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**
E-mail:	**luisct@umich.edu**

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
0.5	13 April 2022	Added corresponding code to the sections	Luis Castaneda-Trejo
0.6	13 April 2022	Added results section	Luis Castaneda-Trejo
1.0	13 April 2022	General Review	Luis Castaneda-Trejo

Table of Contents

Introduction 4
1.1 Concept 4
1.2 Scope 4
Requirements 5
2.1 Hardware 5
2.2 Software 5
Project Elements 7
3.1 Hardware – Distance Measurement Controller (ECU) 7
3.2 Hardware -CAN Network 7
3.3 Software - Distance Measurement Controller (ECU) 8
3.4 Software -Simulated CAN Network (CANoe) 9
3.5 Software -Simulated CAN Network Panel 10
Design 12
4.1 Project Block Diagram 12
4.2 Process of Project Development 13
4.3 Software Design 14
4.2.1 Controller Area Network (CAN) 14
4.2.2 GPIO 15
4.2.3 Real-Time Operative System 16
4.2.4 Tasks 16
4.2.5 Interruptions 18
4.2.6 Configuration Services 21
4.2.7 Special Implementations 22
4.4 Implementation/Code 22
Test Results 23
5.1 Distance Measurement 23
Appendix A Project Picture 24

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 of 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.

1.1 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.

1.2 Scope

The distance measurement controller provides the ability to adjust and set new distance thresholds via CAN as well as the type of distance notification to the network.

.

Requirements

This section describes the project requirements for software and hardware.

2.1 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 1 Mbps.	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

2.2 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

Project Elements

This section describes the parts of the project that were used both in hardware and software.

3.1 Hardware – Distance Measurement Controller (ECU)

The ECU hardware consists of an STM32 (Nucleo-G431KB) microcontroller. The selected board has an Arm 32-bit Cortex-M4 with 128 Kbytes of Flash and 32 Kbytes of RAM. The microcontroller has 1 CAN controller supporting flexible data rate. The CAN 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 the ECHO output. The ECHO is a pulse width signal 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

3.2 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 CANoe setup shown in Fig 1 was applied to achieve a 1Mbps speed network.

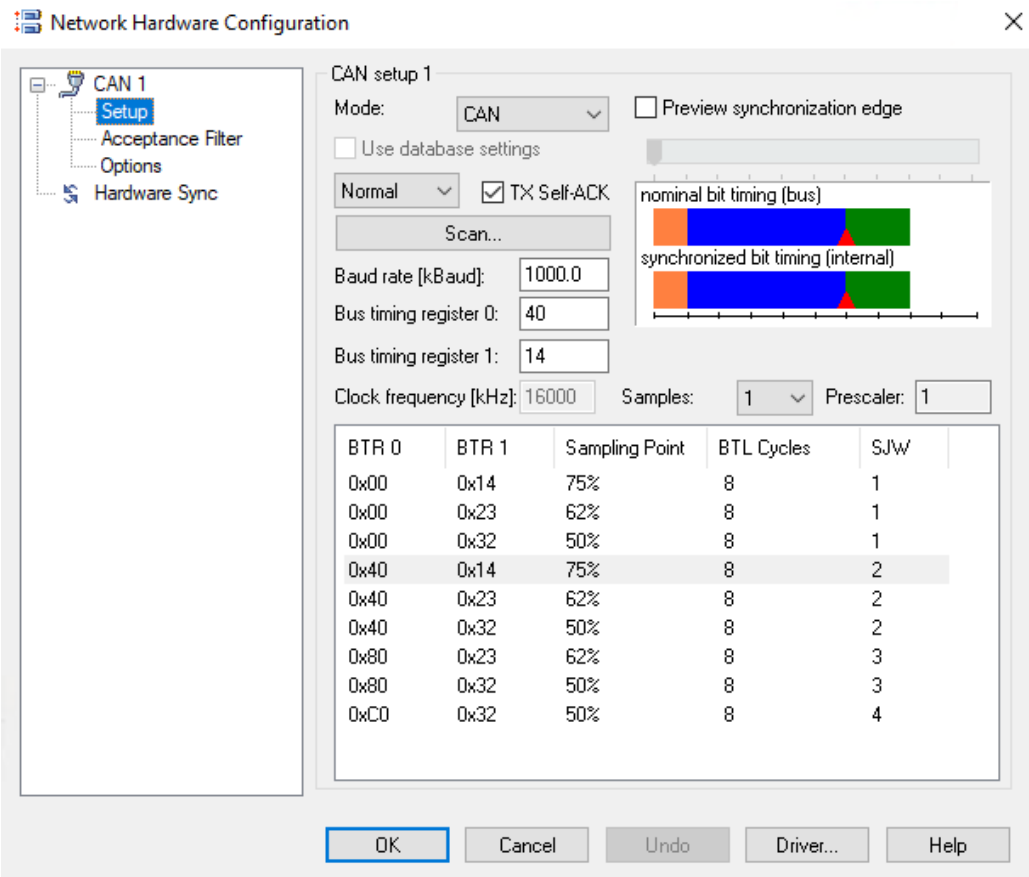

Fig 1. CAN Setup in CANoe

Figure 2 shows the mapping of the CAN channel number used to interface with the ECU.

