CSE522:REAL-TIME EMBEDDED SYSTEMS

Spring 2021

ASSIGNMENT 2 REPORT

Name: Sunjeet Jena
ASUID: 1218420294

The following report contains two parts. First part describes the implementation and device initialization details of the Assignment 2, in which we were asked to write a device driver for HCSR04 Ultrasonic sensor and an application to successfully run two HCSR04 sensors using the device driver. The second part, lists all the devices that were created in the “test_led” program (Assignment 1) and the order these devices were initialized.

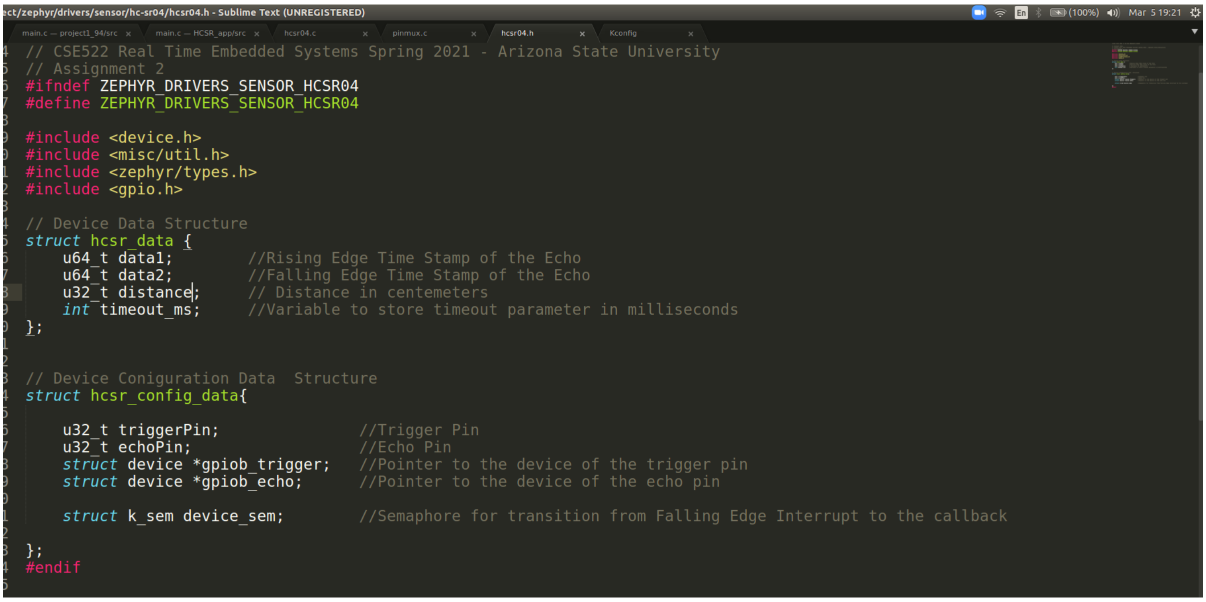
PART 1: Implementation and Device Initialization Details of the Assignment 2

In this assignment we were asked to to write a device driver for HCSR04 Ultrasonic sensor and an application to successfully run two HCSR04 sensors using that device driver. The devices work only in one shot mode and the output are the distances in centimeters, the timestamp of the measurement in microseconds and the device name.

PART 1A:

This part describes how to set up device data object structure and device configuration information structure for the HCSR04 devices and also the configuration parameters for respective devices. We also describe the Kconfig file where the configuration parameters are set.

