4DS4 Project 2 Report:

Autonomous Vehicle: Putting it All Together

Johnnathan Gershkovich - 400408809
Joshua Umansky - 400234265
Shanzeb Immad - 400382928

Name	Contribution
Johnnathan Gershkovich	Co-wrote the C++ code to run on the FMU Assisted in python programming for camera & ultrasonic sensor
Joshua Umansky	Co-wrote the C++ code to run on the FMU Assisted in python programming for camera & ultrasonic sensor
Shanzeb Immad	Co-wrote the C++ code to run on the FMU Assisted in python programming for camera & ultrasonic sensor

Experiment 2: Setup B

After following the provided documentation for running a NuttX application, the following was observed after running our hello_world program.

```
nsh> hello_world
INFO  [hello_world] Hello World 0
INFO  [hello_world] Hello World 1
INFO  [hello_world] Hello World 2
INFO  [hello_world] Hello World 3
INFO  [hello_world] Hello World 4
nsh>
```

Terminal printout of Experiment 2 Setup B

Experiment 3: Setup A

Continuing following the documentation, we were able to listen to a few different topics, two are pictured here. From this experiment, we were able to determine the topic rc_channels was the correct topic for interpreting the values from the controller, which would be used in the next parts of the project.

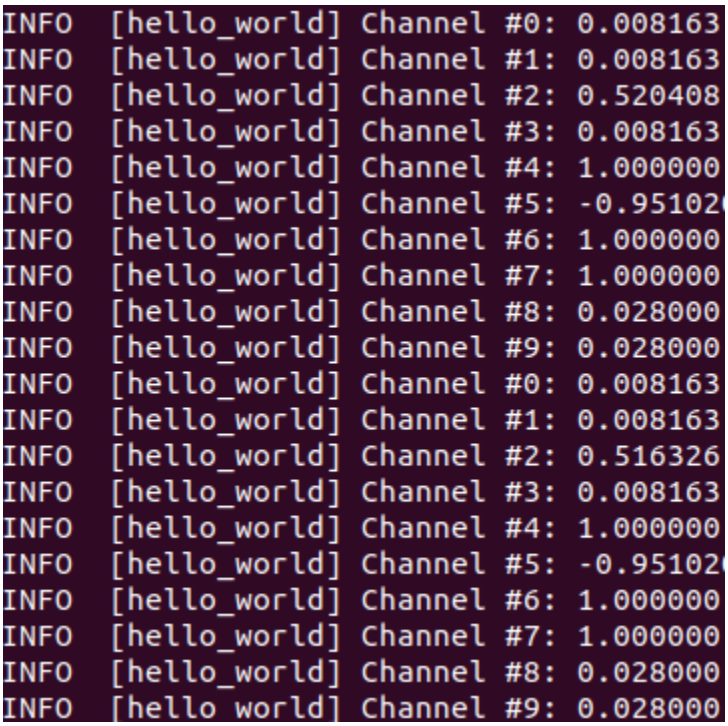
```
TOPIC: sensor_accel
 sensor_accel
    timestamp: 240199915 (0.000343 seconds ago)
    timestamp_sample: 240199839 (76 us before timestamp)
    device_id: 5373970 (Type: 0x52, SPI:2 (0x00))
    x: 0.0383
    y: 0.4979
    z: -10.0652
    temperature: 34.5600
    error_count: 0
    clip_counter: [0, 0, 0]
    samples: 1
```

```
TOPIC: test_motor
 test_motor
    timestamp: 263017826 (0.007770 seconds ago)
    motor_number: 1
    value: -3129926699353900000.0000
    timeout_ms: 0
    action: 1
    driver_instance: 0
```

Examples of topics as a result of listener

Project 2: Step 0

Using the topic we found (RC_Channels), we were able to print the value of each channel inside of the structure every 200ms. This was used to determine which channel corresponded with the appropriate controller stick, and the range that they occupied, which was then used in both steps 1 and 2.



Terminal printout for Project 2 Step 0

Project 2: Step 1

In order to control the DC and Servo motors, another definition was added corresponding to the Angle_Motor, which is defined as motor_number 1. From there, we subscribe to the rc_channels, which allow for continuous updating of the motor speed and angle values. The following code snippet shows the setup information in the hello_world_main function, before the infinite while loop begins;

```
#define DC_MOTOR 0
#define ANGLE_MOTOR 1


extern "C" __EXPORT int hello_world_main(int argc, char *argv[]);

 int hello_world_main(int argc, char *argv[])
 {
    test_motor_s test_motor;
    rc_channels_s rc_channels;
    double motor_value = 0; // a number between 0 to 1

     uORB::Publication<test_motor_s> test_motor_pub(ORB_ID(test_motor));

     int rc_combined_handle = orb_subscribe(ORB_ID(rc_channels));
     orb_set_interval(rc_combined_handle, 200);
```