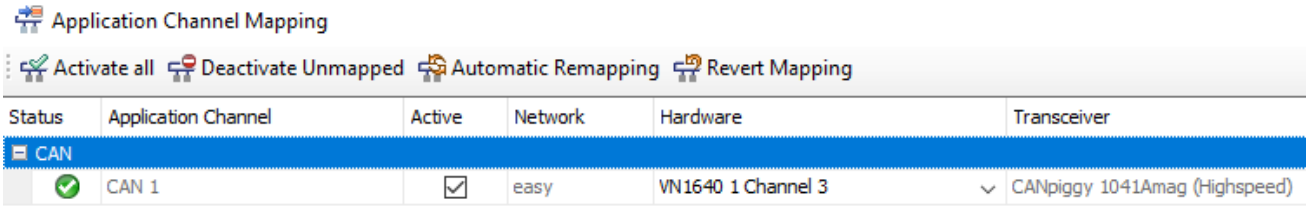

Fig 2. Physical CAN port mapping

3.3 Software - Distance Measurement Controller (ECU)

Software in the ECU uses a Real-Time Operative System (FreeRTOS) to handle the tasks of the project. Figure 3 shows the main software architecture.

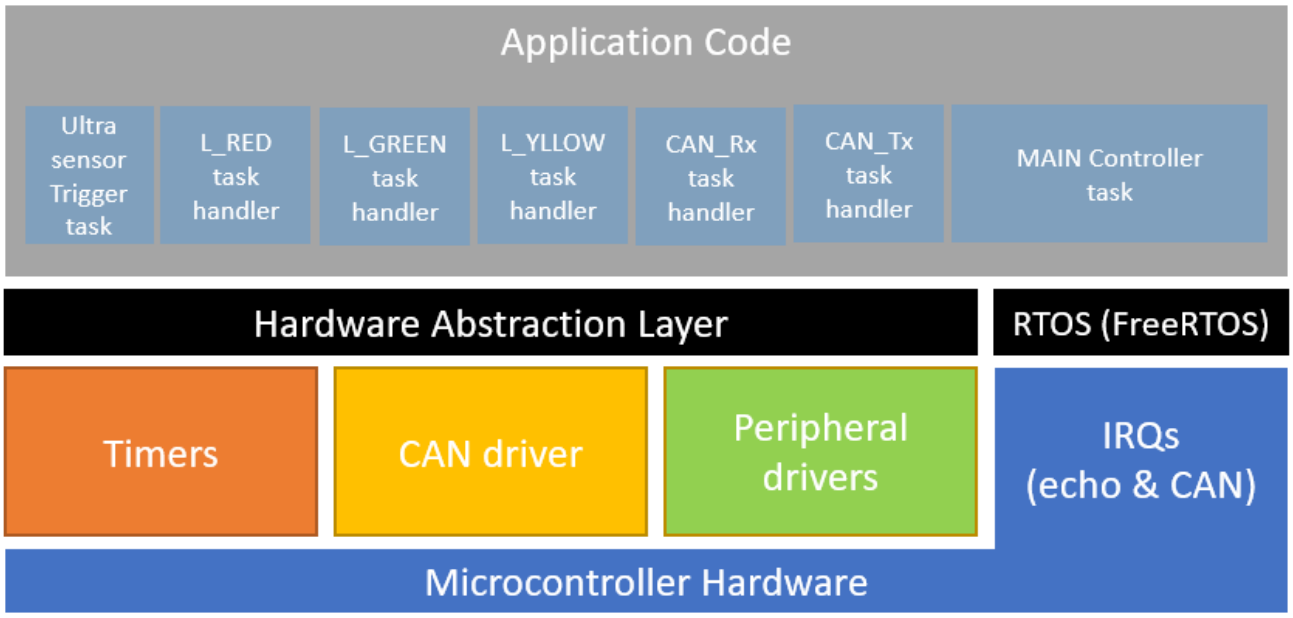

Fig 3. Project SW Architecture

The following table summarizes the FreeRTOS tasks used in the project. Section 4.3 contains a detailed description of each task.

Tasks

Task Name	Priority	Stack...	Entry Function	...	Parameter	Allocation	Buffer Name	Control Bloc...
Controller	osPriorityNormal	300	Controller_handler	...	NULL	Dynamic	NULL	NULL
led_green	osPriorityNormal	128	led_green_handler	...	NULL	Dynamic	NULL	NULL
ultra_sensor_tr	osPriorityNormal	128	Start_ultra_sensor_tr	...	NULL	Dynamic	NULL	NULL
led_yellow	osPriorityNormal	128	led_yellow_handler	...	NULL	Dynamic	NULL	NULL
led_red	osPriorityNormal	128	led_red_handler	...	NULL	Dynamic	NULL	NULL
CAN_Rx_Ctrlr	osPriorityNormal	128	CAN_Rx_Ctrlr_han...	...	NULL	Dynamic	NULL	NULL
CAN_Tx_Ctrlr	osPriorityNormal	128	CAN_Tx_Ctrlr_han...	...	NULL	Dynamic	NULL	NULL

Add Delete

Fig 4. Task List

3.4 Software -Simulated CAN Network (CANoe)

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 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 show the data coming from the distance measurement controller. Figure 5 shows the complete CAN network.

