PX4 Otopilot Yazılımı için Modül Oluşturulması ve MAVLink İletişiminin Sağlanması

Date	**: 14.11.2022**
Version	**: 01**
Change	**: 00**
Document	**: Report**
Author	**: Ömer Can VURAL**

Contents

1 INTRODUCTION

Within the scope of Z-Sistem Havacılık ve Bilişim San. ve Tic. A.Ş. activities, an agricultural spraying copter will be developed. To ensure the device performs its mission as expected, the necessary tasks will be processed by the agricultural module added to the PX4 software. The flight control software of the said device is PX4 autopilot based and the ground control station is QGroundControl based, and the general autopilot software is named Z4. This report summarizes in detail how the agricultural module within the Z4 software was created and what needs to be considered when creating a module in PX4 autopilot software.

To create the PX4-based Z4 autopilot software to be used in the TAR DRONE 1 agricultural spraying device, additional functionality needs to be added to the PX4 software. For this purpose, a module has been created in the PX4 software, and the foundations of the agricultural module software that will provide these functionalities have been laid. The initial version of the agricultural module software has the features of spraying chemicals within geographic boundaries while the device is on its route, using geographic boundary and flight route information determined via the ground control station, and terminating the spraying process if it violates geographic boundaries.

2 SUMMARY

By utilizing the 'geofence' functions within the 'navigator' module of the PX4 autopilot software, it was determined whether there was a geographic boundary violation based on the device's current position information. Then, the conditions for the device to start and stop the chemical spraying operation were determined; accordingly, if the device violates the geographic boundaries, the chemical spraying operation will be terminated. For the condition statuses to be instantly displayed on the ground control station, the 'agricultural module' function defined within the 'navigator' module needs to be able to communicate via MAVLink. For this, the condition status information received from the 'agricultural module' was published via uORB and obtained from the 'MAVLink' module. The information obtained from the 'MAVLink' module was transmitted to the ground control station.

To enable uORB publishing, a 'tarim_modulu.msg' file was first created in the ./msg directory of the PX4-autopilot software and a uORB topic (communication channel) was created by defining it in the 'CMakeList.txt' file in the same directory. After the changes, the software needs to be compiled. Otherwise, the functions and header files containing the necessary parameters for using the publication channel cannot be created. Then, the elapsed time since the beginning of the operation and geographic boundary violation information from the 'tarim_modulu_ilaclama' function defined in the 'navigator' module become available for sharing with other modules via uORB.

To receive the information published via uORB and transmit it to the ground control station, it is necessary to subscribe to the publishing channel and then transmit the received information to the ground control station via MAVLink. Therefore, the module that will subscribe to the channel and transmit it to the ground control station is the 'mavlink' module in the autopilot software. To perform the operation, the 'TARIM_MODULU.hpp' file, where we made the necessary definitions, needs to be created in the 'streams' directory within the 'mavlink' module. In the subsequent process, features such as message publishing, uORB subscription, and MAVLink message sending, whose definitions we made, were also added to the 'mavlink_main' and 'mavlink_messages' files within the 'mavlink' module.

The coding work for the violation module software added to the autopilot software was carried out using the Visual Studio Code editor. Simulation studies were performed using the Gazebo 9 simulation program and QGroundControl ground control station program on the Ubuntu operating system version 20.04 LTS. Inter-module communication within the autopilot is carried out with uORB middleware, and communication between the autopilot and the ground control station is carried out with the MAVLink messaging protocol.

To define the flight area boundaries of the device, a polygon is created using the 'fence' feature of the QGroundControl ground control station software and sent to the autopilot hardware. The autopilot software uses functions in the 'geofence' block of the 'navigator' module (./src/modules/navigator/geofence.cpp) that process the device's polygon boundaries and return a "bool" type value to the 'navigator' module regarding whether the polygon boundaries have been violated.

Polygons can have different shapes, such as polygonal or circular. Therefore, before checking for boundary violations, we must know the polygon type and apply the "Point Inclusion in Polygon Test – PNPOLY" algorithm based on this information. The functions are used under the "geofence_breach_check" function of the 'navigator_main' block in the autopilot software to send an "Approaching on Geofence" warning to the ground control station when the device approaches the polygon boundaries. In other words, its usage is not based on polygon violation, but on the condition of approaching polygon boundaries, and it is determined according to the test point/distances defined in the 'GeofenceBreachAvoidance' block.

