

**Distance Measurement Controller with CAN Notification**

**Project Report**

Document ID: Distance Measurement Controller with CAN notification  
Origin Date: Mar 16, 2022  
Applicable to: ECE-554 Embedded Systems Winter 2022

Student Name: Luis Castaneda-Trejo

**Revision History:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version No.** | **Date** | **Details of Change** | **Modified by** |
| 0.1 | 29 Mar 2022 | Initial Version. | Luis Castaneda-Trejo |
| 0.2 | 6 April 2022 | Added Main Report Sections | Luis Castaneda-Trejo |
| 0.3 | 7 April 202 | Created Design Workflow | Luis Castaneda-Trejo |
| 0.4 | 12 April 2022 | Added scope images and finished SW Design sec | Luis Castaneda-Trejo |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[Introduction 4](#_Toc100657142)

[1.1 Concept 4](#_Toc100657143)

[1.2 Scope 4](#_Toc100657144)

[Requirements 5](#_Toc100657145)

[2.1 Hardware 5](#_Toc100657148)

[2.2 Software 5](#_Toc100657149)

[Project Elements 6](#_Toc100657150)

[3.1 Hardware -Simulated ECU 7](#_Toc100657155)

[3.2 Hardware -CAN Network 7](#_Toc100657156)

[3.3 Software -Simulated ECU 8](#_Toc100657157)

[3.4 Software -Simulated CAN Network 9](#_Toc100657158)

[3.5 Software -Simulated CAN Network Panel 10](#_Toc100657159)

[Design 12](#_Toc100657160)

[4.1 Project Block Diagram 12](#_Toc100657162)

[4.2 Process of Project Development 13](#_Toc100657163)

[4.3 Software Design 14](#_Toc100657165)

[4.2.1 Controller Area Network (CAN) 14](#_Toc100657171)

[4.2.2 GPIO 15](#_Toc100657172)

[4.2.3 Real-Time Operative System 15](#_Toc100657173)

[4.2.4 Tasks 16](#_Toc100657174)

[4.2.5 Interruptions 18](#_Toc100657175)

[4.2.6 Configuration Services 21](#_Toc100657176)

[4.2.7 Special Implementations 22](#_Toc100657177)

[4.4 Implementation/Code 22](#_Toc100657178)

[Test Results 22](#_Toc100657179)

# Introduction

Autonomous mobility technology is becoming more and more sophisticated every year and OEMs are investing heavily in this area. Not so long ago vehicles didn’t have any equipment installed that could notify the driver that a crash can occur if the vehicle was moving at a certain speed or an object was too close in front or behind that could impact the car. Now days, almost all new models from all OEMs have basic safety features included in their most basic package. In Advanced Driver Assistance Systems (ADAS), LIDAR sensors are used to determine the proximity of objects from the vehicle. The sensors are usually located on top of the vehicle and they scan the surrounding area proving distance information to objects to one of the ADAS ECUs. Most automotive LiDAR sensors have a motor that rotates 360 deg to provide distance information but there are also LiDAR sensors that are unidirectional.

Similar to the LiDAR unidirectional sensors there are ultrasonic sensors like the HS-SR04 that work in a similar way. Rather than using reflective light they use ultrasonic sound to measure the distance to an object. They work OK with materials that are able to reflect sound and the best environment where to use them is indoors with no other noises in similar frequencies that could interfere with them.

## Concept

The purpose of this project is emulating the functionality of an automotive distance measurement controller. Automobiles use LIDAR sensors but since this project is a Proof of Concept an ultrasound sensor can provide similar results in a smaller scale. The measured distance will be categorized as Safe, Warning or Danger and depending on each category a visual alarm will be triggered.

## Scope

The distance measurement controller provides the ability to configure and set new distance categories via CAN

.

# Requirements

This section describes the project requirements for software and hardware.



## Hardware

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Description | Comment | Status |
| HW-001 | Distance Sensor | Sensor to provide a voltage output/signal proportionally to the distance of an object. | HC-SR04 ultrasonic sensor will be used. | OK |
| HW-002 | CAN controller | Platform needs to have a CAN controller to communicate with a test bench at 1Mbit. | The STM32G431 has 1 CAN controller suitable for the project needs. | OK |
| HW-003 | Distance Alarm | Platform needs to have at least 3 available GPIO to work as outputs for the Safe, Warning and Danger zones. | The STM32G431 has more than 3 outputs for the project needs. 3 LEDs will be used to display the alarm | OK |
| HW-004 | CAN Transceiver | CAN transceiver to allow communication with a CAN network | NXP TJA1441AT will be used. | OK |
| HW-005 | CAN Interface | CAN interface tool to communicate with the RT target. | A Vector VN1640 will be used. | OK |
| HW-006 | CAN cable | CAN cable with 120Ohm termination resistors |  | OK |
| HW-007 | RT Target | Microcontroller board | An ST NUCLEO-G431kb board will be used for the project. | OK |
| HW-008 | 5v signal to 3.3v | Voltage divider to allow compatibility of ultrasonic sensor ECHO output with microcontroller voltage levels. | Two 10K Ohm were used as voltage divider at the ECHO output. | OK |