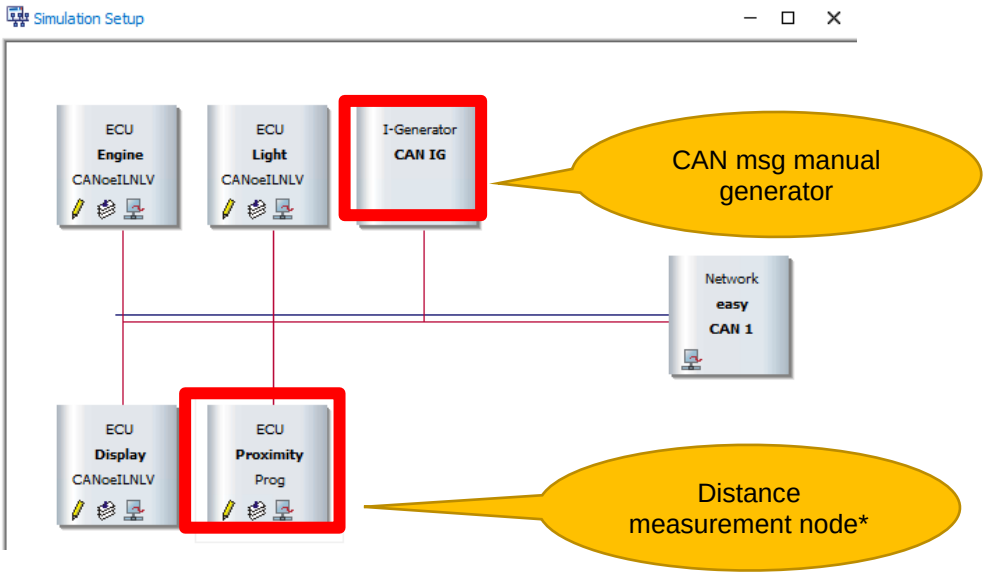

Fig 5. Simulated CAN network

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 3.1. The CAN messages used in the interactive node follow the diagnostics UDS standard described in section 4.2.6.

The proximity ECU shown in Figure 5 is just a virtual representation of the real Distance Measurement Controller. This node was created to allow early testing of the functions and messages without the need of having the real ECU connected.

3.5 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. Figure 6 shows the main panel including the 3 alarm LEDs that represent the distance categories. In real modern vehicles, the distance alarm is usually located inside the cluster or on top of the vehicle dashboard.

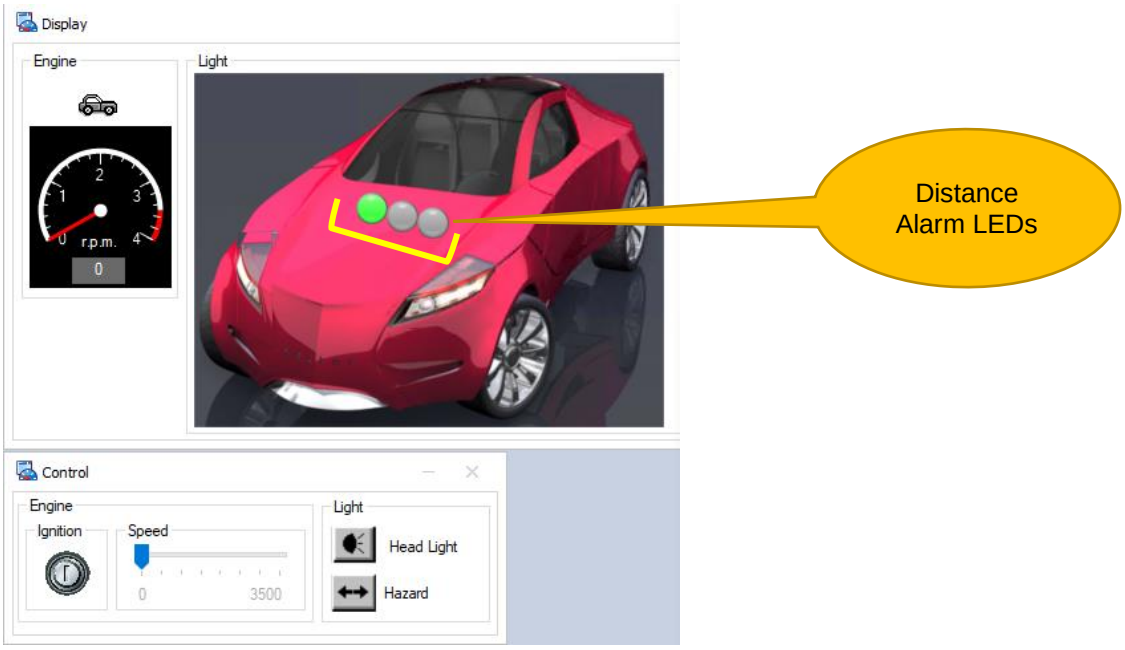

Fig 6. CANoe panels

The trace window in Figure 7 shows the CAN traffic and configuration service messages inside the interactive node mentioned in section 3.4. CAN ID 0x322 is the assigned ID for the distance measurement ECU.