The device data data object structure (s*tores device specific data*) and the device configuration structure (stores device specific configurations eg: trigger pin , echo pin, etc.). Both of these data structures have been defined in the **‘hcsr04.h’** header file *(Path: /zephyr/drivers/sensor/hc-sr04/hcsr04.h)* . These data structures will be used in the initialization function and API functions of the respective devices. These structures also allow user to read the current measurement taken by the device. Seet the below image for details of the structure.



```
// CSE522 Real Time Embedded Systems Spring 2021 - Arizona State University
// Assignment 2
#ifndef ZEPHYR_DRIVERS_SENSOR_HCSR04
#define ZEPHYR_DRIVERS_SENSOR_HCSR04

#include <device.h>
#include <misc/util.h>
#include <zephyr/types.h>
#include <gpio.h>

// Device Data Structure
struct hcsr_data {
    u64_t data1;        //Rising Edge Time Stamp of the Echo
    u64_t data2;        //Falling Edge Time Stamp of the Echo
    u32_t distance;     // Distance in centemeters
    int timeout_ms;     //Variable to store timeout parameter in milliseconds
};


// Device Coniguration Data  Structure
struct hcsr_config_data{

    u32_t triggerPin;               //Trigger Pin
    u32_t echoPin;                  //Echo Pin
    struct device *gpiob_trigger;   //Pointer to the device of the trigger pin
    struct device *gpiob_echo;      //Pointer to the device of the echo pin

    struct k_sem device_sem;        //Semaphore for transition from Falling Edge Interrupt to the callback

};
#endif
```

Figure 1.1: (Source: Screenshot of the hcsr04.h header file in /zephyr/drivers/sensor/hc-sr04/hcsr04.h)

Figure 1.1 is a screenshot of the header file **‘hcsr04.h’** , which defines two data structures: The first structure is “**hcsr_data**” and the second is “**hcsr_config_data**”.

“**hcsr_data**” structure stores 4 types of data:

data1: The time stamp when rising edge was detected by the echo pin of the device.
data2: The time stamp when falling edge was detected by the echo pin of the device.
distance: Object distance in centimeters as detected by the respective HCSR04 device.
timeout_ms: Timeout limit of the device to return an echo signal (Falling Edge).

“**hcsr_config_data**” structure stores 5 types of data:

triggerPin: GPIO trigger pin of the device.
echoPin: GPIO echo pin of the device.
gpiob_trigger: Pointer to the device structure of the gpio device respective to the trigger pin.
gpiob_echo: Pointer to the device structure of the gpio device respective to the echo pin.
device_sem: Semaphore variable to be used by **sample_channel_get()** function to wait till the timeout. This semaphore is given by the interrupt callback of the respective devices.

Both these structures are respective to the individual HCSR04 devices (HCSR0 and HCSR1) and is initialized in the “**hscr04.c**” file in *path: /zephyr/drivers/sensor/hc-sr04/hcsr04.c* .

Now, lets look into the “**Kconfig**” file in *path: /zephyr/drivers/sensor/hc-sr04/ Kconfig* .

```
menuconfig HCSR04
    bool "HCSR04 Distance Measurement Sensor"
    depends on I2C && HAS_DTS_I2C
    help
      Enable driver for HCSR04 distance measurement sensor.

if HCSR04

config HCSR04_DEV0_TRIGGER_PIN
    int
    depends on GPIO
    default 4
    help
        HCSR0 trigger pin.

config HCSR04_DEV1_TRIGGER_PIN
    int
    depends on GPIO
    default 2
    help
        HCSR1 trigger pin.


config HCSR04_DEV0_ECHO_PIN
    int
    depends on GPIO
    default 6
    help
        HCSR0 echo pin.

config HCSR04_DEV1_ECHO_PIN
    int
    depends on GPIO
    default 7
    help
        HCSR1 echo pin.

config HCSR04_0_NAME
    string "Driver Name"
    default "HCSR04_DW_0"
    help
      Name of the first HCSR04 device as exposed to the kernel

config HCSR04_1_NAME
    string "Driver Name"
    default "HCSR04_DW_1"
    help
      Name of the second HCSR04 device as exposed to the kernel
endif # HCSR04
```

Figure 1.2: (Source: Screenshot of the Kconfig file for HCSR04 device driver in /zephyr/drivers/sensor/hc-sr04/Kconfig)

This “Kconfig” file is used to provide configuration details specific to each device like for example **config HCSR04_DEV0_TRIGGER_PIN** is to setup the values trigger pin of the first HCSR04 device(HCSR0). Here we also define the names of both the HCSR04 devices, which is exposed to the system.
(HCSR0: HCSR04_DW_0, HCSR1:HCSR04_DW_1).

