Lab 3/Project2

Autonomous Vehicle: Putting it All Together

---

1 Lab Rules

- You have to stick to your Lab slot assigned to you on Mosaic.
- Prepare a demonstration for all the Lab experiments, and get ready to be asked in any part of the experiments.
- The demonstrations of this lab will be held starting from **April 8 of each lab slot**.
- All the activities and questions written in **blue** should be documented and answered in your lab report.
- Each team needs to submit one report for all the members, and the first page of the report should contain the team number and the names of its members.
- Each team also needs to submit source code of problems (not the whole project) along with the report.
- The report and code should be submitted to github classroom repository before 12pm on your lab demo day. Put the report in a PDF format, name it with your group number.
- The first page (After the title page) of the report must include a **Declaration of Contributions**, where each team member writes his own individual tasks conducted towards the completion of the lab.

General Notes:

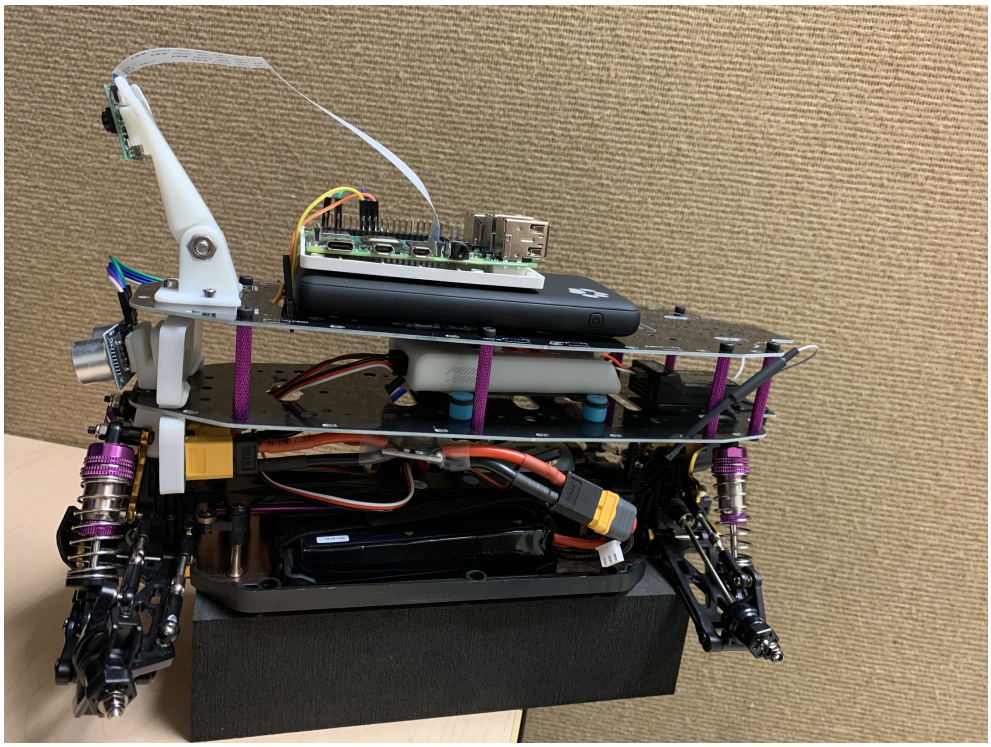
- The lab uses a virtual machine that has an image of Ubuntu operating system. To log into Ubuntu, you don't need a password, but in some cases it asks for it. The password is 123456. You can take a copy of the VM to your laptop if you want to operate using it.
- **Make sure you protect your work after every-time you visit the lab**. So, make sure to transfer your work from the virtual machine to some other place (User drive, flash storage, teams, google drive, etc) and delete your work from the machine before leaving the lab room.
- Similar to the previous point, for Raspberry Pi, make sure you have a copy of your work and delete them from the device.
- DISCONNECT the car's battery and the power bank, and attach them to their chargers. That's urgent for our safety. Turn off the RC controller before leaving the room.

2 Lab Goals

- Putting it All Together: The ultimate target of lab3/project2 is to finish building the autonomous vehicle that is able to use Camera and Ultrasonic sensors to avoid obstacles on its way. Towards this goal, we have the following steps.
- Build PX4 framework and run it on FMUK66.
- Create applications to run on NuttX operating system, using PX4 platform.
- Learn to subscribe and publish uORB messages.
- Use an on-vehicle companion computer for advanced sensor processing. We will be using Raspberry Pi4 as a companion computer in this lab.
- Utilize MAVLINK standard to establish a connection between the FMU and the companion computer.

3 Lab Components

1. 1x Assembled car that contains the following.
    - 1x Electronic Speed Controller (ESC.)
    - 1x Servo motor.
    - 1x DC motor.
    - 1x Lithium polymer battery.
    - 1x Power distribution board.
    - 1x Power module.
    - 1x FMU board.
    - 1x RC receiver module.
    - 1x Ultrasonic sensor.
    - 1x Potentiometer resistor
    - 1x PI camera
    - 1x Raspberry PI
    - 1x Segger J-Link EDU Mini debugger.
    - 1x Debug breakout board with the 7-wire cable.



2. 1x FlySky controller (radio transmitter).

3. 2x micro USB cables.
4. 1x USB C cable.
5. HDMI cable.