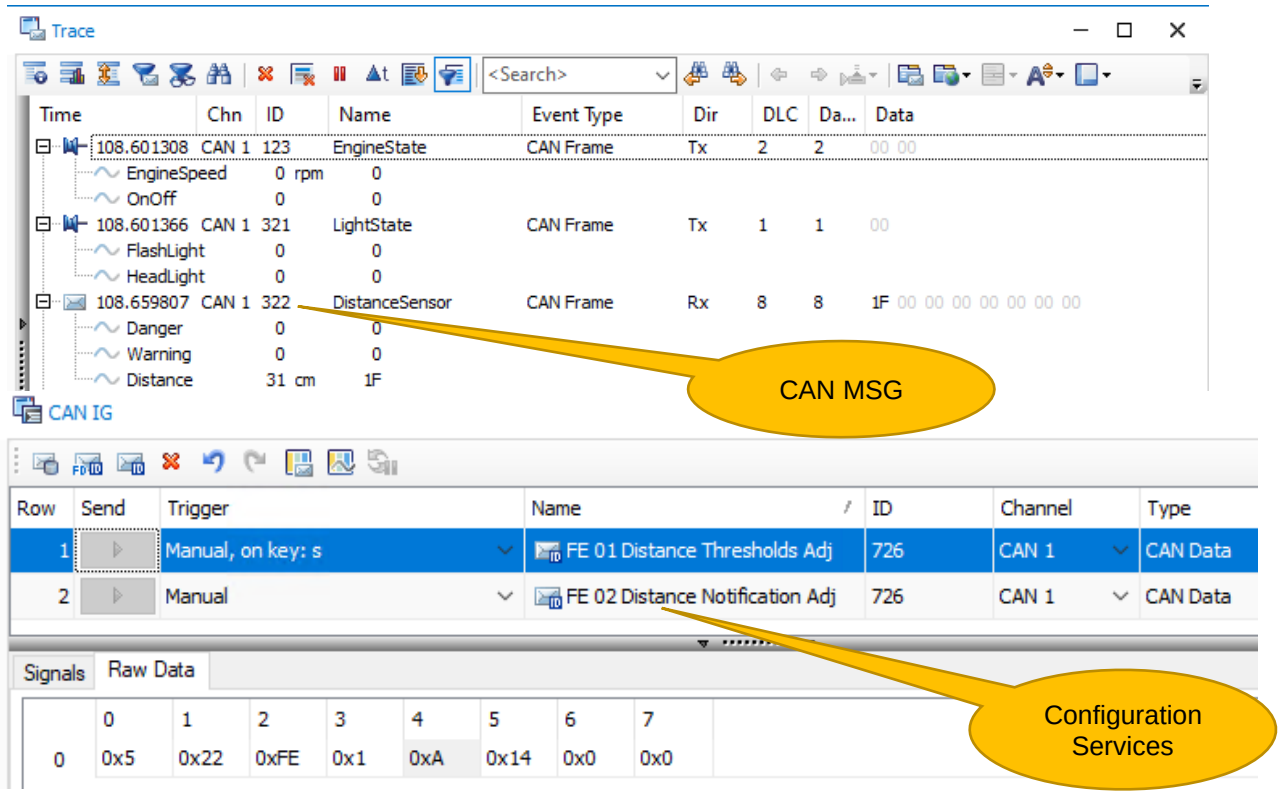

Fig 7. CAN trace and configuration service messages

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.

