University of Manouba

National School of Computer Science

ÉCOLE NATIONALE DES SCIENCES DE L'INFORMATIQUE

Implementation of a CAN Sniffer

"Microcontroller Project"

Supervised by:

Mr Mohammed Masmoudi

Realised by:

Mallouli Wassim

Karaa Hela

Academic year 2015-2016

Summary:

I-Introduction

The increase in automobiles armed with internet-connected technology has opened the door for hackers looking to get into our cars remotely. So most of the people nowadays are wondering if their cars can really be hacked easily!?

Some research have been done lately by many security teams and cars constructors all over the world and prove that we can actually lose control of a car if it has to connect to the internet and possess a particular flaw that allows attackers to access its internal network, or the attackers would need physical access to the car. Even us, we were interested on knowing more information about the car‘s internal network and how it’s possible to connect all the electronics modules such as controls unit or intelligent sensors of our cars together and how we can take control of it? The answer was simple the CAN bus system in our cars was doing all this magic work therefore we decide to learn more about this CAN bus and the way it works and how we can analyze its internal messages. So for this project, we created a 2-MCU CAN bus network between two STM32 Discovery boards and we tried to sniff messages sent between this two boards.

II-CAN Bus Description

The controller area network (CAN) was originally designed for industrial environments. Introduced by Bosch in the mid-1980s for in-vehicle communications, it is used in myriad applications including factory automation, building automation, aircraft and aerospace as well as in cars, trucks and buses.
CAN is actually a standard serial differential bus broadcast interface, allowing the microcontroller to communicate with external devices connect to the same network bus.

The CAN interface is highly configurable allowing nodes to easily connect using two wires. Applications benefit from the low-cost, robust, and direct asynchronous serial asynchronous interface.

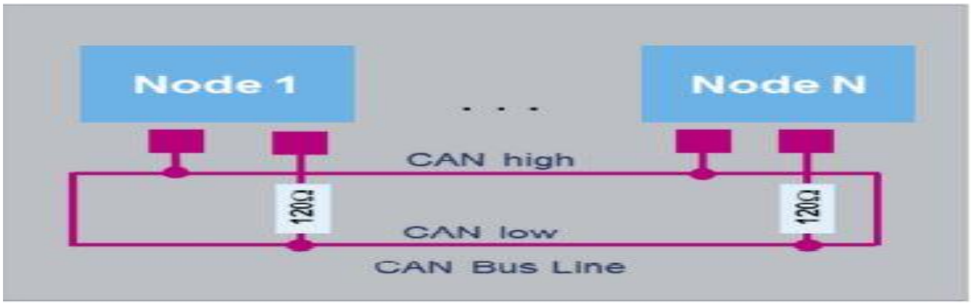


Figure1: CAN Bus: Simple communication with external devices via two pins

CAN provides services at layers 1 and 2 of the OSI model and uses a broadcast method for placing frames on the wire somewhat similar to Ethernet. Bus distance is based on speed, ranging from a maximum of 40 meters at 1 Mbps to a maximum of six kilometers at 10 Kbps. At speeds up to 125 Kbps, CAN provides fault tolerance. If one of the two wires is cut or shorted, the other keeps transmitting.

In a vehicle, both low- and high-speed CAN buses are used. For example, window, lighting and seat control only need low speeds, while engine, cruise control and antilock brakes require high speeds. Two or three CAN buses may be used in a vehicle; for example, a high-speed bus may be dedicated only for safety (air bags, seat belt tensioners, etc.).

In STM32Fx Discovery boards, we found a CAN peripheral supports the basic extended CAN protocol versions 2.0 A and B Active with a maximum bit rate of 1 Mbit/s. This protocol includes 3 transmit mailboxes with a configurable transmit priority option and 2 receive FIFOs with three stages with 14 scalable filter banks. This allows the CAN to efficiently manage a high number of incoming and outgoing messages with a minimum CPU load.
The CAN peripheral also manages four dedicated interrupt vectors.

