Software development for an underwater ROV using PIXHAWK technology and AADL modeling

This work is funded by the ANR «Investissements d'avenir» number ANR-19-GURE-0001 in the framework of the ERASMUS+ SEA UE consortium

Brest, 14.06.2023.

Mentors:
Frank Singhoff
Tonko Kovačević

Written by:
Jure Antunović

Table of Contents

Introduction

The University of Split has built an underwater remote operated vehicle (ROV), which is used to explore the sea. With the objective of advancing their research endeavors, the university's team has set out to develop an upgraded version of the ROV, in collaboration with UBO.

The PIXHAWK autopilot is used as a replacement for the conventional Arduino system.
Data-reading sensors are the integral part of the *autopilot* component of the ROV, working in harmony with the engines. The acquired data is then transmitted to an ARK computer, which serves as the central processing unit, using Ethernet connection.
ROV is controlled via an *external control unit*, consisting of a laptop and a controller.

To create models of the various sensors (and other components), Architecture Analysis and Design Language (AADL) is used.
Source code of the models is written in C.

1.) PROJECT BACKGROUND

1.1.) AADL

Architecture Analysis and Design Language is a modeling language that facilitates the design and analysis of complex software-intensive systems, such as the upgraded underwater ROV developed by the University of Split. AADL enables the representation and specification of the system's architecture, including its components, connections, and behavior, thereby aiding in the seamless integration of the PIXHAWK autopilot, sensors, and control units. By employing AADL, the research team can effectively capture the system's structure and functionality, ensuring a robust and efficient software design for the enhanced ROV project.

1.1.1.) Motivation

Reasons why AADL was choosed lie in its compatibility with real-time operating systems, simple analysis and testing of the entire system and the possibility of generating C source code.

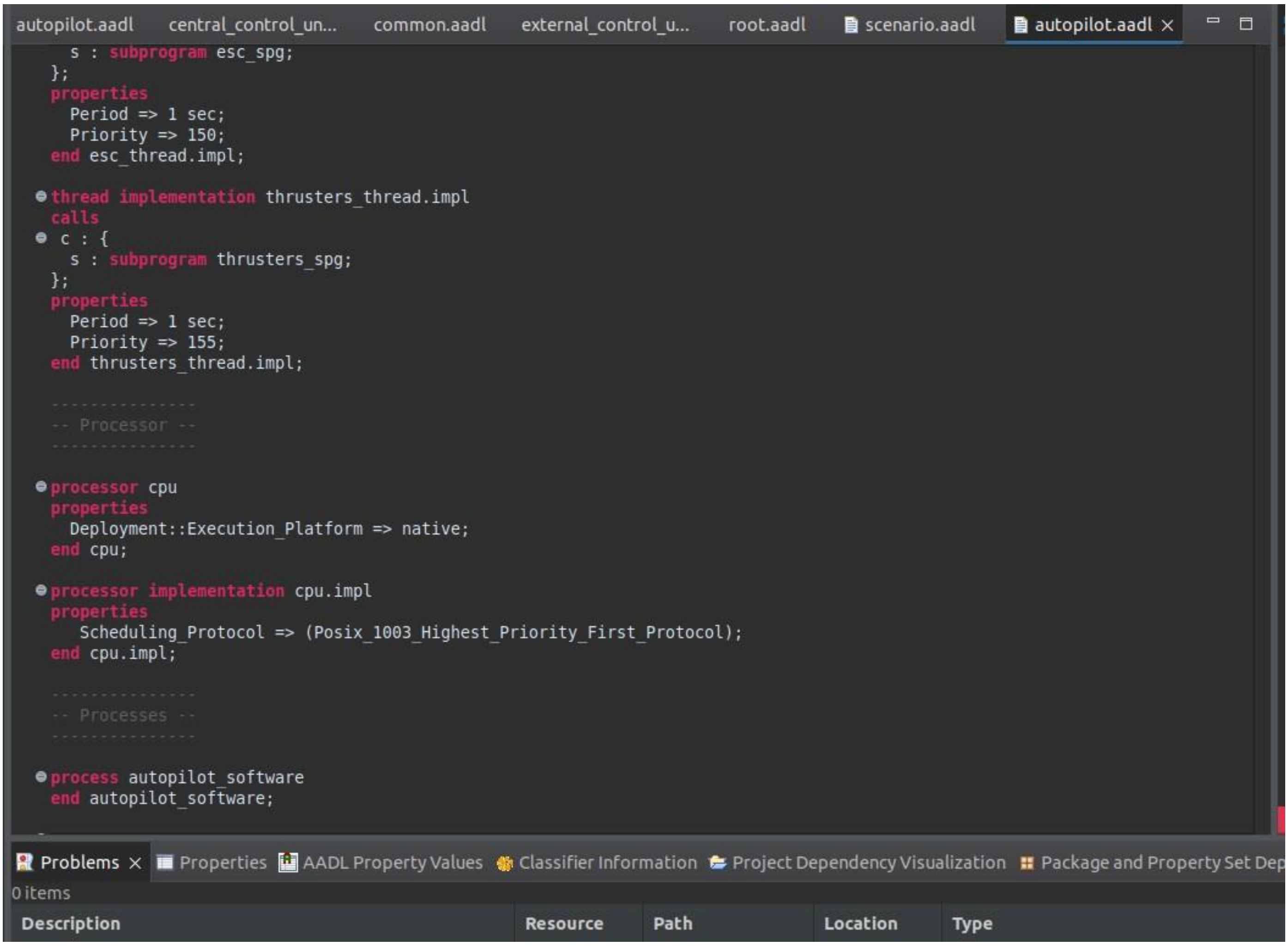
```
data implementation thrusters_data.impl
end thrusters_data.impl;

device thrusters_device
features
  thrusters_input: in data port thrusters_data;
  thrusters_output: out data port thrusters_data;
end thrusters_device;


-----------------
-- Subprograms --
-----------------

subprogram temperature_spg
properties
  source_language => (C);
  source_name     => "temperature_spg";
  source_text     => ("temperature.c");
end temperature_spg;

subprogram magnetometer_spg
properties
  source_language => (C);
  source_name     => "magnetometer_spg";
  source_text     => ("magnetometer.c");
end magnetometer_spg;
```

Picture 1, *a snippet of code from the autopilot.aadl file*

1.1.2.) OSATE

Open Source AADL Tool Environment (AADL) is a tool developer specifically for the AADL.

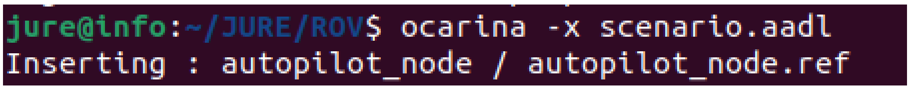
It is used for modeling, ananalysing and generating code for complex software systems. With OSATE, developers can effectively create software architectures.



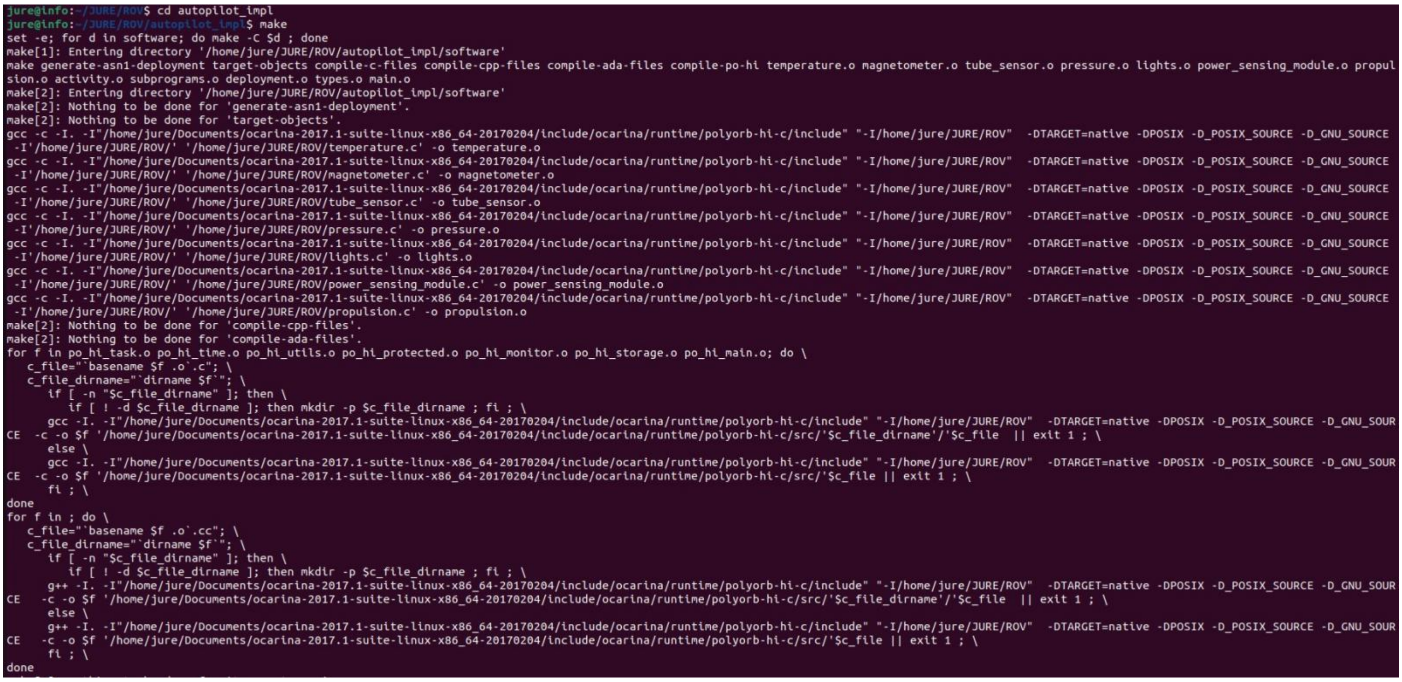
Picture 2, *OSATE interface for the autopilot.aadl file*

1.1.3.) Ocarina

Ocarina is a code generator tool that supports the generation of code from AADL models. It is an accurate representation of a “bridge” between high-level systems (such as this projects’ AADL architecture) and the low-level implementation of the software (coding).



Picture 3, *example of starting Ocarina for the autopilot.aadl file*



Picture 4, *make of the Ocarina code*

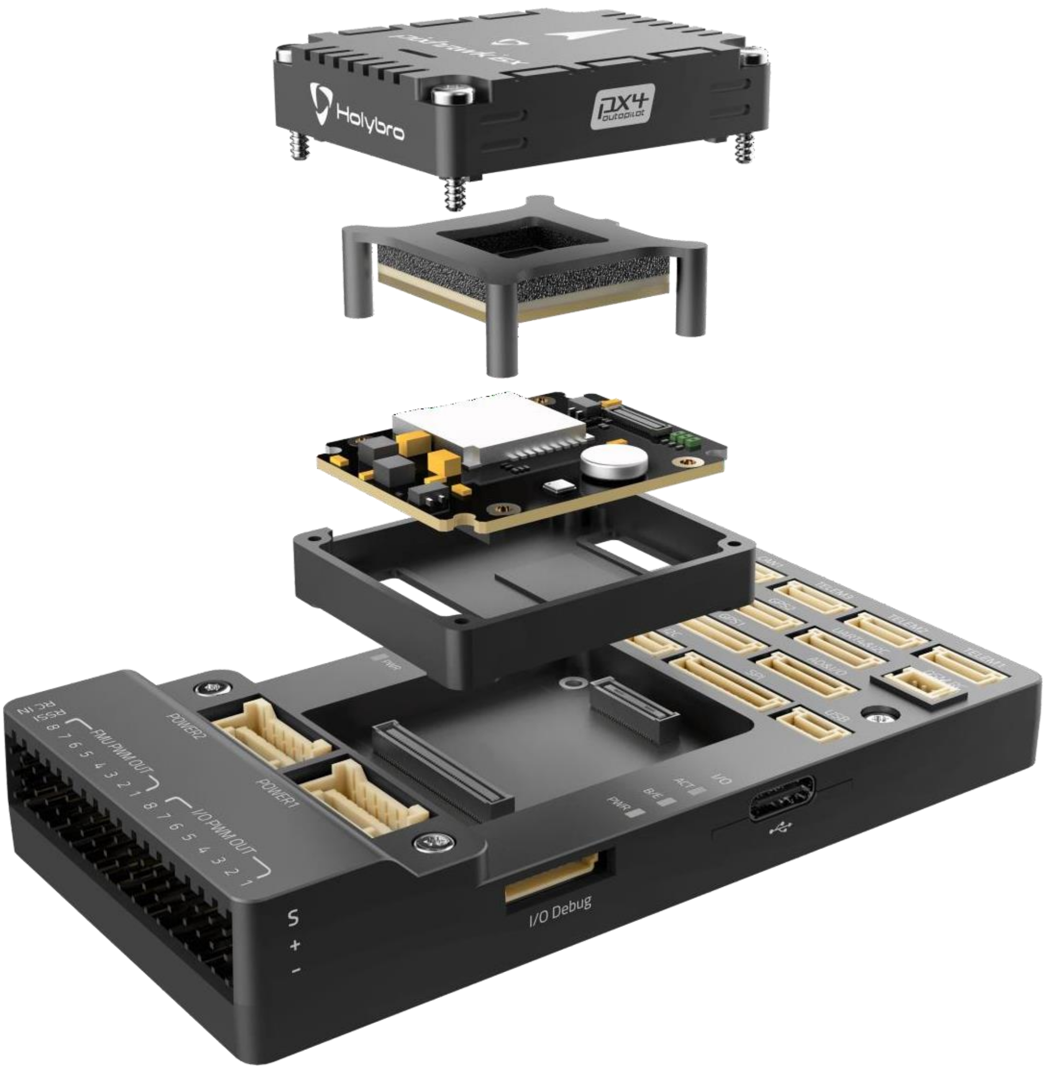
1.2.) PIXHAWK

This is an open-source autopilot system designed for unmanned aerial vehicles (UAVs); however, it can be implemented in underwater ones as well.

It provides hardware and software platform that gives its user efficient control and navigation for autonomous vehicles.

Regarding its connection to this project, PIXHAWK can be integrated into the system architecture as a component or module. AADL can define the interfaces, connections, and behavior of the PIXHAWK autopilot within the larger system design, ensuring seamless integration and coordination between the ROV's software, sensors, and the autonomous capabilities provided by the PIXHAWK autopilot.

For the purposes of this project, PIXHAWK 6X was chosen.



Picture 5, *Holybro PIXHAWK 6X*

1.3.) ARK

Ark computer is a high-performance computing system that is used (almost exclusively) with demanding applications (such as underwater ROVs). It serves as the central processing unit for the underwater ROV project, playing a vital role in data acquisition, processing, and control.