4.1 Project Block Diagram

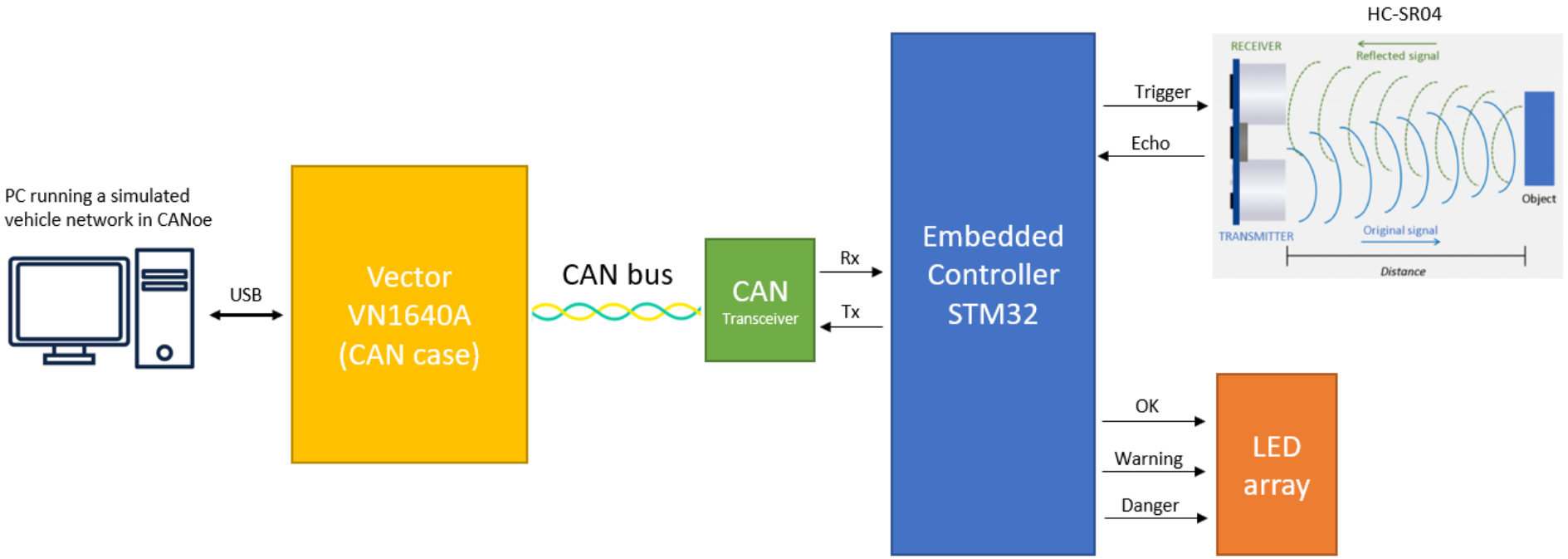

Fig 8. Project Block Diagram

4.2 Process of Project Development

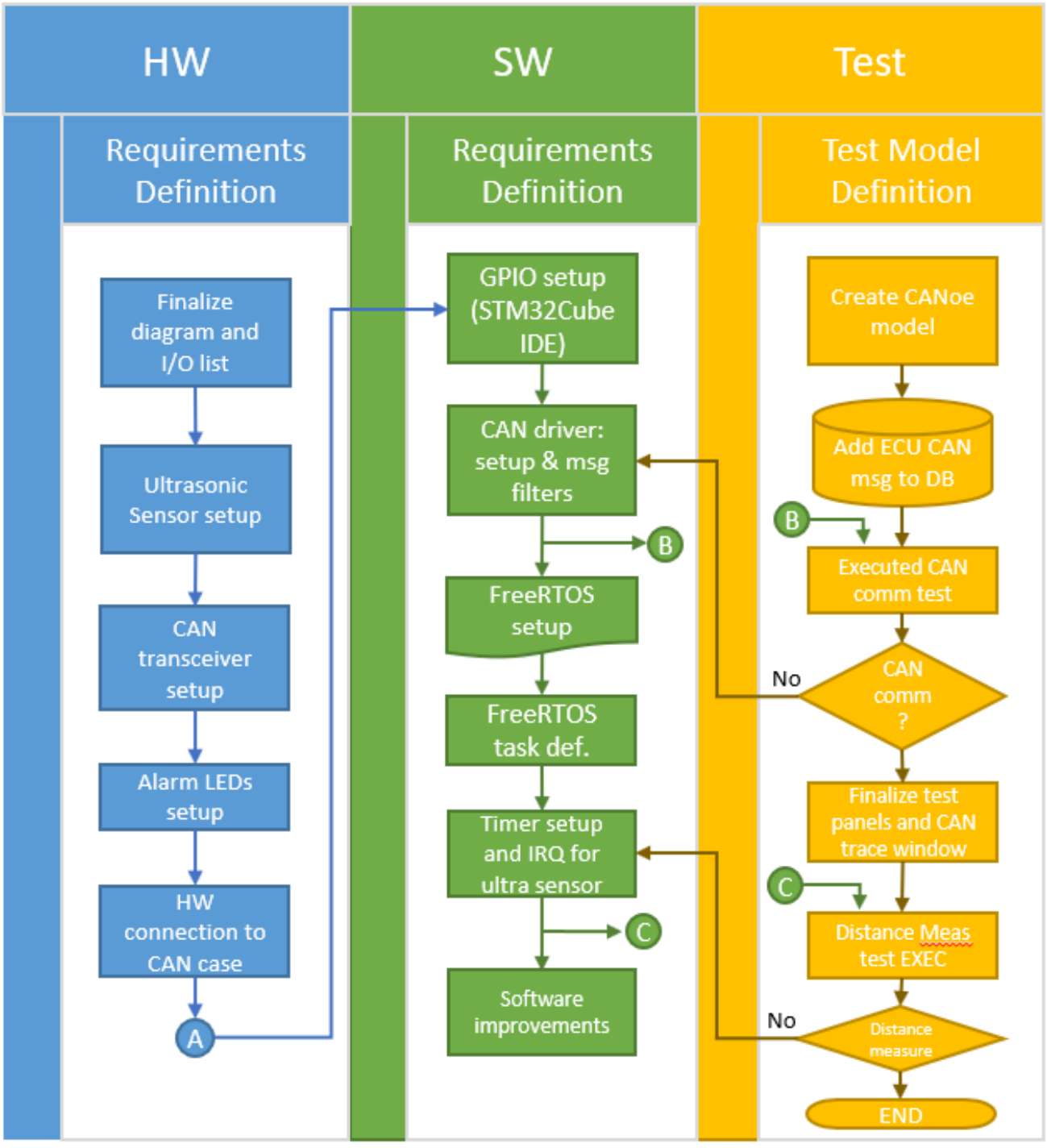

Fig 9. Project Development Process

4.3 Software Design

This section describes the software elements in the distance measurement controller.