## Software

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Description | Comment | Status |
| SW-001 | ECHO signal | Event to handle the ECHO signal coming from the ultrasonic sensor. | An IRQ was setup to catch this event. See section 4.2.5 a | OK |
| SW-002 | TRIG signal | Output signal with >10 usec high time to stimulate TRIG input | PA4 was set as output with a high time of 100 usec. See section 4.2.2 and 4.2.4 b | OK |
| SW-003 | Distance calculation | Calculate the distance to an object. | Calculation handled by the controller\_handler task in app\_freertos.c. See section 4.2.4 a | OK |
| SW-004 | Distance Threshold Adj | System should allow the adjustment of distance thresholds via CAN. Configuration Service $FE01 | Adjustment handled by the CAN\_Rx\_Ctrlr\_handler task in app\_freertos.c. See section 4.2.4 c | OK |
| SW-005 | Distance Notification Adj | System should allow the notification mode via CAN. Continuous and Event-based. Configuration Service $FE02 | Adjustment handled by the CAN\_Rx\_Ctrlr\_handler task in app\_freertos.c. See section 4.2.4 c | OK |
| SW-006 | CAN filter | System should only allow CAN id 0x726 to be processed. | A CAN id filter was set to only react to 0x726. See function Prepare\_CANFilter inside fdcan.c | OK |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Description | Comment | Status |
| SW-007 | Debounce Logic | Provide a debounce algorithm to prevent the LEDs from switching too fast when the distance falls in the middle of 2 categories. | Debounce algorithm was implemented in the controller\_handler task in app\_freertos.c | OK |
| SW-008 | Distance notification | Distance notification should be sent via CAN. | CAN message containing the distance is handled by CAN\_Tx\_Ctrlr\_handler task in app\_freertos.c | OK |
| SW-009 | Distance zones notification | Distance zone notification should be sent via CAN. | CAN message containing the distance zone is handled by CAN\_Tx\_Ctrlr\_handler task in app\_freertos.c | OK |
| SW-010 | Debug mode | System should have a debug mode reporting status of every task to a UART console | A debug macro was added to the main tasks in app\_freertos.c. | OK |
| SW-011 |  |  |  |  |
| SW-012 |  |  |  |  |
| SW-013 |  |  |  |  |

# Project Elements

This section describes the parts of the Distance Measurement Controller, including interconnection of I/O between the simulated ECU (microcontroller), CAN transceiver, Vector interface and LED for alarm visualization.



## Hardware -Simulated ECU

The ECU hardware consists of an STM32 (Nucleo-G431KB) microcontroller. The selected board has an Arm 32-bit Cortex-M4, 128 Kbytes of Flash and 32 Kbytes of RAM. The microcontroller has 1 FDCAN controller supporting flexible data rate. The FDCAN interface is configured as CAN High Speed (HS) only because the distance measurement application does not require more than 8 bytes for payload.

To communicate with a CAN network the TJA1441AT CAN transceiver from NXP was used. This transceiver supports up to 5 Mbit/s in FD mode. The configured speed for the CAN controller is 1 Mbit/s.

The ultrasonic HC-SR04 sensor was used to measure the distance to an object. With a short 10uS pulse to the trigger the module will send out 8 cycle burst of ultrasound at 40kHz and raise its echo. The echo is a pulse width proportional to the measured distance to the object.

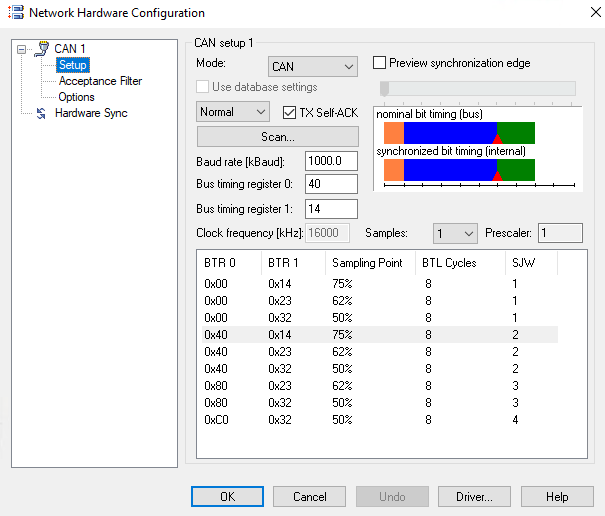
For an easy distance category visualization, an array of 3 LEDs was used. The GREEN LED shows any distance greater than 20 cm. The YELLOW LED shows any distance greater than 10 but less than 20 cm. The RED LED shows any distance less than 10 cm.