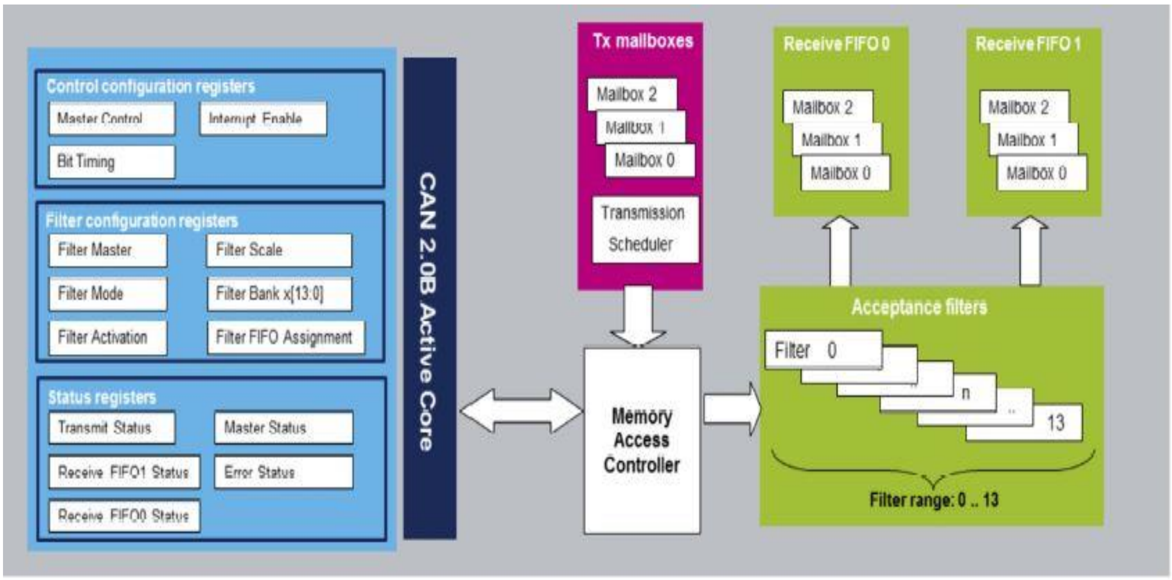


Figure2: A Simplified block diagram of the CAN

For our project we have used BxCAN protocole which had 3 main operating modes:

- Initialization
- Normal
- Sleep

After a hardware reset the BxCAN is in sleep mode which operates at a lower power.
The BxCAN enters Initialization mode via software to allow the configuration of the peripheral.
Before entering Normal mode, the BxCAN must synchronize with the CAN bus, so it waits until the bus is idle.
When the CAN is in Normal mode the user can select whether to run in operation or Test mode.
BxCAN also had 3 test modes: Silent, LoopBack, Combined loopback and Silent.

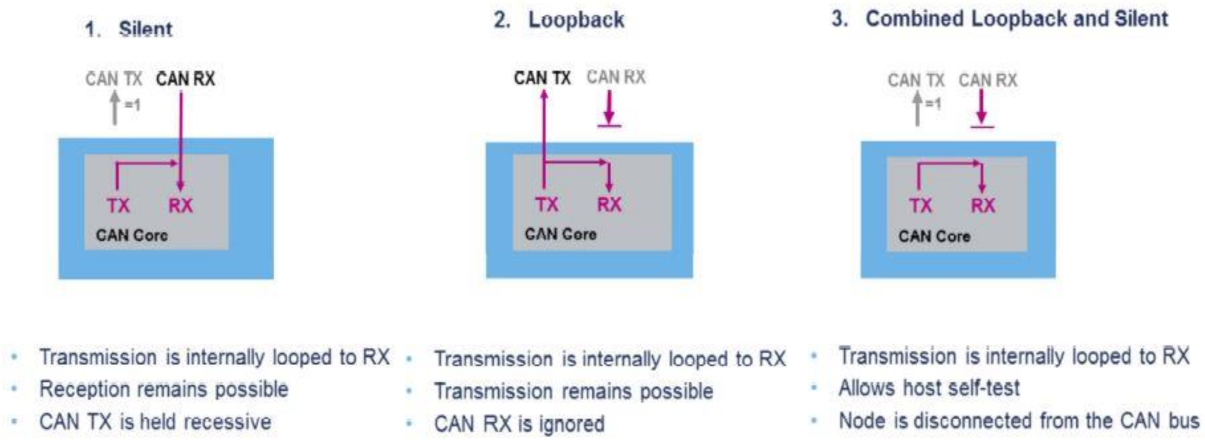


Figure2: BxCAN test modes

We can’t use properly the BxCAN protocol if we are not familiar to the CAN interrupts events, CAN Power modes, and which peripherals may affect our BxCAN behavior

***CAN interrupt events**

CAN interrupt events are Transmit, receive buffers for FIFO0 and FIFO1, and error and status change interrupts

Interrupt event	Description
Transmit interrupt	Set when mailbox is ready to accept a new message.
FIFO 0 interrupt	Set when message is received at FIFO0 (Full or Overrun)
FIFO 1 interrupt	Set when message is received at FIFO1 (Full or Overrun)
Error and Status change interrupts	Set on Error, Wakeup, or Entry into Sleep mode

Figure3: Samples of CAN Interrupt events

***CAN Power Modes**

The CAN low power configuration modes. The device is not able to perform any communications in Stop, Standby of Shutdown modes. It is important to ensure that all CAN traffic is completed before the peripheral enters Stop or Powered-down modes

Mode	Description
Run	Active.
Sleep	Active. Peripheral interrupts cause the device to exit Sleep mode.
Low-power run	Active.
Low-power sleep	Active. Peripheral interrupts cause the device to exit Low-power sleep mode.
Stop 1	Frozen. Peripheral registers content is retained.
Stop 2	Frozen. Peripheral registers content is retained.
Standby	Powered-down. Peripherals must be reinitialized after exiting Standby mode.
Shutdown	Powered-down. Peripherals must be reinitialized after exiting Standby mode.