Initial setup for Project 2 Part 1

The while loop performs the constant updates to the motors, by reading in the values from rc_channels (2 for motor speed, 0 for turning direction), normalized between 0 and 1. That information is then passed to the test_motor (A structure defined by test_motor.h) where various individual parameters are set, motor.value being the primary one being changed, as it controls how fast the motor spins. Once the values are set for one particular motor (drive motor being the first motor set), the motor is published, updating the information, then repeated for the angle motor.

```
while(1)
{
    orb_copy(ORB_ID(rc_channels), rc_combined_handle, &rc_channels);
    if(motor_value > 1.0 || motor_value < 0)
        break;

    float motor_speed = (rc_channels.channels[2] + 1.0f)/2;
    float turn = (rc_channels.channels[0] + 1.0f)/2;

    PX4_INFO("Motor speed is %f", (double)(motor_speed));
    test_motor.timestamp = hrt_absolute_time();
    test_motor.motor_number = DC_MOTOR;
    test_motor.value = motor_speed;
    test_motor.action = test_motor_s::ACTION_RUN;
    test_motor.driver_instance = 0;
    test_motor.timeout_ms = 0;

    test_motor_pub.publish(test_motor);


    // Turn
    test_motor.timestamp = hrt_absolute_time();
    test_motor.motor_number = ANGLE_MOTOR;
    test_motor.value = turn;
    test_motor.driver_instance = 0;
    test_motor.timeout_ms = 0;

    test_motor_pub.publish(test_motor);
}
```

Infinite while loop for Project 2 Part 1

Project 2: Step 2

As we began implementing the second part of the project, we chose to improve our implementation of step one, which involved using the rc channels to control the car when the car was operating in a safe environment. We defined a safe environment to be when the ultrasonic sensor did not find an object within 50 cm of the front of the car. This allowed for a more realistic system, as the user has full control of the car outside a “dangerous” situation.

In order to implement this, we used the MAVLINK to listen for a value being pushed from our python program. When the value is received, the distance and turn variables are filled with real values. These values are used in the if statement to determine if the automated slowing/stopping and turning should be triggered. When our autopilot takes over, there is two situations and both behave differently;

- Situation 1: Distance to object is less than 50cm, but more than 15 cm
    - In this situation, the motor_speed is divided by five, but we allow the user to still control the speed, and the turning direction is manually configured by the information returned from the camera.
- Situation 2: Distance to object is less than 15 cm
    - In this situation, the motor_speed is set to 0.5 (The stopping value) and the turn value is set, to ensure any forward momentum is continued in a safe direction.

```
//float motor_speed = (rc_channels.channels[2] + 1.0f)/2;
float turn = (-rc_channels.channels[0] + 1.0f)/2;
float distance = debug_data.value;
int direction = debug_data.ind-1; // -1=left, 0=straight, 1=right

//turn = (float)((-direction)+1.0f)/2;

if(distance < 50 && distance > 15){
        motor_speed = (rc_channels.channels[2]/5 + 1.0f)/2;
        turn = (float)((-direction)+1.0f)/2;
} else if(distance < 15){
        motor_speed = 0.5f;
        turn = (float)((-direction)+1.0f)/2;
} else{
        motor_speed = (rc_channels.channels[2] + 1.0f)/2;
}
```

Code snippet of body running on the FMU

In order to receive feedback about the proximity of nearby objects and the suggested routing, a raspberry pi in conjunction with an ultrasonic sensor connected through GPIO, and a Pi camera were used. On the raspberry pi, a python file determines the distance of an encountered obstacle, by enabling output on the sensor for a brief period, awaiting the returned input of the

sound waves at the receiving end of the sensor, the distance can be determined, by halving the sonic travel time, and multiplying it by the speed of sound.

```
def distance():
    GPIO.output(GPIO_TRIGGER, True)

    time.sleep(0.00001)
    GPIO.output(GPIO_TRIGGER, False)

    StartTime = time.time()
    StopTime = time.time()

    while GPIO.input(GPIO_ECHO) == 0:
        StartTime = time.time()

    while GPIO.input(GPIO_ECHO) == 1:
        StopTime = time.time()

    TimeElapsed = StopTime - StartTime

    distance = (TimeElapsed * 34300) / 2

    return distance
```

Python function distance()

The raspberry pi determines the best direction using video input from the camera. The process as given in the lab document essentially entails:

1) Reading in an image with open cv
2) Filtering the image to remove noise and reduce error when processing
3) Applying edge detection with the Canny algorithm
4) Dividing the image into three vertical chunks
5) Finding the mean point of every chunk
6) Calculating the distance between the down mid point of the image frame and each chunk mid point
7) And lastly the chunk with the shortest distance is the preferred direction

After both of these inputs, the python code has a float value of distance in millimeters to send to the FMU, as well as an integer state, zero, one or two, for “go left”, “go straight”, “go right”. These values are added to a MAVLink message to send on the debug message topic, where the “value” variable in the debug struct will be the distance, and the “ind” variable of the struct will be the direction. Some basic error management was added to prevent a non-integer value from being sent if the camera algorithm returns non integers.

```
message = mavutil.mavlink.MAVLink_debug_message(0, int(ret) if isinstance(ret, int) else 1, dist)
the_connection.mav.send(message)
```

MavUtil debug_message packing