The following table summarizes the distance thresholds in the controller:

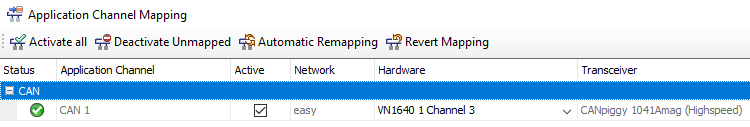
|  |  |  |
| --- | --- | --- |
| Distance Category | Threshold | Color |
| Danger | Less than 10 cm | RED |
| Warning | Greater than 10 cm but less than 20 cm | YELLOW |
| Safe | More than 20 cm | GREEN |

## Hardware -CAN Network

The simulated CAN network provides the right environment to test the ECU. A VN1640A CAN case from Vector was used to interface the ECU to a real CAN network. The VN1640A is a modular interface that supports CAN and LIN interfaces. CAN 3 channel was used as the CAN interface. The following setup was applied to achieve a 1Mbit/s speed network:

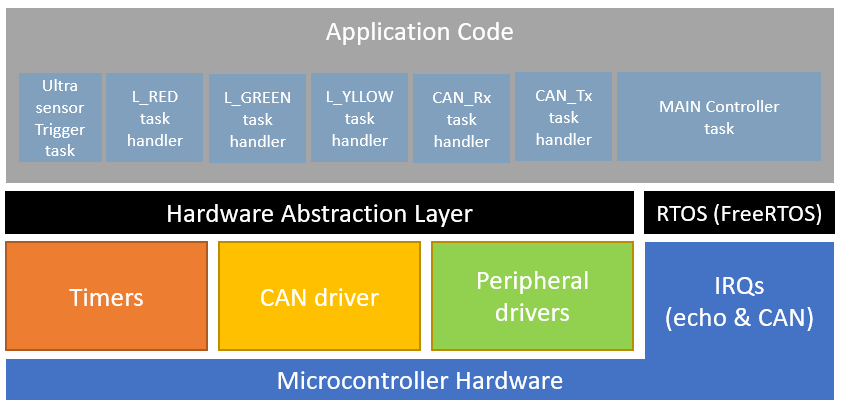


The following image shows the mapping of the CAN channel number used to interface with the ECU.

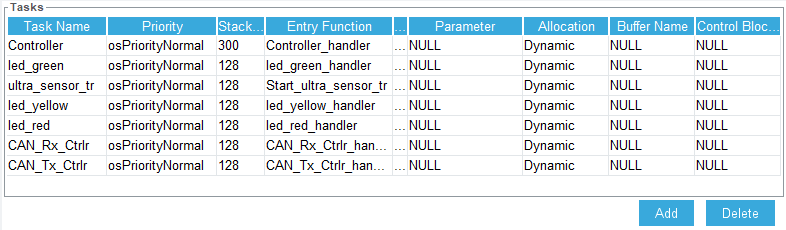


## Software -Simulated ECU

Software in the ECU uses a Real-Time Operative System (FreeRTOS) to handle the tasks of the project. The following figure shows the main software architecture.

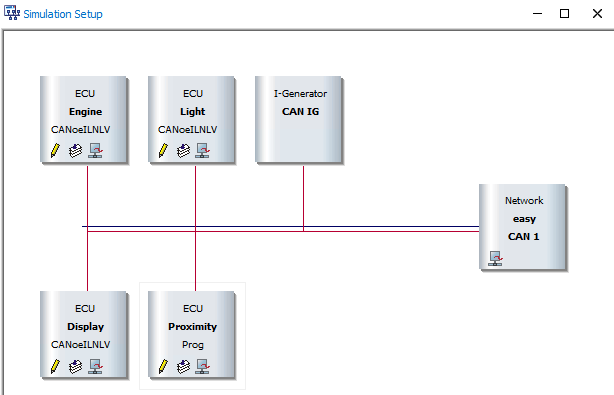


The following table summarizes the FreeRTOS tasks used in the project. All



## Software -Simulated CAN Network

The simulated CAN network was implemented using Vector CANoe. CANoe is a commercial off the shelf software tool to develop, test and analyze individual ECUs and entire ECU networks. It comes preloaded with examples to quickly start analyzing automotive networks. The following CAN network was implemented based on one of the examples that came with the tool and modified to add the proximity ECU in the network. The following figure is the complete CAN network.



