Doze mode */
                    /* RST=0: Master logic not reset */
                    /* MEN=1: Module is enabled */
}

void LPSPI1_init_MC33903 (void)
{
      uint32_t i = 0;                       /* Loop counter */
      uint16_t MC33903_spi_init[] = {       /* SPI commands and
data to initialize MC33903C */
                    0x2580,                 /* Read SAFE register
flags: bits 4:0 contain nonzero ID */
                    0xDF80,                 /* Read Vreg High flags:  */
                    0x5A00,                 /* Write Watchdog reg.: Enter
NORMAL mode*/
                    0x5E10,                 /* Write Regulator reg.: Enable
5V CAN regulator */
                    0x60C0,                 /* Write CAN reg.: CAN in Tx
& Rx modes, fast slew */
                    0x66C4};                /* Write LIN/1 reg.: Tx/Rx
mode, 20 Kbps slew, term. on */
      uint16_t spi_result = 0;              /* Result received SPI
data from SBC */

                  /* Note: MC33904 DBG input on EVB is tied to 9V
nominal, */
                  /*      which puts device in a debug state */
                  /*      which disables the SBC's watchdog. */
      for (i = 0; i < sizeof (MC33903_spi_init) / 2; i++)
      {
            LPSPI1_transmit_16bits (MC33903_spi_init[i]);    /* Transmit
to MC33904 */
            spi_result =  LPSPI1_receive_16bits();         /* Read result */
```

```
                /* Note: It is good practice to verify SPI configuration by
*/
                /*      reading appropriate flags/registers, especially */
                /*      fault flags, after configuration routines. */
        }
}

void LPSPI1_transmit_16bits (uint16_t send)
{
        while((LPSPI1->SR       &       LPSPI_SR_TDF_MASK)      >>
(LPSPI_SR_TDF_SHIFT == 0));
                        /* Wait for Tx FIFO available */
        LPSPI1->TDR = send;             /* Transmit data */
        LPSPI1->SR |= LPSPI_SR_TDF_MASK; /* Clear TDF flag */
}

uint16_t LPSPI1_receive_16bits (void)
{
  uint16_t recieve = 0;

  while((LPSPI1->SR             &       LPSPI_SR_RDF_MASK)      >>
(LPSPI_SR_RDF_SHIFT == 0));
                                /* Wait at least one RxFIFO entry */
  recieve = LPSPI1->RDR;                /* Read received data */
  LPSPI1->SR |= LPSPI_SR_RDF_MASK;          /* Clear RDF flag */
  return recieve;                       /* Return received data */
}

#endif
```

CAN.h

```
/**
 *      \file   CAN.h
 *      \brief
 *                      This is the header of driver in baremetal environment of
 *                      CAN protocol, the driver includes the functions INIT,
 *                      TRANSMITTER and RECEIVER.
 *      \author ACE TEAM
 *                      Andres Hernandez
```

```
 *                  Carem Bernabe
 *                  Eric Guedea
 *      \date   27/03/2019
 */

#ifndef CAN_H_
#define CAN_H_

#define SBC_MC33903   /*Transceiver CAN*/

/*Port CAN*/
typedef enum {CAN_0, CAN_1, CAN_2} PortCAN_t;

/*Clock source variable*/
typedef enum {OSCILLATOR_SRC, PERIPHERAL_SRC} clkSource_t;

/*Time of the frame*/
typedef enum
{
        B10KHZ,
        B20KHZ,
        B50KHZ,
        B125KHZ,
        B250KHZ,
        B500KHZ,
        B800KHZ,
        B1MHZ
} bitTime_t;

typedef struct
{
        uint8_t         propSeg;        /*Propagation Segment*/
        uint8_t         phaseSeg1;      /*Phase Segment 1*/
        uint8_t         phaseSeg2;      /*Phase Segment 2*/
        uint8_t         bitSampling;    /*CAN bit sampling*/
} Timing_t;

/*Variables needed to configure the driver*/
typedef struct
{
```

```
	bitTime_t	bitTime;		/*Bit time of CAN frame*/
	clkSource_t clkSource;		/*Source clock*/
	Timing_t	timing;			/*Timing to CAN bus*/
} CAN_Config_t;

/*Variables needed to Rx*/
typedef struct
{
	uint32_t  RxCode;		/* Received message buffer code */
	uint32_t  RxID;		/* Received message ID */
	uint32_t  RxLength;		/* Received message number of data bytes
*/
	uint32_t  RxData[2];		/* Received message data */
	uint32_t  RxTimeStamp;		/* Received message time */
} Rx_t;