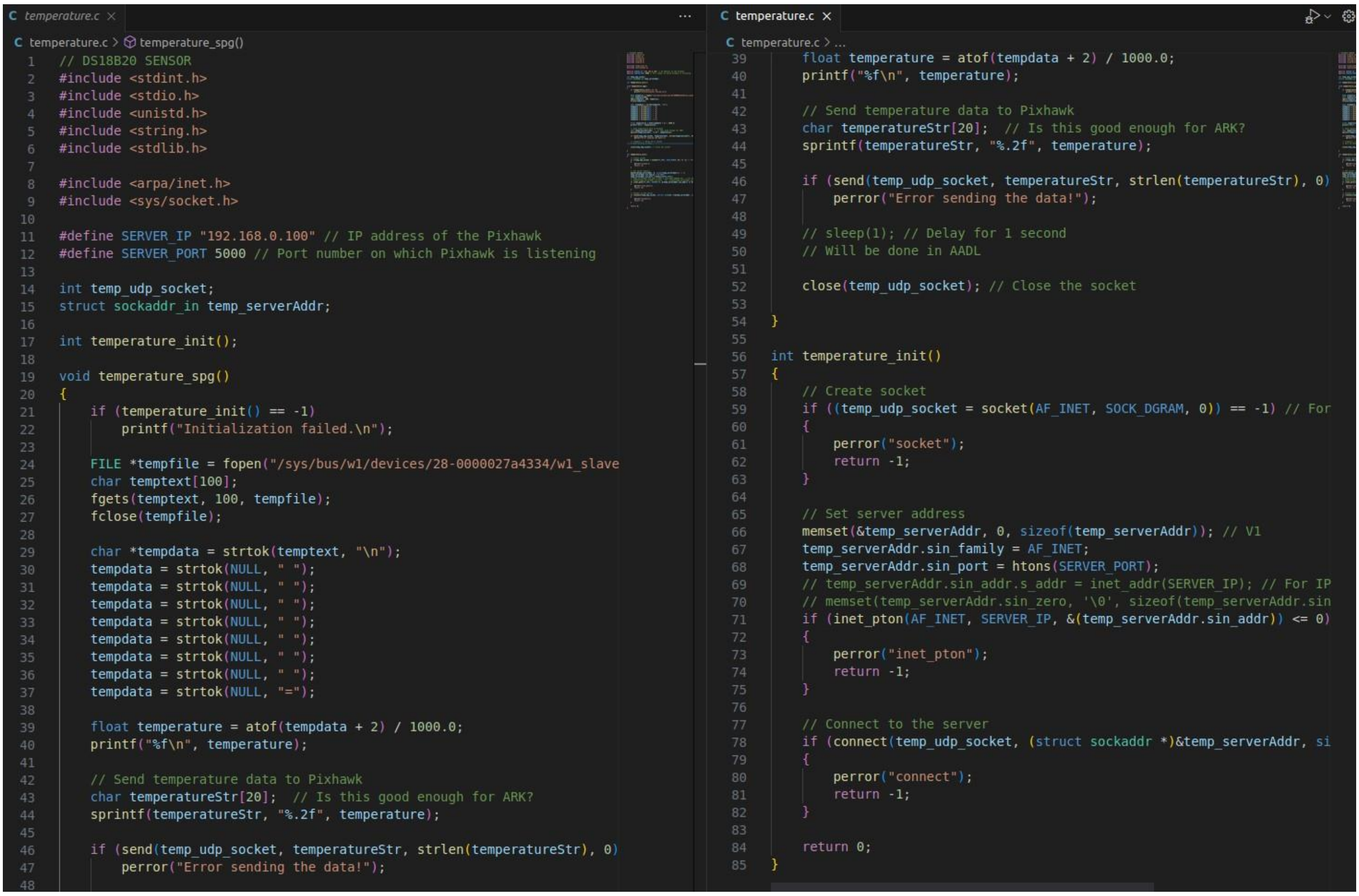
In the context of the ROV project, the ARK-1551-S6A1 computer connects to the PIXHAWK autopilot system to receive and process data from various sensors onboard the ROV. It acts as the intermediary between the autopilot and the rest of the system, facilitating the communication and integration of sensor data with other components (such as cameras and GPS) to enable efficient control and operation of the ROV.

Picture 6, *ARK-1551-S6A1 computer*

1.4.) Sensors and modules

Majority of work done here is related to the various sensors and modules.

The code for these sensors was written in C, using VS Code IDE.



Picture 7, *example of temperature sensor code*

Below are described the above-mentioned sensors and modules, and their code designs are found in the next chapter.

1.4.1.) Temperature

For the temperature sensor, DS18B20 sensor was chosen. It is used in various applications, and in the context of the PIXHAWK autopilot system, it is integrated to provide accurate temperature reading, which is utilised for environmental monitoring and control within the ROV system.

Picture 8, *DS18B20 sensor*

1.4.2.) Tube

The DHT22 sensor, commonly known as the tube sensor, is utilized within the PIXHAWK project to measure both temperature and humidity. Its integration allows for precise environmental monitoring, providing crucial data for optimizing the ROV's performance and ensuring safe operations in varying underwater conditions.

Picture 9, *DHT22 Sensor*

1.4.3.) Pressure

The pressure sensor SKU237545 is incorporated into the PIXHAWK project to accurately measure underwater pressure. By integrating this sensor, the ROV can gather essential data on depth and pressure changes, enabling precise depth control, depth-based operations, and environmental monitoring during underwater exploration.

Picture 10, *pressure sensor*

1.4.4.) Power sensor

The Blue Robotics Power Sensor Module is integrated into the PIXHAWK project to monitor power usage and provide real-time data on the ROV's energy consumption. This information is crucial for efficient power management, allowing for better control over the ROV's energy resources and ensuring optimal operation during extended missions.

Picture 11, *PSM*

1.4.5.) Magnetometer

The HMC5883L sensor, also known as a magnetometer, is employed within the PIXHAWK project to measure magnetic field strength and orientation. By integrating this sensor, the ROV can obtain accurate heading information, aiding in navigation, orientation, and alignment tasks underwater.

Picture 12, *HMC5883L sensor*

1.4.6.) Thrusters

The T200 thrusters play a vital role in the PIXHAWK project as they provide propulsion and maneuverability to the ROV. These high-performance thrusters are integrated into the system and controlled by the PIXHAWK autopilot, allowing for precise and responsive navigation underwater.

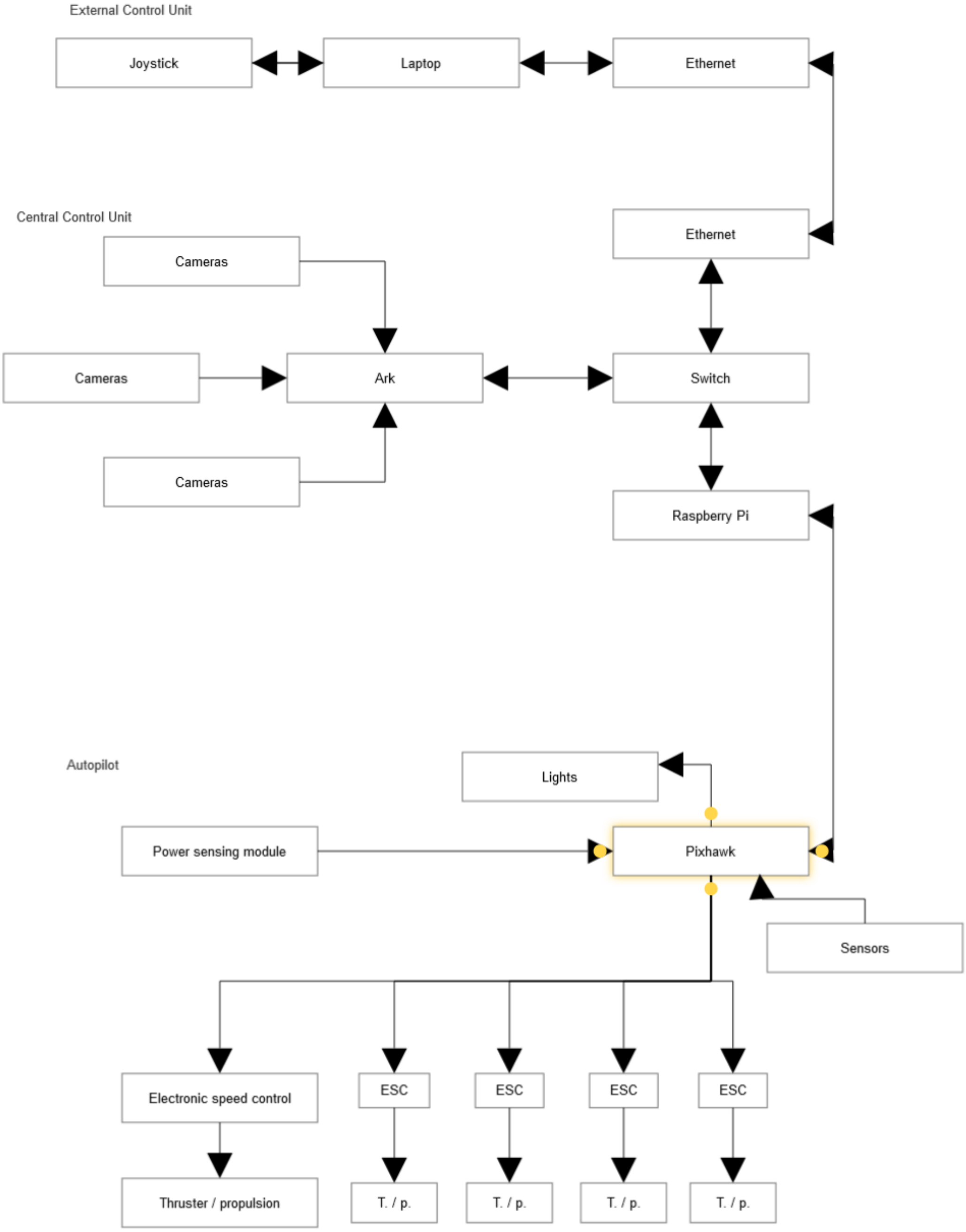
Picture 13, *thruster*

2.) PROJECT REPORT

As mentioned before, software for the ROV was done using AADL and C languages.

2.1.) AADL model

The ROV model was agreed upon with the mentors.
According to the hardware configurations received, the ROV is comprised of 3 main parts: an external control unit, a central control unit and an autopilot.



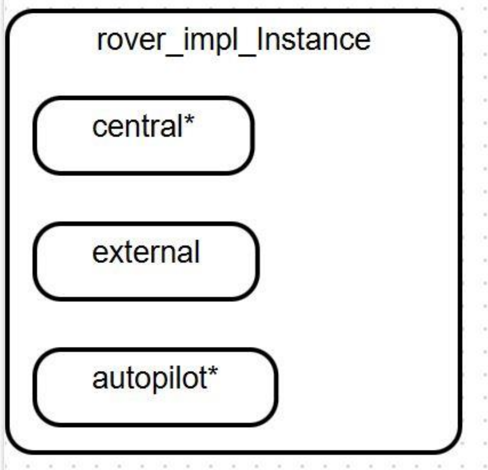
Picture 14, *ROV model*

The 3 components are declared in the “root.aadl” file, as shown on the picture below.

```
package root
public
   with autopilot;
   with external_control_unit;
   with central_control_unit;

------------
-- System --
------------

system rover
end rover;

system implementation rover.impl
subcomponents
   autopilot : system  autopilot::autopilot.impl;
   external  : system  external_control_unit::external.impl;
   central   : system  central_control_unit::central.impl;
end rover.impl;

end root;
```

Picture 15, *root.aadl*

The implemented instance of the rover can be seen on the picture below (generated from OSATE):



Picture 16, *ROV core instances*

While running Ocarina, “scenario.aadl” file is used (custom-modified for each component file differently).

```
system rover_root
properties
  Ocarina_Config::Timeout_Property     => 4000ms;
  Ocarina_Config::Referencial_Files    =>
    ("central_node", "central_node.ref");
  Ocarina_Config::AADL_Files           =>
    ("central_control_unit.aadl","common.aadl");
  Ocarina_Config::Generator            => polyorb_hi_c;
  Ocarina_Config::Needed_Property_Sets =>
    (value (Ocarina_Config::Data_Model),
     value (Ocarina_Config::Deployment),
     value (Ocarina_Config::Cheddar_Properties));
  Ocarina_Config::AADL_Version         => AADLv2;
end rover_root;

system implementation rover_root.Impl
end  rover_root.Impl;
```

Picture 17, *scenario.aadl for the central_control_unit*

2.1.1) Autopilot (PIXHAWK)

Majority of work done here was in regard to the various sensors and modules, which read data. After reading data, sensors send it to the ARK computer via PIXHAWK, using Ethernet communication protocol.

```
autopilot.aadl > ...
package autopilot
public
  with deployment;
  with common;

------------------------
-- Hardware components --
------------------------

data temperature_data
end temperature_data;

data implementation temperature_data.impl
end temperature_data.impl;

device temperature_device
features
  temperature_input: in data port temperature_data;
  temperature_output: out data port temperature_data;
end temperature_device;

data magnetometer_data
end magnetometer_data;

data implementation magnetometer_data.impl
end magnetometer_data.impl;

device magnetometer_device
features
  magnetometer_input: in data port magnetometer_data;
  magnetometer_output: out data port magnetometer_data;
end magnetometer_device;

data tube_sensor_data
end tube_sensor_data;

data implementation tube_sensor_data.impl
end tube_sensor_data.impl;

device tube_sensor_device
features
  tube_input: in data port tube_sensor_data;
  tube_output: out data port tube_sensor_data;
end tube_sensor_device;

data pressure_data
end pressure_data;
```

```
autopilot.aadl > {} autopilot

data implementation pressure_data.impl
end pressure_data.impl;

device pressure_device
features
  pressure_input: in data port pressure_data;
  pressure_output: out data port pressure_data;
end pressure_device;

data lights_data
end lights_data;

data implementation lights_data.impl
end lights_data.impl;

device lights_device
features
  lights_input: in data port lights_data;
  lights_output: out data port lights_data;
end lights_device;

data power_sensing_module_data
end power_sensing_module_data;

data implementation power_sensing_module_data.impl
end power_sensing_module_data.impl;

device power_sensing_module_device
features
  power_input: in data port power_sensing_module_data;
  power_output: out data port power_sensing_module_data;
end power_sensing_module_device;

data esc_data
end esc_data;

data implementation esc_data.impl
end esc_data.impl;

device esc_device
features
  esc_input: in data port esc_data;
  esc_output: out data port esc_data;
end esc_device;

data thrusters_data
end thrusters_data;
```

Picture 18, *autopilot.aadl* ¼

First of all, there are package declarations. This section declares a package named "autopilot" and specifies that it is a public package. It also indicates that the package depends on two other packages, namely "deployment" and "common" (*common* contains definition for periodic and aperiodic properties of threads).

Throughout the code, there are various data types defined, such as "temperature_data," "magnetometer_data," "tube_sensor_data," "pressure_data," "lights_data," "power_sensing_module_data," "esc_data," and "thrusters_data." These data types represent the kind of data that can be transmitted between different hardware components.

For example,

```
data temperature_data
end temperature_data;
```

is the part where a data type named "temperature_data" is defined. The "data" keyword indicates that it is a data type, and the "end temperature_data;" statement marks the end of the data type definition. Additionally, there are corresponding "implementation" sections for each data type, such as "temperature_data.impl," "magnetometer_data.impl," etc. These implementation sections can be used to specify the internal structure or behavior of the data types if needed.

Furthermore, various hardware components in the system are defined, such as temperature sensors, magnetometers, tube sensors, pressure devices, lights devices, power sensing modules, ESCs (Electronic Speed Controllers), and thrusters. Each component is defined using the "device" keyword.

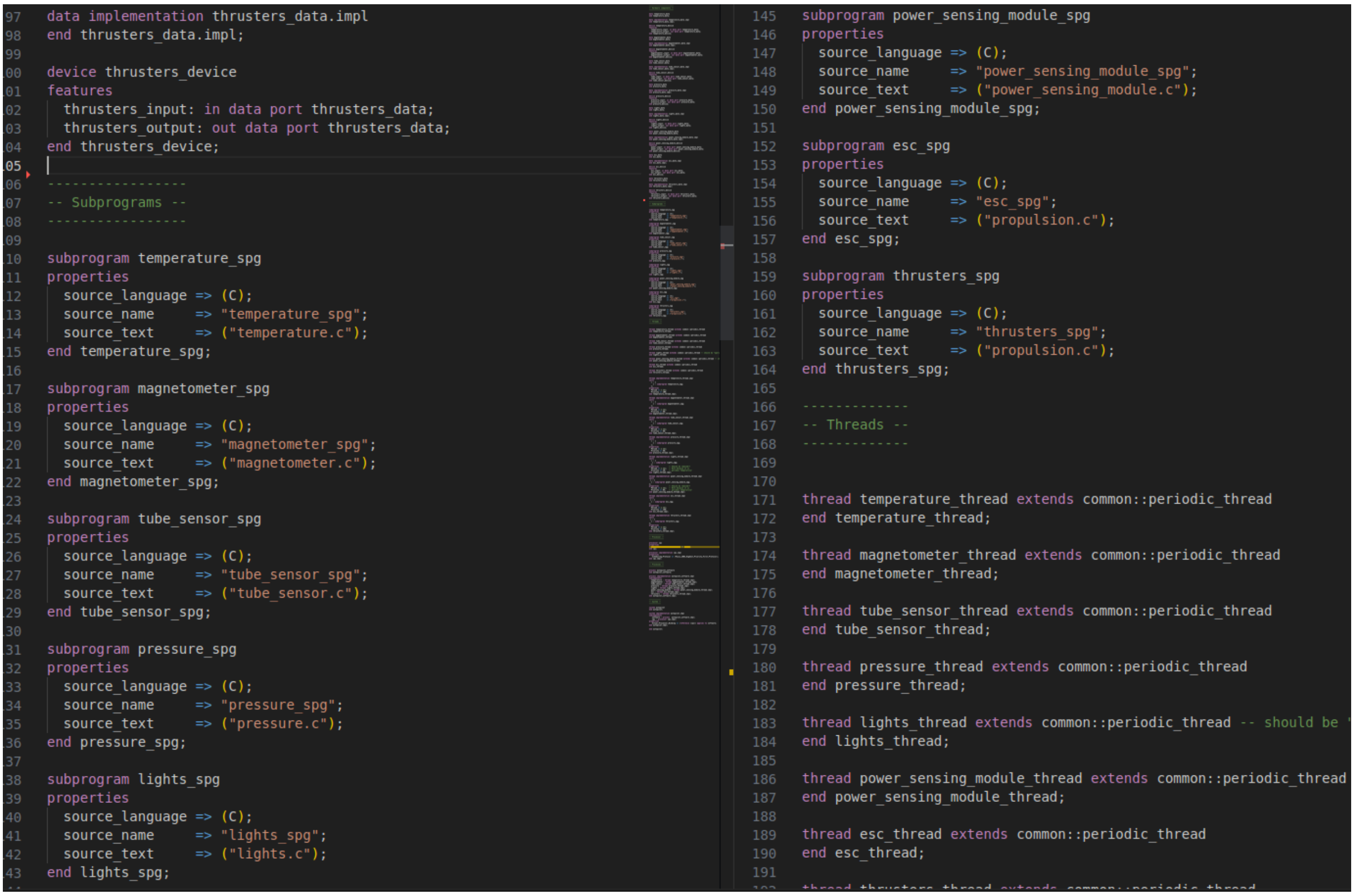
Taking the "temperature_device" as an example:

```
device temperature_device
```

```
features
    temperature_input: in data port temperature_data;
    temperature_output: out data port temperature_data;
end temperature_device;
```

"temperature_device" represents a device in the system. It has two features: "temperature_input" and "temperature_output." These features are defined as data ports of type "temperature_data." A data port is a communication interface that allows the exchange of data between components.