CAN msg manual generator

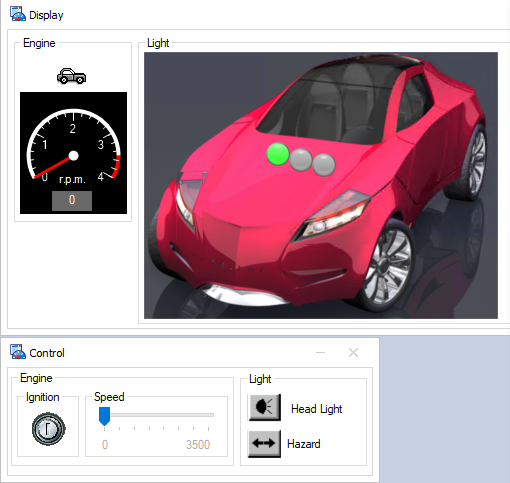
Distance measurement node\*

An interactive node was added to the network to act as a CAN message manual generator. This node is useful to manually change the distance thresholds mentioned in section 2.1. The CAN message structure follows the diagnostics UDS standard described in section zzz.

\*The proximity ECU is just a virtual representation of the real Distance Measurement ECU. This node was created to allow early testing of the real ECU functions and messages without the need of having the real ECU connected.

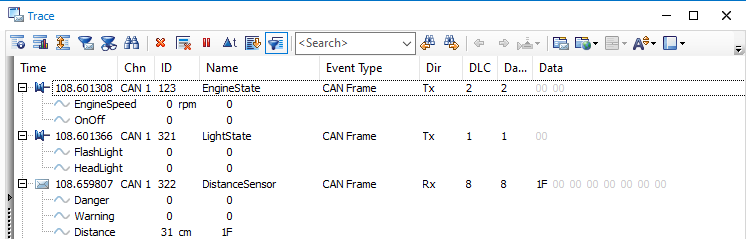
## Software -Simulated CAN Network Panel

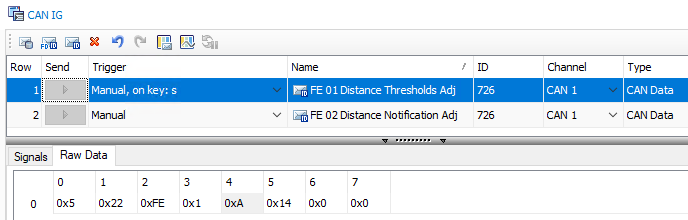
To visualize the distance measurement sent by the ECU, the main control panel of the CANoe example was modified to include the distance values. The following picture shows the main panel including the 3 LEDs that represent the distance categories. In real modern vehicles the distance alarm is usually located inside the cluster or above it.



Distance Alarm LEDs

The following trace window shows the CAN network traffic and configuration services. CAN ID 0x322 is the assigned ID for the distance measurement ECU.