/*****************************************************************
*********************************/
/*****************************************************************
*********************************/
/*****************************************************************
*********************************/
/*!
	\brief		Configure the CAN driver
	\param[in] CAN Port and Pointer with the configuration
	\return		Void
 */
void CAN_init(PortCAN_t portCAN, const CAN_Config_t* CAN_Config);


/*****************************************************************
*********************************/
/*****************************************************************
*********************************/
/*****************************************************************
*********************************/
/*!
	\brief		Tx to send the information
	\param[in]	CAN Port and Data to send in the bus
	\return		Void
```

```
 */
void CAN_Transmitter(PortCAN_t portCAN, uint32_t dataWord1, uint32_t
dataWord2);

/*************************************************************************
*********************************/
/*************************************************************************
*********************************/
/*************************************************************************
*********************************/
/*!
        \brief          Receive the data on different fields
        \param[in] CAN Port and References from variables where the data
received is saved
        \return         Void
 */
void CAN_Receiver(PortCAN_t portCAN, uint32_t *data1, uint32_t
*data2);

#endif /* CAN_H_ */
```

CAN.h

```
/**
 *      \file   CAN.h
 *      \brief
 *                      This is the source of driver in baremetal environment of
 *                      CAN protocol, the driver includes the functions INIT,
 *                      TRANSMITTER and RECEIVER.
 *      \author ACE TEAM
 *                      Andres Hernandez
 *                      Carem Bernabe
 *                      Eric Guedea
 *      \date   27/03/2019
 */

#include "S32K144.h"
#include "CAN.h"

#define MESSAGES_BUFF           (32)            /*Number of MB
for CAN0*/
```

```
#define MESSAGES_BUFF_CAN12 (16)          /*Number of MB
for CAN1 y CAN2*/
#define WORDS_PER_MB        (4)           /*Words per
Messages Buffer*/
#define RAM_LENGTH          (128)         /*Length of the
RAM*/
#define MAX_DATA            (2)           /*Maximum
of words with data*/
#define MB_FILT             (16)          /*Messages Buffer
filtered*/
#define CHECK_ID            (0xFFFFFFFF)  /*Check all IDs for
MB*/
#define CHECK_ALL_ID        (0x1FFFFFFF)  /*Global
acceptance mask*/
#define RX_MB4              (16)          /*Message Buffer
4 for RX*/
#define ENABLE_RX           (0x04000000)  /*Code field of the
Control and Status*/
#define RX_ID_WORD          (0x14440000)  /*Word with ID
for RX*/
#define CANFD_NOT_USED      (0x0000001F)  /*Prior equal to
zero*/
#define CLEAN_MB0           (0x00000001)  /*Clean the flag of
MB0*/
#define TX_MB0              (0)           /*Message
Buffer 0 for TX*/
#define DISABLE_RX          (0x00000000)  /*Disable the
RX*/

#define TX_ID_WORD          (0x15540000)  /*Word with ID
for TX*/
#define DLC_LENGTH          (8)           /*Length of
DLC in Bytes*/
#define CODE_FIELD_TX       (0x0C000000)  /*Code to enable
the transmission of MB*/
#define SRR_TX              (0x400000)    /*Set the TX
frame*/

#define SHIFT_CODE_RX       (24)          /*Shift to obtain
the code of RX*/
```

```
#define CODE_MASK_RX            (0x07000000)    /*Mask to obtain
the code of RX*/
#define RX_DATA_MB4                 (18)            /*Position of
data in MB4*/
#define TIME_STAMP_RX           (0x000FFFF)         /*Mask to
obtain the time stamp*/
#define MB4_CLEAN_FLAG          (0x00000010)    /*Mask to clean
the MB4 flag*/
#define SYNC_SEGMENT            (1)
        /*Synchronization Segment*/
#define SHIFT_PSEG1             (19)            /*Shift to Phase
Segment 1*/
#define SHIFT_PSEG2             (16)            /*Shift to Phase
Segment 2*/
#define SHIFT_RJW               (22)            /*Shift to Resyn
Jump Width*/
#define SHIFT_SMP               (7)                 /*Shift to
Sampling bit*/
#define SHIFT_PRESDIV           (24)            /*Shift to Prescaler
divisor*/

Rx_t  rx;           /*Structure of Rx*/

/*Setup the configurations of the frame between eight options*/
static void CAN_SetBitTime(PortCAN_t portCAN, clkSource_t clkSource,
bitTime_t bitTime, Timing_t timing)
{
        uint32_t time_quanta;       /*Value of time quantum*/
        uint32_t FrequencyTimeQ;    /*Frequency time quantum*/
        uint32_t PresDiv;           /*Prescaler divisor*/
        uint32_t Pseg1;                 /*Phase Segment 1*/
        uint32_t Pseg2;                 /*Phase Segment 2*/
        uint32_t Proseg;            /*Propagation Segment*/
        uint32_t Rjw;                   /*Resync Jump Width*/