Similarly, other hardware components like "magnetometer_device," "tube_sensor_device," "pressure_device," "lights_device," "power_sensing_module_device," "esc_device," and "thrusters_data" are defined in a similar manner.



```
data implementation thrusters_data.impl
end thrusters_data.impl;

device thrusters_device
features
  thrusters_input: in data port thrusters_data;
  thrusters_output: out data port thrusters_data;
end thrusters_device;

----------------
-- Subprograms --
----------------

subprogram temperature_spg
properties
  source_language => (C);
  source_name     => "temperature_spg";
  source_text     => ("temperature.c");
end temperature_spg;

subprogram magnetometer_spg
properties
  source_language => (C);
  source_name     => "magnetometer_spg";
  source_text     => ("magnetometer.c");
end magnetometer_spg;

subprogram tube_sensor_spg
properties
  source_language => (C);
  source_name     => "tube_sensor_spg";
  source_text     => ("tube_sensor.c");
end tube_sensor_spg;

subprogram pressure_spg
properties
  source_language => (C);
  source_name     => "pressure_spg";
  source_text     => ("pressure.c");
end pressure_spg;

subprogram lights_spg
properties
  source_language => (C);
  source_name     => "lights_spg";
  source_text     => ("lights.c");
end lights_spg;
```

```
subprogram power_sensing_module_spg
properties
  source_language => (C);
  source_name     => "power_sensing_module_spg";
  source_text     => ("power_sensing_module.c");
end power_sensing_module_spg;

subprogram esc_spg
properties
  source_language => (C);
  source_name     => "esc_spg";
  source_text     => ("propulsion.c");
end esc_spg;

subprogram thrusters_spg
properties
  source_language => (C);
  source_name     => "thrusters_spg";
  source_text     => ("propulsion.c");
end thrusters_spg;

-------------
-- Threads --
-------------

thread temperature_thread extends common::periodic_thread
end temperature_thread;

thread magnetometer_thread extends common::periodic_thread
end magnetometer_thread;

thread tube_sensor_thread extends common::periodic_thread
end tube_sensor_thread;

thread pressure_thread extends common::periodic_thread
end pressure_thread;

thread lights_thread extends common::periodic_thread -- should be '
end lights_thread;

thread power_sensing_module_thread extends common::periodic_thread
end power_sensing_module_thread;

thread esc_thread extends common::periodic_thread
end esc_thread;
```

Picture 19, *autopilot.aadl 2/4*

Here, there are various subprograms defined, each of them associated with a specific hardware component.
Here's an example for the "temperature_spg" subprogram:

```
subprogram temperature_spg
```

```
properties
```

```
    source_language => (C);
    source_name => "temperature_spg";
    source_text => ("temperature.c");
end temperature_spg;
```

Each subprogram is identified by a unique name, such as "temperature_spg," "magnetometer_spg," etc. The properties section provides additional information about the subprogram, such as the source language (C in this case) and the name and location of the source code file.
In the example, the source code file for the "temperature_spg" subprogram is specified as "temperature.c."

The code also defines several threads, each associated with a specific functionality or task in the system. The threads are derived from a base thread type called "common::periodic_thread," which indicates that these threads have a periodic behavior. Here's an example for the "temperature_thread":

```
thread temperature_thread extends common::periodic_thread

end temperature_thread;
```

Each thread is identified by a unique name, such as "temperature_thread," "magnetometer_thread," etc. The "extends" keyword indicates that the thread inherits properties and behavior from the "common::periodic_thread" type.

In the provided example, the "lights_thread" and "power_sensing_module_thread" should ideally extend "common::aperiodic_thread" instead of "common::periodic_thread." However, there seems to be an issue with the Ocarina tool, which returns an error when using the correct "aperiodic_thread" type.

```
thread thrusters_thread extends common::periodic_thread
end thrusters_thread;


thread implementation temperature_thread.impl
calls
  c : {
    s : subprogram temperature_spg;
  };
properties
  Period => 1 sec;
  Priority => 100;
end temperature_thread.impl;

thread implementation magnetometer_thread.impl
calls
  c : {
    s : subprogram magnetometer_spg;
  };
properties
  Period => 1 sec;
  Priority => 50;
end magnetometer_thread.impl;

thread implementation tube_sensor_thread.impl
calls
  c : {
    s : subprogram tube_sensor_spg;
  };
properties
  Period => 1 sec;
  Priority => 50;
end tube_sensor_thread.impl;

thread implementation pressure_thread.impl
calls
  c : {
    s : subprogram pressure_spg;
  };
properties
  Period => 1 sec;
  Priority => 40;
end pressure_thread.impl;
```

```

thread implementation lights_thread.impl
calls
  c : {
    s : subprogram lights_spg;
  };
properties          -- Should be removed!?
  Period => 1 sec;  -- Here because it is
  Priority => 10;   -- periodic temporarily!
end lights_thread.impl;

thread implementation power_sensing_module_thread.impl
calls
  c : {
    s : subprogram power_sensing_module_spg;
};
properties          -- Should be removed!?
  Period => 1 sec;  -- Here because it is
  Priority => 85;   -- periodic temporarily!
end power_sensing_module_thread.impl;

thread implementation esc_thread.impl
calls
  c : {
    s : subprogram esc_spg;
};
properties
  Period => 1 sec;
  Priority => 150;
end esc_thread.impl;

thread implementation thrusters_thread.impl
calls
  c : {
    s : subprogram thrusters_spg;
};
properties
  Period => 1 sec;
  Priority => 155;
end thrusters_thread.impl;

--------------
-- Processor --
--------------
```

Picture 20, *autopilot.aadl ¾*

This code snippet specifies the subprograms that are called by each thread and defines properties related to their scheduling and execution.

Here's an example for the "temperature_thread.impl":

```
thread implementation temperature_thread.impl
calls
  c : {
    s : subprogram temperature_spg;
  };
properties
    Period => 1 sec;
    Priority => 100;
end temperature_thread.impl;
```

Each thread implementation is identified by a unique name, such as "temperature_thread.impl," "magnetometer_thread.impl," etc. The "calls" section specifies the subprogram called by the thread implementation. In the example, the "temperature_thread.impl" calls the "temperature_spg" subprogram. The properties section of each thread implementation specifies various attributes and characteristics of the thread. Some common properties include "Period," "Priority," etc.

Here's an example for the "temperature_thread.impl" properties:

```
properties
Period => 1 sec;
Priority => 100;
```

In the provided example, the "Period" property indicates that the thread executes with a period of 1 second, and the "Priority" property sets the priority of the thread.

```

processor cpu
properties
  Deployment::Execution_Platform => native;
end cpu;

processor implementation cpu.impl
properties
    Scheduling_Protocol => (Posix_1003_Highest_Priority_First_Protocol);
end cpu.impl;

---------------
-- Processes --
---------------

process autopilot_software
end autopilot_software;

process implementation autopilot_software.impl
subcomponents
  temperature : thread temperature_thread.impl;
  magnetometer : thread magnetometer_thread.impl;
  tube_sensor : thread tube_sensor_thread.impl;
  pressure : thread pressure_thread.impl;
  lights : thread lights_thread.impl;
  power_sensing_module : thread power_sensing_module_thread.impl;
  esc : thread esc_thread.impl;
  thrusters : thread thrusters_thread.impl;
end autopilot_software.impl;

------------
-- System --
------------

system autopilot
end autopilot;

system implementation autopilot.impl
subcomponents
    software : process  autopilot_software.impl;
    cpu : processor cpu.impl;
properties
    Actual_Processor_Binding => (reference (cpu)) applies to software;
end autopilot.impl;

end autopilot;
```

Picture 21, *autopilot.aadl 4/4*

In the example above, the processor, processes, and system implementation in the AADL syntax are described.

The code defines a processor named "cpu" with the following properties:

```
processor cpu
properties
Deployment::Execution_Platform => native;
end cpu;
```

The processor declaration specifies that it belongs to the native execution platform. The "Deployment::Execution_Platform" property provides information about the target execution platform for the processor.

The following code defines an implementation for the processor, which specifies the scheduling protocol:

```
processor implementation cpu.impl
properties
Scheduling_Protocol => (Posix_1003_Highest_Priority_First_Protocol);
end cpu.impl;
```

The processor implementation, "cpu.impl," sets the scheduling protocol to "Posix_1003_Highest_Priority_First_Protocol." This property defines the scheduling policy to be used by the processor.

Next, a process named "autopilot_software" and its implementation, "autopilot_software.impl" are defined. The implementation consists of multiple thread subcomponents:

```
process autopilot_software
end autopilot_software;

process implementation autopilot_software.impl
subcomponents
temperature : thread temperature_thread.impl;
magnetometer : thread magnetometer_thread.impl;
tube_sensor : thread tube_sensor_thread.impl;
pressure : thread pressure_thread.impl;
lights : thread lights_thread.impl;
power_sensing_module : thread power_sensing_module_thread.impl;
esc : thread esc_thread.impl;
thrusters : thread thrusters_thread.impl;
end autopilot_software.impl;
```

The process implementation, "autopilot_software.impl," contains subcomponents that represent the threads in the autopilot software. Each thread implementation is specified, such as "temperature_thread.impl," "magnetometer_thread.impl," etc.

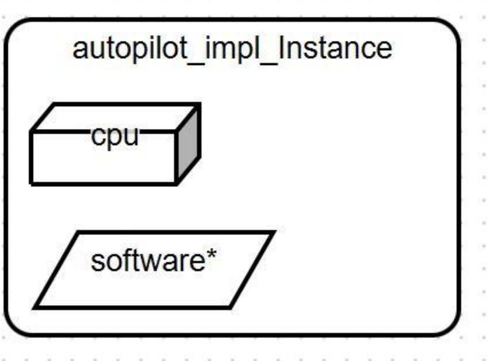
The code defines a system named "autopilot" and its implementation, "autopilot.impl." The implementation consists of two subcomponents: "software" and "cpu." Additionally, a property is applied to bind the software process to the cpu processor:

```
system autopilot
```

```
end autopilot;

system implementation autopilot.impl
subcomponents
software : process autopilot_software.impl;
cpu : processor cpu.impl;
properties
Actual_Processor_Binding => (reference (cpu)) applies to software;
end autopilot.impl;

end autopilot;
```

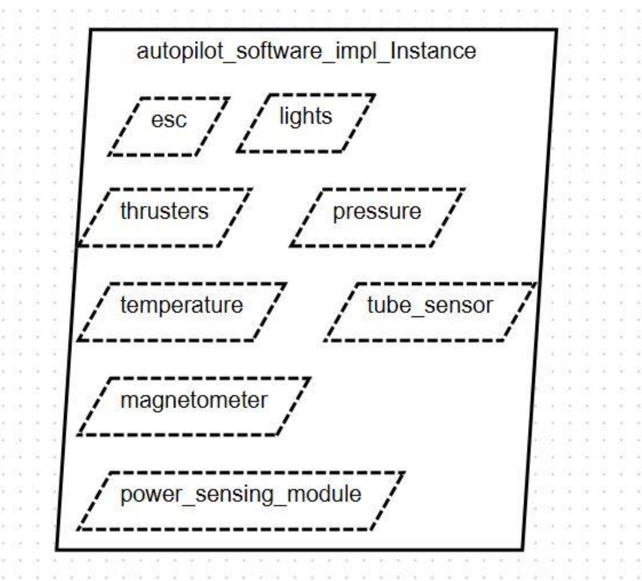
The system implementation, "autopilot.impl," includes the software process and the cpu processor as subcomponents. The "Actual_Processor_Binding" property is used to specify that the software process is bound to the cpu processor.

In the end, the file package is closed.



Picture 22, *implemented instance of the autopilot*

On the picture above, OSATE generated instance of the autopilot module is shown, while on the picture below, all the software modules of the autopilot are presented.



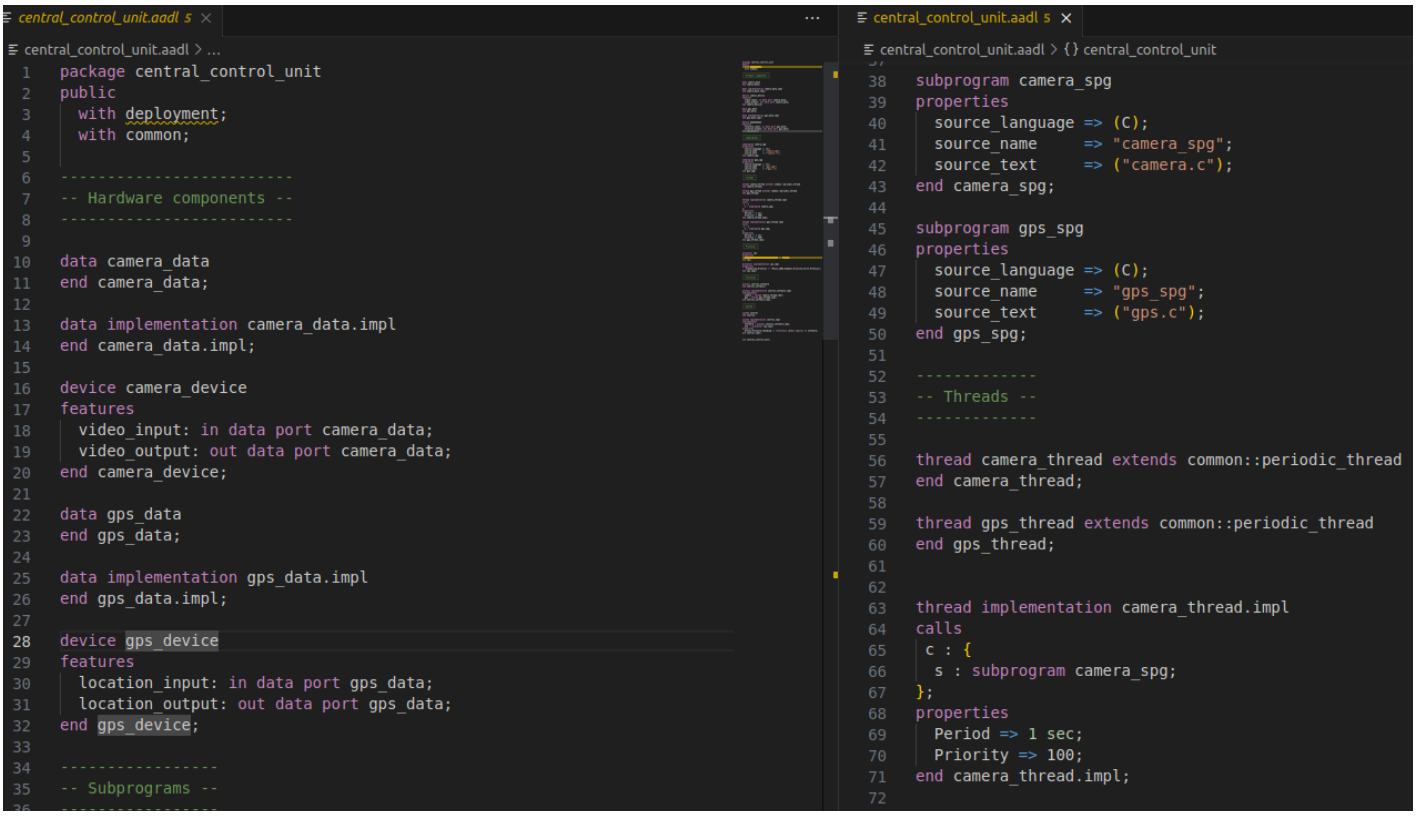
Picture 23, *autopilot modules*

2.1.2.) Central control unit (ARK)

The same principle is present within the central_control_unit.aadl file.