3 METODOLOJİ

The coding work for the agricultural module software added to the autopilot software was carried out using the Visual Studio Code editor. Simulation studies were performed using the Gazebo simulation program and QGroundControl ground control station program on the Ubuntu operating system version 20.04 LTS. Inter-module communication within the autopilot is carried out with uORB, and communication between the autopilot and the ground control station is carried out with the MAVLink protocol.

3.1 Preparing Software Infrastructure for Agricultural Module

Since the aim is to effectively spray agricultural lands and reduce chemical waste and environmental damage, the device must only spray designated areas. To define the agricultural land boundaries, a polygon is created using the 'fence' feature of the QGroundControl ground control station software and sent to the autopilot software.

The autopilot software has functions in the 'geofence' block of the 'navigator' module (./src/modules/navigator/geofence.cpp) that process the device's polygon boundaries and return a "bool" type value to the 'navigator' module regarding whether the polygon boundaries have been violated. Below, the 'insidePolygon' function is shown, which returns :TRUE if a polygonal polygon, with its latitude, longitude, and altitude information, is within the device's polygon boundaries.

```
/*          ./src/modules/navigator/geofence.cpp   */

bool Geofence::insidePolygon(const PolygonInfo &polygon, double lat, double lon, float altitude)
{
        mission_fence_point_s temp_vertex_i;
        mission_fence_point_s temp_vertex_j;
        bool c = false;

        for (unsigned i = 0, j = polygon.vertex_count - 1; i < polygon.vertex_count; j = i++) {
                if (dm_read(DM_KEY_FENCE_POINTS, polygon.dataman_index + i, &temp_vertex_i,
                            sizeof(mission_fence_point_s)) != sizeof(mission_fence_point_s)) {
                        break;
                }

                if (dm_read(DM_KEY_FENCE_POINTS, polygon.dataman_index + j, &temp_vertex_j,
                            sizeof(mission_fence_point_s)) != sizeof(mission_fence_point_s)) {
                        break;
                }

                if (temp_vertex_i.frame != NAV_FRAME_GLOBAL && temp_vertex_i.frame != NAV_FRAME_GLOBAL_INT
                    && temp_vertex_i.frame != NAV_FRAME_GLOBAL_RELATIVE_ALT
                    && temp_vertex_i.frame != NAV_FRAME_GLOBAL_RELATIVE_ALT_INT) {
                        // TODO: handle different frames
                        PX4_ERR("Frame type %i not supported", (int)temp_vertex_i.frame);
                        break;
                }

                if (((double)temp_vertex_i.lon >= lon) != ((double)temp_vertex_j.lon >= lon) &&
                    (lat <= (double)(temp_vertex_j.lat - temp_vertex_i.lat) * (lon - (double)temp_vertex_i.lon) /
                     (double)(temp_vertex_j.lon - temp_vertex_i.lon) + (double)temp_vertex_i.lat)) {
                        c = !c;
                }
        }
        return c;
}
```