        /*Sum all the segments to obtain the total time quantum*/
        time_quanta = timing.phaseSeg1 + timing.phaseSeg2 +
timing.propSeg + SYNC_SEGMENT;
```

```
	/*Assign the Frequency of Time Quantum with each bit time required*/
	switch(bitTime)
	{
	case B10KHZ:
		FrequencyTimeQ = time_quanta * 10000;
		break;
	case B20KHZ:
		FrequencyTimeQ = time_quanta * 20000;
		break;
	case B50KHZ:
		FrequencyTimeQ = time_quanta * 50000;
		break;
	case B125KHZ:
		FrequencyTimeQ = time_quanta * 125000;
		break;
	case B250KHZ:
		FrequencyTimeQ = time_quanta * 250000;
		break;
	case B500KHZ:
		FrequencyTimeQ = time_quanta * 500000;
		break;
	case B800KHZ:
		FrequencyTimeQ = time_quanta * 800000;
		break;
	case B1MHZ:
		FrequencyTimeQ = time_quanta * 1000000;
		break;
	default:
		break;
	}

	/*Calculate the prescaler divisor*/
	PresDiv = (clkSource/FrequencyTimeQ) -  1;

	/*Calculate the phase segment 1*/
	Pseg1 = timing.phaseSeg1 + 1;

	/*Calculate the phase segment 2*/
	Pseg2 = timing.phaseSeg2 + 1;
```

```
/*Calculate the propagation segment*/
Proseg = timing.propSeg - 1;

/*Calculate the resync jump width*/
Rjw = timing.phaseSeg2 - 1;

switch(portCAN)
{
case CAN_0:
        /*AssigN the Prescaler divisor*/
        CAN0->CTRL1 |= PresDiv << SHIFT_PRESDIV;

        /*Assign the Phase Segment 1*/
        CAN0->CTRL1 |= Pseg1 << SHIFT_PSEG1;

        /*Assign the Phase Segment 2*/
        CAN0->CTRL1 |= Pseg2 << SHIFT_PSEG2;

        /*Assign the Resyn Jump Width*/
        CAN0->CTRL1 |= Rjw << SHIFT_RJW;

        /*Assign the Propagation Segment*/
        CAN0->CTRL1 |= Proseg;

        /*Assign the Sampling bit*/
        CAN0->CTRL1 |= timing.bitSampling << SHIFT_SMP;
        break;
case CAN_1:
        /*AssigN the Prescaler divisor*/
        CAN1->CTRL1 |= PresDiv << SHIFT_PRESDIV;

        /*Assign the Phase Segment 1*/
        CAN1->CTRL1 |= Pseg1 << SHIFT_PSEG1;

        /*Assign the Phase Segment 2*/
        CAN1->CTRL1 |= Pseg2 << SHIFT_PSEG2;

        /*Assign the Resyn Jump Width*/
        CAN1->CTRL1 |= Rjw << SHIFT_RJW;
```

```
                /*Assign the Propagation Segment*/
                CAN1->CTRL1 |= Proseg;

                /*Assign the Sampling bit*/
                CAN1->CTRL1 |= timing.bitSampling << SHIFT_SMP;
                break;
        case CAN_2:
                /*AssigN the Prescaler divisor*/
                CAN2->CTRL1 |= PresDiv << SHIFT_PRESDIV;

                /*Assign the Phase Segment 1*/
                CAN2->CTRL1 |= Pseg1 << SHIFT_PSEG1;

                /*Assign the Phase Segment 2*/
                CAN2->CTRL1 |= Pseg2 << SHIFT_PSEG2;

                /*Assign the Resyn Jump Width*/
                CAN2->CTRL1 |= Rjw << SHIFT_RJW;

                /*Assign the Propagation Segment*/
                CAN2->CTRL1 |= Proseg;

                /*Assign the Sampling bit*/
                CAN2->CTRL1 |= timing.bitSampling << SHIFT_SMP;
                break;
        default:
                break;
        }
}

/*Setup the CAN with a selectable clock*/
void CAN_init(PortCAN_t portCAN, const CAN_Config_t* CAN_Config)
{
        uint32_t counter;

        switch(portCAN)
        {
        case CAN_0:
                /*Enable the clock to CAN0*/
```

```
        PCC->PCCn[PCC_FlexCAN0_INDEX]                    |=
PCC_PCCn_CGC_MASK;