4.2.1 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 to achieve a 1 Mbit 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 clock. 80 Mhz were set for the main system and applied to the CAN controller:

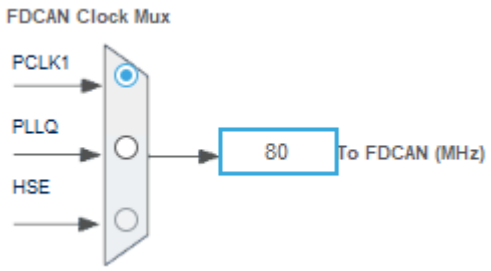

Fig 10. CAN controller source clock

4.2.2 GPIO

The following picture shows the GPIO used in the microcontroller:

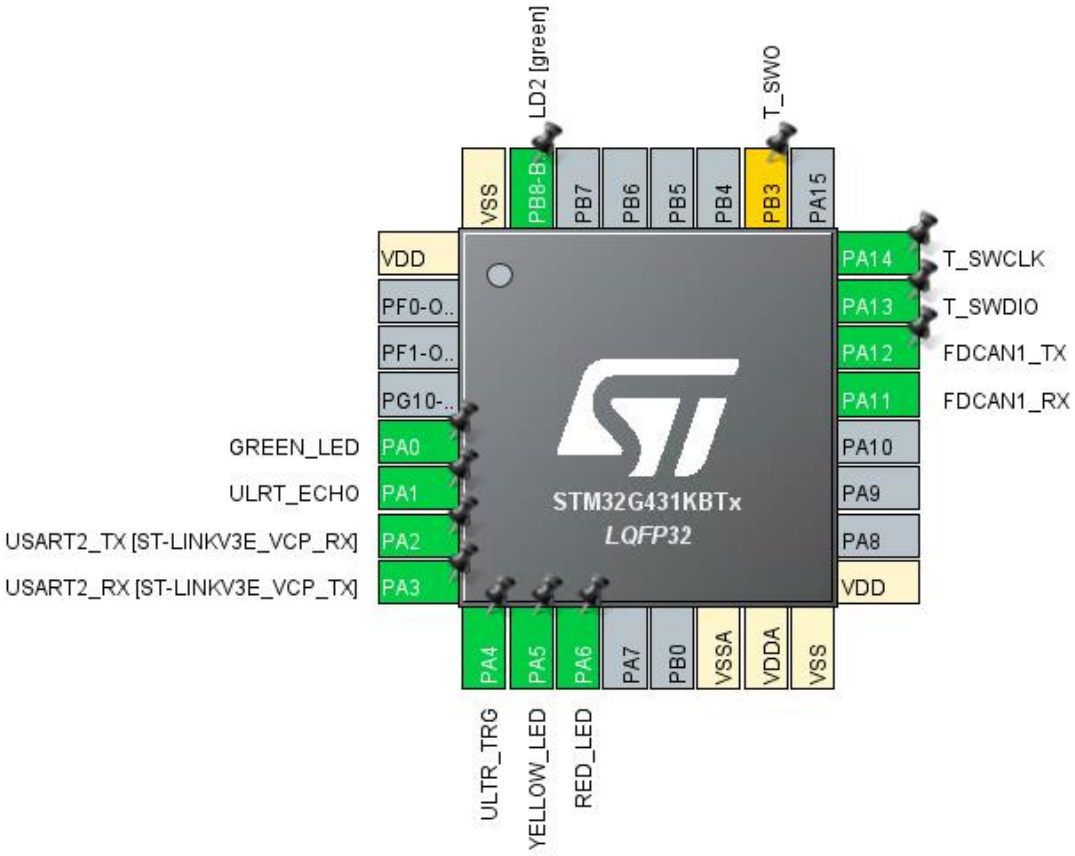

Fig 11. IO Map

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

4.2.3 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.

4.2.4 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

a. **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:

```
//Divided by 2 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

b. **Start_ultra_sensor_tr** This periodic task controls the trigger pin of the ultrasonic sensor. It sets `ULTR_TRG_Pin` to high for 100 usec as and then put back to low. The yellow signal in figure 12 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));
}
```

NOTE: This periodic task was later tested with 10 usec to the trigger and the ultrasonic sensor responded as expected.