Polygons can have different shapes, such as **polygonal or circular**. Therefore, before checking for a boundary violation, we need to know the type of polygon. We then apply the "**Point Inclusion in Polygon Test – PNPOLY**" algorithm based on this information. To determine the polygon type, we utilize the "**isInsidePolygonOrCircle**" function within the "geofence" block of the "navigator" module, which forms the backbone of our 'Agricultural Module'.

```cpp
/*          ./src/modules/navigator/geofence.cpp    */

bool Geofence::isInsidePolygonOrCircle(double lat, double lon, float altitude)
{
        if (dm_trylock(DM_KEY_FENCE_POINTS) != 0) {
                return true;
        }

        mission_stats_entry_s stats;
        int ret = dm_read(DM_KEY_FENCE_POINTS, 0, &stats, sizeof(mission_stats_entry_s));

        if (ret == sizeof(mission_stats_entry_s) && _update_counter != stats.update_counter) {
                _updateFence();
        }

        if (isEmpty()) {
                dm_unlock(DM_KEY_FENCE_POINTS);
                return true;
        }

        if (_altitude_max > _altitude_min) {
                if (altitude > _altitude_max || altitude < _altitude_min) {
                        dm_unlock(DM_KEY_FENCE_POINTS);
                        return false;
                }
        }

        bool outside_exclusion = true;
        bool inside_inclusion = false;
        bool had_inclusion_areas = false;

        for (int polygon_idx = 0; polygon_idx < _num_polygons; ++polygon_idx) {
                if (_polygons[polygon_idx].fence_type == NAV_CMD_FENCE_CIRCLE_INCLUSION) {
                        bool inside = insideCircle(_polygons[polygon_idx], lat, lon, altitude);

                        if (inside) {
                                inside_inclusion = true;
                        }

                        had_inclusion_areas = true;

                } else if (_polygons[polygon_idx].fence_type == NAV_CMD_FENCE_CIRCLE_EXCLUSION) {
                        bool inside = insideCircle(_polygons[polygon_idx], lat, lon, altitude);

                        if (inside) {
                                outside_exclusion = false;
                        }

                } else {
                        bool inside = insidePolygon(_polygons[polygon_idx], lat, lon, altitude);

                        if (_polygons[polygon_idx].fence_type == NAV_CMD_FENCE_POLYGON_VERTEX_INCLUSION) {
                                if (inside) {
                                        inside_inclusion = true;
                                }

                                had_inclusion_areas = true;

                        } else {
                                if (inside) {
                                        outside_exclusion = false;
                                }
                        }
                }
        }

        dm_unlock(DM_KEY_FENCE_POINTS);

        return (!had_inclusion_areas || inside_inclusion) && outside_exclusion;
}

```

These are the **core functions** used in the 'Agricultural Module', but their definitions exist only within the 'geofence' block. In the autopilot software, these functions are used under the "geofence_breach_check" function of the 'navigator_main' block to send an "**Approaching on Geofence**" warning to the ground control station when the device nears the polygon boundaries. This means their usage is based not on a polygon violation, but on the condition of approaching the polygon boundaries, determined by the test points/distances defined in the 'GeofenceBreachAvoidance' block.

Test Mechanism:

```
/*          /src/modules/navigator/GeofenceBreachAvoidance/geofence_breach_avoidance.cpp */

TEST_F(GeofenceBreachAvoidanceTest, waypointFromBearingAndDistance)
{

        GeofenceBreachAvoidance gf_avoidance(nullptr);
        struct map_projection_reference_s ref = {};
        Vector2d home_global(42.1, 8.2);
        map_projection_init(&ref, home_global(0), home_global(1));

        Vector2f waypoint_north_east_local(1.0, 1.0);
        waypoint_north_east_local = waypoint_north_east_local.normalized() * 10.5;
        Vector2d waypoint_north_east_global =
gf_avoidance.waypointFromBearingAndDistance(home_global, M_PI_F * 0.25f, 10.5);

        float x, y;
        map_projection_project(&ref, waypoint_north_east_global(0),
waypoint_north_east_global(1), &x, &y);
        Vector2f waypoint_north_east_reprojected(x, y);

        EXPECT_FLOAT_EQ(waypoint_north_east_local(0),
waypoint_north_east_reprojected(0));
        EXPECT_FLOAT_EQ(waypoint_north_east_local(1),
waypoint_north_east_reprojected(1));

        Vector2f waypoint_south_west_local = -waypoint_north_east_local;

        Vector2d waypoint_southwest_global =
gf_avoidance.waypointFromBearingAndDistance(home_global, M_PI_F * 0.25f, -10.5);

        map_projection_project(&ref, waypoint_southwest_global(0),
waypoint_southwest_global(1), &x, &y);
        Vector2f waypoint_south_west_reprojected(x, y);

        EXPECT_FLOAT_EQ(waypoint_south_west_local(0),
waypoint_south_west_reprojected(0));
        EXPECT_FLOAT_EQ(waypoint_south_west_local(1),
waypoint_south_west_reprojected(1));
```

```

        Vector2d same_as_home_global = gf_avoidance.waypointFromBearingAndDistance(home_global, M_PI_F * 0.25f, 0.0);

        EXPECT_LT(Vector2d(home_global - same_as_home_global).norm(), 1e-4);
}