PART 1B:

This part majorly describes the device driver code for the HCSR4 device from the **"hcsr04.c"** file in the path: */zephyr/drivers/sensor/hc-sr04/hcsr04.c*

The "hcsr04.c" code has four major parts:

a) Device Initialization Functions: Used to setup the PINMUX pins, the trigger pin and the echo pin. This function also adds interrupt callback function to the echo pins.

b) Device API Functions: These functions are used for API interface with the respective devices. There are three API functions to each of the HCSR devices, **sensor_sample_fetch()**, **sensor_channel_get()** and **sensor_attr_set()**.

c) **DEVICE_AND_API_INIT**: Used to initialize the devices, link each device to their respective API functions, and expose the device the kernel.

d) Interrupt Callback Function: Used by the echo pins to test and read the callback data (Input) whenever a Rising edge or a falling edge is detected.

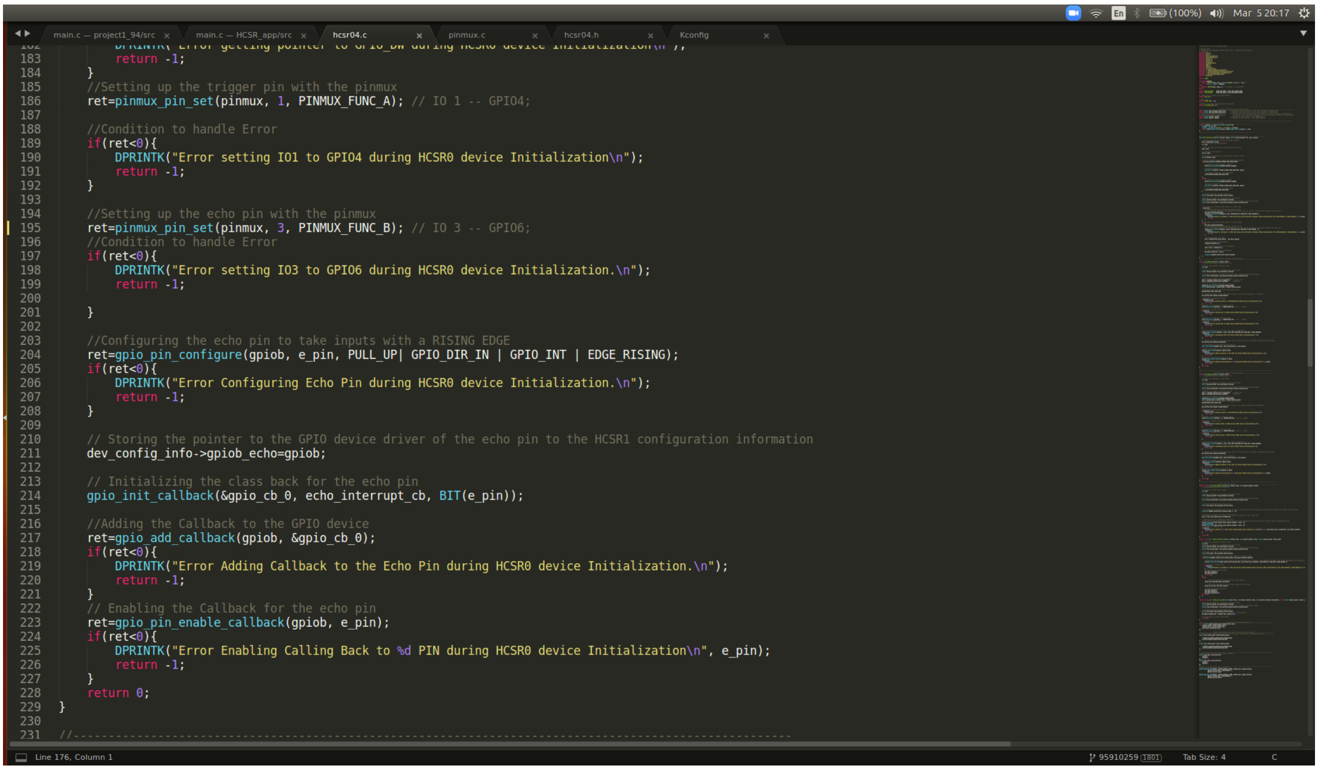
Device Initialization Functions:

Device initialization functions are used by each of the respective HCSR devices to setup the PINMUX, set the trigger GPIO pin to write output, configure the echo GPIO pin to take Rising Edge Input and add a callback function to the echo pin. The screenshots below is the initialization function of HCSR0 device .

```
    }
}
//------------------Initialization Function for HCSR0 device----------------------------------------

static int HCSR0_init(struct device *dev){
    // printk("HCSR0 device initialized. \n");

    //Setting the variables to read returns
    int ret;

    //Getting the address to the HCSR0 device config struct
    struct device_config* dev_config=dev->config;

    // Getiing the address to HCSR0 device config_data struct(hcsr_config_data type);
    struct hcsr_config_data* dev_config_info=dev_config->config_info;

    //Getting the trigger pin and the echo pin;
    u32_t t_pin=dev_config_info->triggerPin;    //Trigger Pin
    u32_t e_pin=dev_config_info->echoPin;       //Echo Pin

    //Getting the PINMUX related data and Binding
    pinmux=device_get_binding(CONFIG_PINMUX_NAME);
    struct galileo_data *pinmux_data = pinmux->driver_data;

    //retrieving gpio_dw driver struct from pinmux driver
    gpiob=pinmux_data->gpio_dw;

    // Storing the pointer to the GPIO device driver to the HCSR0 configuration information
    dev_config_info->gpiob_trigger=gpiob;

    //Condition to handle NULL Pointer
    if(gpiob==NULL){
        DPRINTK("Error getting pointer to GPIO_DW during HCSR0 device Initialization\n");
        return -1;
    }
    //Setting up the trigger pin with the pinmux
    ret=pinmux_pin_set(pinmux, 1, PINMUX_FUNC_A); // IO 1 -- GPIO4;

    //Condition to handle Error
    if(ret<0){
        DPRINTK("Error setting IO1 to GPIO4 during HCSR0 device Initialization\n");
        return -1;
    }

    //Setting up the echo pin with the pinmux
    ret=pinmux_pin_set(pinmux, 3, PINMUX_FUNC_B); // IO 3 -- GPIO6;
    //Condition to handle Error
    if(ret<0){
        DPRINTK("Error setting IO3 to GPIO6 during HCSR0 device Initialization.\n");
```

Figure 1.3: (Source: Screenshot of the hcsr04.c file for HCSR04 device driver in /zephyr/drivers/sensor/hc-sr04/hcsr04.c)

