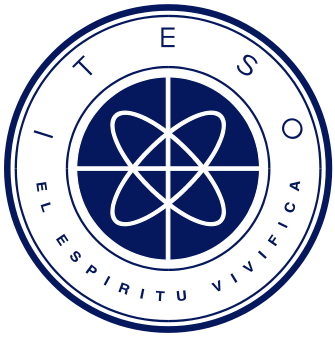
Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Especialidad en Sistemas Embebidos



Implementation of a BUS OFF recovery mechanism on a CAN bus driver.

Tesina para obtener el grado de:

Especialista en sistemas embebidos

Presenta(n)
Alma Karina Pedroza Ornelas
Nombre del director:
Jorge Arturo Pardiñas Mir

San Pedro Tlaquepaque, Jalisco. Diciembre de 2016.

Aknowledgments / Agradecimientos

“To my husband, for being a 4x4 father, otherwise, I would not have been able to attend to school.

A mi marido, pues de no ser un padre 4x4 no hubiera podido atender a clases.

To my dad, who raised me and my siblings without the so called “genres role norms.

A mi papá, que nos crió sin perjuicios ante los llamados ‘roles de género’ “

Abstract

This report presents the implementation of the BUS OFF recovery mechanism on a CAN bus driver.

The CAN protocol defined in 1991 by Robert Bosh GmbH implements 5 error detection mechanisms:

1. Bit Monitoring
2. Bit Stuffing
3. Frame Check
4. Acknowledgement Check
5. Cyclic Redundancy Check

If a node in a CAN bus does not return a recessive bit on its reply for a given time, it will be put in BUS OFF status and will stop transmitting information to the bus, until the BUS OFF status get cleared either manually or automatically. The implementation of this mechanism is made on a Freescale's MC9S12XEP100 and involves modifications to its MSCAN (Scalable Controller Area Network) module configuration bits to enable both the BUS OFF and the Automatic BUS OFF recovery. The implementation also includes functions that will recover the bus from this state, manually, after pressing a button.

This document describes the details on the previously mentioned mechanisms.

Table of Contents

Aknowledgments / Agradecimientos 2
Abstract 3
Introduction 5
Implementation 7
Results 13
Conclusions 15
Table of Figures 16
References 17

Introduction

CAN (Controller Area Network) is a protocol designed by BOSCH in 1991 that has been widely adopted in the automotive sector in order to communicate different devices through a data serial bus, from engine, suspension and traction controls, to lights, doors, seats, instruments and even light and environment devices.

The Data Link Layer of the CAN protocol is defined in the ISO-11898- specification, while the physical layer is defined by the ISO-11898-2 specification. The following image shows a typical CAN bus system with MSCAN (the MC9S12XEP100RMV1 implementation). Each CAN node is physically connected to the CAN bus through a transceiver device.