```

Warning Mechanism:

```
/*          /src/modules/navigator/navigator_main.cpp        */
void Navigator::geofence_breach_check(bool &have_geofence_position_data)
{
……
……
……
gf_violation_type.flags.fence_violation = !_geofence.isInsidePolygonOrCircle(fence_violation_test_point(0),
                                fence_violation_test_point(1),
                                _global_pos.alt);
……
……
……
if (gf_violation_type.value) {
                        _geofence_result.geofence_violated = true;

                        if (!_geofence_violation_warning_sent && _vstatus.arming_state == vehicle_status_s::ARMING_STATE_ARMED) {
                                mavlink_log_critical(&_mavlink_log_pub, "Approaching on Geofence");
……
……
……

```

The "geofence_breach_check" function, as seen in the warning mechanism section, relies on the availability of polygon data provided by the ground control station. The warning message itself is structured using "gf_violation_type," which categorizes polygon violations and takes its value based on what the "isInsidePolygonOrCircle" function in the "geofence" block returns. Depending on this returned value, if the device is approaching a polygon boundary, a warning message is sent to the ground control station via the "mavlink_log_critical" function.

In the "isInsidePolygonOrCircle" function, the 'lat' and 'lon' parameters (referred to here as 'fence_violation_test_point(x)') are generated through functions in the test mechanism and used by being indexed into a 2D vector structure. The device's current altitude ('alt') is obtained from the '_global_pos' structure. Since our goal is to continuously check for polygon violations, we also need to get the 'lat' and 'lon' parameters directly from the '_global_pos' structure in our 'Agricultural Module' function.

This is how the necessary functions and parameters for use in the agricultural module were determined. The agricultural module itself was created within the 'navigator_main' block. The reason for creating it in the 'navigator_main' block is that all the definitions we need are already present in this block.

Agriculture Module infrastructure;

```
/*          /src/modules/navigator/navigator_main   */
void Navigator::tarim_modul_ilaclama(bool &have_geofence_position_data, double lat,
double lon, double alt)
{
        bool ihlal = _geofence.isInsidePolygonOrCircle(lat, lon, alt);
         int ilacAktif;

        if (!ihlal) {
                ilacAktif = 2;
        }
        else {
                ilacAktif = 1;
}}
```

Here's the first function that forms the core of the agricultural module. The tarim_modul_ilaclama function needs the parameter bool &have_geofence_position_data to define polygon data, and double lat, double lon, and double alt (latitude, longitude, altitude) parameters to execute the device's position and "geofence" functions.

The ihlal (violation) variable receives a boolean value indicating whether the device is within the polygon boundaries based on latitude, longitude, and altitude information. If the device is inside the polygon boundaries, it returns TRUE, meaning the ilacAktif (chemicalActive) integer variable should be set to 1, indicating that the device is requested to spray chemicals. Otherwise, it implies the device should not spray chemicals.

All functionalities of the 'navigator' module are executed via the RUN function within the 'navigator_main' block. Therefore, the tarim_modulu_ilaclama function must also be defined here and in the header file.

Execution

```
void
Navigator::run()
{
……
……
……
tarim_modul_ilaclama(have_geofence_position_data, _global_pos.lat, _global_pos.lon,
_global_pos.alt);
……
……
……

```

Definition in Header File ;

```
class Navigator : public ModuleBase<Navigator>, public ModuleParams
{
public:
……
……
……
void           tarim_modul(bool &have_geofence_position_data, double lat, double lon,
double alt);
……
……
……

```

3.2 Communication via uORB Channel

Once the functional infrastructure of the agricultural module is established, it needs to communicate with other modules to physically perform its functions like chemical spraying and ground control station communication. In the PX4 autopilot software, inter-module communication is achieved using the **uORB structure**. uORB is a specialized messaging protocol embedded within the PX4 autopilot software that operates on a **publish/subscribe** model.