        /*Disable the CAN module before selecting clock*/
        CAN0->MCR |= CAN_MCR_MDIS_MASK;


        if (OSCILLATOR_SRC == CAN_Config->clkSource)
            /*Choose the Oscillator Clock 8MHz*/
            CAN0->CTRL1 &= ~CAN_CTRL1_CLKSRC_MASK;
        else
            /*Choose the Peripheral Clock*/
            CAN0->CTRL1 |= CAN_CTRL1_CLKSRC_MASK;


        /*Enable the CAN module*/
        CAN0->MCR &= ~CAN_MCR_MDIS_MASK;

        /*Wait for FRZACK to be frozen*/
        while (!((CAN0->MCR & CAN_MCR_FRZACK_MASK) >>
CAN_MCR_FRZACK_SHIFT));

        /*Now we can change the register in CTRL1*/
        CAN_SetBitTime(portCAN,          CAN_Config->clkSource,
CAN_Config->bitTime, CAN_Config->timing);

        /*Loopback is enabled*/
        CAN0->CTRL1 |= CAN_CTRL1_LPB_MASK;

        /*FIFO is disabled*/
        CAN0->MCR &= ~ CAN_MCR_RFEN_MASK;

        /*Self reception is enabled*/
        CAN0->MCR &= ~ CAN_MCR_SRXDIS_MASK;

        /*Check all IDs*/
        for(counter = 0; counter < MB_FILT; counter++)
            CAN0->RXIMR[counter] = CHECK_ID;

        /*Global acceptance mask to check all the IDs*/
```

```
            CAN0->RXMGMASK = CHECK_ALL_ID;

            /*Disable the RX*/
            CAN0->RAMn[RX_MB4] = DISABLE_RX;

            /*Assign the standard ID to the next word of MB4*/
            CAN0->RAMn[RX_MB4 + 1] = TX_ID_WORD;

            /*Enable the RX*/
            CAN0->RAMn[RX_MB4] = ENABLE_RX;

            /*CAN FD is not used*/
            CAN0->MCR = CANFD_NOT_USED;

            /*Wait for FRZACK to be unfrozen*/
            while ((CAN0->MCR && CAN_MCR_FRZACK_MASK) >>
CAN_MCR_FRZACK_SHIFT);

            /*Wait for CAN Module to be ready*/
            while ((CAN0->MCR && CAN_MCR_NOTRDY_MASK) >>
CAN_MCR_NOTRDY_SHIFT);
            break;

        case CAN_1:
            /*Enable the clock to CAN1*/
            PCC->PCCn[PCC_FlexCAN1_INDEX]                    |=
PCC_PCCn_CGC_MASK;

            /*Disable the CAN module before selecting clock*/
            CAN1->MCR |= CAN_MCR_MDIS_MASK;


            if (OSCILLATOR_SRC == CAN_Config->clkSource)
                /*Choose the Oscillator Clock 8MHz*/
                CAN1->CTRL1 &= ~CAN_CTRL1_CLKSRC_MASK;
            else
                /*Choose the Peripheral Clock*/
                CAN1->CTRL1 |= CAN_CTRL1_CLKSRC_MASK;
```

```
		/*Enable the CAN module*/
		CAN1->MCR &= ~CAN_MCR_MDIS_MASK;

		/*Wait for FRZACK to be frozen*/
		while (!((CAN1->MCR & CAN_MCR_FRZACK_MASK) >>
CAN_MCR_FRZACK_SHIFT));

		/*Now we can change the register in CTRL1*/
		CAN_SetBitTime(portCAN,		CAN_Config->clkSource,
CAN_Config->bitTime, CAN_Config->timing);

		/*Loopback is enabled*/
		CAN1->CTRL1 |= CAN_CTRL1_LPB_MASK;

		/*FIFO is disabled*/
		CAN1->MCR &= ~ CAN_MCR_RFEN_MASK;

		/*Self reception is enabled*/
		CAN1->MCR &= ~ CAN_MCR_SRXDIS_MASK;

		/*Check all IDs*/
		for(counter = 0; counter < MB_FILT; counter++)
			CAN1->RXIMR[counter] = CHECK_ID;

		/*Global acceptance mask to check all the IDs*/
		CAN1->RXMGMASK = CHECK_ALL_ID;

		/*Disable the RX*/
		CAN1->RAMn[RX_MB4] = DISABLE_RX;

		/*Assign the standard ID to the next word of MB4*/
		CAN1->RAMn[RX_MB4 + 1] = TX_ID_WORD;

		/*Enable the RX*/
		CAN1->RAMn[RX_MB4] = ENABLE_RX;