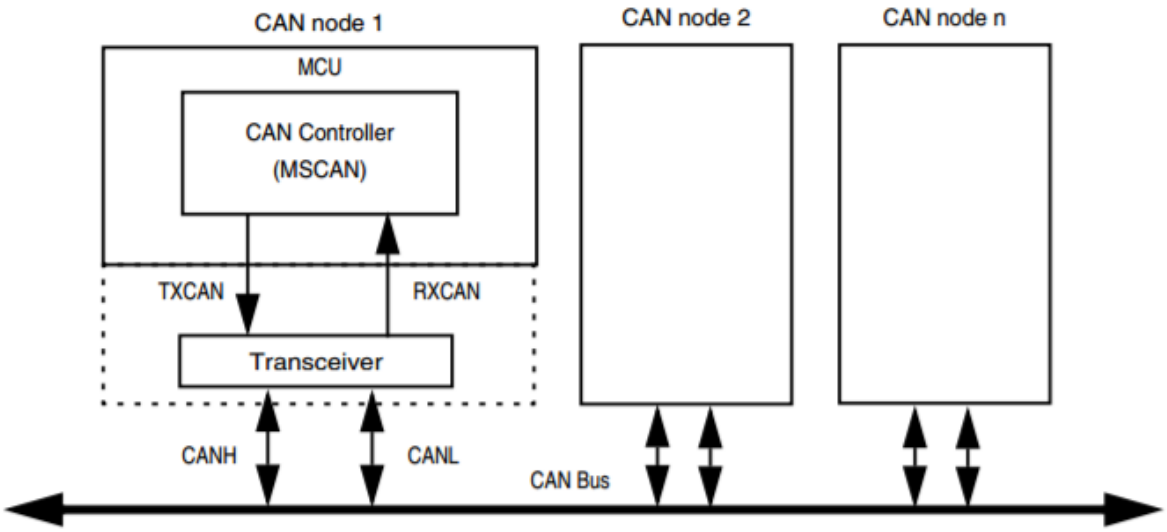


Figure 1- From MC9S12XEP100 Reference Manual, page 608

Please notice that, unlike LIN, the CAN nodes can be connected using different topologies:

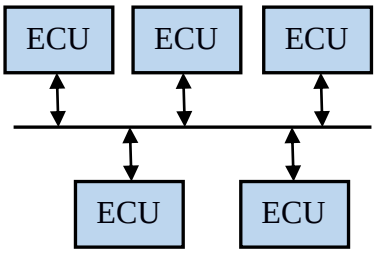


Figure 2- Multi-bus Network

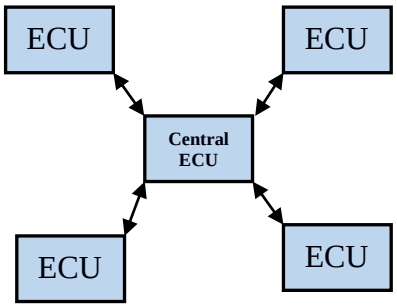


Figure 3- Star Network

The CAN specification is divided in two:

Part A, for the standard format, which implements an 11-bit identifier:

Arbitration Field 11 bits	Control 4 bits	ID	R	Data 8 bits	CRC field 15 bits	ACK Slot	ACK Del	End of frame 7 bits

And part B, which is an extended format that implements a 29-bit identifier:

Arbitration Field 11 bits	Control 4 bits	ID	R2b	Data 8 bits	CRC field 15 bits	ACK Slot	ACK Del	End of frame 7 bits	IFS2-0

The CAN protocol implements 5 error detection mechanisms:

1. Bit Monitoring
2. Bit Stuffing
3. Frame Check
4. Acknowledgement Check
5. Cyclic Redundancy Check

This work is focused on the implementation of the error confinement procedure called BUS OFF, along with the manual and automatic recovery processes.

All nodes in the bus that correctly receive a message shall send a dominant level in the Acknowledgement Slot of the message. The transmitter shall transmit a recessive level in this slot. If the transmitter does not detect a dominant level in the ACK slot, an Acknowledgement Error is reported by transmitting an Error Flag and destroying the bus traffic. When the remaining nodes in the bus detect the error will take actions, for example, discard the message.

Each of the nodes in the bus has two error counters: Transmit Error Counter and Receive Error Counter. The nodes will start in Error Active mode and, if any of the two errors counters raises above 127, the node will be set in the Error Passive state. If the Transmit Error counter rises above 255, the node is set to the Buss Off state.

Nodes in the Buss Off state do not transmit anything on the bus at all.

The error detection and mitigation in a communication protocol is fundamental, especially for CAN, a protocol used in the automotive area, where, an error in a node might lead to life threatening consequences. The Bus OFF mechanism also offers a recovery process for nodes transmitting erroneous information.

Implementation

As mentioned before, there are 5 errors that should be detected and which can be contained by the BUS OFF error mechanism. This work was tested using the Acknowledgement Check error, due to its simplicity to simulate (by configure 0 – Nodes).

Given that embedded systems are finite, the list of nodes has to be setup previously from the beginning. If x (where x > 0) nodes are configured, you can send messages to all, a few or none of them. Nodes are supposed to respond to transmission request with a dominant bit in the ACK slot within the response. If no dominant bit is detected, it is assumed that the message could not be acknowledged.