4 Experiments

4.1 Experiment 1: Build and Program PX4

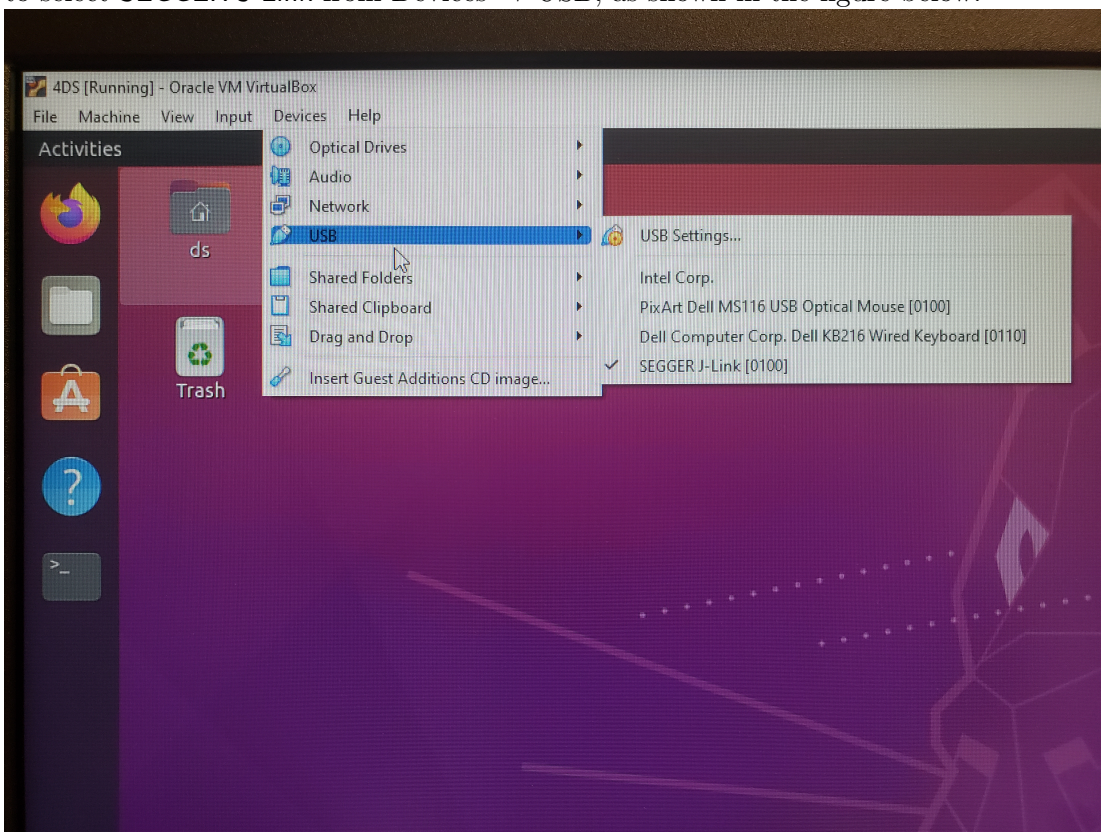
In this experiment, we will build PX4 for FMUK66, and then we will program and run a simple program using PX4 on the board. For more information about PX4, check its documentation from here.

Experiment Setup:

Virtual Machine

1. Navigate to C:\4DS4_VM. Double-click 4DS with the blue-cube icon to start the 4DS virtual machine.

2. Plug in the debugger to the computer. To connect it to the virtual machine, make sure to select SEGGER J-Link from Devices → USB, as shown in the figure below.



Build PX4

1. Open the Ubuntu terminal, and navigate to the PX4-Autopilot folder by typing cd Documents/PX4-Autopilot.

2. You will find a new directory created and named PX4-Autopilot.

3. To build PX4, type `make nxp_fmuk66-v3`. (it may require install kconfiglib library, So, first run *pip3 install kconfiglib*)

4. Make sure that PX4 is built successfully without error, and you get a message similar to the following figure.

```
Memory region         Used Size  Region Size  %age Used
       vectflash:         464 B         1 KB     45.31%
      cfmprotect:          16 B         16 B    100.00%
       progflash:     1879088 B    2071536 B     90.71%
        datasram:       40148 B       256 KB     15.32%
[1090/1090] Creating /home/ds/Documents/Group_0/PX4-Aut...t/build/nxp_fmuk66-v3_default/nxp_fmuk66-v3_default.px4
ds@ds-VirtualBox:~/Documents/Group_0/PX4-Autopilot$
```

Program PX4 to the FMU

1. First we have to program PX4 bootloader before PX4 itself. Bootloader is a piece of software that initializes the FMU before running PX4, and it enables us to program PX4 to FMU through the FMU's USB.

2. In the terminal, run `JLinkExe` (Make sure the debugger is connected with the virtual machine. If not, restart the virtual machine and run the JLinkExe command). JLinkExe will let us to connect to Jlink module to program FMU.

3. Write `connect` to start the connection.

4. It will ask you about the MCU name. First, check what is written besides <Default>, if `MK66FN2M0XXX18` is written, you can just press Enter. Otherwise, write `MK66FN2M0XXX18` and hit Enter.

5. Afterwards, it will ask you to choose between JTAG and SWD protocols. Write `s` to choose SWD and hit Enter.

6. For the speed, if the default is 4000 kHz, press Enter. Otherwise, write `4000000` and hit Enter.

```
Type "connect" to establish a target connection, '?' for help
J-Link>connect
Please specify device / core. <Default>: MK66FN2M0XXX18
Type '?' for selection dialog
Device>
Please specify target interface:
  J) JTAG (Default)
  S) SWD
  T) cJTAG
TIF>s
Specify target interface speed [kHz]. <Default>: 4000 kHz
Speed>
```

7. If JLink connects successfully, it will show a message similar to the message in the following figure.

```
Device "MK66FN2M0XXX18" selected.


Connecting to target via SWD
InitTarget()
Found SW-DP with ID 0x2BA01477
DPIDR: 0x2BA01477
Scanning AP map to find all available APs
AP[2]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x24770011)
AP[1]: JTAG-AP (IDR: 0x001C0000)
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FF000
CPUID register: 0x410FC241. Implementer code: 0x41 (ARM)
Found Cortex-M4 r0p1, Little endian.
FPUnit: 6 code (BP) slots and 2 literal slots
CoreSight components:
ROMTbl[0] @ E00FF000
[0][0]: E000E000 CID B105E00D PID 000BB00C SCS-M7
[0][1]: E0001000 CID B105E00D PID 003BB002 DWT
[0][2]: E0002000 CID B105E00D PID 002BB003 FPB
[0][3]: E0000000 CID B105E00D PID 003BB001 ITM
[0][4]: E0040000 CID B105900D PID 000BB9A1 TPIU
[0][5]: E0041000 CID B105900D PID 000BB925 ETM
[0][6]: E0042000 CID B105900D PID 003BB907 ETB
[0][7]: E0043000 CID B105900D PID 001BB908 CSTF
Cortex-M4 identified.
J-Link>
```