		/*CAN FD is not used*/
		CAN1->MCR = CANFD_NOT_USED;

		/*Wait for FRZACK to be unfrozen*/
```

```
            while ((CAN1->MCR && CAN_MCR_FRZACK_MASK) >>
CAN_MCR_FRZACK_SHIFT);

            /*Wait for CAN Module to be ready*/
            while ((CAN1->MCR && CAN_MCR_NOTRDY_MASK) >>
CAN_MCR_NOTRDY_SHIFT);
            break;

        case CAN_2:
            /*Enable the clock to CAN0*/
            PCC->PCCn[PCC_FlexCAN2_INDEX]                    |=
PCC_PCCn_CGC_MASK;

            /*Disable the CAN module before selecting clock*/
            CAN2->MCR |= CAN_MCR_MDIS_MASK;


            if (OSCILLATOR_SRC == CAN_Config->clkSource)
                /*Choose the Oscillator Clock 8MHz*/
                CAN2->CTRL1 &= ~CAN_CTRL1_CLKSRC_MASK;
            else
                /*Choose the Peripheral Clock*/
                CAN2->CTRL1 |= CAN_CTRL1_CLKSRC_MASK;


            /*Enable the CAN module*/
            CAN2->MCR &= ~CAN_MCR_MDIS_MASK;

            /*Wait for FRZACK to be frozen*/
            while (!((CAN2->MCR & CAN_MCR_FRZACK_MASK) >>
CAN_MCR_FRZACK_SHIFT));

            /*Now we can change the register in CTRL1*/
            CAN_SetBitTime(portCAN,          CAN_Config->clkSource,
CAN_Config->bitTime, CAN_Config->timing);

            /*Loopback is enabled*/
            CAN2->CTRL1 |= CAN_CTRL1_LPB_MASK;

            /*FIFO is disabled*/
```

```
                CAN2->MCR &= ~ CAN_MCR_RFEN_MASK;

                /*Self reception is enabled*/
                CAN2->MCR &= ~ CAN_MCR_SRXDIS_MASK;

                /*Check all IDs*/
                for(counter = 0; counter < MB_FILT; counter++)
                        CAN2->RXIMR[counter] = CHECK_ID;

                /*Global acceptance mask to check all the IDs*/
                CAN2->RXMGMASK = CHECK_ALL_ID;

                /*Disable the RX*/
                CAN2->RAMn[RX_MB4] = DISABLE_RX;

                /*Assign the standard ID to the next word of MB4*/
                CAN2->RAMn[RX_MB4 + 1] = TX_ID_WORD;

                /*Enable the RX*/
                CAN2->RAMn[RX_MB4] = ENABLE_RX;

                /*CAN FD is not used*/
                CAN2->MCR = CANFD_NOT_USED;

                /*Wait for FRZACK to be unfrozen*/
                while ((CAN2->MCR && CAN_MCR_FRZACK_MASK) >>
CAN_MCR_FRZACK_SHIFT);

                /*Wait for CAN Module to be ready*/
                while ((CAN2->MCR && CAN_MCR_NOTRDY_MASK) >>
CAN_MCR_NOTRDY_SHIFT);
                break;

        default:
                break;
        }
}

/*Transmit the data through of channel CAN 0 with two data */
```

```
void CAN_Transmitter(PortCAN_t portCAN, uint32_t dataWord1, uint32_t
dataWord2)
{
	switch(portCAN)
	{
	case CAN_0:
		/*Clean MB0 flag*/
		CAN0->IFLAG1 = CLEAN_MB0;

		/*Data word 1 in the third position of MB0*/
		CAN0->RAMn[TX_MB0 + 2] = dataWord1;

		/*Data word 2 in the fourth position of MB0*/
		CAN0->RAMn[TX_MB0 + 3] = dataWord2;

		/* Standard ID 0x555, first position of MB0*/
		CAN0->RAMn[TX_MB0 + 1] = TX_ID_WORD;

		/*SETUP OF THE TX*/
		/*Set the length of DLC*/
		CAN0->RAMn[TX_MB0 + 0] |= (DLC_LENGTH <<
CAN_WMBn_CS_DLC_SHIFT);

		/*Assign the code to transmit MB*/
		CAN0->RAMn[TX_MB0 + 0] |= CODE_FIELD_TX;

		/*Set TX frame*/
		CAN0->RAMn[TX_MB0 + 0] |= SRR_TX;
		break;

	case CAN_1:
		/*Clean MB0 flag*/
		CAN1->IFLAG1 = CLEAN_MB0;