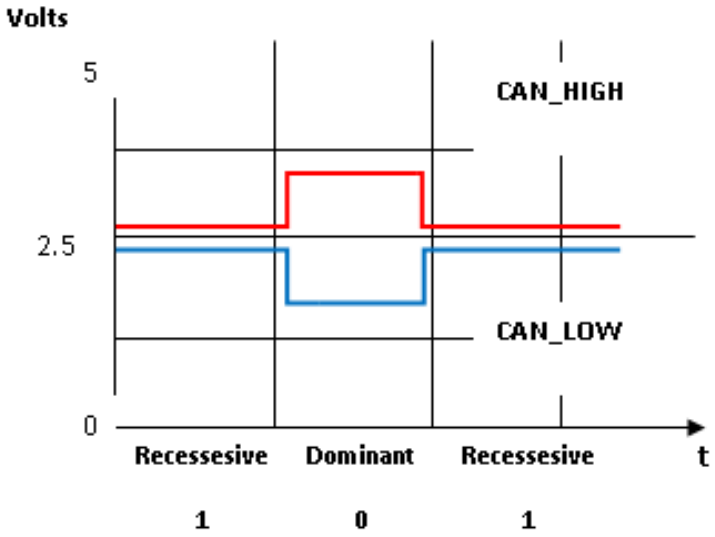


Figure 4- CAN Bus Signal

This behavior will increase either error counter that applies (Transmit or Receive), which, sill set the node into the Error Passive state if they raise 127 or BUS OFF, if the errors raise above 255.

The CAN protocol is very popular in the automotive industry, therefore, the devices connected into the bus might transmit information that might turn into a life threatening event. Not only the implementation must obtain information quickly, but, it must also apply the proper mechanisms to verify its veracity and reliability.

The BUS OFF confinement mechanism helps the failing units to be identified in time and avoids those units to perform harm into the bus, therefore, no more information or nodes can be affected.

Hardware and CAN facilities.

The BUS OFF implementation might be easy to be implemented if the right hardware is selected. The Freescale's MC9S12XEP100, used in the automotive industry, is an example or a complete yet, easy to use, learn and configure card with a CAN implementation. This card already has a MSCAN (Scalable Controller Area Network) and its architecture is shown in Figure 5.

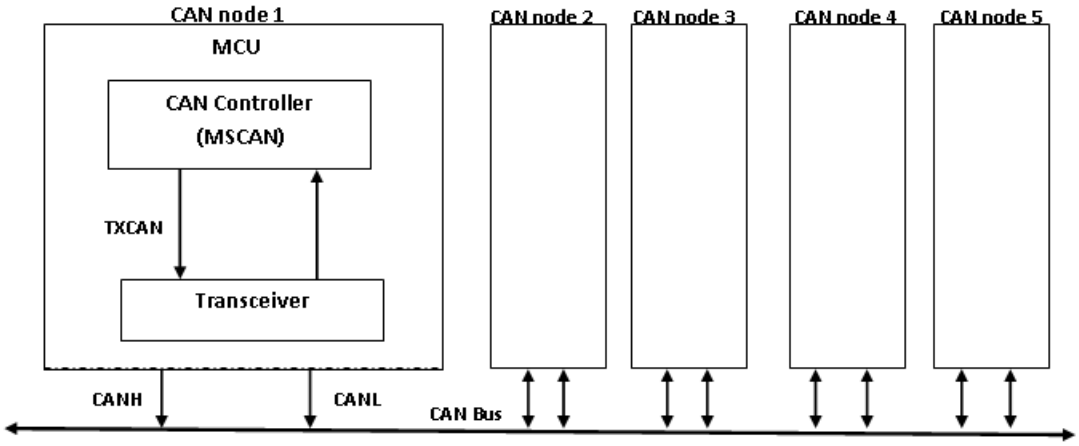


Figure 5 - MSCAN architecture

There are two CAN Control Registers: **CANCTL0** and **CANCTL1.** The CANCTL1 register contains the BROM control bit which configures the **BUS OFF** state recovery mode of the MSCAN as follows:

0 – Automatic BUS OFF recovery
1 – BUS OFF recovery upon user request (manual).

If a BUS OFF condition occurs, an error interrupt is generated. The MSCAN's Receiver Flag Register (**CANRFLG**) would indicate **CAN Status Change.** The error is indicated by the **TSTAT** and **RSTAT** flags.