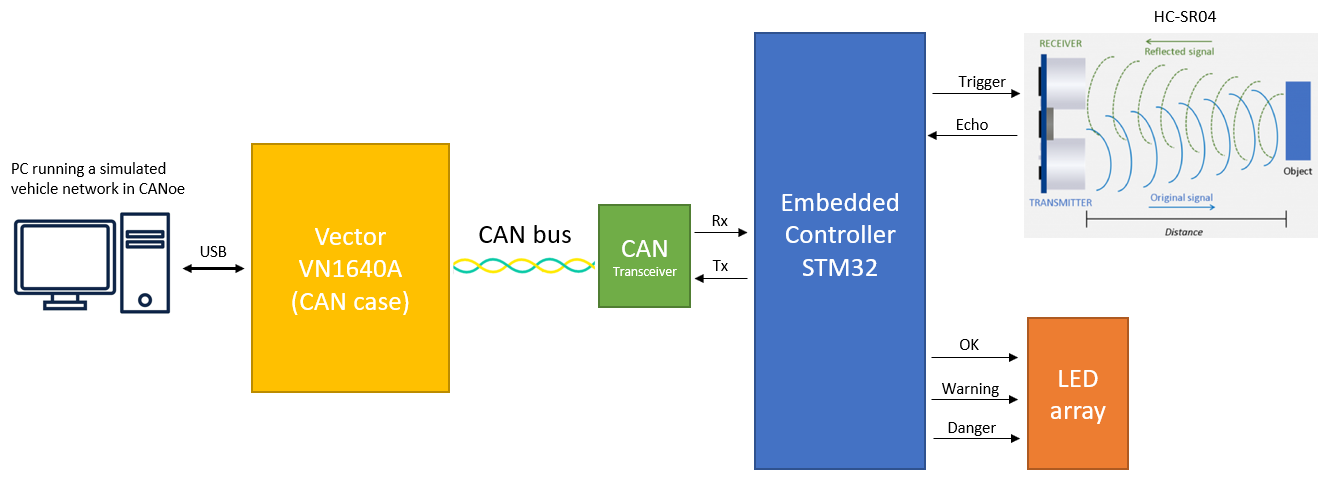


# Design

This section describes the design approach used for the project. The project design has 3 sections: Hardware, Software and Testing. Section 4.2 describes the process for each section.



## Project Block Diagram

The following figure shows the general block diagram of the project.

## Process of Project Development

## 

## Software Design

Software in the embedded controller consist of the following parts: CAN driver



### Controller Area Network (CAN)

The STM32G431 has 1 CAN controller with FD support. This project only uses the controller as CAN High Speed (1 Mbit). The following configuration was applied to the driver in order to achieve a 1Mbit speed:

hfdcan1.Instance = FDCAN1;

hfdcan1.Init.ClockDivider = FDCAN\_CLOCK\_DIV1;

hfdcan1.Init.FrameFormat = FDCAN\_FRAME\_FD\_BRS;

hfdcan1.Init.Mode = FDCAN\_MODE\_NORMAL;

hfdcan1.Init.AutoRetransmission = *ENABLE*;

hfdcan1.Init.TransmitPause = *DISABLE*;

hfdcan1.Init.ProtocolException = *DISABLE*;

hfdcan1.Init.NominalPrescaler = 1;

hfdcan1.Init.NominalSyncJumpWidth = 16;

hfdcan1.Init.NominalTimeSeg1 = 63;

hfdcan1.Init.NominalTimeSeg2 = 16;

hfdcan1.Init.DataPrescaler = 1;

hfdcan1.Init.DataSyncJumpWidth = 4;

hfdcan1.Init.DataTimeSeg1 = 5;

hfdcan1.Init.DataTimeSeg2 = 4;

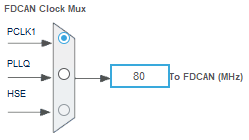
hfdcan1.Init.StdFiltersNbr = 1;

hfdcan1.Init.ExtFiltersNbr = 0;

hfdcan1.Init.TxFifoQueueMode = FDCAN\_TX\_FIFO\_OPERATION;

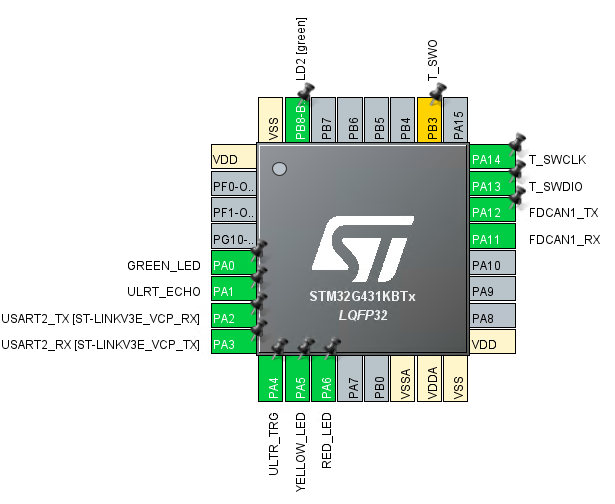
HAL\_FDCAN\_Init(&hfdcan1);

No prescalers were used for the CAN clock. 80 Mhz were set for the main system and applied to the CAN controller:



### GPIO

The following picture shows the GPIO used in the microcontroller:



Main inputs:

* PA1 – Ultrasonic Sensor ECHO
* PA11 – FDCAN1\_Rx

Main outputs:

* PA4 – Ultrasonic Sensor Trigger
* PA12 – FDCAN1\_Tx
* PA0 – Green LED
* PA5 – Yellow LED
* PA6 – Red LED

### Real-Time Operative System

The chosen OS was FreeRTOS. FreeRTOS provides a lightweight RTOS and allows modularization of tasks making it a good framework for this project.