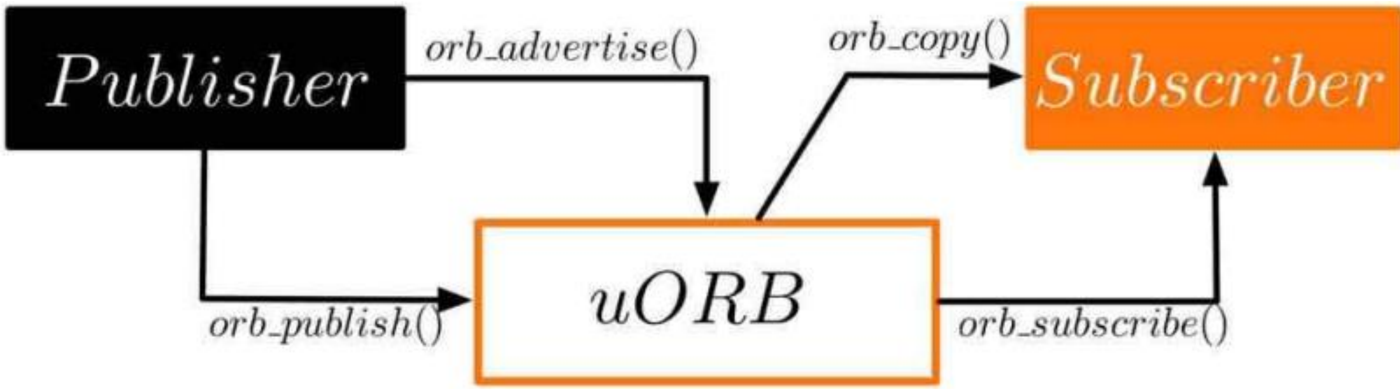


Figure 1 – uORB Diyagramı - https://i0.wp.com/uav-lab.org/wp-content/uploads/2016/08/uORB-1-e1489045507250.jpg?fit=554%2C385&ssl=1

For the Agricultural Module, the following will be implemented:

- A **publication channel (TOPIC)** will be created for uORB messaging.
- The value received by the "**ilacAktif**" variable in the tarim_modulu_ilaclama function will be assigned to a parameter within the struct created for uORB messaging.
- The assigned value will then be published on a uORB channel containing the parameters from the struct.

3.2.1 Creating a Publication Channel (TOPIC) for uORB Messaging

To create a uORB channel, I'll navigate to the PX4/msg/ directory within the autopilot software and create a **.msg file** with the desired channel name. Since module communication will occur via this channel, I'm creating the tarim_modulu.msg file here.

Channels used for uORB messaging **must include a timestamp parameter**. This parameter holds information about the time elapsed since a publication was made on the channel, allowing the system to transmit desired information to modules without significant delays.

uORB Channel ;

```
# ./PX4-Autopilot/msg/tarim_modulu.msg
uint64 timestamp
uint16 ilac_durum

```

When creating a message file, you should keep the following in mind:

- It **must** include a **timestamp** parameter of type **uint64**.
- Defined parameters **must be entirely in lowercase or entirely in uppercase**. For example, a parameter cannot be named "ilacDurum."

After creating the tarim_modulu.msg file, it must then be defined in the **CMakeList.txt** block, also located in the PX4/msg/ directory. CMakeList.txt is a rule definer that specifies where and for what purpose structures like modules and blocks will be used within the system.

CMakeList.txt ;

```
# ./PX4-Autopilot/msg/CMakeList.txt
cmake_policy(SET CMP0057 NEW)

set(msg_files
  …
  …
  tarim_modulu.msg
  …
  …

```

After these steps, the autopilot software **must be compiled**. This is necessary for the system to generate the required software and definition files based on the tarim_modulu.msg block we defined in CMakeList.txt. Without compilation, the necessary definition files won't be created, and uORB messaging cannot be established.

3.2.2 Publishing Messages on the uORB Channel

After creating my message file, I'll go back to my tarim_modul_ilaclama function and proceed with creating the struct that will contain the parameters I want to publish on uORB, and then publish the message on uORB.

- When creating a struct for uORB publication, make sure the struct name is the same as the created message file name.
- The created struct must contain the same parameters as the message file.