To setup the trigger pin to generate output/trigger we first, get a binding to the PINMUX device using the **device_get_binding()** and use **pinmux_pin_set()** to set the trigger pin. Please note that by default it has been set to set IO1 to GPIO 4. In case the trigger pin is changed, respective changes has be made to set pinmux correctly, based on how GPIO pins are connected to the IO pins.



```c
        return -1;
    }
    //Setting up the trigger pin with the pinmux
    ret=pinmux_pin_set(pinmux, 1, PINMUX_FUNC_A); // IO 1 -- GPIO4;

    //Condition to handle Error
    if(ret<0){
        DPRINTK("Error setting IO1 to GPIO4 during HCSR0 device Initialization\n");
        return -1;
    }

    //Setting up the echo pin with the pinmux
    ret=pinmux_pin_set(pinmux, 3, PINMUX_FUNC_B); // IO 3 -- GPIO6;
    //Condition to handle Error
    if(ret<0){
        DPRINTK("Error setting IO3 to GPIO6 during HCSR0 device Initialization.\n");
        return -1;

    }

    //Configuring the echo pin to take inputs with a RISING EDGE
    ret=gpio_pin_configure(gpiob, e_pin, PULL_UP| GPIO_DIR_IN | GPIO_INT | EDGE_RISING);
    if(ret<0){
        DPRINTK("Error Configuring Echo Pin during HCSR0 device Initialization.\n");
        return -1;
    }

    // Storing the pointer to the GPIO device driver of the echo pin to the HCSR1 configuration information
    dev_config_info->gpiob_echo=gpiob;

    // Initializing the class back for the echo pin
    gpio_init_callback(&gpio_cb_0, echo_interrupt_cb, BIT(e_pin));

    //Adding the Callback to the GPIO device
    ret=gpio_add_callback(gpiob, &gpio_cb_0);
    if(ret<0){
        DPRINTK("Error Adding Callback to the Echo Pin during HCSR0 device Initialization.\n");
        return -1;
    }
    // Enabling the Callback for the echo pin
    ret=gpio_pin_enable_callback(gpiob, e_pin);
    if(ret<0){
        DPRINTK("Error Enabling Calling Back to %d PIN during HCSR0 device Initialization\n", e_pin);
        return -1;
    }
    return 0;
}

//-----------------------------------------------------------------------------------
```

Figure 1.4: (Source: Screenshot of the hcsr04.c file for HCSR04 device driver in /zephyr/drivers/sensor/hc-sr04/hcsr04.c)

The above screenshot shows, how the echo pin was setup to take rising edge input signal and a callback was added to the echo pin of the HCSR0 device using the **gpio_init_callback()** function and **gpio_add_callback()** function. Finally we also enable the echo pin to take call backs using the **gpio_pin_enable_callback()** function.

Device API Functions:

Device specific API functions have been defined in the "**hcsr04.c**" file to enable the user/application to fetch a sample for the indvidual devices and access the data (distance in centemeters) from the driver data. An API function has also been implemented to set the timeout parameter of each of the devices. This timeout parameter is used by the device to wait, till a measurement is completed and in case a measurement was not completeted within the timeout limit, sample buffered is cleared and -1 is returned.

Specifically there are three API functions defined:

- hcsr04_sample_fetch() ---> corresponds to sensor_sample_fetch()

  The critical role of this function is to write a trigger signal in the GPIO trigger pin of the respective devices. Trigger pulse generated by first writing a '1' using the **gpio_pin_write()** function into the GPIO trigger pin, then we write '0' using the same **gpio_pin_write()** function.

- hcsr04_channel_get() --> corresponds to sensor_sample_fetch()