In the pictures below, a similar code is shown; package declaration, declarations and implementations of hardware components, subprograms, threads (of the execution units) and the parts defining the processors, its processes and the system itself.

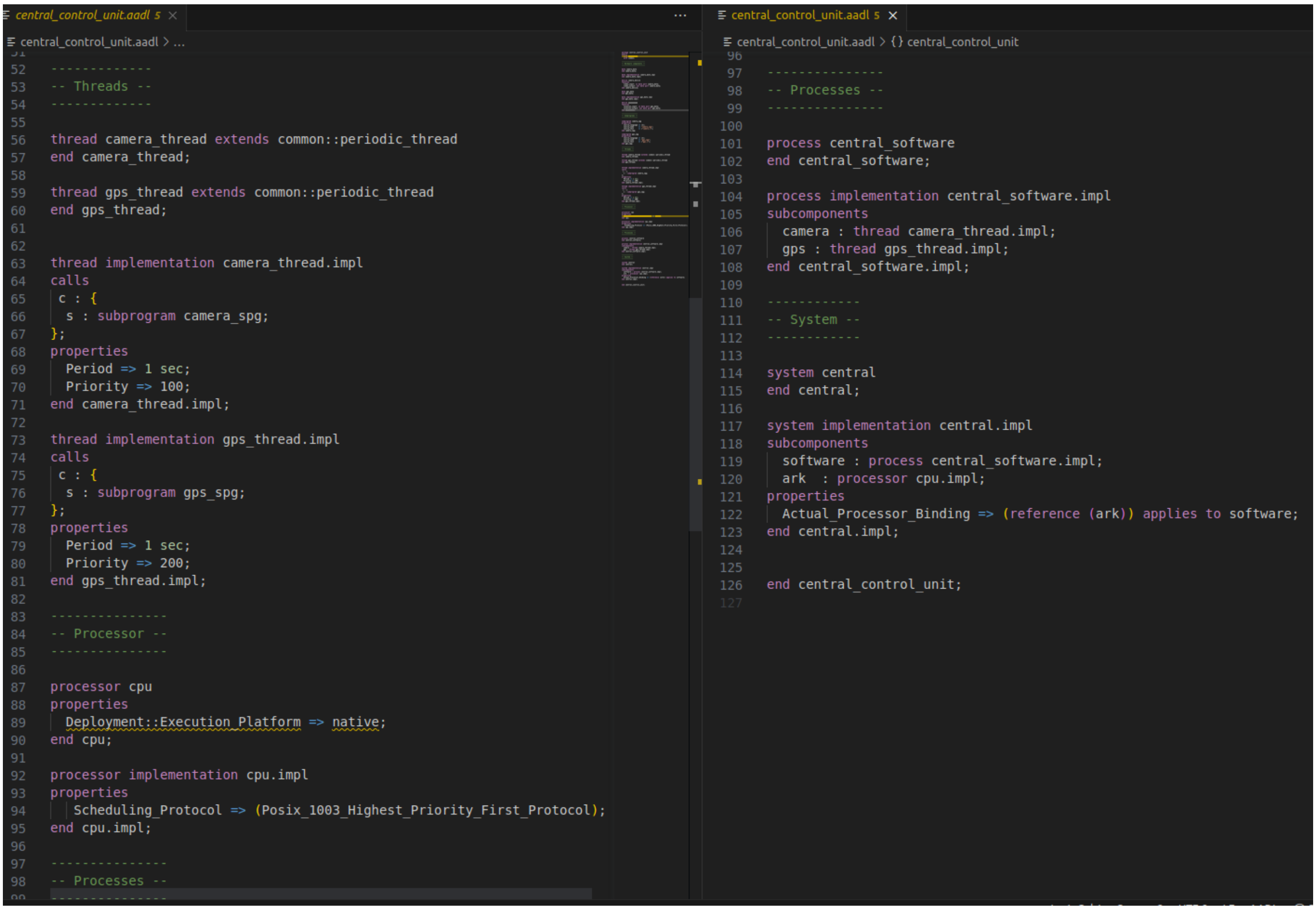
However, in this case, there are only 2 hardware components (connected directly to the ARK computer): camera and gps.



```
package central_control_unit
public
  with deployment;
  with common;

------------------------
-- Hardware components --
------------------------

data camera_data
end camera_data;

data implementation camera_data.impl
end camera_data.impl;

device camera_device
features
  video_input: in data port camera_data;
  video_output: out data port camera_data;
end camera_device;

data gps_data
end gps_data;

data implementation gps_data.impl
end gps_data.impl;

device gps_device
features
  location_input: in data port gps_data;
  location_output: out data port gps_data;
end gps_device;

----------------
-- Subprograms --
```

```
subprogram camera_spg
properties
  source_language => (C);
  source_name     => "camera_spg";
  source_text     => ("camera.c");
end camera_spg;

subprogram gps_spg
properties
  source_language => (C);
  source_name     => "gps_spg";
  source_text     => ("gps.c");
end gps_spg;

------------
-- Threads --
------------

thread camera_thread extends common::periodic_thread
end camera_thread;

thread gps_thread extends common::periodic_thread
end gps_thread;


thread implementation camera_thread.impl
calls
  c : {
   s : subprogram camera_spg;
};
properties
  Period => 1 sec;
  Priority => 100;
end camera_thread.impl;
```

Picture 24, *central_control_unit.aadl 1/2*

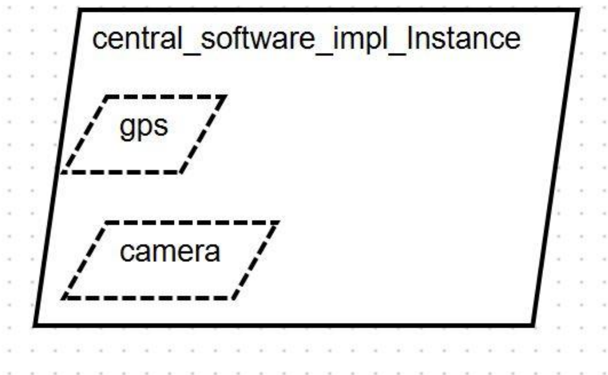


```
------------
-- Threads --
------------

thread camera_thread extends common::periodic_thread
end camera_thread;

thread gps_thread extends common::periodic_thread
end gps_thread;


thread implementation camera_thread.impl
calls
  c : {
    s : subprogram camera_spg;
};
properties
    Period => 1 sec;
    Priority => 100;
end camera_thread.impl;

thread implementation gps_thread.impl
calls
  c : {
    s : subprogram gps_spg;
};
properties
    Period => 1 sec;
    Priority => 200;
end gps_thread.impl;

--------------
-- Processor --
--------------

processor cpu
properties
    Deployment::Execution_Platform => native;
end cpu;

processor implementation cpu.impl
properties
     Scheduling_Protocol => (Posix_1003_Highest_Priority_First_Protocol);
end cpu.impl;

--------------
-- Processes --
```

```
--------------
-- Processes --
--------------

process central_software
end central_software;

process implementation central_software.impl
subcomponents
    camera : thread camera_thread.impl;
    gps : thread gps_thread.impl;
end central_software.impl;

-----------
-- System --
-----------

system central
end central;

system implementation central.impl
subcomponents
    software : process central_software.impl;
    ark  : processor cpu.impl;
properties
    Actual_Processor_Binding => (reference (ark)) applies to software;
end central.impl;


end central_control_unit;
```

Picutre 25, *central_control_unit.aadl 2/2*

In the picture below, OSATE generated instance of the central computer is shown.



Picture 26, *implemened instance of the central computer modules*

2.1.3.) External control unit

```
external_control_unit.aadl ×
external_control_unit.aadl > ...

package external_control_unit
public

------------
-- System --
------------

system external
end external;

system implementation external.impl
end external.impl;

end external_control_unit;
```

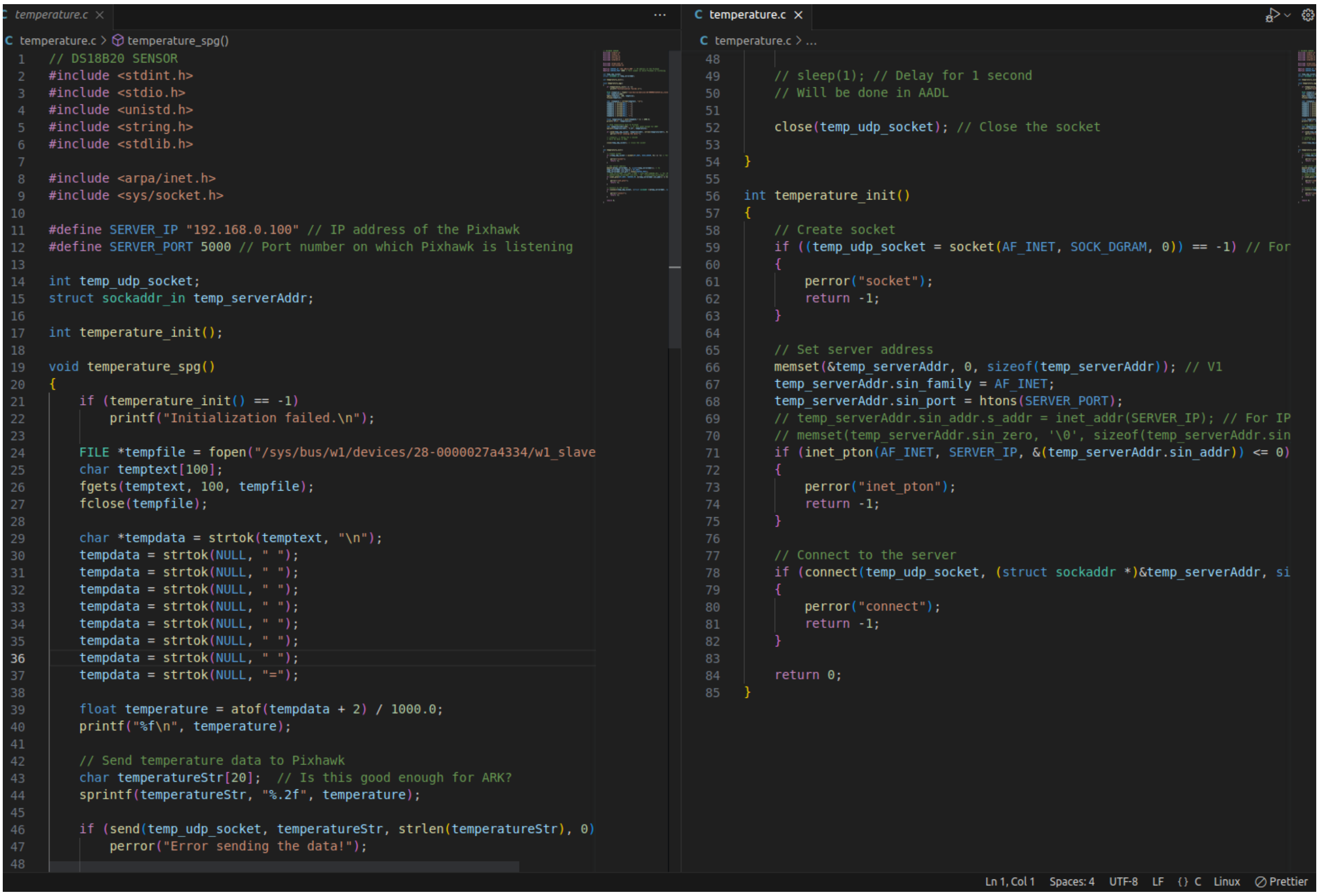
Picture 27, *external_control_unit.aadl*

This file contains the representation of the external controller.

2.2.) Source code

As mentioned before, source code was written in C, in the VS Code IDE.

2.2.1.) Temperature sensor



```c
// DS18B20 SENSOR
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening

int temp_udp_socket;
struct sockaddr_in temp_serverAddr;

int temperature_init();

void temperature_spg()
{
    if (temperature_init() == -1)
        printf("Initialization failed.\n");

    FILE *tempfile = fopen("/sys/bus/w1/devices/28-0000027a4334/w1_slave
    char temptext[100];
    fgets(temptext, 100, tempfile);
    fclose(tempfile);

    char *tempdata = strtok(temptext, "\n");
    tempdata = strtok(NULL, " ");
    tempdata = strtok(NULL, " ");
    tempdata = strtok(NULL, " ");
    tempdata = strtok(NULL, " ");
    tempdata = strtok(NULL, " ");
    tempdata = strtok(NULL, " ");
    tempdata = strtok(NULL, " ");
    tempdata = strtok(NULL, "=");

    float temperature = atof(tempdata + 2) / 1000.0;
    printf("%f\n", temperature);

    // Send temperature data to Pixhawk
    char temperatureStr[20];  // Is this good enough for ARK?
    sprintf(temperatureStr, "%.2f", temperature);

    if (send(temp_udp_socket, temperatureStr, strlen(temperatureStr), 0)
        perror("Error sending the data!");

    // sleep(1); // Delay for 1 second
    // Will be done in AADL

    close(temp_udp_socket); // Close the socket

}

int temperature_init()
{
    // Create socket
    if ((temp_udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) == -1) // For
    {
        perror("socket");
        return -1;
    }

    // Set server address
    memset(&temp_serverAddr, 0, sizeof(temp_serverAddr)); // V1
    temp_serverAddr.sin_family = AF_INET;
    temp_serverAddr.sin_port = htons(SERVER_PORT);
    // temp_serverAddr.sin_addr.s_addr = inet_addr(SERVER_IP); // For IP
    // memset(temp_serverAddr.sin_zero, '\0', sizeof(temp_serverAddr.sin
    if (inet_pton(AF_INET, SERVER_IP, &(temp_serverAddr.sin_addr)) <= 0)
    {
        perror("inet_pton");
        return -1;
    }

    // Connect to the server
    if (connect(temp_udp_socket, (struct sockaddr *)&temp_serverAddr, si
    {
        perror("connect");
        return -1;
    }

    return 0;
}
```

Picture 28, *temperature.c*

The code for this sensor is a C program that reads temperature data from a DS18B20 sensor and sends it to a PIXHAWK device over a UDP socket connection.

Header files include several standard C library header files and some additional header files for networking functionality.
Two constants are defined: SERVER_IP and SERVER_PORT. These represent the IP address and port number of the PIXHAWK device to which the temperature data will be sent.
There are 2 global variables: *temp_udp_socket*, an integer representing the UDP socket, and *temp_serverAddr*, a structure which represents the server address.

There is a prototype of the initialisation function, *temperature_init();* This function is responsible for creating a UDP socket, setting up the server address, and connecting to the PIXHAWK device.
It creates a UDP socket using the *socket()* function. If the socket creation fails, an error message is printed, and *-1* is returned.
It then sets the server address by populating the *temp_serverAddr* structure with the corresponding values. The server IP address is converted from a string to a binary form using *inet_pton()*. If the conversion fails, an error message is printed, and *-1* is returned.

Finally, the socket connects to the server using the *connect()* function. If the connection fails, an error message is printed, and *-1* is returned.