Struct ;

```
/*          /src/modules/navigator/navigator.h      */
class Navigator : public ModuleBase<Navigator>, public ModuleParams
{
public:
……
void            tarim_modul(bool &have_geofence_position_data, double lat, double lon,
double alt);
struct tarim_modulu_nav_s {
                uint64_t timestamp;
                uint16_t ilacDurum;
        } ;
……

```

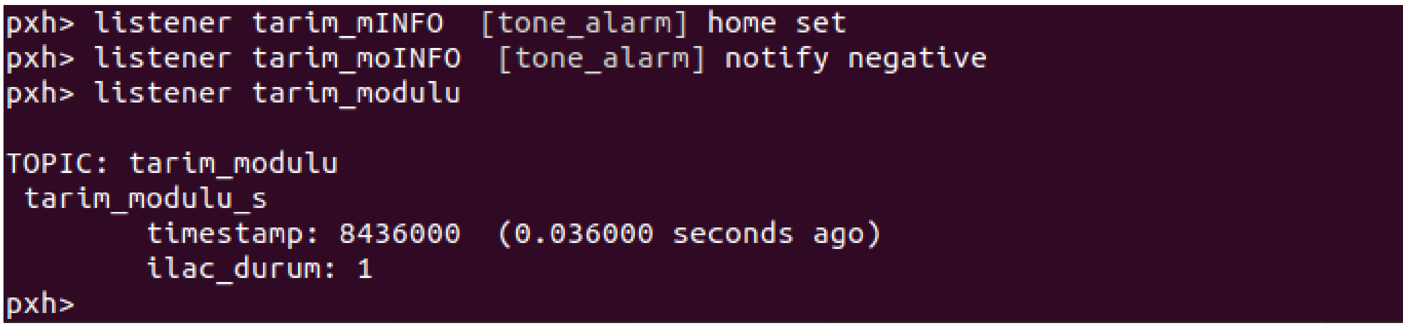
tarim_modul_ilaclama function ;

```
/*          /src/modules/navigator/navigator_main.cpp         */
void Navigator::tarim_modul_ilaclama(bool &have_geofence_position_data, double lat,
double lon, double alt)
{
        bool ihlal = _geofence.isInsidePolygonOrCircle(lat, lon, alt);

        int ilacAktif;

        if (ihlal) {
                ilacAktif = 2;
        }
        else {
                ilacAktif = 1;
        }

        struct tarim_modulu_s tarimNavStruct;

        memset(&tarimNavStruct, 0, sizeof(tarimNavStruct));

        orb_advert_t tarim_modulu_pub = orb_advertise(ORB_ID(tarim_modulu),
&tarimNavStruct);

        tarimNavStruct.timestamp = hrt_absolute_time();
        tarimNavStruct.ilac_durum = ilacAktif;

        orb_publish(ORB_ID(tarim_modulu), tarim_modulu_pub, &tarimNavStruct);
}

```

This is the complete code for the tarim_modul_ilaclama function. A struct named **tarim_modulu_s** has been created, containing the timestamp and ilac_durum parameters defined in the message block. This struct is used as tarimNavStruct within the function.

After the struct is created, its parameters are first **zeroed out** to prevent them from holding random values. Subsequently, the struct to be published is designated as a **'publish' structure for uORB**. The timestamp, which tracks the elapsed time since the function started, and the ilacAktif parameter, indicating whether the device is spraying chemicals within the polygon boundaries, are then assigned to the struct's parameters. Finally, the struct is **published on the 'tarim_modulu' channel**.

You can verify that a uORB channel has been created and is publishing by using the **listener <module_name>** command syntax in the autopilot console.



```
pxh> listener tarim_mINFO  [tone_alarm] home set
pxh> listener tarim_moINFO  [tone_alarm] notify negative
pxh> listener tarim_modulu

TOPIC: tarim_modulu
 tarim_modulu_s
        timestamp: 8436000  (0.036000 seconds ago)
        ilac_durum: 1
pxh>
```

As seen, the timestamp and ilac_durum parameters are being published under the "tarim_modulu" topic.