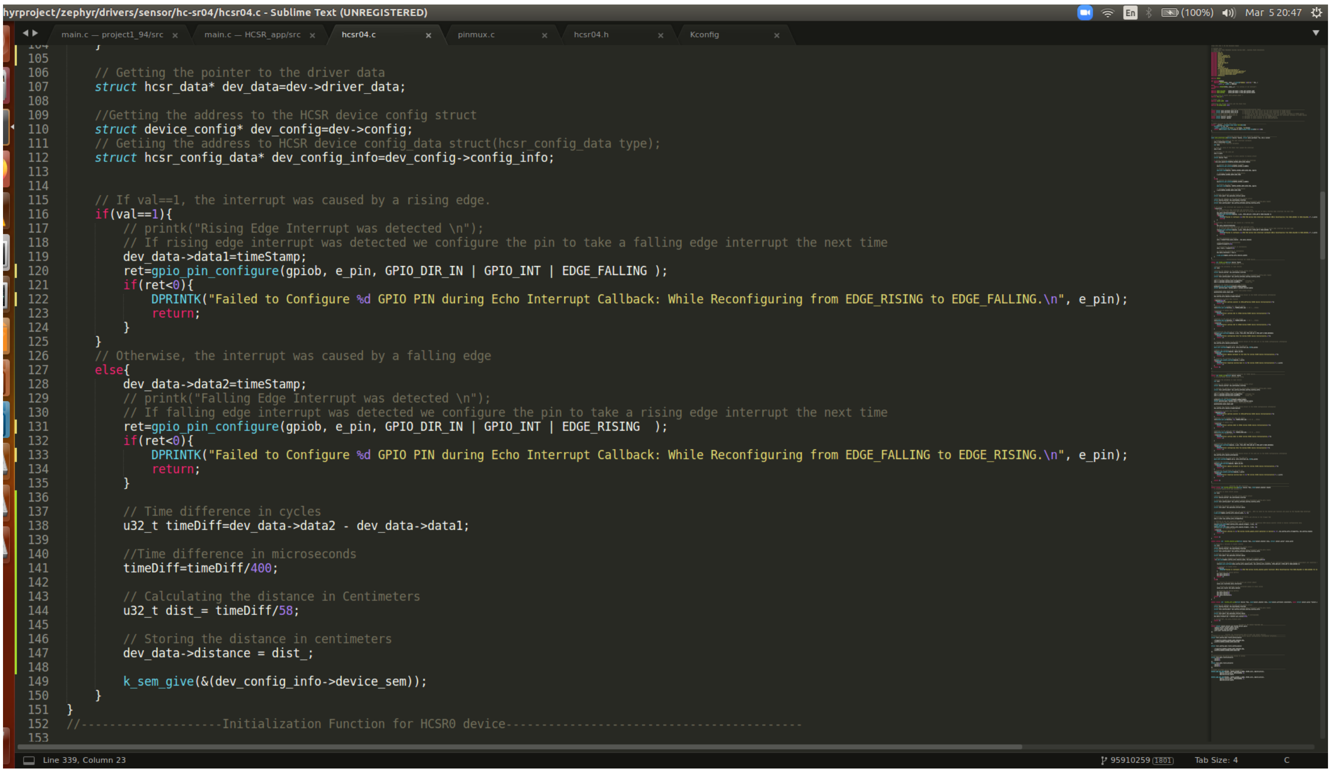
  This function is used to fetch the time stamp of the measurement, as well the distance stored in the device buffer. At the end of this call, the device buffer is cleared for the next sample. This function is blocked until (using the semaphore defined in the **hcsr_config_data** structure**)** the measurement is complete or timeout duration has expired. If timeout duration has been expired, we return '-1' and clear the buffer.

- hcsr04_attr_set() --> corresponds to sensor_attr_set()

  This function is been used to set the “timeout_ms” parameters (defined in **hcsr_data**) of the respective devices. The user/application provides the timeout duration in microseconds.

Interrupt Callback Function:

This function is used by the respective echo pin as a callback function, whenever a RISING EDGE or a FALLING edge is detected. See the below screen shot for implementation.



```c
    }

    // Getting the pointer to the driver data
    struct hcsr_data* dev_data=dev->driver_data;

    //Getting the address to the HCSR device config struct
    struct device_config* dev_config=dev->config;
    // Getiing the address to HCSR device config_data struct(hcsr_config_data type);
    struct hcsr_config_data* dev_config_info=dev_config->config_info;


    // If val==1, the interrupt was caused by a rising edge.
    if(val==1){
        // printk("Rising Edge Interrupt was detected \n");
        // If rising edge interrupt was detected we configure the pin to take a falling edge interrupt the next time
        dev_data->data1=timeStamp;
        ret=gpio_pin_configure(gpiob, e_pin, GPIO_DIR_IN | GPIO_INT | EDGE_FALLING );
        if(ret<0){
            DPRINTK("Failed to Configure %d GPIO PIN during Echo Interrupt Callback: While Reconfiguring from EDGE_RISING to EDGE_FALLING.\n", e_pin);
            return;
        }
    }
    // Otherwise, the interrupt was caused by a falling edge
    else{
        dev_data->data2=timeStamp;
        // printk("Falling Edge Interrupt was detected \n");
        // If falling edge interrupt was detected we configure the pin to take a rising edge interrupt the next time
        ret=gpio_pin_configure(gpiob, e_pin, GPIO_DIR_IN | GPIO_INT | EDGE_RISING  );
        if(ret<0){
            DPRINTK("Failed to Configure %d GPIO PIN during Echo Interrupt Callback: While Reconfiguring from EDGE_FALLING to EDGE_RISING.\n", e_pin);
            return;
        }

        // Time difference in cycles
        u32_t timeDiff=dev_data->data2 - dev_data->data1;

        //Time difference in microseconds
        timeDiff=timeDiff/400;

        // Calculating the distance in Centimeters
        u32_t dist_= timeDiff/58;

        // Storing the distance in centimeters
        dev_data->distance = dist_;

        k_sem_give(&(dev_config_info->device_sem));
    }
}
//-------------------Initialization Function for HCSR0 device----------------------------------------
```