This work is based on a basic CAN driver for the XEP100 card which manages the transmission, reception and storage of messages on the CAN bus by the MSCAN modules. In order to apply the BUS OFF functionality to the driver, some parameters and configuration files of the driver must be modified.

BUSS OFF functionality

The **UserApp** interface has been added in order be able to identify the BUS OFF status. This interface includes two functions: **LedApp_vBlink** and **App_vCANSupervisor.** The function **App_vCANSupervisor** is the one in charge of checking on the **CAN0RFLG_RSTAT0** and **CAN0RFLG_RSTAT1** registers. If both of them get the value 1, it means that the node has been set into BUS OFF state.

```
void App_vCANSupervisor( void )
{
    static BOOL boErrorFound = FALSE;


    if (boErrorFound == TRUE)
    {

        if (PP0_push_button == 0)
        {
            PTA_PTA1 = 1;
            CAN0MISC_BOHOLD = 1;
            boErrorFound =FALSE;
        }
        else
        {
            PTA_PTA1 = 0;

        }
    }


    if((CAN0RFLG_RSTAT0 == 1) && (CAN0RFLG_RSTAT1 == 1))
    {
     PTA_PTA2 = 1;
     boErrorFound = TRUE;
    }
    else
    {
     PTA_PTA2 = 0;
```

Figure 6 - App_vCANSupervisor: Function in charge to monitor if the BUS OFF status has been set up

When this implementation reaches the bus BUS ERROR and the manual recovery has been configured, one must use the **App_vCANSupervisor** in order to trigger the node recovery mechanism. This function will also poll the **PP0** button's status, to verify if it has been pressed, therefore, the BUS OFF status has been cleared.

The function **LedApp_vBlink** allows you to visually identify by a LED when the node has set itself in the **BUS ERROR** mode, as a feedback to help you debug and take the proper signals observations easier.

```
void LedApp_vBlink( void )
{
    static UINT8 u8Time_Ctr = 0;

    u8Time_Ctr++;
    if (u8Time_Ctr == 5)
    {
        PTA_PTA0 = !PTA_PTA0;
        u8Time_Ctr = 0;
    }

    return;
}
```

Figure 7 - LedApp_vBlink

CAN node configuration structures.

The file conf_mscan.h contains the CAN node configuration structures. As mentioned before, two recovery modes are implemented: manual and automatic and managed by the **tMSCAN_BusRecovery** structure. Given that they are declared as *enum*, the values are as follows:

BUS_OFF_AUTOMATIC_RECOVERY – 0
BUS_OFF_USER_RECOVERY – 1

```
/*****************************************************************************************
* Declaration of module wide TYPES
*****************************************************************************************
typedef enum
{
    BUS_OFF_AUTOMATIC_RECOVERY,                     /**<  RECOVERY default*/
    BUS_OFF_USER_RECOVERY,                          /**<  RECOVERY for user request*/
}tMSCAN_BusRecovery;

typedef enum
{
    MSCAN_A,                            /**< CAN A */
    MSCAN_B,                            /**< CAN B */
```

The following structure **tMSCAN_DeviceConfig** holds the CAN nodes configuration. The *enum* value **tMSCAN_BusRecovery** has been added in order to pass the Bus Recovery mechanism to the device.

```
typedef struct
{
  UINT32                                baudrate;       /**< Baudrate */
    const tMSCAN_RxHWObjectConfig       *rx_hwObj_cfg;   /**< pointer to static Rx hw objects configuration */
    const tMSCAN_TxHwFifoConfig         *tx_hw_fifo_cfg; /**< pointer to static Tx fifo configuration */
    enum tMSCAN_Device                   device;         /**< Device ID */
    enum tMSCAN_AccFilterModeCfg         filter_cfg;     /**< Rx filter acceptance mode configuration */
    UINT8                                nr_of_rx_hwOb;  /**< number of rx hw objects being configured */
    UINT8                                rx_buffer_depth;/**< software buffer depth for each rx hw filter */
    UINT8                                tx_fifo_depth;  /**< software queue depth for transmission purposes */
    enum tMSCAN_BusRecovery              enBusOffMode_cfg;/**< Bus off recovery mode configuration */
}tMSCAN_DeviceConfig;
```

The structures shown above are used for modules, variables and types to be used by **cnf_mscan.c**.