### Tasks

A total of 7 RTOS tasks were created. **Controller\_handler, Start\_ultra\_sensor\_tr, CAN\_Rx\_Ctrlr\_handler, CAN\_Tx\_Ctrlr\_handler, led\_green\_handler** and **led\_yellow\_handler, led\_red\_handler.** All the tasks are in app\_freertos.c

1. **Controller\_handler** This is the main task of the application code, it receives the count value from TIMER2 (timer\_ticks) and calculates the distance using the following formula:

//Multiplied by 0.5 because sound travels from the sensor to object and back.

distance = SPEED\_OF\_SOUND \* TIMER\_PERIOD \* timer\_ticks \* 0.5;

where:

**#define** SPEED\_OF\_SOUND 34300 //Speed of sound in cm/s.

**#define** TIMER\_PERIOD 0.0000000125 //80 Mhz clock. Period = 0.0125 us.

The calculated distance will be categorized as SAFE, WARNING or DANGER depending on the distance threshold values. The **case** *INIT* inside this task initializes the default values and the *MAIN* case contains the zone selection. Once the distance is selected, the *controller\_handler* task will notify the corresponding LED task with the output selection.

This task contains the debounce logic described in section 4.2.7

1. **Start\_ultra\_sensor\_tr** This periodic task controls the trigger of the ultrasonic sensor. It sets ULTR\_TRG\_Pin to high for 100 usec and then puts it to low. The yellow signal in the following picture displays the behavior.

**for**(;;){

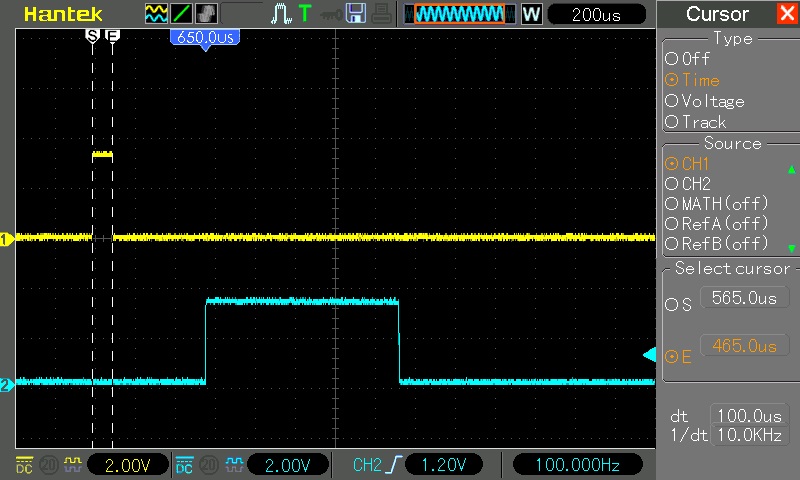
HAL\_GPIO\_WritePin(ULTR\_TRG\_GPIO\_Port, ULTR\_TRG\_Pin, *GPIO\_PIN\_SET*);

delay\_us(100); //The HC-SR10 trigger needs a delay of 10 us as a minimum.

HAL\_GPIO\_WritePin(ULTR\_TRG\_GPIO\_Port, ULTR\_TRG\_Pin, *GPIO\_PIN\_RESET*);

vTaskDelayUntil(&xLastWakeTime, pdMS\_TO\_TICKS(10));

}



**Fig. Trigger and Echo signals**

1. **CAN\_Rx\_Ctrlr\_handler** This task receives confirmation from the CAN interrupt that a CAN message matching the 0x726 identifier has been received. Once a new CAN message is available it gets filtered to see if the payload matches with one of the Configuration Services described in section 4.2.6

If the payload bytes are correct a positive response will be sent back to CANoe with the following format:

CAN\_MessageRx->Tx\_Payload[0] = 0x07; //Sets the 0x762 header for response part 1.

CAN\_MessageRx->Tx\_Payload[1] = 0x62; //Sets the 0x762 header for response part 2.

CAN\_MessageRx->Tx\_Payload[2] = 0xFE; //Sets the Service number.

CAN\_MessageRx->Tx\_Payload[3] = 0x01; //Sets the DID configuration case.

CAN\_MessageRx->Tx\_Payload[4] = CAN\_MSG\_Received.Rx\_Payload[4]; //ACK danger threshold.

CAN\_MessageRx->Tx\_Payload[5] = CAN\_MSG\_Received.Rx\_Payload[5]; //ACK warning threshold.

HAL\_FDCAN\_AddMessageToTxFifoQ(&hfdcan1, &CAN\_MessageRx->TxHeader, CAN\_MessageRx->Tx\_Payload); //Sends positive acknowledgment.

1. **CAN\_Tx\_Ctrlr\_handler.** This task receives notification from the *Controller\_handler* task informing that the distance calculation has been processed and ready to be sent via CAN either in continuous mode or event based. The differences between the two modes are described in Notification Mode Adjustment Service in section 4.2.6

status = xTaskNotifyWait(0, 0, NULL, pdMS\_TO\_TICKS(20));

**if**(status == pdPASS && CAN\_MsgContinious == ON)

HAL\_FDCAN\_AddMessageToTxFifoQ(&hfdcan1, &CAN\_Message1->TxHeader, CAN\_Message1->Tx\_Payload); //Sends the distance to the CAN network.