8. To program the bootloader, run loadbin "/home/ds/Documents/PX4-Autopilot/boards/nxp/fmuk66-v3/extras/nxp_fmuk66-v3_bootloader.bin" 0x0. For the bootloader, the address should be programmed at address 0x0.

```
J-Link>loadbin "/home/ds/Documents/Group_0/PX4-Autopilot/boards/nxp/fmuk66-v3/extras/nxp_fmuk66-v3_bootloader.bin" 0x0
Downloading file [/home/ds/Documents/Group_0/PX4-Autopilot/boards/nxp/fmuk66-v3/extras/nxp_fmuk66-v3_bootloader.bin]...
J-Link: Flash download: Bank 0 @ 0x00000000: 1 range affected (20480 bytes)
J-Link: Flash download: Total: 0.487s (Prepare: 0.098s, Compare: 0.018s, Erase: 0.016s, Program & Verify: 0.315s, Restore:
0.038s)
J-Link: Flash download: Program & Verify speed: 63 KB/s
O.K.
J-Link>
```

   Note: The path must not contain any spaces, and do not use '∼' in your path.

9. Now, we have the choice to use JLink or FMU's USB to program PX4.

10. For JLink, type the command loadbin "/home/ds/Documents/PX4-Autopilot/build/nxp_fmuk66-v3_default/nxp_fmuk66-v3_default.bin" 0x6000. The binary of PX4 will be stored under the build folder after the successful build of PX4, but the address will be 0x6000.

```
J-Link>loadbin "/home/ds/Documents/Group_0/PX4-Autopilot/build/nxp_fmuk66-v3_default/nxp_fmuk66-v3_default.bin" 0x6000
Downloading file [/home/ds/Documents/Group_0/PX4-Autopilot/build/nxp_fmuk66-v3_default/nxp_fmuk66-v3_default.bin]...
J-Link: Flash download: Bank 0 @ 0x00000000: 1 range affected (1884160 bytes)
J-Link: Flash download: Total: 25.608s (Prepare: 0.141s, Compare: 0.393s, Erase: 0.000s, Program & Verify: 25.003s, Restore
: 0.069s)
J-Link: Flash download: Program & Verify speed: 73 KB/s
O.K.
```

11. Reset the FMU by pressing its side push button, or by unplugging and replugging plug the USB.

12. To confirm that PX4 is running properly, check the FMU LED. It should be blinking in blue or red.

13. Now, we will reprogram PX4 using FMU's USB only. Remember, you can use USB for programming FMU only after programming the bootloader using JLink.

14. Open a terminal window, and navigate to PX4-Autopilot path.

15. Run make nxp_fmuk66-v3 upload. And when the terminal waits for the bootloader, reset the FMU by pressing the re-start button on the board or plug off/on the USB, . This command will build and program PX4 to FMUk66.

```
Found board id: 28,0 bootloader version: 5 on /dev/serial/by-id/usb-NXP_SEMICONDUCTORS_PX4_BL_FMUK66_v3.x_0-if00
sn: 0030ffffffffffff4e456018
chip: 660083ca
family: b'MK66FN2M0VMD18'
revision: b'8'
flash: 2072576 bytes
Windowed mode: False

Erase  : [====================] 100.0%
Program: [====================] 100.0%
Verify : [====================] 100.0%
Rebooting. Elapsed Time 80.271
```

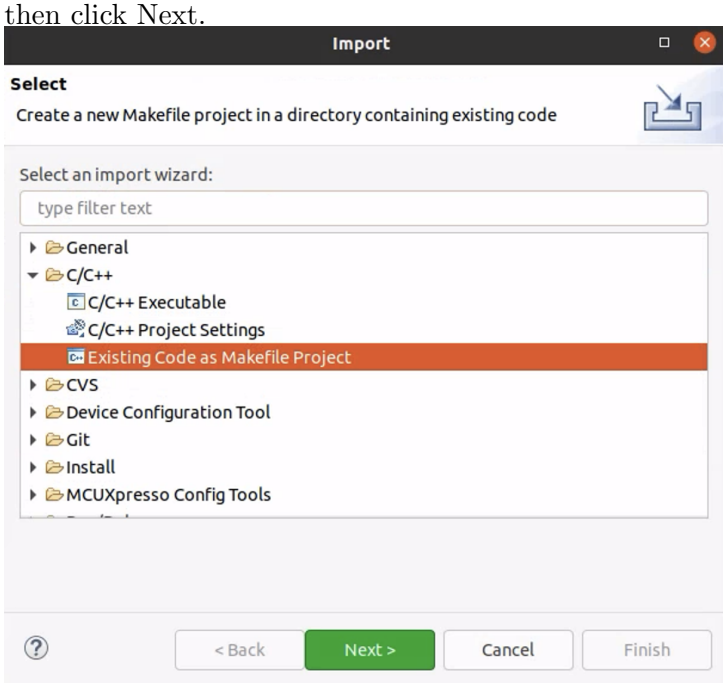
Using FMU's USB is simpler than using JLink, but it is slower in programming.

4.2 Experiment 2: Write Applications on NuttX with PX4

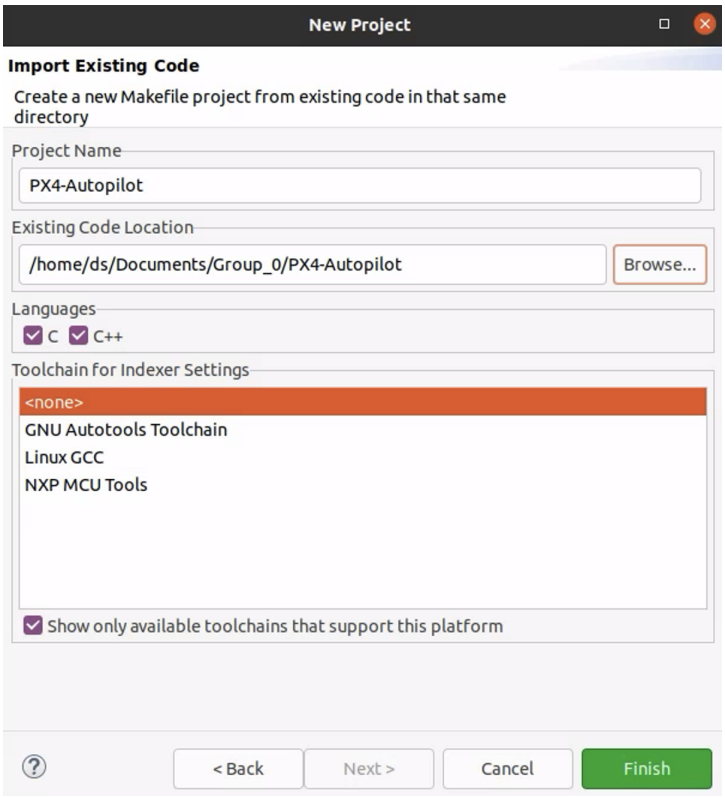
The purpose of this experiment is to introduce some of the PX4 features and write applications that run inside the framework. In order to facilitate the development and integration with the platform, you may find using an IDE helping for coding and development. Thus, we will MCUXpresso for this purpose.