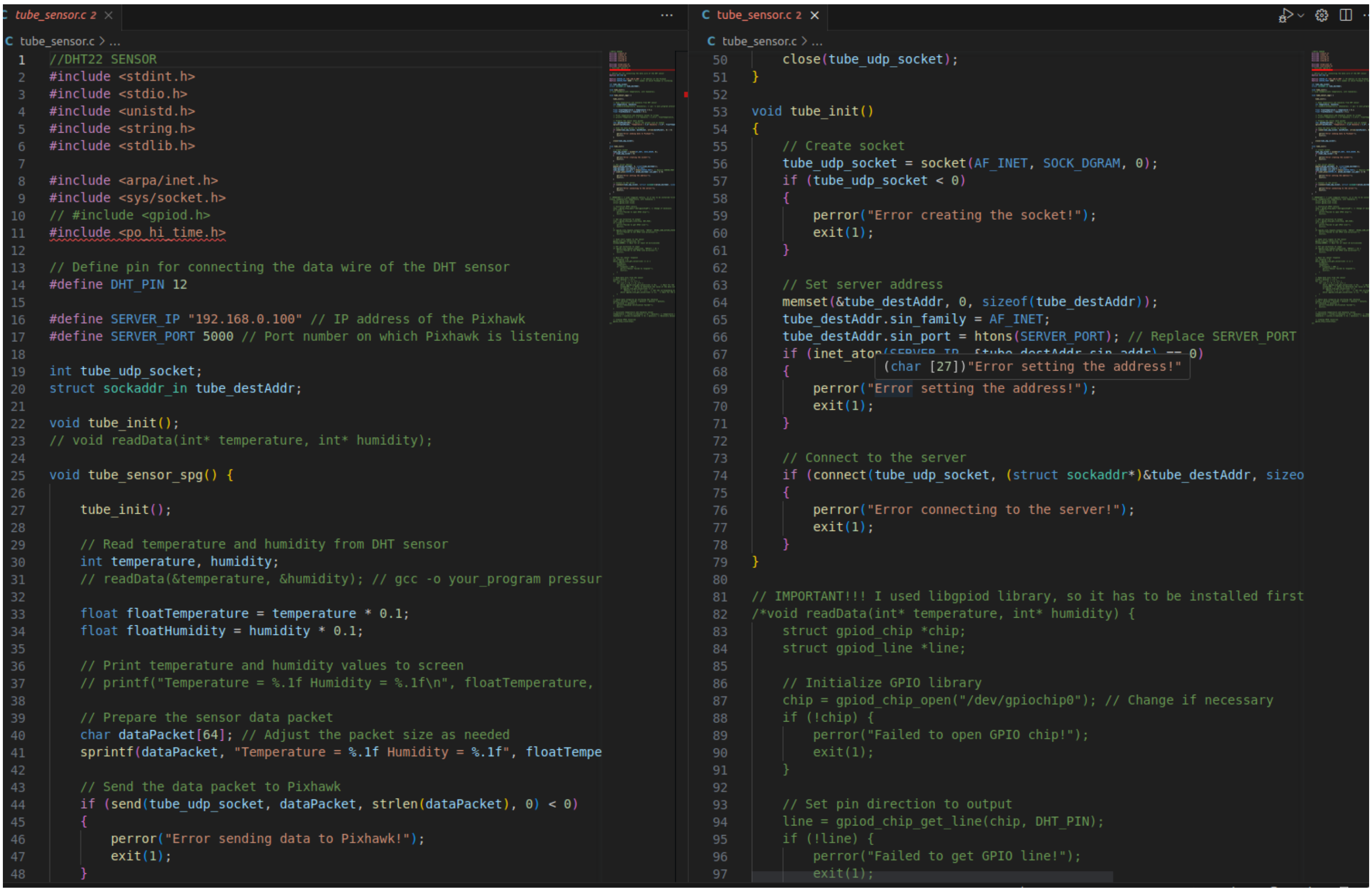
If all the initialization steps are successful, *0* is returned.

temperature_spg() is the main entry point. It first calls the initialisation function to initialize the UDP socket and establish a connection to the PIXHAWK device. If the initialization fails, an error message is printed.
It then opens a file `"/sys/bus/w1/devices/28-0000027a4334/w1_slave", "r"` to read the temperature data from the DS18B20 sensor. The file is read line by line, and the relevant temperature data is extracted and converted to a floating-point value.

The temperature data is then sent to the PIXHAWK device by converting it to a string and using the *send()* function to transmit it over the UDP socket. If an error occurs during sending, an error message is printed.

Finally, the UDP socket is closed using the *close()* function.

2.2.2.) Tube sensor



```c
//DHT22 SENSOR
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>
// #include <gpiod.h>
#include <po_hi_time.h>

// Define pin for connecting the data wire of the DHT sensor
#define DHT_PIN 12

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening

int tube_udp_socket;
struct sockaddr_in tube_destAddr;

void tube_init();
// void readData(int* temperature, int* humidity);

void tube_sensor_spg() {

    tube_init();

    // Read temperature and humidity from DHT sensor
    int temperature, humidity;
    // readData(&temperature, &humidity); // gcc -o your_program pressur

    float floatTemperature = temperature * 0.1;
    float floatHumidity = humidity * 0.1;

    // Print temperature and humidity values to screen
    // printf("Temperature = %.1f Humidity = %.1f\n", floatTemperature,

    // Prepare the sensor data packet
    char dataPacket[64]; // Adjust the packet size as needed
    sprintf(dataPacket, "Temperature = %.1f Humidity = %.1f", floatTempe

    // Send the data packet to Pixhawk
    if (send(tube_udp_socket, dataPacket, strlen(dataPacket), 0) < 0)
    {
        perror("Error sending data to Pixhawk!");
        exit(1);
    }
```

```c
    close(tube_udp_socket);
}

void tube_init()
{
    // Create socket
    tube_udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (tube_udp_socket < 0)
    {
        perror("Error creating the socket!");
        exit(1);
    }

    // Set server address
    memset(&tube_destAddr, 0, sizeof(tube_destAddr));
    tube_destAddr.sin_family = AF_INET;
    tube_destAddr.sin_port = htons(SERVER_PORT); // Replace SERVER_PORT
    if (inet_aton(SERVER_IP, &tube_destAddr.sin_addr) == 0)
    {
        perror("Error setting the address!");
        exit(1);
    }

    // Connect to the server
    if (connect(tube_udp_socket, (struct sockaddr*)&tube_destAddr, sizeo
    {
        perror("Error connecting to the server!");
        exit(1);
    }
}

// IMPORTANT!!! I used libgpiod library, so it has to be installed first
/*void readData(int* temperature, int* humidity) {
    struct gpiod_chip *chip;
    struct gpiod_line *line;

    // Initialize GPIO library
    chip = gpiod_chip_open("/dev/gpiochip0"); // Change if necessary
    if (!chip) {
        perror("Failed to open GPIO chip!");
        exit(1);
    }

    // Set pin direction to output
    line = gpiod_chip_get_line(chip, DHT_PIN);
    if (!line) {
        perror("Failed to get GPIO line!");
        exit(1);
```

Picture 29, *tube_sensor ½*

This code reads temperature and humidity data from a DHT22 sensor and sends it to a PIXHAWK device over a UDP socket connection.

There are several standard header files and several additional ones.
There are 3 constants defined: DHT_PIN is used to connect the data wire, while the SERVER macros define the IP PIXHAWK data.
The global variables are an integer representing the UDP socket, and a structure which represents the server address.

tube_init() is responsible for creating a UDP socket, setting up the server address, and connecting to the PIXHAWK device.
It creates a UDP socket using the *socket()* function. If the socket creation fails, an error message is printed, and the program exits.
It then sets the server address by populating the *tube_destAddr* structure with the appropriate values. The server IP address is converted from a string to a binary form using *inet_aton*. If the conversion fails, an error message is printed, and the program exits.
Finally, the socket is connected to the server using the *connect()* function. If the connection fails, an error message is printed, and the program exits.

tube_sensor_spg() is the main entry point. It first calls *tube_init()* to initialize the UDP socket and establish a connection to the PIXHAWK device.
It then reads temperature and humidity data from the DHT22 sensor. However, the actual reading of data is commented out, along with the necessary dependencies on the *libgpiod* library.
Next, the temperature and humidity values are converted to floating-point values and printed to the screen (commented out).

The sensor data packet is prepared by formatting the temperature and humidity values into a string. Finally, the data packet is sent to the PIXHAWK device using the UDP socket connection. If an error occurs during sending, an error message is printed, and the program exits.
The UDP socket is then closed using the *close()* function.

tube_sensor.c 2

tube_sensor.c > ...

```c
    }
    if (gpiod_line_request_output(line, "DHT22", GPIOD_LINE_ACTIVE_STATE_LOW) < 0) { // Change if necessary
        perror("Failed to set GPIO line direction!");
        exit(1);
    }
    // Send start signal to the sensor
    gpiod_line_set_value(line, 0);
    usleep(18000); // Wait for at least 18 milliseconds

    // Set pin direction to input
    if (gpiod_line_request_input(line, "DHT22") < 0) {
        perror("Failed to set GPIO line direction!");
        exit(1);
    }
    // Wait for sensor response
    int response = 0;
    while (gpiod_line_get_value(line) == 1) {
        usleep(1);
        response++;
        if (response > 100) {
            perror("Sensor failed to respond!");
            exit(1);
        }
    }
    // Read data bits from the sensor
    uint8_t data[5] = { 0 };
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 8; j++) {
            while (gpiod_line_get_value(line) == 0);    // Wait for the start of the data bit
            usleep(30);  // Delay to determine the value of the data bit
            if (gpiod_line_get_value(line) == 1)
                data[i] |= (1 << (7 - j));   // Set the corresponding bit in the data array
            while (gpiod_line_get_value(line) == 1);   // Wait for the end of the data bit
        }
    }
    // Check data integrity by verifying the checksum
    uint8_t checksum = data[0] + data[1] + data[2] + data[3];
    if (checksum != data[4]) {
        perror("Checksum verification failed!");
        exit(1);
    }

    // Calculate temperature and humidity values
    *temperature = (int16_t)((data[2] << 8) | data[3]); // temperature = (int16_t)(data[2] * 256 + data[3]);
    *humidity = (int16_t)((data[0] << 8) | data[1]); // Necessary because of the DHT data format

    // Cleanup GPIO resources
    gpiod_chip_close(chip);
```

Picture 30, *tube_sensor 2/2*

Two integer pointers are declared, which are used to store the data. Two structure variables are declared, which are to interact with GPIO pins.

GPIO is initialised in 3 steps:

```c
chip = gpiod_chip_open("/dev/gpiochip0");
```

This line opens a connection to the GPIO chip by providing the device file path (`/dev/gpiochip0`). This path may be changed, based on the specific setup. If the connection fails, an error message is printed, and the program exits.

```c
line = gpiod_chip_get_line(chip, DHT_PIN);
```

This line obtains a reference to a specific GPIO line on the chip, indicated by the DHT_PIN constant. If obtaining the line fails, an error message is printed, and the program exits.

```
gpiod_line_request_output(line, "DHT22", GPIOD_LINE_ACTIVE_STATE_LOW
```

This line configures the GPIO line as an output. The DHT22 string is a label for the line, and GPIOD_LINE_ACTIVE_STATE_LOW indicates that the line should be initially set to a low (0) state. If the configuration fails, an error message is printed, and the program exits.

The sensor communication is done in 3 steps as well:

```
gpiod_line_set_value(line, 0);
```

This sets the GPIO line to a low (0) state, which acts as a start signal to the sensor.

```
usleep(18000);
```

This pauses the program execution for at least 18 milliseconds to allow the sensor to respond.

```
gpiod_line_request_input(line, "DHT22"
```

This reconfigures the GPIO line as an input to prepare for reading data. If the reconfiguration fails, an error message is printed, and the program exits.

Waiting for sensor response: the code waits for the sensor to respond by continuously checking the value of the GPIO line. If the line remains high (1) for more than 100 iterations, indicating no response from the sensor, an error message is printed, and the program exits.

The data is read inside a nested loop; code reads 40 bits of data (5 bytes) from the sensor.
It waits for the start of each data bit by checking the GPIO line value. Once the bit starts, it waits for a short delay to determine its value.
If the value is high (*1*), it sets the corresponding bit in the *data* array using bitwise OR and left shift operations.
After each data bit, it waits for the end of the bit by checking the GPIO line value.
The code then calculates a checksum by adding the first four bytes of the *data* array and compares it with the fifth byte. If the checksum verification fails, an error message is printed, and the program exits.

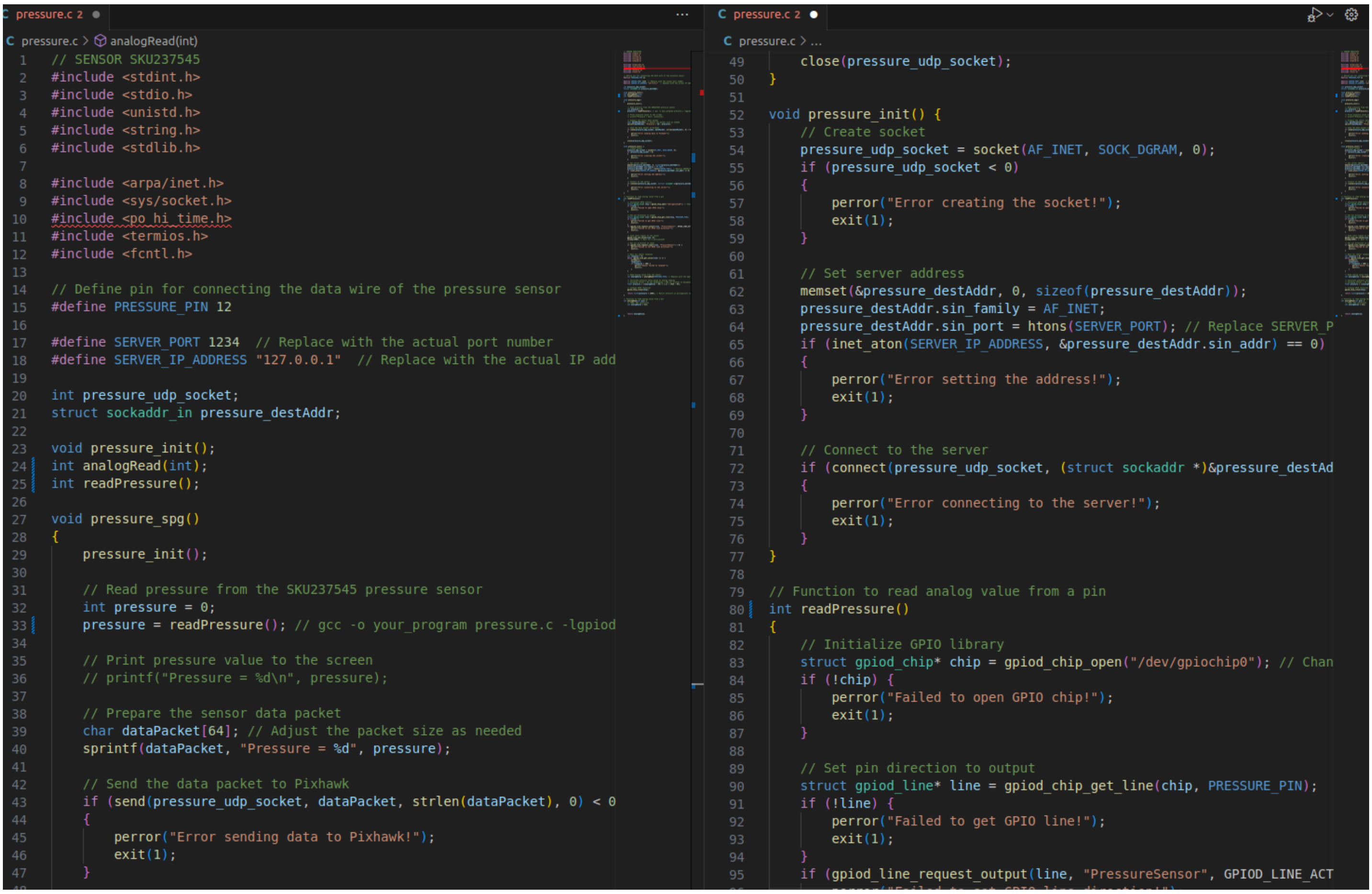
The data is processed by the code calculating the temperature and humidity values from the received data. It combines the third and fourth bytes to form the temperature value and the first and second bytes for the humidity value. The values are stored in the memory locations of the pointers.

In the end, a cleanup is done (`gpiod_chip_close(chip);`).