1. **led\_green\_handler** This and the rest of the LED tasks receive a notification from the *controller\_handler* task to turn on/off their respective LED once the distance has been set and categorized. The following code shows the main logic for all LED tasks:

**for**(;;)

{

status = xTaskNotifyWait(0, 0, &flag, pdMS\_TO\_TICKS(10));

**if**(status == pdPASS){

**switch**(flag){

**case** ON: HAL\_GPIO\_WritePin(GREEN\_LED\_GPIO\_Port, GREEN\_LED\_Pin, ON); **break**;

**case** OFF: HAL\_GPIO\_WritePin(GREEN\_LED\_GPIO\_Port, GREEN\_LED\_Pin, OFF); **break**;

**case** TOGGLE: HAL\_GPIO\_TogglePin(GREEN\_LED\_GPIO\_Port, GREEN\_LED\_Pin); **break**;

**default**: **break**;

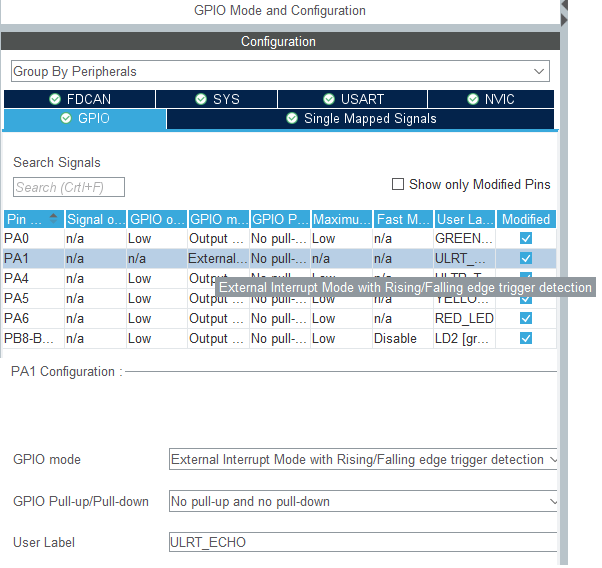
}

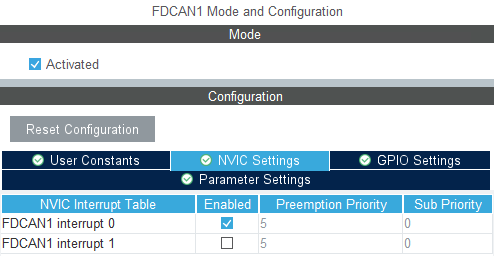
}

### Interruptions

There were 2 interruptions setup for the project: **HAL\_GPIO\_EXTI\_Callback** and **HAL\_FDCAN\_RxFifo0Callback.**

These are callback functions that the IRQ handler provide when the ECHO signal from the ultrasonic sensor gets triggered. The IRQ calling functions can be found in stm32g4xx\_it.c The callback functions are auto generated when an input is set as an interrupt. See the picture below.





1. **Ultrasonic Sensor ECHO signal interrupt**

This interrupt handles the ECHO signal coming from the ultrasonic sensor. The HAL GPIO ReadPin provided the state of the pin, that allowed the interrupt to be classified as a rising or falling edge. Once a falling edge was detected the value of a 32-bit timer was sent to the controller task in FreeRTOS.

**void** **HAL\_GPIO\_EXTI\_Callback**(uint16\_t GPIO\_Pin){

GPIO\_PinState state;

**if**(GPIO\_Pin == ULRT\_ECHO\_Pin) /\* Interrupt function for ECHO signal \*/

{

state = HAL\_GPIO\_ReadPin(ULRT\_ECHO\_GPIO\_Port, ULRT\_ECHO\_Pin);

**switch**(state){

**case** *GPIO\_PIN\_SET*: \_\_HAL\_TIM\_SET\_COUNTER(&htim2, 0); **break**; /\*Rising Edge\*/

**case** *GPIO\_PIN\_RESET*: xTaskNotifyFromISR((TaskHandle\_t)ControllerHandle,

\_\_HAL\_TIM\_GET\_COUNTER(&htim2), *eSetValueWithOverwrite*, NULL); /\*Falling Edge\*/

**break**;

**default**: **break**;

}

}

}

1. **CAN Rx Interrupt**

This interrupt handles when the distance controller ECU receives a message with 0x