Experiment Setup A: Import PX4 to MCUXpresso

1. In MCUXpresso, click on File menu and choose Import....
2. Select from the pop-up window C/C++, and choose "Existing Code as Makefile Project" then click Next.



3. From the next window, click on Browse... and navigate to the path of PX4-Autopilot then click Open.



4. Close the window by click on Finish. You will find a project created with "PX4-Autopilot" name, where you can create and edit files.

Experiment Setup B: Hello World Application

In this part, we will create a simple application that prints "Hello World" on the terminal. The steps of creating this application will be common in all the following applications.

1. Open the Ubuntu terminal and navigate to PX4-Autopilot path.
2. Create a new directory under src/examples/, and name it "hello_world".
3. Create three files under hello_world directory, and name them hello.cpp, CMakeLists.txt, and Kconfig. Note that CMakeLists.txt and Kconfig are used by PX4's tool chain to build the code.
4. In CMakeLists.txt, add the following code.

```
px4_add_module(
    MODULE examples__hello_world
    MAIN hello_world
    SRCS
        hello.cpp
    DEPENDS
)
```

In this code, we define the module name and the name of its main function in lines 2 and 3. And under SRCS tag in line 4, we write the names of the source files (in this case, we only have hello.cpp).

5. In Kconfig, add the following code.

```
1 menuconfig EXAMPLES_HELLO_WORLD
2     bool "hello_world"
3     default n
4     ---help---
5         Enable support for hello_world
```

The name in line 1 is the most important information in this file. This name will be used later to include the application inside NuttX (the operating system used by PX4), and the convention to choose the name is to write the directory name then the application name (hence, we chose EXAMPLES_HELLO_WORLD). More information about Kconfig is available in PX4 docs here.

6. In hello.cpp, add the following code that prints "Hello World".

```
#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/log.h>

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

int hello_world_main(int argc, char *argv[])
{
    for(int i = 0; i < 5; i++)
    {
        PX4_INFO("Hello World %d", i);
        px4_sleep(1);
    }

    return 0;
}
```

Note, the name of the main function should match the name that is written in line 3 in CMakeLists.txt plus the suffix "_main". This code prints "Hello World" every 1 second for five times. It is recommended to used PX4_INFO instead of the regular printf as it prints the application name along with the printed string, so it will be helpful in debugging if multiple applications print data in the same time. To use PX4_INFO, log.h header should be included. For the delays, there are two functions, px4_sleep which delays the execution for a number of seconds and px4_usleep which delays the execution for a number of microseconds.

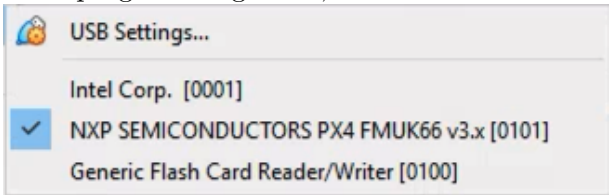
7. Now, the application is ready to be added to the NuttX applications. Navigate to "PX4-Autopilot/boards/nxp/fmuk66-v3"

8. Open default.px4board. This file should contain all the applications that run inside NuttX.

9. Add at the end of default.px4board "CONFIG_EXAMPLES_HELLO_WORLD=y". Remember, the name after "CONFIG" should match the name in line 1 in Kconfig.

10. Build and program PX4 after these edits, same as step 15 in Experiment 1.

11. After programming FMU, check the VM USB and make sure that FMUK66 is connected.



12. Open another terminal, and write `screen /dev/ttyACM0 57600 8N1`. Note, FMUK66 may appear with a different name other than "ttyACM0", if you encounter this case, try `ls /dev/tty` which lists all the devices and you can figure out the device name by disconnecting and connecting it.

13. The screen command will connect us to the NuttX terminal. Note after running the screen command, hit Enter key a couple of times until something appears on the terminal.

14. Write in the NuttX terminal `help` to view all the available applications.

15. To run our application, write `hello_world`.

16. Take a screenshot of the output and add them to your lab report.

4.3 Experiment 3: uORB Messaging

This experiment introduces uORB messaging which is the mechanism that PX4 uses for inter-thread/inter-process communication. Any application in PX4 can subscribe to a number of messages and publish some other messages. For instance, the application (process) that interfaces with the accelerometer sensor in FMU, publishes a uORB message that contains the sensor data, and other applications that require to use the accelerometer data, subscribe to that message to get the updated values of the sensor.

This concept of message passing simplifies the synchronization between processes without the need to create queues or semaphores similar to the case of FreeRTOS. Nevertheless, uORB is not a replacement to queues or semaphores, it is just an abstraction layer that is provided by PX4 to hide the details of the synchronization techniques which still run behind the scenes. For more information, check PX4 documentation from here.