The *libgpiod* library was used to fetch the data, therefore installation is necessary before use;
sudo apt-get install libgipod-dev

To use this library, compilation process needs update:
gcc -o *program* tube_sensor.c -lgpiod

2.2.3.) Pressure



```c
// SENSOR SKU237545
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>
#include <po_hi_time.h>
#include <termios.h>
#include <fcntl.h>

// Define pin for connecting the data wire of the pressure sensor
#define PRESSURE_PIN 12

#define SERVER_PORT 1234  // Replace with the actual port number
#define SERVER_IP_ADDRESS "127.0.0.1"  // Replace with the actual IP add

int pressure_udp_socket;
struct sockaddr_in pressure_destAddr;

void pressure_init();
int analogRead(int);
int readPressure();

void pressure_spg()
{
    pressure_init();

    // Read pressure from the SKU237545 pressure sensor
    int pressure = 0;
    pressure = readPressure(); // gcc -o your_program pressure.c -lgpiod

    // Print pressure value to the screen
    // printf("Pressure = %d\n", pressure);

    // Prepare the sensor data packet
    char dataPacket[64]; // Adjust the packet size as needed
    sprintf(dataPacket, "Pressure = %d", pressure);

    // Send the data packet to Pixhawk
    if (send(pressure_udp_socket, dataPacket, strlen(dataPacket), 0) < 0
    {
        perror("Error sending data to Pixhawk!");
        exit(1);
    }
```

```c
    close(pressure_udp_socket);
}

void pressure_init() {
    // Create socket
    pressure_udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (pressure_udp_socket < 0)
    {
        perror("Error creating the socket!");
        exit(1);
    }

    // Set server address
    memset(&pressure_destAddr, 0, sizeof(pressure_destAddr));
    pressure_destAddr.sin_family = AF_INET;
    pressure_destAddr.sin_port = htons(SERVER_PORT); // Replace SERVER_P
    if (inet_aton(SERVER_IP_ADDRESS, &pressure_destAddr.sin_addr) == 0)
    {
        perror("Error setting the address!");
        exit(1);
    }

    // Connect to the server
    if (connect(pressure_udp_socket, (struct sockaddr *)&pressure_destAd
    {
        perror("Error connecting to the server!");
        exit(1);
    }
}

// Function to read analog value from a pin
int readPressure()
{
    // Initialize GPIO library
    struct gpiod_chip* chip = gpiod_chip_open("/dev/gpiochip0"); // Chan
    if (!chip) {
        perror("Failed to open GPIO chip!");
        exit(1);
    }

    // Set pin direction to output
    struct gpiod_line* line = gpiod_chip_get_line(chip, PRESSURE_PIN);
    if (!line) {
        perror("Failed to get GPIO line!");
        exit(1);
    }
    if (gpiod_line_request_output(line, "PressureSensor", GPIOD_LINE_ACT
```

Picture 31, *pressure sensor ½*

As in the previous sensors, first there are various header files imported and macros defined. PRESSURE_PIN is defined as the pin number to which the data wire of the pressure sensor is connected. SERVER_PORT and SERVER_IP_ADDRESS are defined as the port number on which the UDP server is running and the IP address of the UDP server.
pressure_udp_socket is an integer variable that will store the socket descriptor for the UDP socket, and *pressure_destAddr* is a *sockaddr_in* type structure, which represents the server's address (IP address and port number).

pressure_init() is a function that initializes the UDP socket, sets the server's address, and connects to the server.
pressure_spg() is the main function that reads the pressure from the sensor, prepares a data packet, and sends it to the PIXHAWK device.
Inside this function, pressure value is set to 0, but is then changed by the returned value of *readPressure()*. This function is used to read the pressure value from the sensor. However, using this function requires, as with the *tube_sensor*, different compilation command.

dataPacket, a character array used to store the pressure value as a string, is sent to the PIXHAWK using the *send()* function.
analogRead() is to be implemented so that it reads analog values from the pressure sensor (using GPIO operations), but now it returns a placeholder value of 512 (which is to be replaced with actual ADC conversion).

```c
        exit(1);
    }

    // Send start signal to the sensor
    gpiod_line_set_value(line, 0);
    usleep(2000); // Wait for 2 milliseconds

    // Set pin direction to input
    if (gpiod_line_request_input(line, "PressureSensor") < 0) {
        perror("Failed to set GPIO line direction!");
        exit(1);
    }

    // Wait for sensor response
    int response = 0;
    while (gpiod_line_get_value(line) == 1) {
        usleep(1);
        response++;
        if (response > 100) {
            perror("Sensor failed to respond!");
            exit(1);
        }
    }

    // Read analog value from the sensor
    int analogValue = analogRead(PRESSURE_PIN); // Replace with the appropriate function to read analog value

    // Calculate pressure value based on analog reading
    // Assuming offset = 94 and maxReading = 920 (as written in documentation)
    float pressure = ((analogValue - 94) * 1.2) / (920 - 94);

    // Cleanup GPIO resources
    gpiod_chip_close(chip);

    return (int)(pressure * 1000); // Return pressure in millipascals (mPa)
}

// Function to read analog value from a pin
int analogRead(int pin) {
    // TODO: ADC conversion
    int analogValue = 512;




    return analogValue;
}
```

Picture 32, *pressure 2/2*

2.2.4.) Magnetometer

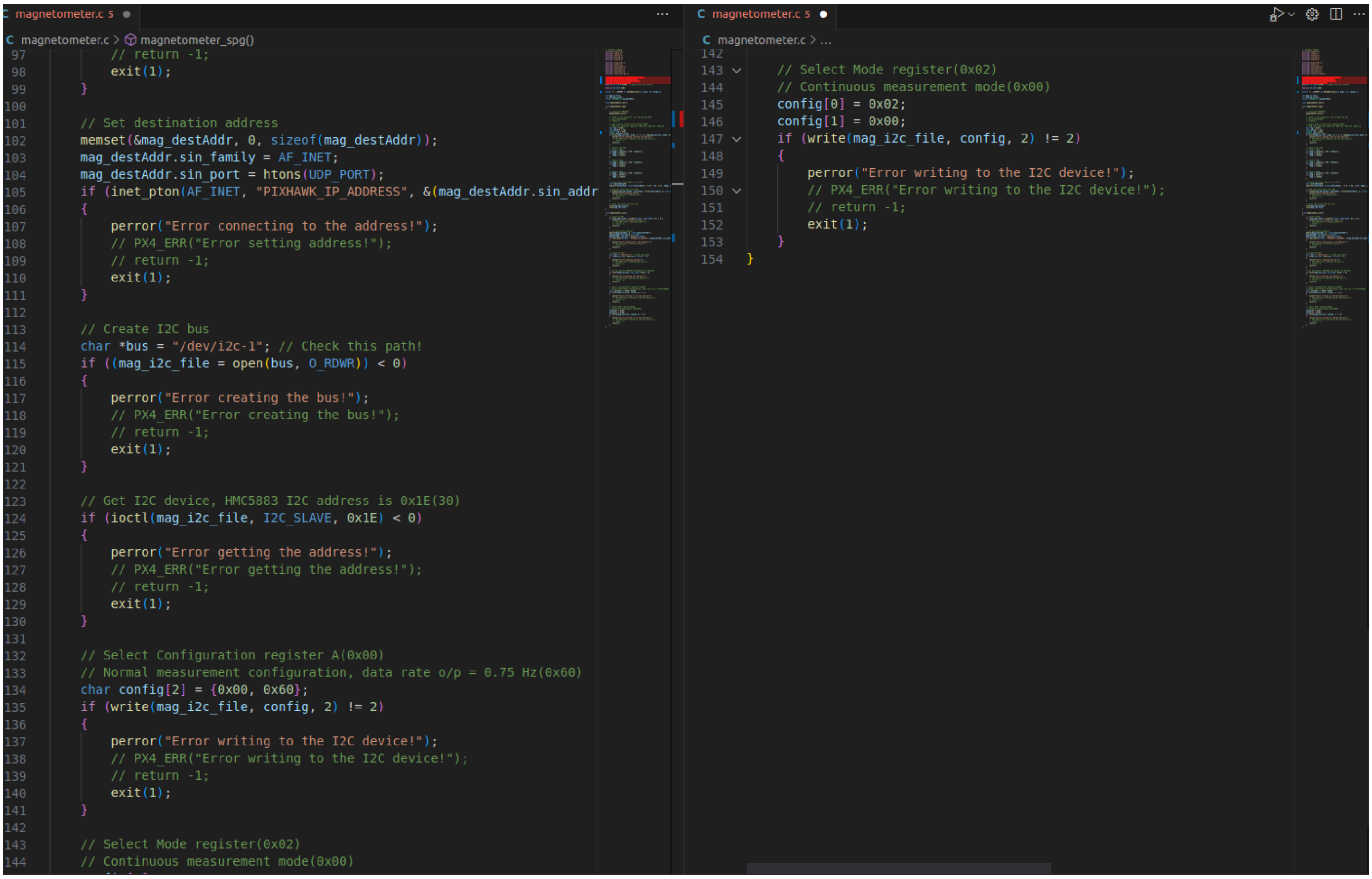
```c
// HMC5883L SENSOR
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/fcntl.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>

#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/log.h>
#include <px4_platform_common/tasks.h>
#include <drivers/drv_hrt.h>
#define I2C_SLAVE 0x0703  // Dummy value for testing

#define UDP_PORT 5000

extern "C" __EXPORT int hmc5883l_main(int argc, char *argv[]);

int mag_i2c_file;
int mag_udp_socket;
struct sockaddr_in mag_destAddr;

void magnetometer_init();

void magnetometer_spg()
{

    // Initialize HMC5883L
    magnetometer_init();

    // Sleep is not necessary, will be done by AADL
    // px4_usleep(1000);
    // sleep(1);

    // Read 6 bytes of data from register(0x03)
    // xMag msb, xMag lsb, zMag msb, zMag lsb, yMag msb, yMag lsb
    char reg = 0x03;
    char data[6] = {0};
    uint8_t reg = 0x03;
    uint8_t data[6] = {0};
    if (write(mag_i2c_file, &reg, 1) != 1 || read(mag_i2c_file, data, 6)
```

```c
        perror("Error reading from the I2C device");
        // PX4_ERR("Error reading from the I2C device");
        // return -1;
        exit(1);
    }

    // Convert the data
    // int16_t xMag
    int xMag = (data[0] * 256 + data[1]);
    if (xMag > 32767) {
        xMag -= 65536;
    }

    // int16_t zMag
    int zMag = (data[2] * 256 + data[3]);
    if (zMag > 32767) {
        zMag -= 65536;
    }

    // int16_t yMag
    int yMag = (data[4] * 256 + data[5]);
    if (yMag > 32767) {
        yMag -= 65536;
    }

    // Format the sensor data as a string
    char sensorData[50];
    snprintf(sensorData, sizeof(sensorData), "X:%d, Y:%d, Z:%d", xMag, y

    // Send the sensor data over UDP
    if (sendto(mag_udp_socket, sensorData, strlen(sensorData), 0, (struc
        perror("Error sending data");
        // PX4_ERR("Error sending data");
        // return -1;
        exit(1);
    }

    // Close the socket and I2C file
    close(mag_udp_socket);
    close(mag_i2c_file);
}

void magnetometer_init()
{
    // Create socket
    if ((mag_udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Error creating the socket!");
        // PX4_ERR("Error creating socket!");
```

Picture 33, *magnetometer ½*

As before, first there are various header files imported.
There are 2 macros defined: I2C_SLAVE is defined as a dummy value for testing purposes, and UDP_PORT is defined as the port number on which the UDP socket will communicate.
mag_i2c_file is an integer variable that will store the file descriptor for the I2C bus.
mag_udp_socket is an integer variable that will store the socket descriptor for the UDP socket.
mag_destAddr is a struct of *sockaddr_in* type that represents the destination address (IP address and port number) for the UDP communication.

magnetometer_init() is a function that creates a UDP socket, sets the destination address, creates the I2C bus, and initializes the HMC5883L sensor by writing to its configuration and mode registers.
magnetometer_spg() is the main function that reads data from the HMC5883L sensor, formats it as a string, and sends it over UDP.
Inside it, the HMC5883L sensor is initialized by setting the measurement configuration and mode registers.
Data is read from the sensor by writing the register address (0x03) and then reading 6 bytes of data, after which is converted to 16-bit signed integers (int16_t) for each axis.
Then, it is formatted as a string using *sprintf()*, and it is sent over UDP using the *sendtio()* function.
After the data is sent, the socket and the file are closed.



```c
        // return -1;
        exit(1);
    }

    // Set destination address
    memset(&mag_destAddr, 0, sizeof(mag_destAddr));
    mag_destAddr.sin_family = AF_INET;
    mag_destAddr.sin_port = htons(UDP_PORT);
    if (inet_pton(AF_INET, "PIXHAWK_IP_ADDRESS", &(mag_destAddr.sin_addr
    {
        perror("Error connecting to the address!");
        // PX4_ERR("Error setting address!");
        // return -1;
        exit(1);
    }

    // Create I2C bus
    char *bus = "/dev/i2c-1"; // Check this path!
    if ((mag_i2c_file = open(bus, O_RDWR)) < 0)
    {
        perror("Error creating the bus!");
        // PX4_ERR("Error creating the bus!");
        // return -1;
        exit(1);
    }

    // Get I2C device, HMC5883 I2C address is 0x1E(30)
    if (ioctl(mag_i2c_file, I2C_SLAVE, 0x1E) < 0)
    {
        perror("Error getting the address!");
        // PX4_ERR("Error getting the address!");
        // return -1;
        exit(1);
    }

    // Select Configuration register A(0x00)
    // Normal measurement configuration, data rate o/p = 0.75 Hz(0x60)
    char config[2] = {0x00, 0x60};
    if (write(mag_i2c_file, config, 2) != 2)
    {
        perror("Error writing to the I2C device!");
        // PX4_ERR("Error writing to the I2C device!");
        // return -1;
        exit(1);
    }

    // Select Mode register(0x02)
    // Continuous measurement mode(0x00)
    config[0] = 0x02;
    config[1] = 0x00;
    if (write(mag_i2c_file, config, 2) != 2)
    {
        perror("Error writing to the I2C device!");
        // PX4_ERR("Error writing to the I2C device!");
        // return -1;
        exit(1);
    }
}
```

Picture 34, *magnetometer 2/2*

There are small differences between this sensor and the others; such as the

*extern "C" __EXPORT int hmc5883l_main(int argc, char *argv[]);*

This is an embedded PX6 function; it is commented in this project, but it was left, because this sensor can be found in the official build of the autopilot.

2.2.5.) Lights

lights.c > lights_init()

```c
// LUMEN SUBSEA LIGHTS
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>
#include <gpiod.h>

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening
#define OUTPUT 1 // voltage/ground
#define HIGH 1 // 5V or logical 1
#define LOW 0

int ledPin1 = 33;
int ledPin2 = 35;
int lights_udp_socket;
struct sockaddr_in server_address;
struct gpiod_chip* chip;
struct gpiod_line* line;

void lights_init();
void update_led_state(int value);
void lightsPinMode(int pin, int mode);
void lightsDigitalWrite(int pin, int value);

void lights_spg()
{
    lights_init();

    // Read LED control value from Pixhawk
    // ...

    int ledControlValue = 1111; // Replace with the value received from

    // Perform LED control based on the received value
    update_led_state(ledControlValue);

    // Send the LED control value back to Pixhawk
    sendto(lights_udp_socket, &ledControlValue, sizeof(ledControlValue),

    close(lights_udp_socket);
}

void lights_init()
```

lights.c > lights_init()

```c
{
    // Create socket
    lights_udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (lights_udp_socket == -1) {
        perror("Socket creation failed!");
        exit(-1);
    }

    // Set server address
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(SERVER_PORT);
    if (inet_pton(AF_INET, SERVER_IP, &(server_address.sin_addr)) <= 0)
    {
        perror("Invalid address!");
        exit(-1);
    }

    // Initialize LED pins for GPIO?
    // ...

    // Initialize LED pins - commented because of the library used
    // lightsPinMode(ledPin1, OUTPUT);
    // lightsPinMode(ledPin2, OUTPUT);
    // lightsDigitalWrite(ledPin1, HIGH);
    // lightsDigitalWrite(ledPin2, HIGH);
}

void update_led_state(int value)
{
    switch (value) // commented because of the library used
    {
        case 1234: // defined as such in the original code
            // lightsDigitalWrite(ledPin1, LOW);
            // lightsDigitalWrite(ledPin2, LOW);
            break;
        case 1111: // defined as such in the original code
            // lightsDigitalWrite(ledPin1, HIGH);
            // lightsDigitalWrite(ledPin2, HIGH);
            break;
        // Add more cases if needed

        default:
            // Handle unsupported values or errors
            break;
    }
}
```

Picture 35, *lights sensor ½*

After importing various header files, several macros are described as constants for server IP address, server port, and pin modes (OUTPUT, HIGH, LOW).
Various global variables are declared in order to ease the code flow.
Various function prototypes are defined.