		/*Data word 1 in the third position of MB0*/
		CAN1->RAMn[TX_MB0 + 2] = dataWord1;

		/*Data word 2 in the fourth position of MB0*/
		CAN1->RAMn[TX_MB0 + 3] = dataWord2;
```

```
            /* Standard ID 0x555, first position of MB0*/
            CAN1->RAMn[TX_MB0 + 1] = TX_ID_WORD;

            /*SETUP OF THE TX*/
            /*Set the length of DLC*/
            CAN1->RAMn[TX_MB0 + 0] |= (DLC_LENGTH << CAN_WMBn_CS_DLC_SHIFT);

            /*Assign the code to transmit MB*/
            CAN1->RAMn[TX_MB0 + 0] |= CODE_FIELD_TX;

            /*Set TX frame*/
            CAN1->RAMn[TX_MB0 + 0] |= SRR_TX;
            break;

        case CAN_2:
            /*Clean MB0 flag*/
            CAN2->IFLAG1 = CLEAN_MB0;

            /*Data word 1 in the third position of MB0*/
            CAN2->RAMn[TX_MB0 + 2] = dataWord1;

            /*Data word 2 in the fourth position of MB0*/
            CAN2->RAMn[TX_MB0 + 3] = dataWord2;

            /* Standard ID 0x555, first position of MB0*/
            CAN2->RAMn[TX_MB0 + 1] = TX_ID_WORD;

            /*SETUP OF THE TX*/
            /*Set the length of DLC*/
            CAN2->RAMn[TX_MB0 + 0] |= (DLC_LENGTH << CAN_WMBn_CS_DLC_SHIFT);

            /*Assign the code to transmit MB*/
            CAN2->RAMn[TX_MB0 + 0] |= CODE_FIELD_TX;

            /*Set TX frame*/
            CAN2->RAMn[TX_MB0 + 0] |= SRR_TX;
            break;
        default:
```

```
		break;
	}
}

/*Receive the data though the channel and only is received two data*/
void CAN_Receiver(PortCAN_t portCAN, uint32_t *data1, uint32_t
*data2)
{
	uint8_t counter;
	uint32_t dummy;

	switch(portCAN)
	{
	case CAN_0:
		/*Obtain the code of RX with the first word of MB4*/
		rx.RxCode = (CAN0->RAMn[RX_MB4] &
CODE_MASK_RX) >> SHIFT_CODE_RX;

		/*Obtain the ID of RX with the next word of MB4*/
		rx.RxID = (CAN0->RAMn[RX_MB4 + 1] &
CAN_WMBn_ID_ID_MASK) >> CAN_WMBn_ID_ID_SHIFT;

		/*Obtain the length of RX with the first word of MB4*/
		rx.RxLength = (CAN0->RAMn[RX_MB4] &
CAN_WMBn_CS_DLC_MASK) >> CAN_WMBn_CS_DLC_SHIFT;

		/*Read the data received*/
		for (counter = 0; counter < MAX_DATA; counter++)
		  rx.RxData[counter] = CAN0->RAMn[RX_DATA_MB4 +
counter];

		/*Save the data received*/
		data1 = rx.RxData[0];
		data2 = rx.RxData[1];

		/*Obtain the time stamp*/
		rx.RxTimeStamp = (CAN0->RAMn[RX_MB4] &
TIME_STAMP_RX);

		/*Unlock message buffers*/
```

```
            dummy = CAN0->TIMER;

            /*Clean MB4 flag*/
            CAN0->IFLAG1 = MB4_CLEAN_FLAG;
            break;

        case CAN_1:
            /*Obtain the code of RX with the first word of MB4*/
            rx.RxCode = (CAN1->RAMn[RX_MB4] &
CODE_MASK_RX) >> SHIFT_CODE_RX;

            /*Obtain the ID of RX with the next word of MB4*/
            rx.RxID = (CAN1->RAMn[RX_MB4 + 1] &
CAN_WMBn_ID_ID_MASK)  >> CAN_WMBn_ID_ID_SHIFT;

            /*Obtain the length of RX with the first word of MB4*/
            rx.RxLength = (CAN1->RAMn[RX_MB4] &
CAN_WMBn_CS_DLC_MASK) >> CAN_WMBn_CS_DLC_SHIFT;

            /*Read the data received*/
            for (counter = 0; counter < MAX_DATA; counter++)
              rx.RxData[counter] = CAN1->RAMn[RX_DATA_MB4 +
counter];

            /*Save the data received*/
            data1 = rx.RxData[0];
            data2 = rx.RxData[1];

            /*Obtain the time stamp*/
            rx.RxTimeStamp = (CAN1->RAMn[RX_MB4] &
TIME_STAMP_RX);