Experiment Setup A: Interact with uORB

1. Build and program PX4.

2. Open NuttX terminal (similar to the previous experiment).

3. Write uorb top. This command will show you all the messages that are published and their rate of publishing.

```
update: 1s, topics: 66, total publications: 10295, 572.7 kB/s
TOPIC NAME                    INST #SUB RATE #Q SIZE
actuator_armed                   0    7    2  1   16
actuator_controls_0              0    5  794  1   48
actuator_outputs                 0    2  783  1   80
adc_report                       0    2  100  1   96
battery_status                   0    5  100  1  168
commander_state                  0    1    2  1   16
control_allocator_status         0    2  783  1   80
cpuload                          0    3    2  1   16
estimator_innovation_test_ratios 0    1   99  1  128
estimator_innovation_variances   0    1   99  1  128
estimator_innovations            0    1   99  1  128
estimator_sensor_bias            0    3    1  1  120
estimator_states                 0    1   99  1  216
estimator_status                 0    3   99  1  112
estimator_status_flags           0    2    1  1   96
failure_detector_status          0    1    2  1   24
magnetometer_bias_estimate       0    2   20  1   64
rate_ctrl_status                 0    1  794  1   24
rtl_time_estimate                0    2    1  1   24
safety                           0    2    1  1   16
sensor_accel                     0    5  399  8   48
sensor_baro                      0    2   45  1   32
sensor_baro                      1    2   11  1   32
sensor_combined                  0    2  198  1   48
sensor_gyro                      0    5  794  8   48
sensor_mag                       0    3   20  4   40
sensors_status_imu               0    1  198  1   96
system_power                     0    2  100  1   40
telemetry_status                 0    2    2  1   88
vehicle_acceleration             0    2  199  1   32
vehicle_air_data                 0    6   15  1   40
vehicle_angular_acceleration     0    1  794  1   32
vehicle_angular_velocity         0    6  794  1   32
vehicle_attitude                 0    5  198  1   56
vehicle_attitude_setpoint        0    3  198  1   64
vehicle_control_mode             0   12    2  1   24
vehicle_imu                      0    3  198  1   56
vehicle_imu_status               0    5   10  1  120
vehicle_land_detected            0   10    1  1   24
vehicle_local_position           0   11   99  1  168
vehicle_magnetometer             0    2   10  1   40
vehicle_rates_setpoint           0    3  198  1   32
vehicle_status                   0   18    2  1   88
vehicle_status_flags             0    4    2  1   48
```

4. To read one of these messages, write listener <THE NAME OF THE MESSAGE>

5. Write listener sensor_combined to read the accelerometer and the gyroscope data.

```
nsh> listener sensor_combined

TOPIC: sensor_combined
 sensor_combined
    timestamp: 161332413303 (0.003576 seconds ago)
    gyro_rad: [0.0056, 0.0067, -0.0031]
    gyro_integral_dt: 5000
    accelerometer_timestamp_relative: 0
    accelerometer_m_s2: [-0.0592, 0.0913, -9.7491]
    accelerometer_integral_dt: 4991
    accelerometer_clipping: 0
```

6. Read a couple of different messages and add them to the report.

Experiment Setup B: Subscribe to a message

In this experiment, we will write an application that subscribes to sensor_combined message and prints the accelerometer value every 200 ms.

1. Create a new application similar to Experiment 2B or use the same application.

2. Add the following code to your .cpp file. Note you may need to change the name of the main function according to your configuration.

```
#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/log.h>

#include <uORB/topics/sensor_combined.h>

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

int hello_world_main(int argc, char *argv[])
{
    int sensor_combined_handle;
    sensor_combined_s sensor_data;

    sensor_combined_handle = orb_subscribe(ORB_ID(sensor_combined));
    orb_set_interval(sensor_combined_handle, 200);

    while(1)
    {
        orb_copy(ORB_ID(sensor_combined), sensor_combined_handle, &sensor_data);

        PX4_INFO("X = %f, Y = %f, Z = %f", (double)sensor_data.accelerometer_m_s2[0],
                                           (double)sensor_data.accelerometer_m_s2[1],
                                           (double)sensor_data.accelerometer_m_s2[2]);

        px4_usleep(200000);
    }

    return 0;
}
```

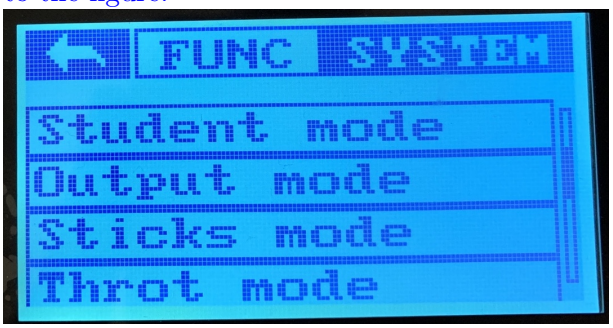
Initially, to subscribe to any message, you have to include its header file from "uORB/-topics" and to refer to the message using its ORB_ID which is defined in uORB/topic-s/uORBTopic.hpp. The next step is to call "orb_subscribe" which returns a reference to this subscription to be used later to read the data. Finally, the data can be read using "orb_copy" after passing the id of the required message and the subscription reference (sensor_combined_handle).

Note, the structure of "sensor_combined_s" is the same as the structure appeared with "listener" command, and also it can be accessed from "sensor_combined.h" file to know the data type of each field. you can find all the messages types definitions in /home/ds/Documents/PX4-Autopilot/build/nxp_fmuk66-v3_default/uORB/topics/

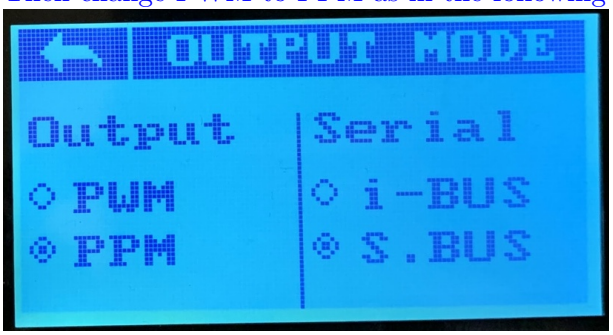
Project2 – Step 0

Write an application that prints the values of RC channels every 200 ms.
Note, the RC message will not be published unless the receiver is connected to FMU and the controller is powered on. Additionally, the controller's output should be changed to PPM

instead of PWM to be compatible with PX4. To change the controller's output, open the controller's settings and choose SYSTEM, and from SYSTEM choose Output mode similar to the figure.



Then change PWM to PPM as in the following figure.



Finally, after powering on FMU and the RC connection is established, you will notice a change in the blinking color of FMU from blue to violet.