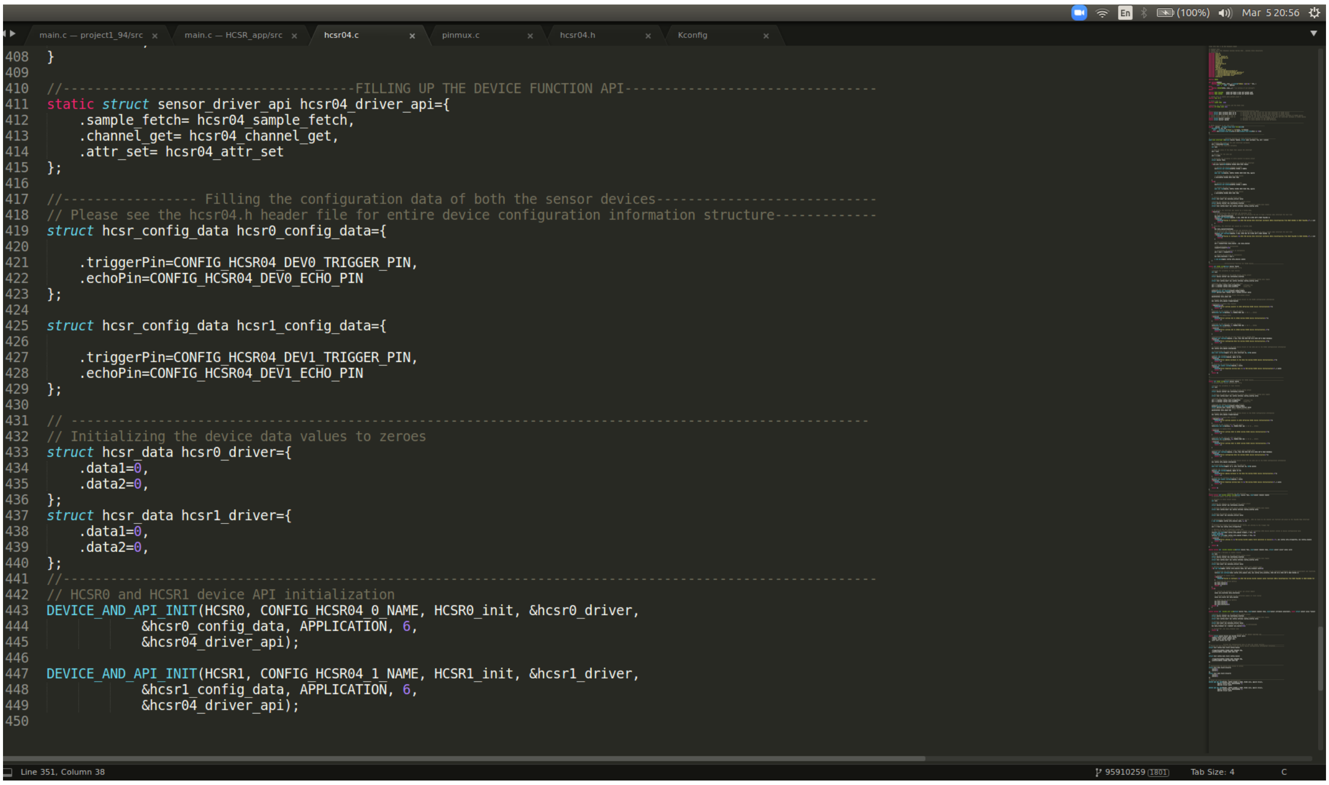
Figure 1.5: (Source: Partial Screenshot of the echo_interrupt_cb() function defined in hcsr04.c file for HCSR04 device driver in /zephyr/drivers/sensor/hc-sr04/hcsr04.c)