Figure4: CAN Power Modes

***Peripheral that affect the BxCAN behavior**

-Reset and Clock controller (RCC)

-Interrupts

-General purpose I/Os (GPIO)

II-Sniffer CAN

When connecting external devices to CAN bus in the car or other system, there is always risk of influencing the communication on the bus because of this device. Like elementary mistake as incorrectly set bus speed, if the device is not in Listen only mode, disturbs the communication on the bus, thus the car control units report error and the car can be out of order. **Listen** only mode eliminates this risk and to figure out which device don't work correctly!

In a vehicle, CAN SNIFFER device allows data reading from CAN bus by means of tapping and signal reconstruction at the bus through its scanning from conductors through their insulation. CAN SNIFFER is not conductibly connected with the car bus, thus, the conductors are not disturbed, and communication at this bus cannot be influenced.

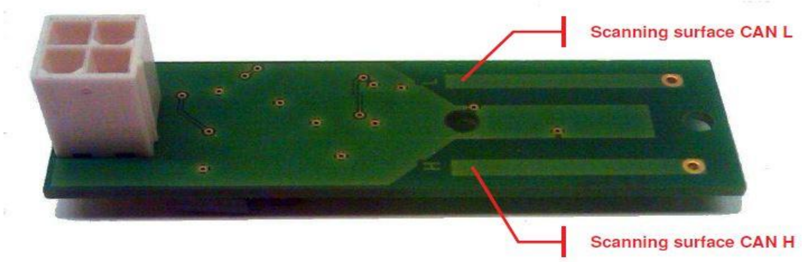


Figure5: Sniffer CAN for cars

Like we see Sniffer CAN are used to avoid risks and to do some tests to devices that are connected to the CAN Bus but we are going to implement a sniffer CAN just to analyze the CAN messages while two STM32 Discovery boards are communicating to each other.

III-System Design:

The system design is divided into part one is about the hardware design and the other part is the Software design.

For the hardware design, we will present the characteristics of our STM32 Discovery boards and the CAN transceiver that help us to realize a communication between two boards and for the software design we will talk about the peripherals that we use it to implement our project.

1-Hardware Design

a- Stm32F407

The datasheet of both of the STM boards were provided by **STMicroelectronics**

STM32F407xx family is based on the high-performance ARM® Cortex®-M4 32-bit RISC core operating at a frequency of up to 168 MHz. The Cortex-M4 core features a Floating point unit (FPU) single precision which supports all ARM single precision data-processing instructions and data types. It also implements a full set of DSP instructions and a memory protection unit (MPU) which enhances application security.

The STM32F407xx family incorporates high-speed embedded memories (Flash memory up to 1 Mbyte, up to 192 Kbytes of SRAM), up to 4 Kbytes of backup SRAM, and an extensive range of enhanced I/Os and peripherals connected to two APB buses, three AHB buses and a 32-bit multi-AHB bus matrix. All devices offer three 12-bit ADCs, two DACs, a low-power RTC, twelve general-purpose 16-bit timers including two PWM timers for motor control, two general-purpose 32-bit timers. A true random number generator (RNG). They also feature standard and advanced communication interfaces.

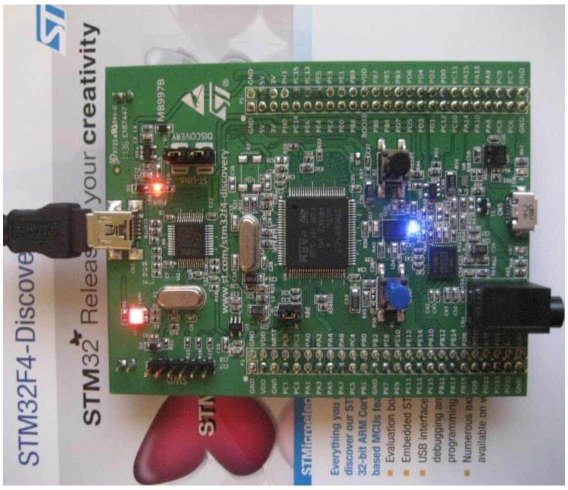


Figure6: Photo of Stm32F407 board

b- Stm32f429i

The STM32F429 Discovery kit (32F429IDISCOVERY) allows users to easily develop applications with the STM32F429 high-performance MCUs with ARM®Cortex®-M4 core.

It includes an ST-LINK/V2 or ST-LINK/V2-B embedded debug tool, a 2.4" QVGA TFT LCD, an external 64-Mbit SDRAM, an ST MEMS gyroscope, a USB OTG micro-AB connector, LEDs and push-buttons.

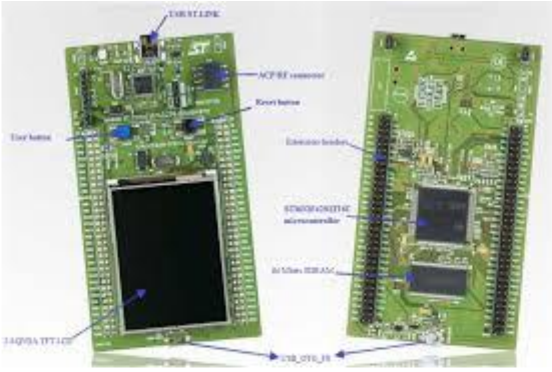


Figure7: Stm32f429i

c- CAN transceiver

The CAN transceiver is the CAN physical layer and interfaces the single ended host CAN protocol controller with the differential CAN bus found in industrial, building automation, and automotive applications. These devices operate over a -2 V to 7 V common mode range on the bus, and can withstand common mode transients of ±25V

There are 3 types of a well-known CAN transceiver SN65HVD230, SN65HVD231, and SN65HVD232.Those transceivers are compatible to the specifications of the ISO 11898-2 high Speed CAN physical layer standard. These devices are designed for data rates up to 1 megabit per second (Mbps), and include protection features providing device and CAN network robustness.