Experiment Setup C: Publish a message

In this experiment, we will control the motors of the car using uORB messages, but unlike the previous experiment, we have to publish the motor messages.

1. Create a new application similar or use your existing application.
2. Add the following code in you source file.

```
#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/log.h>

#include <drivers/drv_hrt.h>
#include <uORB/Publication.hpp>
#include <uORB/topics/test_motor.h>


#define DC_MOTOR        0


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

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

```
18
19      uORB::Publication<test_motor_s> test_motor_pub(ORB_ID(test_motor));
20
21      while(1)
22      {
23          PX4_INFO("Enter speed value (0 to 1). If you enter a value outside the range,
        the motor will be stopped and the application will be terminated.");
24          scanf("%lf", &motor_value);
25          if(motor_value > 1.0 || motor_value < 0)
26              break;
27
28          PX4_INFO("Motor speed is %f", motor_value);
29          test_motor.timestamp = hrt_absolute_time();
30          test_motor.motor_number = DC_MOTOR;
31          test_motor.value = (float)motor_value;
32          test_motor.action = test_motor_s::ACTION_RUN;
33          test_motor.driver_instance = 0;
34          test_motor.timeout_ms = 0;
35
36          test_motor_pub.publish(test_motor);
37      }
38
39      PX4_INFO("The motor will be stopped");
40      test_motor.timestamp = hrt_absolute_time();
41      test_motor.motor_number = DC_MOTOR;
42      test_motor.value = 0.5;
43      test_motor.driver_instance = 0;
44      test_motor.timeout_ms = 0;
45
46      test_motor_pub.publish(test_motor);
47
48      return 0;
49  }
```

This code publishes "test_motor" messages to control the dc motor speed. In line 19, we create an object from Publication class to handle the publication process of the message, and the constructor of the class requires the id of the message for the object creation. This object is used in line 36 and line 46 to publish the content of test_motor message.

The the value stored in "test_motor" ranges from 0 to 1 to configure the ESC to drive the motor in the forward and reverse directions. **The first value you enter determines the stopping point of the motor and it is recommended to be 0.5 or less**. As such, zero value will drive the motor with the maximum speed in one direction and one will move it in the other direction. In Line 42, we assume that 0.5 is the stopping value of the motor.

Note that "test_motor" expects the motor number starts from zero, so if pin 1 is used for connecting the motor, it should be zero in the code, and pin 2 is 1, and so on.

3. Build and program the code.

4. Run the application from NuttX shell, and try to input different values between 0 and 1. Remember, the first value you enter configures the stopping value.

5. To terminate the application and stop the motor, enter any value smaller than 0 or larger than 1.

Project2 – Part 1 (Revisiting Project 1 using PX4)

(a) Edit the code in Experiment 3C to control both the dc motor and the servo.

(b) Write a code that integrates reading RC channel, and controlling the motors accordingly.

(c) BONUS Write a code that publishes led_control message to control the LED color. Note, write 0xFF in led_mask field in led_control, and set priority field to led_contro_s::MAX_PRIORITY.

(d) BONUS Modify the code you wrote in b to mimic Project 1. i.e., by integrates reading the accelerometer value, reading RC channels (with the three gear/speed modes), controlling the motors, and control the LED colors. That code should provide the same functions required in Project 1 except for controlling the car from the terminal.

4.4 Experiment 4: MAVLink

MAVLink is another communication protocol similar to uORB messaging system, and it is used for communicating with the outside modules such as a companion computer, or a camera. In this experiment, we will use MAVLink to establish a communication link between FMU and your PC (PC is consider as a companion computer for this experiment only, but it will be replaced later with Raspberry PI). The detailed documentation of MAVLink is available on this link, and there is a useful tutorial provided from PX4 on this link.

Experiment Setup A: Setting up the FMU side and Reading Messages from FMU

In this experiment, we will write a Python application that reads messages from FMU.

1. Build and program PX4.
2. From the NuttX terminal, write this command:
   `mavlink start -d /dev/ttyACM0 -b 2000000 -r 800000 -x`

   This command starts MAVLink on the specified serial link after '-d', so it starts MAVLink on USB which is ACM0. Also, the command specify the baud rate and the maximum data rate which they are 2000000 and 800 KB/s, respectively.
3. After starting MAVLink, we no longer need the NuttX terminal, and MAVLink requires the USB connection as well. Thus, to close the Screen session, press CTRL+A then press \. You may need to tap the three keys simultaneously multiple times to close the screen session.
4. We will switch to Python application that reads MAVLink messages.

5. Open a file, and write the following code.

```
1  from pymavlink import mavutil
2
3
4  # Start a connection
5  the_connection = mavutil.mavlink_connection('/dev/ttyACM0')
6
7  # Wait for the first heartbeat
8  #   This sets the system and component ID of remote system for the link
9  the_connection.wait_heartbeat()
10 print("Heartbeat from system (system %u component %u)" % (the_connection.target_system,
       the_connection.target_component))
11
12 # Once connected, use 'the_connection' to get and send messages
13 while 1:
14     msg = the_connection.recv_match(blocking=True)
15     print(msg)
```

In line 5, we create an object from mavlink_connection that connects to ttyACM0 (note that USB may appear with a different name. To confirm its name, run ls /dev/tty* in the terminal to view all the devices). Then the code waits at line 9 till the connection starts. Afterwards, the message are read one by one at line 14.

6. Run the code, and you will receive a stream of messages.

7. To filter the messages, we can choose to receive a certain type of messages by adding the following parameter to line 14.

```
msg = the_connection.recv_match(type='ATTITUDE', blocking=True)
```

This parameter will instruct the function to receive 'ATTITUDE' messages only. This message contains the angles of car in space and their rates of changing, try to lift the car and rotate it in different angles, and observe the change of the values in real-time.

Experiment Setup B:(OPTIONAL) Read Custom Messages from FMU

This experiment is similar to Experiment 4A, but instead of reading the default PX4 messages, we will read a message that is sent from a created application that runs on FMU.
(OPTIONAL Note:) Since this direction (messaging from FMU to companion computer) is not exercised in the final outcome of the lab/project, this part is optional (please note that unlike bonus, optional is not graded). We put it here in case some one is interested in exploring this direction.

1. Create a new application inside PX4.

2. Add the following code in your source file.