**void** **HAL\_FDCAN\_RxFifo0Callback**(FDCAN\_HandleTypeDef \*hfdcan, uint32\_t RxFifo0ITs){

/\* Retrieve Rx messages from RX FIFO0\*/

HAL\_FDCAN\_GetRxMessage(hfdcan, FDCAN\_RX\_FIFO0, &CAN\_MSG\_Received.RxHeader, CAN\_MSG\_Received.Rx\_Payload);

/\* Check if CAN msg is standard CAN and if identifier is the calibration id (0x726) \*/

**if** ((CAN\_MSG\_Received.RxHeader.Identifier == CALIBRATION\_ID) && (CAN\_MSG\_Received.RxHeader.IdType == FDCAN\_STANDARD\_ID)){

xTaskNotifyFromISR((TaskHandle\_t)CAN\_Rx\_CtrlrHandle, 0, *eNoAction*, NULL);

}

}

### Configuration Services

Most ECUs in vehicles support different types of configurations and flashing modes via CAN. The distance controller ECU supports 2 types of configuration services via CAN: $FE01 *Distance Threshold Adj* and $FE02 *Notification Adj.*

The services setup and byte configuration were implemented similar to the ones using the UDS standard. The following tables provide the byte information for each configuration service.

1. **Distance Threshold Adjustment Service**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte  # | Request (0x726 for ECU) à  (CANoe or Node to ECU) | Data/ range  (Hex) | ß Response (0x762)  (ECU to CANoe) | Data/ range  (Hex) |
| 0 | # of Bytes Sent by CANoe  (ex., excluding this byte) | 05 | # of Bytes Sent by ECU  (ex., excluding this byte) | 05 |
| 1 | Service ID (SID) | 22 | Positive response to SID  (Request + 0x40) | 62 |
| 2 | DID number  (ex 0xFE01 = Distance Thresholds) | $FE | Same as Request | $FE |
| 3 | $01 | $01 |
| 4 | Danger threshold | 0-FF | Threshold Acknowledgment | 0-FF |
| 5 | Warning threshold | 0-FF | Threshold Acknowledgment | 0-FF |
| 6 | Not used | 0 | Not used | 0 |
| 7 | Not used | 0 | Not used | 0 |

1. **Notification Mode Adjustment Service**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte  # | Request (0x726 for ECU) à  (CANoe or Node to ECU) | Data/ range  (Hex) | ß Response (0x762)  (ECU to CANoe) | Data/ range  (Hex) |
| 0 | # of Bytes Sent by CANoe  (ex., excluding this byte) | 05 | # of Bytes Sent by ECU  (ex., excluding this byte) | 05 |
| 1 | Service ID (SID) | 22 | Positive response to SID  (Request + 0x40) | 62 |
| 2 | DID number  (ex 0xFE02 = Notification Mode) | $FE | Same as Request | $FE |
| 3 | $02 | $02 |
| 4 | Notification Mode | 1-2 | Notification Acknowledgment | 1-2 |
| 5 | Not used | 0 | Not used | 0 |
| 6 | Not used | 0 | Not used | 0 |
| 7 | Not used | 0 | Not used | 0 |

The implementation of the configuration services can be found in the **CAN\_Rx\_Ctrlr\_handler** task inside app\_freertos.c

### Special Implementations

1. **Debounce Algorithm**

During the testing phase if the distance measurement controller was set to CONTINIOUS REPORTING MODE it was noticed that if the object was in the edge of any of the 3 zones (safe, warning or danger) the alarm LEDs would toggle quickly passing from one zone to the next one.

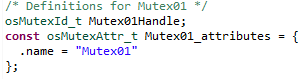
A debounce algorithm was needed to avoid this effect in the physical LEDs and in the CANoe configuration LEDs.

The debounce algorithm is basically a delay that consist of a countdown value that gets triggered every time the measured distance fell into one of the 3 distance categories. The counter will go from 100 (**#define** RESTART 100) to ZERO and during that dwell time the reported distance measurement is the one saved in the memory buffer. Once the 100 cycles passed the debounce flag was set to OFF and would allow a new distance value to be processed.

This implementation can be found in the **Controller\_handler** task inside app\_freertos.c

1. **Debug Mode**

A debug macro was implemented (but not finished) to allow the developer to see basic messages in the UART terminal. This macro was implemented in each of the RTOS tasks to prevent resource sharing issues. The following mutex was used:



## Implementation/Code

The full project can be found in the following GitHub repository:

[ECE-554\_Embedded\_Systems\_Final\_Project](https://github.com/ecastanedat/ECE-554_Embedded_Systems_Final_Project)

# Test Results

1. Project Pictures