PART NUMBER[1]	LOW POWER MODE	INTEGRATED SLOPE CONTROL	V_{ref} PIN	T_A	MARKED AS:
SN65HVD230	Standby mode	Yes	Yes	40°C to 85°C	VP230
SN65HVD231	Sleep mode	Yes	Yes		VP231
SN65HVD232	No standby or sleep mode	No	No		VP232

Figure 8: Device comparison Table

The different pin configuration and functions of the CAN transceiver are explained throw this picture and the table of pin functions

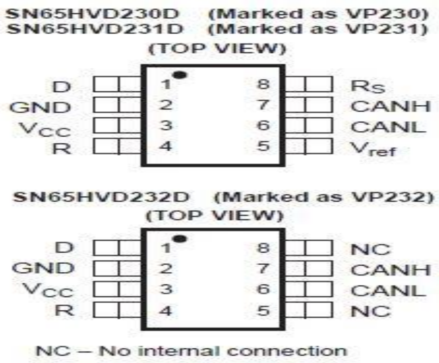


Figure9 : Transceiver‘s4 pins

PIN		TYPE	DESCRIPTION
NAME	NO.		
D	1	I	CAN transmit data input (LOW for dominant and HIGH for recessive bus states), also called TXD, driver input
GND	2	GND	Ground connection
V_{CC}	3	Supply	Transceiver 3.3V supply voltage
R	4	O	CAN receive data output (LOW for dominant and HIGH for recessive bus states), also called RXD, receiver output
V_{ref}	5	O	SN65HVD230 and SN65HVD231: V_{CC} / 2 reference output pin
NC		NC	SN65HVD232: No Connect
CANL	6	I/O	Low level CAN bus line
CANH	7	I/O	High level CAN bus line
R_S	8	I	SN65HVD230 and SN65HVD231: Mode select pin: strong pull down to GND = high speed mode, strong pull up to V_{CC} = low power mode, 10kΩ to 100kΩ pull down to GND = slope control mode
NC		I	SN65HVD232: No Connect

Figure 10: Transceiver‘s pin functions

In our project, we have chosen the transceiver SN65HVD230 because it was the only one available in the market

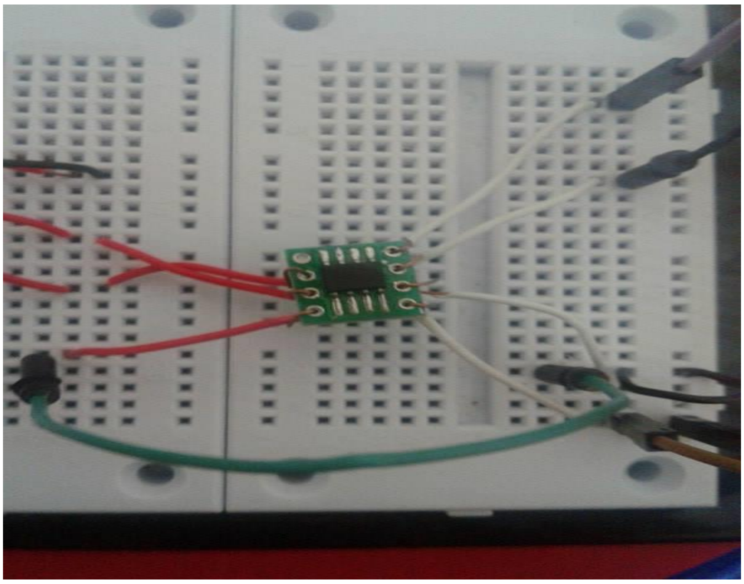
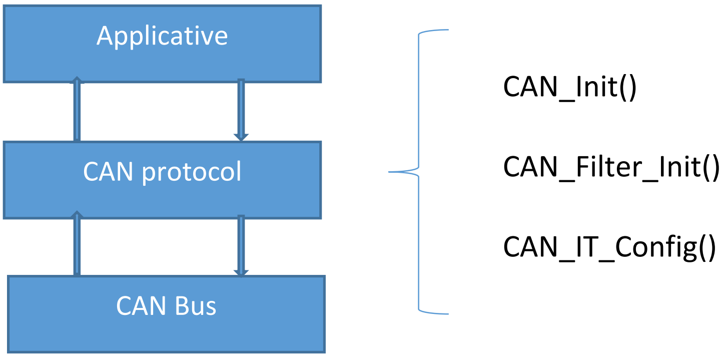