```
#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/log.h>
```

```
#include <drivers/drv_hrt.h>
#include <uORB/Publication.hpp>
#include <uORB/topics/debug_value.h>

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

int hello_world_main(int argc, char *argv[])
{
    printf("The Application starts!");

    int counter = 0;
    debug_value_s debug_data;

    uORB::Publication<debug_value_s> debug_value_pub(ORB_ID(debug_value));


    while (1) {
        debug_data.timestamp = hrt_absolute_time();
        debug_data.value = counter / 2.0f;
        debug_data.ind = counter;

        debug_value_pub.publish(debug_data);

        counter++;
        px4_sleep(1);
    }

    return 0;
}
```

This code publish debug_value message which is a regular uORB message that is mapped to a MAVLink message. Consequently, we can deal with the MAVLink messages from FMU side the same way we dealt with uORB messages (subscribe to it or publish it).

3. In the NuttX terminal, run your application in a new thread by adding a space and '&' after its name.
4. Start MAVLink, and exit the Screen session.
5. On the Python side, the same code from Experiment 4A with changing the filter to 'DEBUG'.
6. Take a screenshot of the receive messages.

Experiment Setup C: Send Messages to FMU

In this experiment, we will send data from a Python application to FMU and based on the data, an application on the FMU side will turn LED on and off.

1. Add the following code in your source file.

```
#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/log.h>
#include <uORB/Publication.hpp>
```

```
#include <uORB/topics/led_control.h>
#include <uORB/topics/debug_value.h>

extern "C" __EXPORT int hello_world_main(int argc, char *argv[]);
int hello_world_main(int argc, char *argv[])
{
  px4_sleep(2);

  debug_value_s debug_data;
  int debug_handle = orb_subscribe(ORB_ID(debug_value));
  orb_set_interval(debug_handle, 500);

  led_control_s led_control;
  uORB::Publication<led_control_s> led_control_pub(ORB_ID(led_control));

  while(1)
  {
    orb_copy(ORB_ID(debug_value), debug_handle, &debug_data);

    led_control.timestamp = hrt_absolute_time();
    led_control.color = led_control_s::COLOR_GREEN;
    led_control.priority = led_control_s::MAX_PRIORITY;
    led_control.led_mask = 0xff;

    if(debug_data.ind == 1)
      led_control.mode = led_control_s::MODE_ON;
    else
      led_control.mode = led_control_s::MODE_OFF;

    led_control_pub.publish(led_control);
    px4_usleep(500000);
  }
  return 0
}
```

This code subscribe to debug value message which is a regular uORB message that is mapped to a MAVLink message. The led will blink every 1 sec. Notice adding 2 second at the beginning of the application, to make sure it is not starting before the system initialization.

2. Instead of running the application from the NuttX terminal, we will start the application and MAVLink automatically with the board startup.

3. Write the name of the application with '&' at the end of the file, "PX4-Autopilot/boards/nxp/fmuk66-v3/init/rc.board_defaults".

4. Write MAVLink start command at the end of the file "PX4-Autopilot/boards/nxp/fmuk66-v3/init/rc.board_sensors".

5. On the Python side, write the following code that sends Debug message that contains zero or one.

```
from pymavlink import mavutil
import time
```

```
# Start a connection
the_connection = mavutil.mavlink_connection('/dev/ttyACM0')

# Wait for the first heartbeat
#   This sets the system and component ID of remote system for the link
the_connection.wait_heartbeat()
print("Heartbeat from system (system %u component %u)" % (the_connection.target_system,
    the_connection.target_component))

# Once connected, use 'the_connection' to get and send messages
value = 0
while 1:
    message = mavutil.mavlink.MAVLink_debug_message(0, value, 0.0)
    the_connection.mav.send(message)
    time.sleep(1)
    value = (value + 1) % 2
    print("Message sent")
```

MAVLink_debug_message forms a debug message where the first parameter is mapped to debug_value.timestamp, while the second and the third are mapped to debug_value.ind and debug_value.value, respectively.

6. Run the code, and you should observe that LED blinks every second.

4.5 Experiment 5: Ultrasonic Sensor

In this experiment, we will interface with an ultrasonic sensor using Raspberry PI (RPI) to read the distance between the car and the closest obstacle.

1. Power on RPI board and connect it to a monitor.
2. Open Thonny application which is a simple Python IDE.
3. Add the following code.

```
#Libraries
import RPi.GPIO as GPIO
import time


#set GPIO Pins
GPIO_TRIGGER = 23
GPIO_ECHO = 24


def ultrasonic_setup():
    #GPIO Mode (BOARD / BCM)
    GPIO.setmode(GPIO.BCM)

    #set GPIO direction (IN / OUT)
    GPIO.setup(GPIO_TRIGGER, GPIO.OUT)
    GPIO.setup(GPIO_ECHO, GPIO.IN)
```

```
def distance():
    # set Trigger to HIGH
    GPIO.output(GPIO_TRIGGER, True)

    # set Trigger after 0.01ms to LOW
    time.sleep(0.00001)
    GPIO.output(GPIO_TRIGGER, False)

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

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

    # save time of arrival
    while GPIO.input(GPIO_ECHO) == 1:
        StopTime = time.time()

    # time difference between start and arrival
    TimeElapsed = StopTime - StartTime
    # multiply with the sonic speed (34300 cm/s)
    # and divide by 2, because there and back
    distance = (TimeElapsed * 34300) / 2