In the main function, *lights_spg()*, subsea lights are controlled. The function initializes the lights, reads the LED control value from the PIXHAWK, performs LED control based on the received value, sends the LED control value back to the PIXHAWK, and then closes the UDP socket.
lights_init() function is responsible for creating the UDP socket, setting the server address, and initializing the GPIO pins for controlling the lights.

update_led_state() function is used to change the LED state based on the received LED control value. The actual GPIO control is commented out and should be implemented based on the chosen GPIO library.

lightsPinMode() function sets the pin mode for the specified GPIO pin based on the mode parameter (either OUTPUT or INPUT). It uses the *libgpiod* library to open the GPIO chip, get the GPIO line, and set the pin direction.

lightsDigitalWrite function sets the pin state for the specified GPIO pin based on the value parameter (either HIGH or LOW). It uses the libgpiod library to set the pin value.

To use this code, as with other sensors, *libgpiod* library must be installed. That is the reason why the GPIO chip (/dev/gpiochip0) is used.

```c
// IMPORTANT!!! I used libgpiod library, so it has to be installed first: sudo apt-get install libgpiod-dev
void lightsPinMode(int pin, int mode) {
    int temp;

    // Open the GPIO chip
    chip = gpiod_chip_open("/dev/gpiochip0");
    if (!chip) {
        perror("Failure opening GPIO chip!");
        exit(-1);
    }

    // Get the GPIO line
    line = gpiod_chip_get_line(chip, pin);
    if (!line) {
        perror("Failure getting GPIO line!");
        exit(-1);
    }

    // Set the pin direction
    if (mode == OUTPUT) {
        temp = gpiod_line_request_output(line, "my-output", 0);
    } else {
        temp = gpiod_line_request_input(line, "my-input");
    }

    if (temp < 0) {
        perror("Failure setting pin mode!");
        exit(-1);
    }
}

void lightsDigitalWrite(int pin, int value)
{
    // Set the pin state
    int temp;
    if (value == HIGH) {
        temp = gpiod_line_set_value(line, 1);
    } else {
        temp = gpiod_line_set_value(line, 0);
    }

    if (temp < 0) {
        perror("Failure setting pin value!");
        exit(-1);
    }
}
```

Picture 36, *lights sensor 2/2*

2.2.6.) Power sensing module

power_sensing_module.c

```c
// BLUE ROBOTICS PSM
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening

int power_udp_socket;
struct sockaddr_in power_serverAddr;

void power_sensing_module_init();
void readPowerData(float *current, float *voltage);

void power_sensing_module_spg()
{
    power_sensing_module_init();

    // Read power data
    float current = 0.0, voltage = 0.0;
    readPowerData(&current, &voltage);

    printf("Current: %.2f A\n", current);
    printf("Voltage: %.2f V\n", voltage);

    // Send power data to Pixhawk
    char powerStr[50]; // Modify the size as per your data requirements
    sprintf(powerStr, "Current: %.2f A, Voltage: %.2f V", current, volta

    if (send(power_udp_socket, powerStr, strlen(powerStr), 0) == -1)
        perror("Error sending the data!");

    close(power_udp_socket);
}

```

power_sensing_module.c

```c
void power_sensing_module_init()
{
    // Create socket
    if ((power_udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    { // For TCP, SOCK_STREAM
        perror("socket");
        exit(1);
    }

    // Set server address
    memset(&power_serverAddr, 0, sizeof(power_serverAddr)); // V1
    power_serverAddr.sin_family = AF_INET;
    power_serverAddr.sin_port = htons(SERVER_PORT);
    if (inet_pton(AF_INET, SERVER_IP, &(power_serverAddr.sin_addr)) <= 0
    {
        perror("inet_pton");
        exit(1);
    }

    // Connect to the server
    if (connect(power_udp_socket, (struct sockaddr *)&power_serverAddr,
    {
        perror("connect");
        exit(1);
    }
}

void readPowerData(float *current, float *voltage)
{
    // Replace the code below with implementation to read the current an
    // GPIO, maybe?
    *current = 1.23;
    *voltage = 12.34;
}
```

Picture 37, *PSM*

This Blue Robotics Power Sensor Module communicates with a PIXHAWK flight controller using UDP.

After importing various header files and defining macros for PIXHAWK IP address and its port, global variables (for easier code handling) and function prototypes are declared.

In the main function, *power_sensing_module_spg()*, the module is initialised, the current and voltage data is read and (after constructing a power data string) sent to the Pixhawk flight controller via UDP.

In the initialisation function, the UDP socket is created, the server address is set and the connection to the server is established. *socket()*, *memset()*, *inet_pton()* and *connect()* functions are used for this purpose.

readPowerData() is a placeholder for reading the current and voltage values from the power sensing module. In the provided implementation, it simply assigns static values to the *current* and *voltage* pointers. This code is to be replaced with the actual implementation of data reading.

2.2.7.) Propulsion

propulsion.c 2

propulsion.c > esc_spg()

```c
// T200
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <time.h>
#include <po_hi_time.h>

#define ESC_SERVO_PIN 17 // Defined as such in the original code

// Macros for easier code reading
#define DRIVE_INPUT 0
#define INPUT 0
#define DRIVE_OUTPUT 1
#define OUTPUT 1

// T200 Thruster Pins
#define THRUSTER_1_PIN 2
#define THRUSTER_2_PIN 3
#define THRUSTER_3_PIN 5
#define THRUSTER_4_PIN 6
#define THRUSTER_5_PIN 7

int signal = 1700; // Defined as such in the original code

int thruster;
int speed;

void esc_init(int pin, int mode);
void escWriteMicroseconds(int pin, int value);
void escPinMode(uint8_t pin, uint8_t mode);

void thrusters_init();

void esc_spg()
{
    // Configure the servo pin
    esc_init(ESC_SERVO_PIN, OUTPUT);

    // Send stop signal to ESC
    escWriteMicroseconds(ESC_SERVO_PIN, 1500);

    // Define the value of the signal, ranging from 1100 to 1900; 1700 b
    signal = 1700;

    // Send the signal to the ESC
```

propulsion.c 2

propulsion.c > ...

```c
    escWriteMicroseconds(ESC_SERVO_PIN, signal);

}

void esc_init(int pin, int mode)
{
    if (mode == DRIVE_INPUT)
    {
        // Implementation specific to the hardware for input mode
        // Configure the pin as an input for receiving PWM signal from P
        // Initialize the necessary communication protocol for receiving
        // Set up the appropriate interrupt or polling mechanism to read

        // Configure the pin as an input
        escPinMode(pin, INPUT);

        // Initialize the necessary communication protocol for receiving
        // For example, if you are using a microcontroller with built-in

        // Set up the appropriate interrupt or polling mechanism to read
        // Here, you would define the necessary interrupt handlers or po
    }
    else if (mode == DRIVE_OUTPUT) {
        // Implementation specific to the hardware for output mode
        // Configure the pin as an output for sending PWM signal to ESC
        // Initialize the necessary communication protocol for sending P
        // Set up the appropriate timing and duty cycle to generate the
        // Connect the output pin to the ESC control input

        // Configure the pin as an output
        escPinMode(pin, OUTPUT);

        // Initialize the necessary communication protocol for sending P
        // For example, if you are using a microcontroller with built-in

        // Set up the appropriate timing and duty cycle to generate the
        // You would define the necessary code to set the desired timing

        // Connect the output pin to the ESC control input
        // Here, you would connect the output pin to the appropriate ESC
    }
}

// In this version, the Servo library is replaced with "escWriteMicrosec
void escWriteMicroseconds(int pin, int value) {
    // Write code to send a signal specific to the hardware
}
```

Picture 38, *electronic speed controller*

This file contains the code for both the ESCs and its thrusters.
After importing various header files, macros for easier code reading and thruster pins are defined.

esc_spg() function initializes the ESC servo pin as an output and sends a stop signal to the ESC. It then sets the signal value and sends it to the ESC.
esc_init() function is responsible for configuring the ESC servo pin as either an input or output based on the specified mode. Depending on the mode, the function initializes the necessary communication protocol and sets up interrupt handlers or polling logic.
This part of the code is unfinished with some instructions left.

escWriteMicroseconds() function is a placeholder for sending a signal specific to the hardware.
escPinMode() function is a placeholder for configuring the pin mode based on the provided parameters. It is based on Arduino principle.

thrusters_spg() function initializes the thrusters, and based on the specified thruster number, it sets the speed using the *escWriteMicroseconds()* function.
thrusters_init() function initializes the ESC pins and modes for each thruster by calling *esc_init()* with the appropriate parameters.

propulsion.c 2

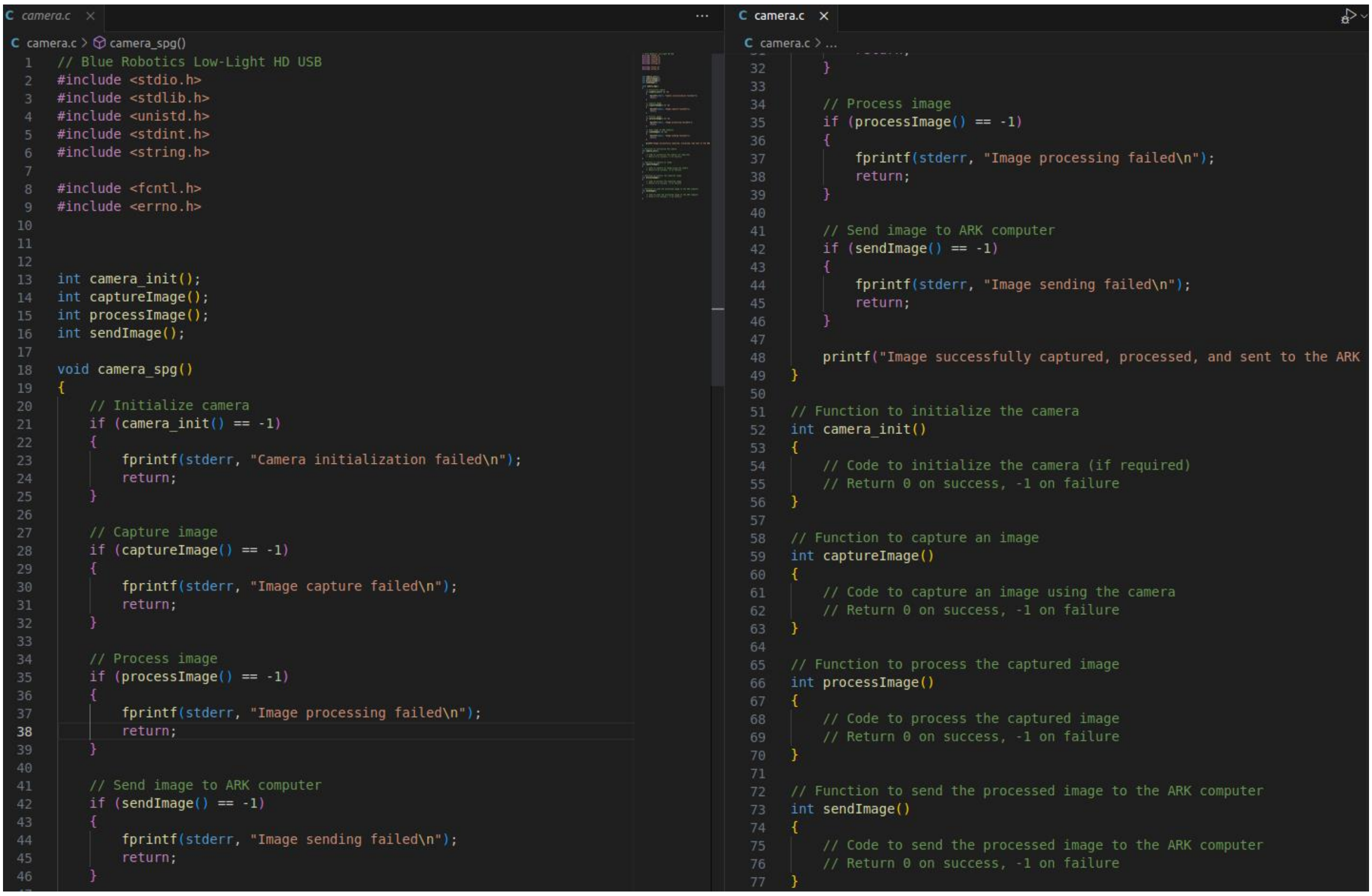
propulsion.c > esc_spg()

```c
// Implementation of escPinMode function
void escPinMode(uint8_t pin, uint8_t mode) {
    // Implementation specific to the platform or library
    // Configure the pin mode based on the provided parameters
    // This implementation assumes the use of the Arduino framework

    // Example implementation using the Arduino framework
    // Here, the pin mode is set using the escPinMode function provided by the Arduino library
    // The implementation would vary based on the specific platform or library being used
}

void thrusters_spg()
{
    thrusters_init();

    // Set the speed of the specified thruster
    switch (thruster) {
        case 1:
            escWriteMicroseconds(THRUSTER_1_PIN, speed);
            break;
        case 2:
            escWriteMicroseconds(THRUSTER_2_PIN, speed);
            break;
        case 3:
            escWriteMicroseconds(THRUSTER_3_PIN, speed);
            break;
        case 4:
            escWriteMicroseconds(THRUSTER_4_PIN, speed);
            break;
        case 5:
            escWriteMicroseconds(THRUSTER_5_PIN, speed);
            break;
        default:
            // Handle unsupported thruster number
            break;
    }
}

void thrusters_init()
 {
    // Initialize the ESC pins and modes for each thruster
    esc_init(THRUSTER_1_PIN, DRIVE_OUTPUT);
    esc_init(THRUSTER_2_PIN, DRIVE_OUTPUT);
    esc_init(THRUSTER_3_PIN, DRIVE_OUTPUT);
    esc_init(THRUSTER_4_PIN, DRIVE_OUTPUT);
    esc_init(THRUSTER_5_PIN, DRIVE_OUTPUT);
}
```

Picture 39, *thrusters*

2.2.8.) Camera and GPS

The 2 modules that connect directly to the ARK computer are camera and GPS. There are only the basics of the source code written for these modules.



```c
// Blue Robotics Low-Light HD USB
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>

#include <fcntl.h>
#include <errno.h>



int camera_init();
int captureImage();
int processImage();
int sendImage();

void camera_spg()
{
    // Initialize camera
    if (camera_init() == -1)
    {
        fprintf(stderr, "Camera initialization failed\n");
        return;
    }

    // Capture image
    if (captureImage() == -1)
    {
        fprintf(stderr, "Image capture failed\n");
        return;
    }

    // Process image
    if (processImage() == -1)
    {
        fprintf(stderr, "Image processing failed\n");
        return;
    }

    // Send image to ARK computer
    if (sendImage() == -1)
    {
        fprintf(stderr, "Image sending failed\n");
        return;
    }

    printf("Image successfully captured, processed, and sent to the ARK
}

// Function to initialize the camera
int camera_init()
{
    // Code to initialize the camera (if required)
    // Return 0 on success, -1 on failure
}

// Function to capture an image
int captureImage()
{
    // Code to capture an image using the camera
    // Return 0 on success, -1 on failure
}

// Function to process the captured image
int processImage()
{
    // Code to process the captured image
    // Return 0 on success, -1 on failure
}

// Function to send the processed image to the ARK computer
int sendImage()
{
    // Code to send the processed image to the ARK computer
    // Return 0 on success, -1 on failure
}
```

Picture 40, *camera*

There are standard libraries imported. A structure of the code is assumed, divided into 4 parts.
In the main function (*camera_spg()*), *camera_init()* would be called, after which the image is to be captured, processed and sent to the ARK computer.

The same code structure is written for GPS:

```c
void gps_spg()
{
    // Initialize GPS
    if (gps_init() == -1)
    {
        fprintf(stderr, "GPS initialization failed\n");
        return;
    }

    // Read GPS data
    if (readGPSData() == -1)
    {
        fprintf(stderr, "GPS data read failed\n");
        return;
    }

    // Process GPS data
    if (processGPSData() == -1)
    {
        fprintf(stderr, "GPS data processing failed\n");
        return;
    }

    // Send GPS data to ARK computer
    if (sendGPSData() == -1)
    {
        fprintf(stderr, "GPS data sending failed\n");
        return;
    }

    printf("GPS data successfully read, processed, and sent to the ARK computer!\n");
}

int gps_init()
{
    // Code to initialize the GPS device
    // Return 0 on success, -1 on failure
}

int readGPSData()
{
    // Code to read GPS data from the device
    // Return 0 on success, -1 on failure
}

int processGPSData()
```

Picture 41, *GPS*

CONCLUSION

In this paper, software development of an upgraded underwater ROV is described.

Using AADL has proven to be an excellent choice: its ability to clearly develop models for this system proved to be efficient. The related tools (OSATE and Ocarina) allow the developer easieraccess to code/model testing.