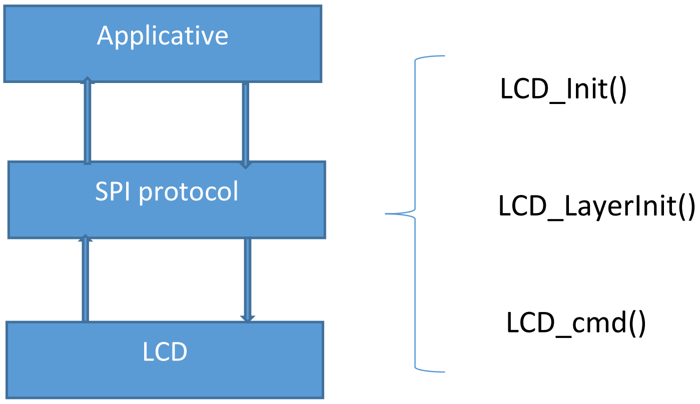
Figure11: A Real Snapshot of the SN65HVD230 transceiver

2-Software Design

- CAN module



- LCD module



V-Achievement:

For this part of the report, we are going to introduce all the tools that we have used to create our project and also we will represent some samples of our software codes

1-IAR Embedded Workbench

IAR Embedded Workbench is a set of development tools for building and debugging embedded system applications using assembler, C and C++.

It provides a completely integrated development environment that includes a project manager, editor, build tools and the C-SPY debugger.

2- Examples of our codes:

To implement our Sniffer CAN, We had to do first the configuration of the CAN protocol on both of the STM boards and be certain that the communication between those two boards worked correctly.

All the following pictures represent parts of our code written in IAR Embedded Workbench in order to implement the CAN protocol configuration on our STM boards.

main.c * | stm32f4xx_it.c | stm32f4xx_can.h | main.h | stm32f4xx_can.c | system_stm32f4xx.c

```c
/* CAN CONFIG */
void CAN_Config(CAN_InitTypeDef CAN_InitStructure)
{
  /* Enable GPIO clock */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
  /* Connect CAN pins to AF9 */
  GPIO_PinAFConfig( GPIOD, GPIO_PinSource0, GPIO_AF_CAN1);
  GPIO_PinAFConfig( GPIOD, GPIO_PinSource1 , GPIO_AF_CAN1);
  /* Configure CAN RX and TX pins */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
  /* CAN configuration ********************************************************/
  /* Enable CAN clock */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
   /* CAN register init */
  CAN_DeInit(CAN1);
 /* CAN cell init */
  CAN_InitStructure.CAN_TTCM = DISABLE;
  CAN_InitStructure.CAN_ABOM = DISABLE;
  CAN_InitStructure.CAN_AWUM = DISABLE;
  CAN_InitStructure.CAN_NART = DISABLE;
  CAN_InitStructure.CAN_RFLM = DISABLE;
  CAN_InitStructure.CAN_TXFP = DISABLE;
  CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
  CAN_Init(CAN1, &CAN_InitStructure);
  if(CAN_Init(CAN1, &CAN_InitStructure))
  {
    GPIO_SetBits(GPIOD,GPIO_Pin_15);
  }
   /* CAN filter init */
```