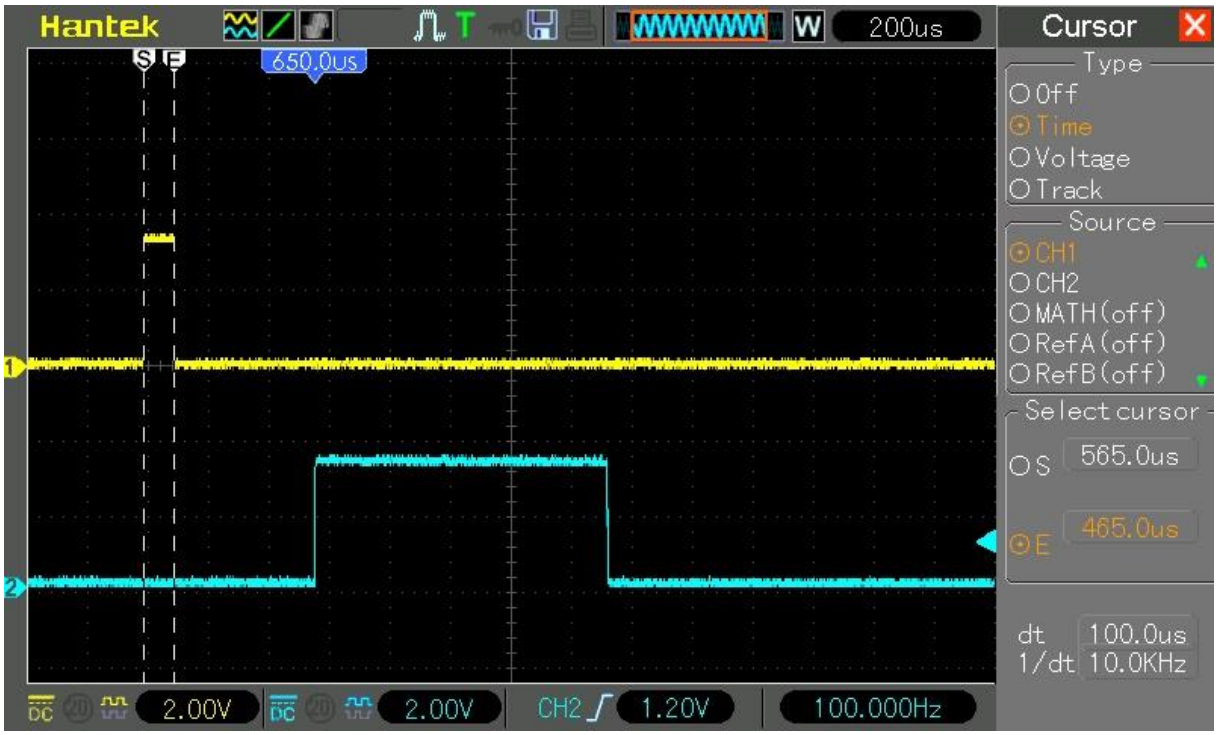

Fig 12. Trigger and Echo signals

c. **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 CAN network 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.
```

d. **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 or event-based mode. The differences between the two modes are described in the Notification Mode Adjustment Service in section 4.2.6

```
for(;;)
{
   status = xTaskNotifyWait(0, 0, &c_distance_zone, 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.

       else if(status == pdPASS && CAN_MsgContinious == OFF && c_distance_zone !=
       p_distance_zone){
           HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &CAN_Message1->TxHeader,
           CAN_Message1->Tx_Payload); //Sends the distance to the CAN network.

           p_distance_zone = c_distance_zone;
       }

}
```

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. All LED tasks share the same logic.

```
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;
         }
}
```

4.2.5 Interruptions

There are 2 interruption setups for the project: `HAL_GPIO_EXTI_Callback` and `HAL_FDCAN_RxFifo0Callback.`

These are callback functions that the IRQ handler provides when the ECHO signal from the ultrasonic sensor gets triggered or when a CAN message matching the expected ID is received. The IRQ calling functions can be found in stm32g4xx_it.c The callback interrupt functions are auto generated by the IDE when an input is set as an interrupt. See figure 13 below.

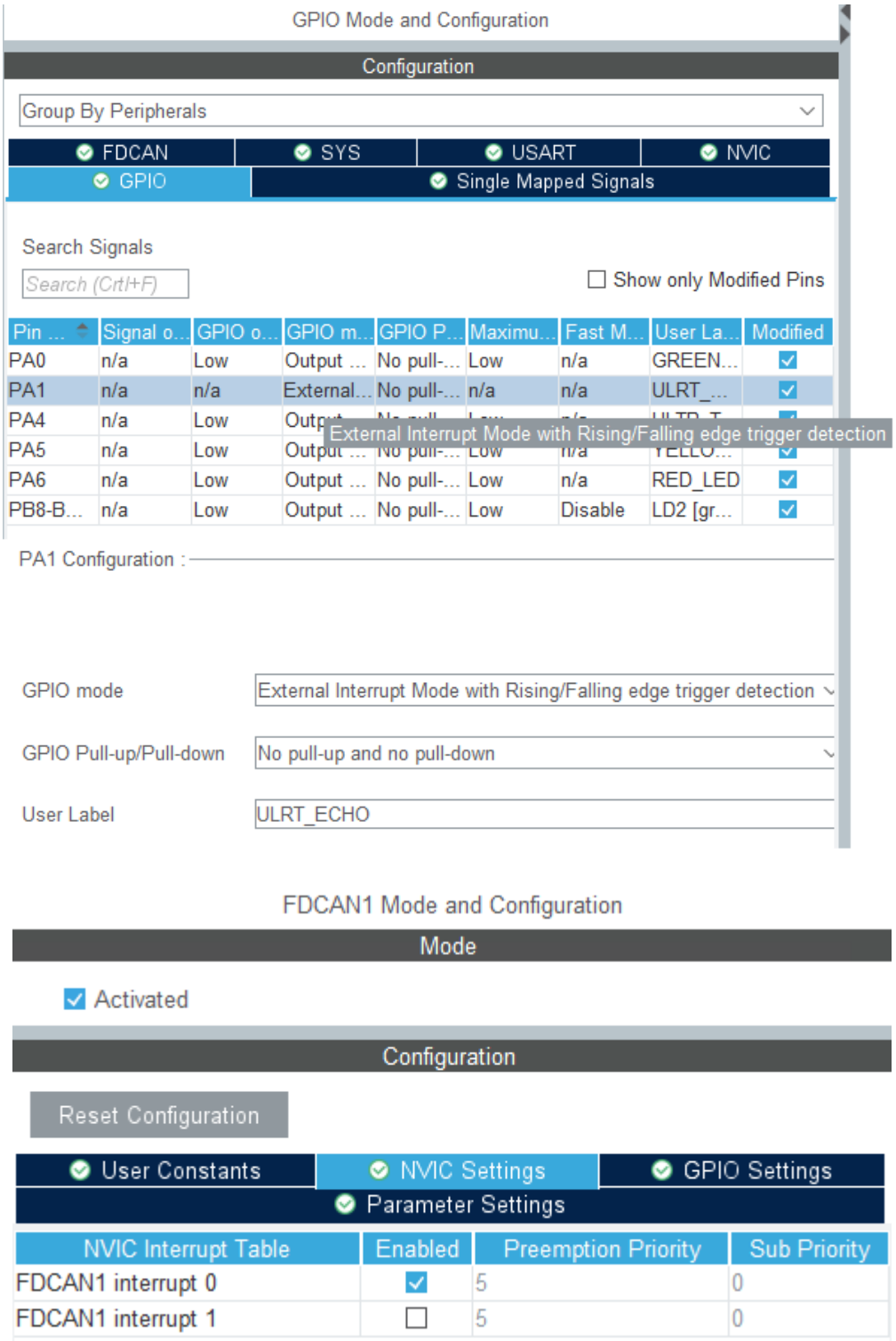

Fig 13. IO and Interrupt configuration

a. Ultrasonic Sensor ECHO signal interrupt

This interrupt handles the ECHO signal coming from the ultrasonic sensor. The HAL GPIO ReadPin function provides the state of the pin, that allows the interrupt to be classified as a rising or falling edge. Once a falling edge is detected, the value of TIMER2 (32-bit) is sent to the controller_handler task in FreeRTOS using *xTaskNotifyFromISR.*