3.3 uORB Subscription and Communication with Ground Station via MAVLink

3.3.1 Creating a MAVLink Message

To create a new MAVLink message, you need to edit the **common.xml** file located in the mavlink/include/mavlink/v2.0/message_definitions/ directory within the PX4 autopilot software. common.xml is the file that contains the definitions/rules for MAVLink messages.

```
-- mavlink/include/mavlink/v2.0/message_definitions/common.xml
……
<messages>
……
……
  <message id="15000" name="TARIM_MODULU">
     <description>Z4 TARIM MODULU MESAJI</description>
            <field type="uint64_t" name="time_usec" units="us">Timestamp (UNIX Epoch time or time since system boot). The receiving end can infer timestamp format (since 1.1.1970 or since system boot) by checking for the magnitude of the number.
            </field>
            <field type="uint16_t" name="ilaclama_durumu">Hava aracının ilaçlama durumu bilgisini verir. 1 = İlaçlama Açık, 2 = İlaçlama Kapalı
            </field>
  </message>
……
……
</messages>
……
```

MAVLink messages defined in common.xml must have a **unique message ID**. These ID numbers can vary based on MAVLink versions. MAVLink v1.x uses 1 byte (8-bit), allowing message IDs from 0-255. MAVLink v2.x uses 4 bytes (32-bit), with 3 bytes available for message IDs. We are using **MAVLink v2.x**, and our message ID is **15000**.

Based on the message definition we've made in common.xml, we use the **mavgenerate.py** program, downloaded from https://github.com/mavlink/mavlink, to generate the necessary communication definitions and functions. Afterwards, we copy the output folder from the program to the mavlink/include/mavlink/v2.0/ directory in our autopilot software.

3.3.1 uORB Subscription and Sending MAVLink Messages

Now that we've created the MAVLink message, we need to set up the system that will receive the data we publish on the uORB channel and send this information to the ground control station. In the src/modules/mavlink/streams/ directory, I'm creating the **TARIM_MODULU.hpp header file**, which will contain the definitions of the functions responsible for subscription and message sending. I'm leveraging other existing modules that are already functional and used for MAVLink messaging to create this file.

TARIM_MODULU.hpp ;

```
/*          /src/modules/mavlink/streams/TARIM_MODULU.hpp    */
#ifndef TARIM_ILAC_MODUL_HPP
#define TARIM_ILAC_MODUL_HPP

#include <uORB/topics/tarim_modulu.h>

class MavlinkStreamTarimModulu : public MavlinkStream
{
public:
    static MavlinkStream *new_instance(Mavlink *mavlink)
    {
        return new MavlinkStreamTarimModulu(mavlink);
}

    static constexpr const char *get_name_static
    {
        return "TARIM_MODULU";
    }
    static constexpr uint16_t get_id_static()
    {
        return MAVLINK_MSG_ID_TARIM_MODULU;
    }
    const char *get_name() const override
    {
        return get_name_static();
    }
    uint16_t get_id() override
    {
        return get_id_static();
    }
    unsigned get_size() override
    {
        return _tarim_modulu_sub.advertised() ? (MAVLINK_MSG_ID_TARIM_MODULU_LEN +
MAVLINK_NUM_NON_PAYLOAD_BYTES) : 0;
    }

private:
    explicit MavlinkStreamTarimModulu(Mavlink *mavlink) : MavlinkStream(mavlink) {}

    uORB::Subscription _tarim_modulu_sub
    {
        ORB_ID(tarim_modulu), 0
    };
    bool send() override
    {
        tarim_modulu_nav_s tarimNavStruct;

        if (_tarim_modulu_sub.update(&tarimNavStruct))
        {
            mavlink_tarim_modulu_t msg_tarim_modul{};

            msg_tarim_modul.time_usec = tarimNavStruct.timestamp;
            msg_tarim_modul.ilaclama_durumu = tarimNavStruct.ilacDurum;

            mavlink_msg_tarim_modulu_send_struct(_mavlink->get_channel(),
&msg_tarim_modul);
55.
56.                 return true;
57.             }
58.             return false;
59.         }
60. };
61. #endif
62.
```

- Lines 2 and 3, which contain **#ifndef** and **#define**, are necessary for the mavlink_messages.cpp block that manages MAVLink messages.
- The **#include <uORB/topics/tarim_modulu.h>** line adds the uORB channel definition file.
- In the **Public** block of the MavlinkStreamTarimModulu class, tagging is performed to determine which functions or blocks will call the functions, definitions, and their parameters that make up the message content when the message is sent.
- In the **Private** block, we subscribe to the uORB message we created. Then, a struct (mavlink_tarim_modulu_t) is created to hold the content of the MAVLink message. The values of the tarim_modulu_nav_s struct parameters, whose content is sent in the uORB message, are assigned to the mavlink_tarim_modulu_t parameters, and the message is packaged for sending via MAVLink.