            /*Unlock message buffers*/
            dummy = CAN1->TIMER;

            /*Clean MB4 flag*/
            CAN1->IFLAG1 = MB4_CLEAN_FLAG;
            break;

        case CAN_2:
```

```
            /*Obtain the code of RX with the first word of MB4*/
            rx.RxCode = (CAN2->RAMn[RX_MB4] & CODE_MASK_RX) >> SHIFT_CODE_RX;

            /*Obtain the ID of RX with the next word of MB4*/
            rx.RxID = (CAN2->RAMn[RX_MB4 + 1] & CAN_WMBn_ID_ID_MASK) >> CAN_WMBn_ID_ID_SHIFT;

            /*Obtain the length of RX with the first word of MB4*/
            rx.RxLength = (CAN2->RAMn[RX_MB4] & CAN_WMBn_CS_DLC_MASK) >> CAN_WMBn_CS_DLC_SHIFT;

            /*Read the data received*/
            for (counter = 0; counter < MAX_DATA; counter++)
              rx.RxData[counter] = CAN2->RAMn[RX_DATA_MB4 + counter];

            /*Save the data received*/
            data1 = rx.RxData[0];
            data2 = rx.RxData[1];

            /*Obtain the time stamp*/
            rx.RxTimeStamp = (CAN2->RAMn[RX_MB4] & TIME_STAMP_RX);

            /*Unlock message buffers*/
            dummy = CAN2->TIMER;

            /*Clean MB4 flag*/
            CAN2->IFLAG1 = MB4_CLEAN_FLAG;
            break;
        default:
            break;
        }
}
```

Clock_and_modules.h

```
#ifndef CLOCKS_AND_MODES_H_
#define CLOCKS_AND_MODES_H_
```

```
void SOSC_init_8MHz (void);
void SPLL_init_160MHz (void);
void NormalRUNmode_80MHz (void);
void ClockConfig (void);

#endif /* CLOCKS_AND_MODES_H_ */
```

Clock_and_modules.c

```
/**
 *      \file    main.c
 *      \brief
 *                      This is the main of driver where the driver is
 *                      tested with the transmission and reception.
 *      \author ACE TEAM
 *                      Andres Hernandez
 *                      Carem Bernabe
 *                      Eric Guedea
 *      \date   27/03/2019
 */

#include "S32K144.h"
#include "CAN.h"
#include "LPSPI.h"
#include "GPIO.h"
#include "clock_and_modes.h"

#define DATA_WORD_1                     (0xA5112233)      /*Data word
1 to transmit*/
#define DATA_WORD_2                     (0x44556677)      /*Data word
2 to transmit*/


/*Pointer that saves the information about the configuration about the CAN
frame*/
const CAN_Config_t      CAN_Config =
{
        OSCILLATOR_SRC,             /*Source clock*/
        B500KHZ,                    /*Bit time*/
        {7,                         /*Propagation Segment*/
```

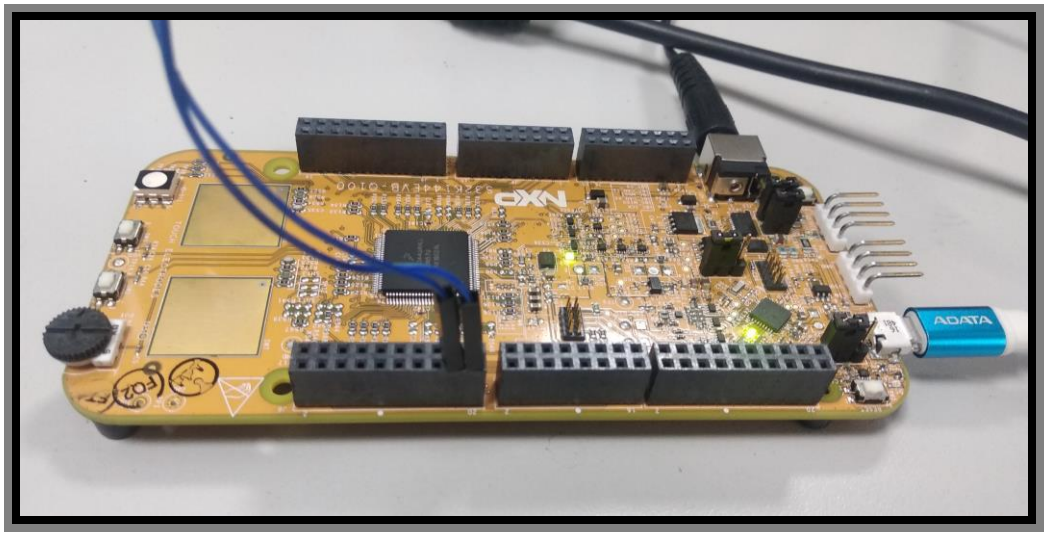
```
        4,                          /*Phase 1 Segment*/
        4,                          /*Phase 2 Segment*/
        1}                          /*Sampling bit*/
};

int main(void)
{
        WDOG_disable();                     /*Disable the watchdog*/
        ClockConfig();                      /*Configure the clock*/