As seen from the image above, in case a RISING EDGE is detected by the echo pin, we configure the echo pin to detect a FALLING EDGE next timeand in case a FALLING EDGE interrupt is detected by the echo, we configure it to detect a RISING EDGE next time.

In the callback function, we also calculate the distance in centimeters and store the values in device data buffer variable “distance” defined in the “hcsr_data” structure. After the completion of an Interrupt when a FALLING EDGE was detected, the semaphore is given to the waiting hcsr04_channel_get() function.

DEVICE_AND_API_INIT:

This macro is been used to the initialize the device(s), expose the device to the kernel, expose the device data structure(hcsr_data) to kernel expose the device configuration to kernal and finally to link the device to their respective API functions. See the image in the next page for the implementation details.



```c
408 }
409
410 //-----------------------------------FILLING UP THE DEVICE FUNCTION API-------------------------------
411 static struct sensor_driver_api hcsr04_driver_api={
412     .sample_fetch= hcsr04_sample_fetch,
413     .channel_get= hcsr04_channel_get,
414     .attr_set= hcsr04_attr_set
415 };
416
417 //---------------- Filling the configuration data of both the sensor devices---------------------------
418 // Please see the hcsr04.h header file for entire device configuration information structure------------
419 struct hcsr_config_data hcsr0_config_data={
420
421     .triggerPin=CONFIG_HCSR04_DEV0_TRIGGER_PIN,
422     .echoPin=CONFIG_HCSR04_DEV0_ECHO_PIN
423 };
424
425 struct hcsr_config_data hcsr1_config_data={
426
427     .triggerPin=CONFIG_HCSR04_DEV1_TRIGGER_PIN,
428     .echoPin=CONFIG_HCSR04_DEV1_ECHO_PIN
429 };
430
431 // ----------------------------------------------------------------------------------------------------
432 // Initializing the device data values to zeroes
433 struct hcsr_data hcsr0_driver={
434     .data1=0,
435     .data2=0,
436 };
437 struct hcsr_data hcsr1_driver={
438     .data1=0,
439     .data2=0,
440 };
441 //-----------------------------------------------------------------------------------------------------
442 // HCSR0 and HCSR1 device API initialization
443 DEVICE_AND_API_INIT(HCSR0, CONFIG_HCSR04_0_NAME, HCSR0_init, &hcsr0_driver,
444             &hcsr0_config_data, APPLICATION, 6,
445             &hcsr04_driver_api);
446
447 DEVICE_AND_API_INIT(HCSR1, CONFIG_HCSR04_1_NAME, HCSR1_init, &hcsr1_driver,
448             &hcsr1_config_data, APPLICATION, 6,
449             &hcsr04_driver_api);
450
```

Figure 1.5: (Source: Screenshot of hcsr04.c file for HCSR04 device driver in /zephyr/drivers/sensor/hc-sr04/hcsr04.c) showing the implementation of DEVICE_AND_API_INIT and driver API function.

In figure 1.5 we can see that both the devices are initialized and exposed to the kernel using the **DEVICE_AND_API_INIT()** macro. *(line 443 and 447).*

Please also note that a data structure named "**sensor_driver_api**" has been created to link the user defined API function to the structure defined by the ZephyrOS. This structure/variable is used by the kernel to implement the sensor specific APIs.

We also setup of the configuration parameters of the individual devices using the parameters obtained from the **"Kconfig"** file and passing them the to respective "**hcsr_config_data**" structure of the devices. Please see the image above (Figure 1.5) for the details of the implementation details (*line 433 and line 437*).