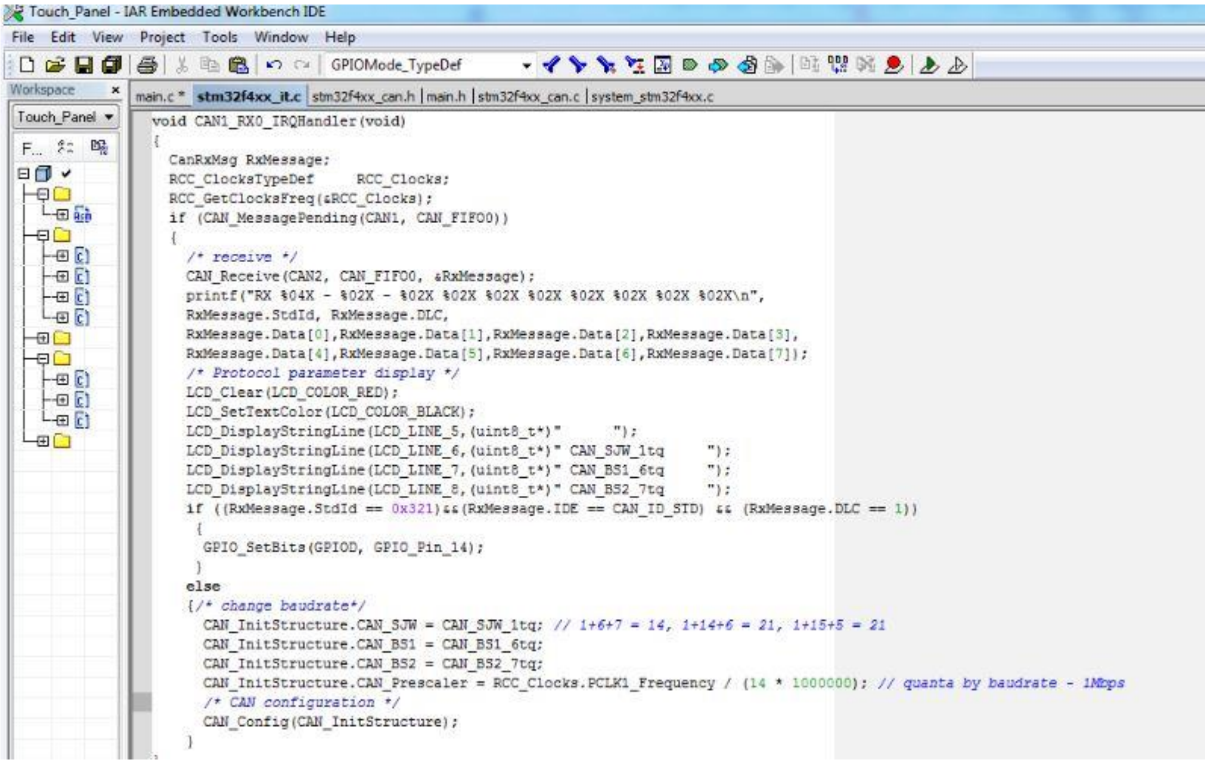
main.c * | stm32f4xx_it.c | stm32f4xx_can.h | main.h | stm32f4xx_can.c | system_stm32f4xx.c

```c
   /* CAN filter init */
  CAN_FilterInitStructure.CAN_FilterNumber = 0;
  CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;
  CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
  CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
  CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
  CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0x0000;
  CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
  CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;
  CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
  CAN_FilterInit(&CAN_FilterInitStructure);

  /* Transmit Structure preparation */
  TxMessage.StdId = 0x321;
  TxMessage.ExtId = 0x01;
  TxMessage.RTR = CAN_RTR_DATA;
  TxMessage.IDE = CAN_ID_STD;
  TxMessage.DLC = 1;

  /* Enable FIFO 0 message pending Interrupt */
  CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);
}
```

Figure 12: CAN configuration




```c
void CAN1_RX0_IRQHandler(void)
{
  CanRxMsg RxMessage;
  RCC_ClocksTypeDef     RCC_Clocks;
  RCC_GetClocksFreq(&RCC_Clocks);
  if (CAN_MessagePending(CAN1, CAN_FIFO0))
  {
    /* receive */
    CAN_Receive(CAN2, CAN_FIFO0, &RxMessage);
    printf("RX %04X - %02X - %02X %02X %02X %02X %02X %02X %02X %02X\n",
    RxMessage.StdId, RxMessage.DLC,
    RxMessage.Data[0],RxMessage.Data[1],RxMessage.Data[2],RxMessage.Data[3],
    RxMessage.Data[4],RxMessage.Data[5],RxMessage.Data[6],RxMessage.Data[7]);
    /* Protocol parameter display */
    LCD_Clear(LCD_COLOR_RED);
    LCD_SetTextColor(LCD_COLOR_BLACK);
    LCD_DisplayStringLine(LCD_LINE_5,(uint8_t*)"      ");
    LCD_DisplayStringLine(LCD_LINE_6,(uint8_t*)" CAN_SJW_1tq     ");
    LCD_DisplayStringLine(LCD_LINE_7,(uint8_t*)" CAN_BS1_6tq     ");
    LCD_DisplayStringLine(LCD_LINE_8,(uint8_t*)" CAN_BS2_7tq     ");
    if ((RxMessage.StdId == 0x321)&&(RxMessage.IDE == CAN_ID_STD) && (RxMessage.DLC == 1))
     {
      GPIO_SetBits(GPIOD, GPIO_Pin_14);
     }
    else
    {/* change baudrate*/
      CAN_InitStructure.CAN_SJW = CAN_SJW_1tq; // 1+6+7 = 14, 1+14+6 = 21, 1+15+5 = 21
      CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;
      CAN_InitStructure.CAN_BS2 = CAN_BS2_7tq;
      CAN_InitStructure.CAN_Prescaler = RCC_Clocks.PCLK1_Frequency / (14 * 1000000); // quanta by baudrate - 1Mbps
      /* CAN configuration */
      CAN_Config(CAN_InitStructure);
    }
```