Our message is now tagged and packaged for sending. We perform the sending operation using the mavlink_messages.cpp and mavlink_main.cpp blocks located in the /src/modules/mavlink/ directory.

mavlink_messages.cpp ;

```
……
#include <uORB/topics/tarim_modulu.h>
……
#include "streams/TARIM_MODULU.hpp"
……
……
#if defined(TARIM_ILAC_MODUL_HPP)
        create_stream_list_item<MavlinkStreamTarimModulu>()
#endif // TARIM_ILAC_MODUL_HPP
……
……

```

mavlink_main.cpp ;

```
void
Mavlink::send_protocol_version()
{
……
switch (_mode) {
        case MAVLINK_MODE_NORMAL:
……
……
configure_stream_local("TARIM_MODULU", 1.0f);// configure_stream_local(<message_def_name>, HZ cinsinden gönderme hızı);
……

```

4 CONCLUSION

Z-Sistem Havacılık ve Bilişim San. ve Tic. A.Ş. is developing an **agricultural spraying copter**. To ensure the device performs its mission as expected, tasks will be handled by the **agricultural module** integrated into the PX4 software. The flight control software for this device is **PX4 autopilot-based**, and the ground control station is **QGroundControl-based**, with the overall autopilot software named **Z4**. This report details how the agricultural module within the Z4 software was created and outlines key considerations for developing a new module in PX4 autopilot software.

The **TAR DRONE 1 agricultural spraying device** utilizes PX4-based Z4 autopilot software. To enable its functionality, additional capabilities needed to be added to the PX4 software. This was achieved by creating a new module within PX4, laying the groundwork for the agricultural module software. The initial version of this agricultural module can spray chemicals within defined **geographic boundaries** while the device is on its route, using boundary and flight path information from the ground control station. It also has the ability to stop spraying if these geographic boundaries are violated.

This report specifically covers the foundational coding work for the **'AGRICULTURAL MODULE' software**, which is part of the TAR DRONE 1 agricultural spraying copter and its PX4-based Z4 autopilot software, developed by Z-Sistem Havacılık ve Bilişim San. ve Tic. A.Ş. It elaborates on the following tasks undertaken for adding a new module to the PX4 autopilot software:

- **Creating a message block/file** for a uORB channel and defining necessary parameters.
- **Writing the required functions** for the device to perform agricultural module functions and **publishing their results as messages** over uORB.
- **Extracting necessary parameters** from the messages published via the uORB channel and **sending and utilizing them through MAVLink** to the ground control station.

These areas have been addressed and thoroughly examined in this report.

5 KAYNAKÇA

- https://nxp.gitbook.io/hovergames/developerguide/px4-tutorial-example-code/hg-px4-example-lab-1
- https://nxp.gitbook.io/hovergames/developerguide/px4-tutorial-example-code/hg-px4-example-lab-2
- https://nxp.gitbook.io/hovergames/developerguide/px4-tutorial-example-code/hg-px4-example-lab-3
- https://nxp.gitbook.io/hovergames/developerguide/px4-tutorial-example-code/hg-px4-example-lab-4
- https://nxp.gitbook.io/hovergames/developerguide/px4-tutorial-example-code/hg-px4-example-lab-5
- https://www.hackster.io/mdobrea/communication-through-custom-uorb-and-mavlink-messages-269ebf
- https://github.com/mavlink/mavlink
- https://uav-lab.org/2016/08/02/px4-research-log4-a-first-look-at-px4-architecture/
- https://px4.io/px4-uorb-explained-part-1/
- https://px4.io/px4-uorb-explained-part-2/

https://px4.io/px4-uorb-explained-part-3-the-deep-stuff/