        CAN_init(CAN_0, &CAN_Config);   /* Init FlexCAN0 */
        PORT_init(CAN_0, PORT_E);           /* Configure ports */

#ifdef SBC_MC33903                          /* SPI and transceiver
initialization is required */
        LPSPI1_init_master();       /*  Initialize  LPSPI1  for
communication with MC33903 */
        LPSPI1_init_MC33903();      /* Configure SBC via SPI for
CAN transceiver operation */
#endif

        uint32_t *dataReceived1;    /*Data to save the information
from RX*/
        uint32_t *dataReceived2;    /*Data to save the information
from RX*/

        for(;;)
        {
                delay(5000);
                CAN_Transmitter(CAN_0,          DATA_WORD_1,
DATA_WORD_2);
                CAN_Receiver (CAN_0, dataReceived1, dataReceived2);
        }

        return 0;
}
```

main.h

```
/**
 *	\file	main.c
 *	\brief
 *			This is the main of driver where the driver is
 *			tested with the transmission and reception.
 *	\author ACE TEAM
 *			Andres Hernandez
 *			Carem Bernabe
 *			Eric Guedea
 *	\date	27/03/2019
 */

#include "S32K144.h"
#include "CAN.h"
#include "LPSPI.h"
#include "GPIO.h"
#include "clock_and_modes.h"

#define DATA_WORD_1		(0xA5112233)	/*Data word 1 to transmit*/
#define DATA_WORD_2		(0x44556677)	/*Data word 2 to transmit*/


/*Pointer that saves the information about the configuration about the CAN frame*/
const CAN_Config_t	CAN_Config =
{
	OSCILLATOR_SRC,		/*Source clock*/
	B500KHZ,		/*Bit time*/
		{7,			/*Propagation Segment*/
		4,			/*Phase 1 Segment*/
		4,			/*Phase 2 Segment*/
		1}			/*Sampling bit*/
};

int main(void)
{
	WDOG_disable();				/*Disable the watchdog*/
	ClockConfig();				/*Configure the clock*/

	CAN_init(CAN_0, &CAN_Config);	/* Init FlexCAN0 */
	PORT_init(CAN_0, PORT_E);	/* Configure ports */

#ifdef SBC_MC33903				/* SPI and transceiver initialization is
required */
	  LPSPI1_init_master();		/* Initialize LPSPI1 for communication with MC33903 */
	  LPSPI1_init_MC33903();		/* Configure SBC via SPI for CAN transceiver
operation */
#endif

	  uint32_t *dataReceived1;	/*Data to save the information from RX*/
	  uint32_t *dataReceived2;	/*Data to save the information from RX*/

	  for(;;)
	  {
		delay(5000);
		CAN_Transmitter(CAN_0, DATA_WORD_1, DATA_WORD_2);
```

```
                CAN_Receiver (CAN_0, dataReceived1, dataReceived2);
            }

        return 0;
}
```

6 FUNCTIONAL EVIDENCE



TX payload load and configuration

CAN0->RAMn[0]	uint32_t	138936384
CAN0->RAMn[1]	uint32_t	357826560
CAN0->RAMn[2]	uint32_t	3148528554
CAN0->RAMn[3]	uint32_t	4291624908
CAN0->RAMn[4]	uint32_t	0

RX receiving, data arrived correctly.

CAN0->RAMn[16]	uint32_t	34078784
CAN0->RAMn[17]	uint32_t	357826560
CAN0->RAMn[18]	uint32_t	3148528554
CAN0->RAMn[19]	uint32_t	4291624908
CAN0->RAMn[20]	uint32_t	0

APPENDIX A - GLOSSARY

MCU. - Microcontroller.

CAN. - Controller Area Network.

USB. - Universal Serial Bus.

MB0. –Message Buffer 0.

CANFD. – Controller Area Network Flexible Data-Rate.

FRZACK. – Freeze acknowledge.

TX. – Transfer.

ID. – Identification.

CTRL1. – Control1.

SOF: Start of frame

CRC: cyclical redundancy check

ACK: acknowledge

EOF: end of frame.