Node configuration.

The configuration for each node in the CAN network must be declared in the file cnf_mscan.c The following declaration shows the configuration for the node A. Similar configurations will be made for the N numbers of nodes you want to add on the bus. As shown in **cnf_mscan.h**, the **BUS OFF** configuration value is set to 0 (automatic)

.

```
/**< Configuration of MSCAN A device */
const tMSCAN_DeviceConfig CAN_device_cfg[] =
{
    {
        (UINT32)CAN_BAUDRATE_500Kbps,     /**< Baudrate */
        &MSCAN_A_rx_msg_cfg[0],             /**< pointer to static Rx hw objects configuration */
        &MSCAN_A_tx_msg_cfg,                /**< pointer to static Tx fifo configuration */
        MSCAN_A,                            /**< Device ID */
        MSCAN_ACC_FILTERS_8_BIT_MODE,       /**< Rx filter acceptance mode configuration */
        sizeof(MSCAN_A_rx_msg_cfg)/sizeof(MSCAN_A_rx_msg_cfg[0]),    /**< number of rx hw objects being configured */
        CAN_RX_BUFFER_DEPTH,                /**< software buffer depth for all the rx FIFO */
        CAN_TX_QUEUE_DEPTH,                 /**< software queue depth for transmission purposes */
        BUS_OFF_AUTOMATIC_RECOVERY          /*or BUS_OFF_USER_RECOVERY*/,
    },
};

    /*
    BUS_OFF_AUTOMATIC_RECOVERY
```

No further modifications were done to this section. This screenshot can be taken as an example, since, you can add as many nodes as you want, as far as you declare the proper values as shown in all of the ***MSCAN_A_<values>***, lines.

The mscan.h file exposes the functions created in the mscan.c, plus, adds different objects needed by the CAN driver.

The macro **MSCAN_CTL1_BORM_MASK** is used to modify the 4th bit of the **MSCAN_CTL1** register, in order to modify the BORM field needed to configure the BUS OFF recovery mechanism.

```
#define MSCAN_MAIN_NODE_OFS                 64

#define MSCAN_CTL1_INITAK_MASK              (UINT8)1
#define MSCAN_CTL1_BORM_MASK                (UINT8)8
#define MSCAN_CTL1_LISTEN_MASK              (UINT8)16
#define MSCAN_CTL1_LOOPB_MASK               (UINT8)32
#define MSCAN_CTL1_CLKSRC_MASK              (UINT8)64
#define MSCAN_CTL1_CANE_MASK                (UINT8)128
```

Initialization of the CAN driver using BUS-OFF recovery.

The last step in the implementation is to initialize the CAN driver using the BUS-OFF recovery mechanism you want to use. This process is done just once, in **the vfnCAN_Init** function.

```
/* Read settings from CTL1 register */
MSCANxCTL1_temp = MSCAN_READ_CTL1(device);
/* Enable MSCAN module */
MSCANxCTL1_temp |= MSCAN_CTL1_CANE_MASK;
/* LoopBack Mode Disabled */
MSCANxCTL1_temp &= ~MSCAN_CTL1_LOOPB_MASK;
/* Listen only mode Disabled */
MSCANxCTL1_temp &= ~MSCAN_CTL1_LISTEN_MASK;
/* Clock source is XTAL */
MSCANxCTL1_temp &= ~MSCAN_CTL1_CLKSRC_MASK;

/*Config Bus Recovery for user request*/
if(BUS_OFF_USER_RECOVERY == mscan_cfg->mscan_device_cfg[device].enBusOffMode_cfg)
{
    MSCANxCTL1_temp |= MSCAN_CTL1_BORM_MASK;
}

/* Write settings back onto CTL1 register */
MSCAN_WRITE_CTL1(device, MSCANxCTL1_temp);

/* Configure Baud Rate as per customer settings */
baudrate =  mscan_cfg->mscan_device_cfg[device].baudrate;
vfnCAN_BaudRateConfig(device, baudrate);
```

Application Program Interface (API)

Once the driver is configured and compiled the user may initialize the driver and be able to apply the BUSS OFF recovery mechanism by using the application program interface (API) comprising the function explained next (definition, input and output).