```
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;
        }
   }
 }
```

b. CAN Rx Interrupt

This interrupt is triggered when the distance controller receives a 0x726 CAN message ID (calibration ID). Verifies that the message is a standard CAN message and notifies the CAN_Rx_Ctrlr task in FreeRTOS that a new message is available to be processed.

```
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);
     }
}
```

4.2.6 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 diagnostics UDS standard. The following tables provide the byte information for each configuration service.

a. Distance Threshold Adjustment Service

Byte #	Request (0x726 for ECU) → (CANoe or Node to ECU)	Data/ range (Hex)	← 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

b. Notification Mode Adjustment Service

Byte #	Request (0x726 for ECU) → (CANoe or Node to ECU)	Data/ range (Hex)	← 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

4.2.7 Special Implementations

a. Debounce Algorithm

During the testing phase if the distance measurement controller was set to CONTINIOUS MODE it was noticed that if an 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 a delay that consist of a countdown value that gets triggered every time the measured distance falls into the Danger or Warning categories. The counter goes from 100 (`#define RESTART 100`) to ZERO and during that window time, the reported distance is the one saved in the memory buffer. Once the 100 cycles pass the debounce flag is set to OFF and allows a new distance value to be processed.

This implementation can be found in the **`Controller_handler`** task inside app_freertos.c

b. 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:

```
/* Definitions for Mutex01 */
osMutexId_t Mutex01Handle;
const osMutexAttr_t Mutex01_attributes = {
.name = "Mutex01"
};
```

4.4 Implementation/Code

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

ECE-554_Embedded_Systems_Final_Project

Test Results

5.1 Distance Measurement

The following table shows some distance measurements.

Distance	Measurement	Zone
30 cm	30 cm	Safe
16 cm	16 cm	Warning
7 cm	7 cm	Danger

5.2 Configuration Services

The following table shows the CAN message response when a configuration adjustment was set.

Threshold Adjustment ($FE01) Positive Response:

19907.41...	CAN 1	726	CAN Frame	Tx	8	8	05 22 FE 01 09 14 00 00
19907.41...	CAN 1	72E	CAN Frame	Rx	8	8	05 62 FE 01 09 14 00 00

Distance Notification ($FE02) Continuous Mode Positive Response:

19608.52...	CAN 1	726	CAN Frame	Tx	8	8	05 22 FE 02 01 00 00 00
19608.52...	CAN 1	72E	CAN Frame	Rx	8	8	05 62 FE 02 01 00 00 00

Distance Notification ($FE02) Event-Based Positive Response:

19561.07...	CAN 1	726	CAN Frame	Tx	8	8	05 22 FE 02 02 00 00 00
19561.07...	CAN 1	72E	CAN Frame	Rx	8	8	05 62 FE 02 02 00 00 00

Negative Response (7F) for invalid service:

19433.76...	CAN 1	726	CAN Frame	Tx	8	8	05 22 FE 02 03 00 00 00
19433.76...	CAN 1	72E	CAN Frame	Rx	8	8	07 62 7F 00 00 00 00 00

Appendix A **Project Picture**