Figure13: Interrupt Handler

main.c * | stm32f4xx_it.c | stm32f4xx_can.h | main.h | stm32f4xx_can.c | system_stm32f4xx.c

```c
/* LCD CONFIG*/
static void TP_Config(void)
{
  /* Clear the LCD */
  LCD_Clear(LCD_COLOR_WHITE);

  /* Configure the IO Expander */
  if (IOE_Config() == IOE_OK)
  {
     LCD_Clear(LCD_COLOR_RED);
    LCD_SetTextColor(LCD_COLOR_RED);
    LCD_SetFont(&Font16x24);
    LCD_DrawFullCircle(100, 50, 3);
    LCD_DisplayStringLine(LCD_LINE_4, (uint8_t*) "                   ");
    LCD_DisplayStringLine(LCD_LINE_5, (uint8_t*) "                   ");
    LCD_DisplayStringLine(LCD_LINE_6, (uint8_t*) " WAITING FOR       ");
    LCD_DisplayStringLine(LCD_LINE_7, (uint8_t*) "   MESSAGE         ");
    LCD_DisplayStringLine(LCD_LINE_8, (uint8_t*) "                   ");
    LCD_DisplayStringLine(LCD_LINE_9, (uint8_t*) "      :D           ");
    LCD_DisplayStringLine(LCD_LINE_10, (uint8_t*)"                   ");
    LCD_DisplayStringLine(LCD_LINE_10, (uint8_t*)"                   ");
  }
  else
  {
    LCD_Clear(LCD_COLOR_RED);
    LCD_SetTextColor(LCD_COLOR_BLACK);
    LCD_DisplayStringLine(LCD_LINE_5, (uint8_t*)"      ");
    LCD_DisplayStringLine(LCD_LINE_6, (uint8_t*)"      ");
    LCD_DisplayStringLine(LCD_LINE_7, (uint8_t*)"      ");
    LCD_DisplayStringLine(LCD_LINE_8, (uint8_t*)"      ");
  }
}
```

Figure13: LCD Configuration

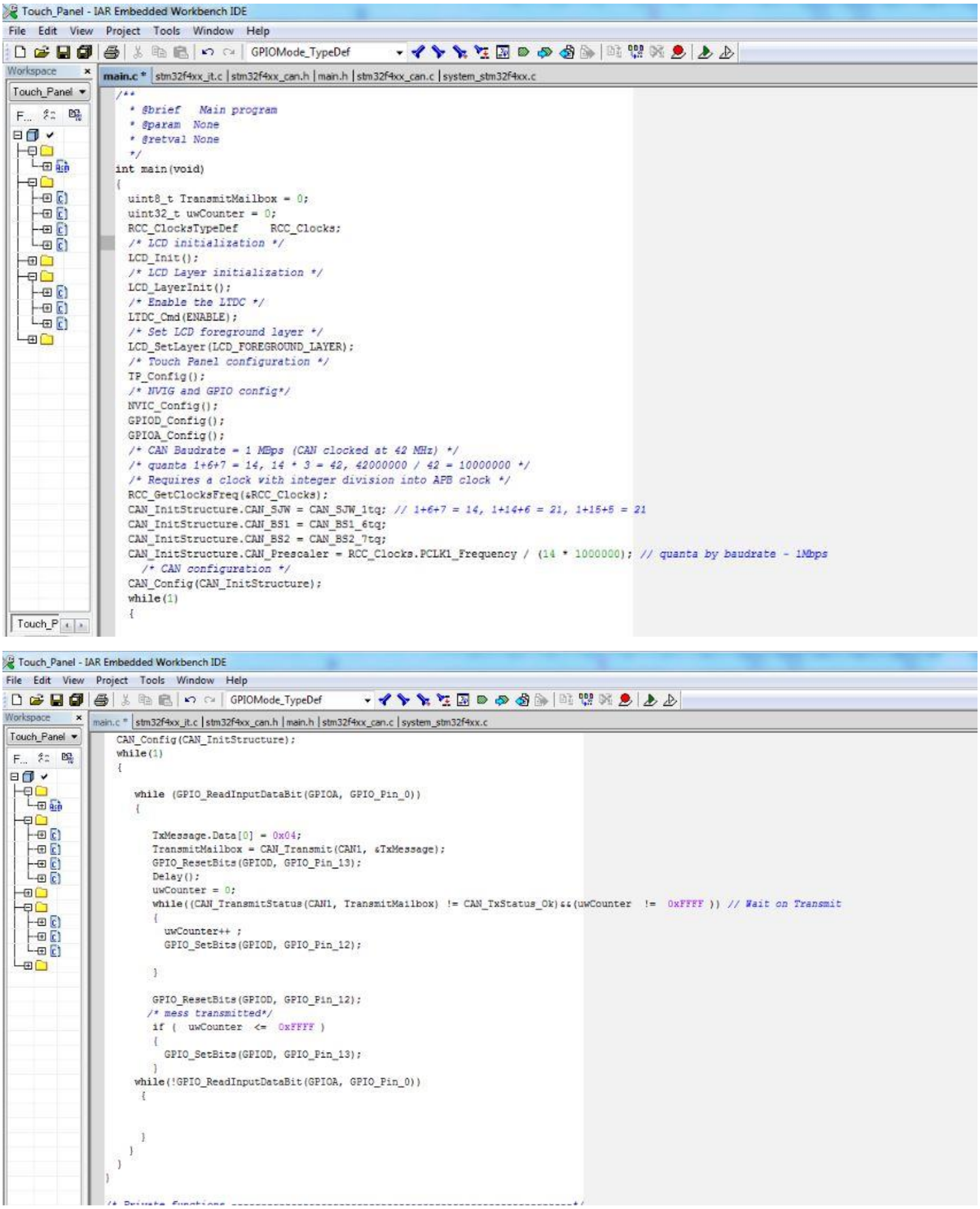


Figure14:MAIN.c

To test our codes, we had to build this whole circuit and to connect those 2 STM boards together using 2 CAN transceivers:

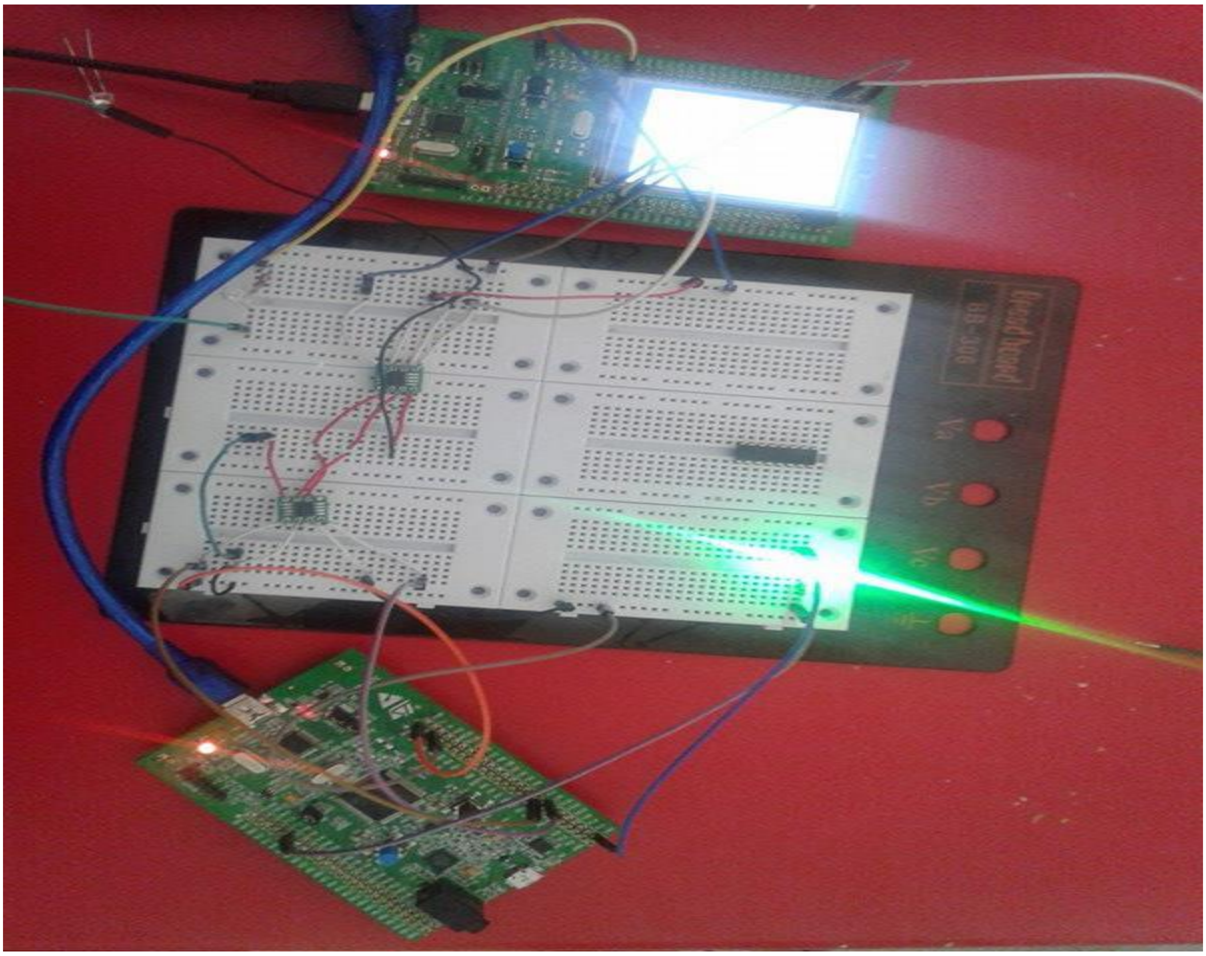


Figure15: STM32F4 Discovery CAN Bus Communication

3-Constraints and difficulties

- We wasted a lot of time on reading documentations
- The communication between the two ST32 boards took a long time to be fixed and done properly.

VI-Conclusion:

In order to analyze and sniff CAN bus messages, we had studied all the characteristics of CAN bus and how we could connect to STM32 devices together using a CAN transceiver and also we found out that it is possible to hack a car if we had an access to its internal network and we used CAN bus to send wrong data for example a hacker can sniff the bus for steering wheel button presses. He pretend to be controllers by sending spoofed data onto the bus like he could send a fake engine RPM to the instrument cluster.

It really interesting to implement a sniffer can for cars and try to write all the data send between devices and to find easily the vulnerable device and try to limit the damage and even our simple project inspired us to work in the field of automobiles and learn more CAN bus features in the future!

Neptography

1-http://www.volkspage.net/technik/ssp/ssp/SSP_238.pdf

2-http://www.ti.com/

3-http://www.st.com/

4- https://www.iar.com/iar-embedded-workbench/

5- http://www.st.com/web/en/resource/technical/document/datasheet/DM00037051.pdf

6-http://hackaday.com/