Service Name:	LedApp_vBlink
Syntax:	LedApp_vBlink(void)
Sync/Async:	
Reentrancy:	
Parameters(in):	None
Parameters(in/out):	None
Return Value:	
Description:	Blinks the LED if BUS ERROR takes place

Service Name:	App_vCANSupervisor
Syntax:	App_vCANSupervisor(void)
Sync/Async:	
Reentrancy:	
Parameters(in):	None
Parameters(in/out):	None
Return Value:	
Description:	Checks if the button PP0 has been pushed.

Results

The following test case has been executed in order to validate the BUS OFF condition:

1. Create a two nodes bus
2. Send a message from Node A to node B
3. Configure the node B so no ACK is transmitted
4. The ACK error shall be reported by node A
5. Node B enters in BUS ERROR state
6. Node B does not transmit anymore

The following are the signals taken from two nodes in a CAN bus during the test case. Both nodes send and receive the information properly.

Figure 8 ad 9 shows the signals corresponding to two nodes in a CAN bus, both, sending and receiving data.

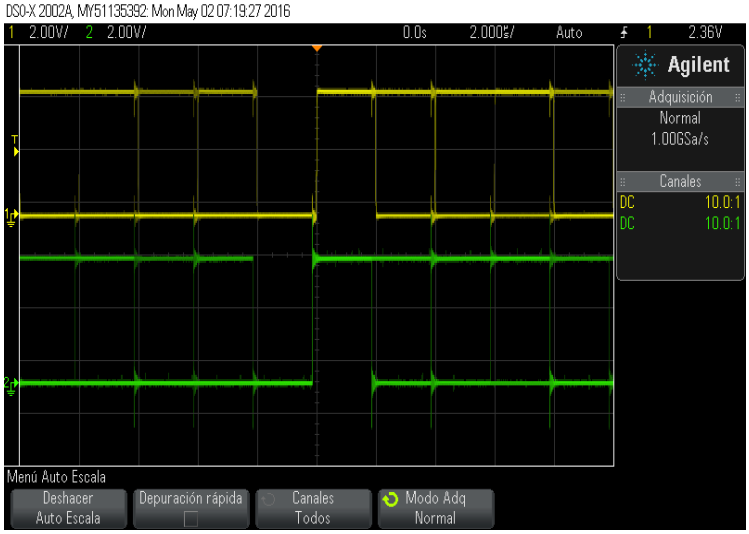


Figure 8 - CAN bus, both, sending and receiving data

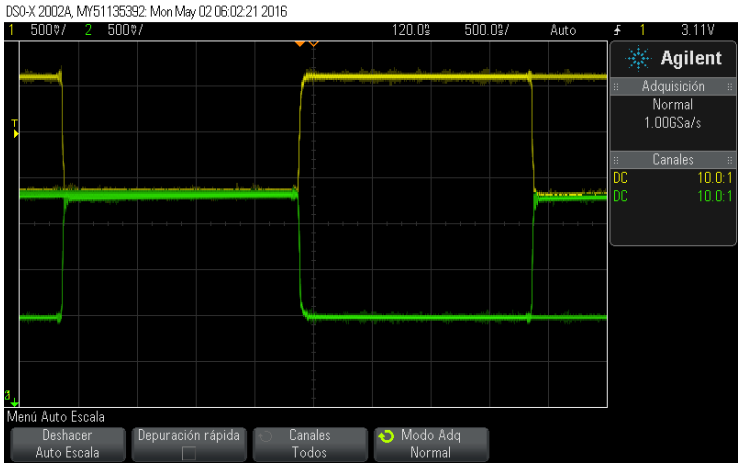


Figure 9- CAN High and CAN Low signals

Figure 10 shows the signals corresponding to the BUS OFF status. As you can see, the node does not send nor receive any information.