    return distance


ultrasonic_setup();
while True:
    dist = distance()
    print ("Measured Distance = %.1f cm" % dist)
    time.sleep(0.1)
```

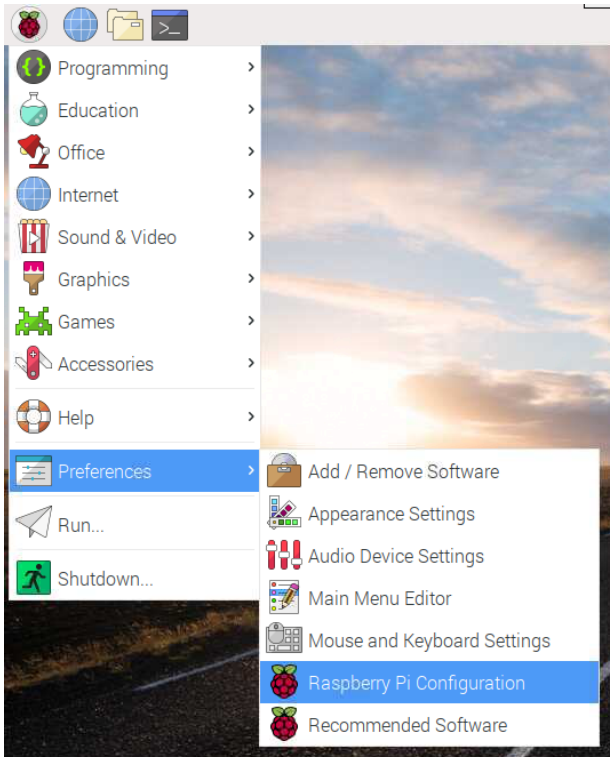
This code contains two functions, ultrasonic_setup and distance. Ultrasonic_setup function initializes the two pins, Trigger and Echo, required to interface the sensor, while distance function start the sensor operation by putting zero on Trigger pin then it calculates the duration of the pulse appears on Echo pin. The duration of the pulse generated on Echo pin is directly proportion to the distance.

4. Run the code and observe changing the distance with approaching any object to the sensor.

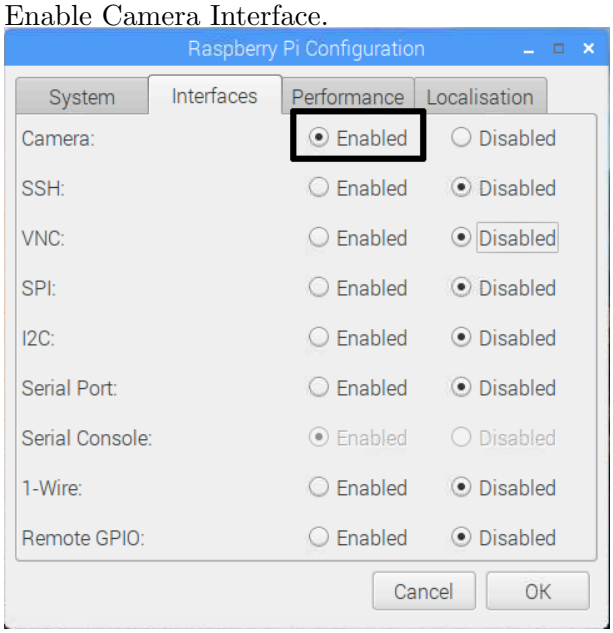
4.6 Experiment 6: Camera

In this experiment, we will interface with a camera connected to RPI to choose which direction the car should move. First, make sure the camera interface is enables.

1. Open Raspberry Pi Configuration.



2. Enable Camera Interface.



3. Reboot your Raspberry Pi.

4. Now, write the code to read data from the camera. Open Thonny application.

5. Add the following code.

```
import cv2
```

```
import numpy as np
import math

cap = cv2.VideoCapture(0)
StepSize = 5

def getChunks(l, n):
    """Yield successive n-sized chunks from l."""
    a = []
    for i in range(0, len(l), n):
        a.append(l[i:i + n])
    return a

def process_camera_frame():
    ret,frame = cap.read()
    img = cv2.flip(frame, 0)
    blur = cv2.bilateralFilter(img,9,40,40)
    edges = cv2.Canny(blur,50,100)
    img_h = img.shape[0] - 1
    img_w = img.shape[1] - 1
    EdgeArray = []

    for j in range(0,img_w,StepSize):
        pixel = (j,0)
        for i in range(img_h-5,0,-1):
            if edges.item(i,j) == 255:
                pixel = (j,i)
                EdgeArray.append(pixel)
                break

    if len(EdgeArray) != 0:
        chunks = getChunks(EdgeArray, math.ceil(len(EdgeArray)/3))
    else:
        return

    #c = []
    distance = []
    for i in range(len(chunks)):
        x_vals = []
        y_vals = []
        for (x,y) in chunks[i]:
            x_vals.append(x)
            y_vals.append(y)
        avg_x = int(np.average(x_vals))
        avg_y = int(np.average(y_vals))
        #c.append([avg_y,avg_x])
        distance.append(math.sqrt((avg_x - 320)**2 + (avg_y - 640)**2))
        cv2.line(img, (320, 640), (avg_x,avg_y), (0,0,255), 2)

    cv2.imshow("frame", img)
    cv2.waitKey(5)
    if(distance[0] < distance[1]):
        if(distance[0] < distance[2]):
            return 0
        else:
            return 2
```

```
    else:
        if(distance[1] < distance[2]):
            return 1
        else:
            return 2

while(1):
    ret = process_camera_frame()
    if ret == 0:
        print('Left direction is preferred');
    elif ret == 1:
        print('Forward direction is preferred');
    elif ret == 2:
        print('Right direction is preferred');
```

This code applies a simple computer vision to help the car to avoid any obstacles by steering it away from them. The algorithm follows these steps:

(a) Read an image from the camera and flip it vertically.

(b) Filter the image to remove the noise.

(c) Apply an edge detection algorithm (Canny) to convert the image to edges.

(d) Divide the image into three vertical chunks.

(e) Find the mean point in every chunk.

(f) Calculate the distance between the down mid point of the image frame and mean point of every chunk.

(g) The chunk that gives the shortest distance will be the preferred direction as it means no object appear in that chunk.

6. Run the code and observe changing the preferred direction of the car.

Project2 – Part 2

In this part of the project, you will be using the given algorithm above that gives you the distance from the ultrasonic and directions from camera, then write a small logic/function that does the following:

1. Reads the data from sensors, sends them to FMU board and actuates the motors accordingly.

2. It is recommended to have the car decelerates if the distance between the car and the closest obstacle is less than 50 cm, and stops completely if the distance becomes less than 15 cm. (you can define different safe distances to decrease the speed and stop, based on the speed of your car, but please discuss that in the demo and the report)

3. Likewise, you can use another camera algorithm for directions detection, but you need to discuss that in the demo and the report.

Note that in this experiment, you are required to establish the MAVlink connection between the RPI and the FMU to communicate data between them.