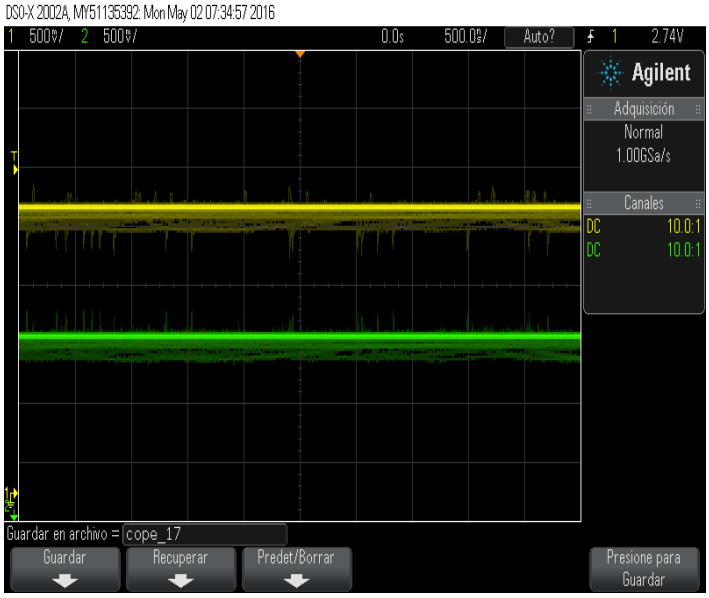


Figure 10 - BUS OFF signal (zoom)

Figure 11 shows the transition from a normal transmission to the BUS OFF status set and recovery, proving that the implementation works correctly.

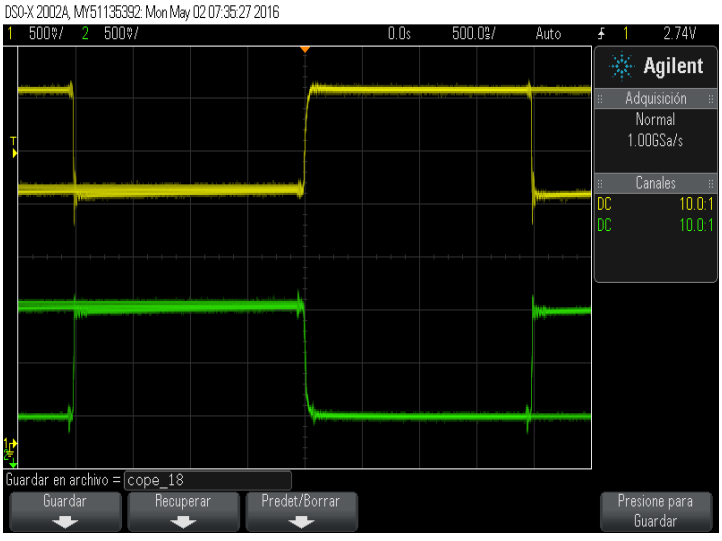


Figure 11 - BUS OFF Set and recovery

Conclusions

The CAN protocol is very popular in the automotive industry, therefore, the devices connected into the bus might transmit information that might turn into a life threatening event. Not only the implementation must obtain information quickly, but, it must also apply the proper mechanisms to verify its veracity and reliability.

The BUS OFF confinement mechanism helps the failing units to be identified in time and avoids those units to perform harm into the bus, therefore, no more information or nodes can be affected.

The BUS OFF implementation might be easy to be implemented if the right hardware is selected. The Freescale's MC9S12XEP100 is an example or a complete yet, easy to use, learn and configure card with a CAN implementation.

Table of Figures

Figure 1- From MC9S12XEP100 Reference Manual, page 608 5
Figure 2- Multi-bus Network 5
Figure 3- Star Network 5
Figure 4- CAN Bus Signal 7
Figure 5 - MSCAN architecture 8
Figure 6 - App_vCANSupervisor: Function in charge to monitor if the BUS OFF status has been set up 9
Figure 7 - LedApp_vBlink 9
Figure 8 - CAN bus, both, sending and receiving data 13
Figure 9- CAN High and CAN Low signals 13
Figure 10 - BUS OFF signal (zoom) 14
Figure 11 - BUS OFF Set and recovery 14

References

Semiconductors, F., s.f. *MC9S12XEP100RMV1 Reference.* [Online] Available at: http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XEP100RMV1.pdf?pspll=1