Writing of the code for the sensors represents the bulk of the work done here. Its integration withPIXHAWK and AADL proved to be a challenging obstacle to overcome.

After researching AADL, OSATE and Ocarina, I started working on creating a functional ROV model (using already available ROV data as the starting template). For a little while, there were some technical difficulties (Linux-related), but those were quickly dealt with.
After I created acceptable ROV models, I started working on the source code for various modules and sensors, using ARDUINO code of the old ROV model as a starting point.

Many problems were encountered during this step, mostly related to (un)successful integration of the written code and the hardware (*how can ARK computer successfully interpret the data received*?). After doing further research, the solution was found (existing Linux libraries are to be used as a connection between data reading, data processing and data sending to the ARK computer). The chosen “gpiod” libraries are shown to be compatible.

Of the 7 sensors/modules connected to the autopilot, the *pressure* sensor, the *power sensing* module and the *propulsion* are missing the implementations of the crucial data-reading functions.
The *lights* module’s structure of data flow needs confirmation (it was written on the penultimate day of the project, so it was not properly tested).

Modules connected to the central computer (*camera* and *gps*) have merely been declared; they require the most work.

After the next student finishes implementing these sensors, (s)he can complete the *pixhawk_interface* file (to succesfully integrate the sofwtare and hardware) and the work should be done.

PICTURES

1 – screenshot of AADL code

2 – screenshot of AADL code in OSATE

3 – screenshot of Ocarina use

4 – screenshot of make command

5 – PX6, 09.06.2023., https://docs.px4.io/main/en/flight_controller/pixhawk6x.html

6 – ARK computer, 09.06.2023., https://www.advantech.com/en/products/92d96fda-cdd3-409d-aae5-2e516c0f1b01/ark-1551/mod_47d30ee7-28b6-41bc-83a1-a7ca416e68cd

7 – screenshot of the sensor code

8 – temperature sensor, 12.06.2023., https://components101.com/sites/default/files/components/DS18B20-Sensor_0.jpg

9 – tube sensor, 12.06.2023., https://cityos-air.readme.io/docs/4-dht22-digital-temperature-humidity-sensor

10 – pressure sensor, 12.06.2023., https://xianyunyi2020.en.made-in-china.com/product/UwftapmVRLcg/China-Flat-Connector-Mini-4-20mA-Auto-Fuel-Oil-Pressure-Sensor.html

11 – power sensor module, 12.06.2023., https://bluerobotics.com/store/comm-control-power/control/psm-asm-r2-rp/

12 – magnetometer sensors, 12.06.2023., https://www.electronicwings.com/sensors-modules/hmc5883l-magnetometer-module

13 – thruster, 12.06.2023., https://www.carcinus.co.uk/product/blue-robotics-t200-thruster/

14 - ROV model, screenshot of the model

15 : 41 - screenshots of the modules’ code and instances

ANNEX

In the annex, code of this project can be found.

AADL files

scenario.aadl

```
system rover_root
properties
	Ocarina_Config::Timeout_Property => 4000ms;
	Ocarina_Config::Referencial_Files =>
		("central_node", "central_node.ref");
	Ocarina_Config::AADL_Files =>
		("central_control_unit.aadl","common.aadl");
	Ocarina_Config::Generator => polyorb_hi_c;
	Ocarina_Config::Needed_Property_Sets =>
		(value (Ocarina_Config::Data_Model),
		value (Ocarina_Config::Deployment),
		value (Ocarina_Config::Cheddar_Properties));
	Ocarina_Config::AADL_Version => AADLv2;
end rover_root;

system implementation rover_root.Impl
end rover_root.Impl;
```

root.aadl

```
package root
public
	with autopilot;
	with external_control_unit;
	with central_control_unit;
------------
-- System --
------------

system rover
end rover;

system implementation rover.impl
subcomponents
	autopilot : system autopilot::autopilot.impl;
	external : system external_control_unit::external.impl;
	central : system central_control_unit::central.impl;
end rover.impl;
```

```
end root;
```

common.aadl

```
package common
public
thread periodic_thread
properties
	Dispatch_Protocol => periodic;
end periodic_thread;

thread aperiodic_thread
properties
	Dispatch_Protocol => aperiodic;
end aperiodic_thread;

end common;
```

external_control_unit.aadl

```
package external_control_unit
public
------------
-- System --
------------

system external
end external;

system implementation external.impl
end external.impl;

end external_control_unit;
```

central_control_unit.aadl

```
package central_control_unit
public
	with deployment;
	with common;

---------------------------
-- Hardware components --
---------------------------

data camera_data
end camera_data;

data implementation camera_data.impl
end camera_data.impl;
device camera_device
features
	video_input: in data port camera_data;
	video_output: out data port camera_data;
end camera_device;

data gps_data
end gps_data;

data implementation gps_data.impl
end gps_data.impl;
device gps_device
features
	location_input: in data port gps_data;
	location_output: out data port gps_data;
end gps_device;

------------------
-- Subprograms --
------------------

subprogram camera_spg
properties
	source_language => (C);
	source_name => "camera_spg";
	source_text => ("camera.c");
```

```
end camera_spg;

subprogram gps_spg
properties
	source_language => (C);
	source_name => "gps_spg";
	source_text => ("gps.c");
end gps_spg;

-------------
-- Threads --
-------------

thread camera_thread extends common::periodic_thread
end camera_thread;

thread gps_thread extends common::periodic_thread
end gps_thread;


thread implementation camera_thread.impl
calls
	c : {
		s : subprogram camera_spg;
	};
properties
	Period => 1 sec;
	Priority => 100;
end camera_thread.impl;

thread implementation gps_thread.impl
calls
	c : {
		s : subprogram gps_spg;
	};
properties
	Period => 1 sec;
	Priority => 200;
end gps_thread.impl;

---------------
-- Processor --
---------------
```

```
processor cpu
properties
	Deployment::Execution_Platform => native;
end cpu;

processor implementation cpu.impl
properties
	Scheduling_Protocol => (Posix_1003_Highest_Priority_First_Protocol);
end cpu.impl;

---------------
-- Processes --
---------------

process central_software
end central_software;

process implementation central_software.impl
subcomponents
	camera : thread camera_thread.impl;
	gps : thread gps_thread.impl;
end central_software.impl;

------------
-- System --
------------

system central
end central;

system implementation central.impl
subcomponents
	software : process central_software.impl;
	ark : processor cpu.impl;
properties
	Actual_Processor_Binding => (reference (ark)) applies to software;
end central.impl;


end central_control_unit;
```

autopilot.aadl

```
package autopilot
public
	with deployment;
	with common;

-------------------------
-- Hardware components --
-------------------------

data temperature_data
end temperature_data;

data implementation temperature_data.impl
end temperature_data.impl;
device temperature_device
features
	temperature_input: in data port temperature_data;
	temperature_output: out data port temperature_data;
end temperature_device;

data magnetometer_data
end magnetometer_data;

data implementation magnetometer_data.impl
end magnetometer_data.impl;

device magnetometer_device
features
	magnetometer_input: in data port magnetometer_data;
	magnetometer_output: out data port magnetometer_data;
end magnetometer_device;

data tube_sensor_data
end tube_sensor_data;

data implementation tube_sensor_data.impl
end tube_sensor_data.impl;
device tube_sensor_device
features
	tube_input: in data port tube_sensor_data;
	tube_output: out data port tube_sensor_data;
end tube_sensor_device;
```

```
data pressure_data
end pressure_data;

data implementation pressure_data.impl
end pressure_data.impl;

device pressure_device
features
	pressure_input: in data port pressure_data;
	pressure_output: out data port pressure_data;
end pressure_device;

data lights_data
end lights_data;

data implementation lights_data.impl
end lights_data.impl;

device lights_device
features
	lights_input: in data port lights_data;
	lights_output: out data port lights_data;
end lights_device;

data power_sensing_module_data
end power_sensing_module_data;

data implementation power_sensing_module_data.impl
end power_sensing_module_data.impl;
device power_sensing_module_device
features
	power_input: in data port power_sensing_module_data;
	power_output: out data port power_sensing_module_data;
end power_sensing_module_device;

data esc_data
end esc_data;

data implementation esc_data.impl
end esc_data.impl;

device esc_device
features
```

```
        esc_input: in data port esc_data;
        esc_output: out data port esc_data;
end esc_device;

data thrusters_data
end thrusters_data;

data implementation thrusters_data.impl
end thrusters_data.impl;

device thrusters_device
features
        thrusters_input: in data port thrusters_data;
        thrusters_output: out data port thrusters_data;
end thrusters_device;


-----------------
-- Subprograms --
-----------------

subprogram temperature_spg
properties
        source_language => (C);
        source_name => "temperature_spg";
        source_text => ("temperature.c");
end temperature_spg;

subprogram magnetometer_spg
properties
        source_language => (C);
        source_name => "magnetometer_spg";
        source_text => ("magnetometer.c");
end magnetometer_spg;

subprogram tube_sensor_spg
properties
        source_language => (C);
        source_name => "tube_sensor_spg";
        source_text => ("tube_sensor.c");
end tube_sensor_spg;

subprogram pressure_spg
```

```
properties
	source_language => (C);
	source_name => "pressure_spg";
	source_text => ("pressure.c");
end pressure_spg;

subprogram lights_spg
properties
	source_language => (C);
	source_name => "lights_spg";
	source_text => ("lights.c");
end lights_spg;

subprogram power_sensing_module_spg
properties
	source_language => (C);
	source_name => "power_sensing_module_spg";
	source_text => ("power_sensing_module.c");
end power_sensing_module_spg;

subprogram esc_spg
properties
	source_language => (C);
	source_name => "esc_spg";
	source_text => ("propulsion.c");
end esc_spg;

subprogram thrusters_spg
properties
	source_language => (C);
	source_name => "thrusters_spg";
	source_text => ("propulsion.c");
end thrusters_spg;

-------------
-- Threads --
-------------


thread temperature_thread extends common::periodic_thread
end temperature_thread;

thread magnetometer_thread extends common::periodic_thread
```

```
end magnetometer_thread;

thread tube_sensor_thread extends common::periodic_thread
end tube_sensor_thread;

thread pressure_thread extends common::periodic_thread
end pressure_thread;

thread lights_thread extends common::periodic_thread -- should be "aperiodic_thread" (!?), but
Ocarina returns an error
end lights_thread;

thread power_sensing_module_thread extends common::periodic_thread -- should be
"aperiodic_thread" (!?), but Ocarina returns an error
end power_sensing_module_thread;

thread esc_thread extends common::periodic_thread
end esc_thread;

thread thrusters_thread extends common::periodic_thread
end thrusters_thread;


thread implementation temperature_thread.impl
calls
	c : {
		s : subprogram temperature_spg;
	};
properties
	Period => 1 sec;
	Priority => 100;
end temperature_thread.impl;

thread implementation magnetometer_thread.impl
calls
	c : {
		s : subprogram magnetometer_spg;
	};
properties
	Period => 1 sec;
	Priority => 50;
end magnetometer_thread.impl;

thread implementation tube_sensor_thread.impl
```

```
calls
	c : {
		s : subprogram tube_sensor_spg;
	};
properties
	Period => 1 sec;
	Priority => 50;
end tube_sensor_thread.impl;

thread implementation pressure_thread.impl
calls
	c : {
		s : subprogram pressure_spg;
	};
properties
	Period => 1 sec;
	Priority => 40;
end pressure_thread.impl;

thread implementation lights_thread.impl
calls
	c : {
		s : subprogram lights_spg;
	};
properties -- Should be removed!?
	Period => 1 sec; -- Here because it is
	Priority => 10; -- periodic temporarily!
end lights_thread.impl;

thread implementation power_sensing_module_thread.impl
calls
	c : {
		s : subprogram power_sensing_module_spg;
	};
properties -- Should be removed!?
	Period => 1 sec; -- Here because it is
	Priority => 85; -- periodic temporarily!
end power_sensing_module_thread.impl;

thread implementation esc_thread.impl
calls
	c : {
```

```
                s : subprogram esc_spg;
        };
properties
        Period => 1 sec;
        Priority => 150;
end esc_thread.impl;

thread implementation thrusters_thread.impl
calls
        c : {
                s : subprogram thrusters_spg;
        };
properties
        Period => 1 sec;
        Priority => 155;
end thrusters_thread.impl;

---------------
-- Processor --
---------------

processor cpu
properties
        Deployment::Execution_Platform => native;
end cpu;

processor implementation cpu.impl
properties
        Scheduling_Protocol => (Posix_1003_Highest_Priority_First_Protocol);
end cpu.impl;

---------------
-- Processes --
---------------

process autopilot_software
end autopilot_software;

process implementation autopilot_software.impl
subcomponents
        temperature : thread temperature_thread.impl;
        magnetometer : thread magnetometer_thread.impl;
        tube_sensor : thread tube_sensor_thread.impl;
```

```
        pressure : thread pressure_thread.impl;
        lights : thread lights_thread.impl;
        power_sensing_module : thread power_sensing_module_thread.impl;
        esc : thread esc_thread.impl;
        thrusters : thread thrusters_thread.impl;
end autopilot_software.impl;

------------
-- System --
------------

system autopilot
end autopilot;

system implementation autopilot.impl
subcomponents
        software : process autopilot_software.impl;
        cpu : processor cpu.impl;
properties
        Actual_Processor_Binding => (reference (cpu)) applies to software;
end autopilot.impl;

end autopilot;
```

Sensor files

camera.c

```
// Blue Robotics Low-Light HD USB
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>

#include <fcntl.h>
#include <errno.h>



int camera_init();
int captureImage();
int processImage();
int sendImage();

void camera_spg()
{
// Initialize camera
if (camera_init() == -1)
{
fprintf(stderr, "Camera initialization failed\n");
return;
}

// Capture image
if (captureImage() == -1)
{
fprintf(stderr, "Image capture failed\n");
return;
}

// Process image
if (processImage() == -1)
{
fprintf(stderr, "Image processing failed\n");
return;
}
```

```c
// Send image to ARK computer
if (sendImage() == -1)
{
fprintf(stderr, "Image sending failed\n");
return;
}

printf("Image successfully captured, processed, and sent to the ARK computer!\n");
}

// Function to initialize the camera
int camera_init()
{
// Code to initialize the camera (if required)
// Return 0 on success, -1 on failure
}

// Function to capture an image
int captureImage()
{
// Code to capture an image using the camera
// Return 0 on success, -1 on failure
}

// Function to process the captured image
int processImage()
{
// Code to process the captured image
// Return 0 on success, -1 on failure
}

// Function to send the processed image to the ARK computer
int sendImage()
{
// Code to send the processed image to the ARK computer
// Return 0 on success, -1 on failure
}
```

gps.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>

#include <fcntl.h>
#include <errno.h>



int gps_init();
int readGPSData();
int processGPSData();
int sendGPSData();

void gps_spg()
{
// Initialize GPS
if (gps_init() == -1)
{
fprintf(stderr, "GPS initialization failed\n");
return;
}

// Read GPS data
if (readGPSData() == -1)
{
fprintf(stderr, "GPS data read failed\n");
return;
}

// Process GPS data
if (processGPSData() == -1)
{
fprintf(stderr, "GPS data processing failed\n");
return;
}

// Send GPS data to ARK computer
if (sendGPSData() == -1)
{
```

```
fprintf(stderr, "GPS data sending failed\n");
return;
}

printf("GPS data successfully read, processed, and sent to the ARK computer!\n");
}

int gps_init()
{
// Code to initialize the GPS device
// Return 0 on success, -1 on failure
}

int readGPSData()
{
// Code to read GPS data from the device
// Return 0 on success, -1 on failure
}

int processGPSData()
{
// Code to process the GPS data
// Return 0 on success, -1 on failure
}

int sendGPSData()
{
// Code to send the GPS data to the ARK computer
// Return 0 on success, -1 on failure
}
```

lights.c

```c
// LUMEN SUBSEA LIGHTS
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>
#include <gpiod.h>

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening
#define OUTPUT 1 // voltage/ground
#define HIGH 1 // 5V or logical 1
#define LOW 0

int ledPin1 = 33;
int ledPin2 = 35;
int lights_udp_socket;
struct sockaddr_in server_address;
struct gpiod_chip* chip;
struct gpiod_line* line;

void lights_init();
void update_led_state(int value);
void lightsPinMode(int pin, int mode);
void lightsDigitalWrite(int pin, int value);

void lights_spg()
{
lights_init();

// Read LED control value from Pixhawk
// ...

int ledControlValue = 1111; // Replace with the value received from Pixhawk

// Perform LED control based on the received value
update_led_state(ledControlValue);

// Send the LED control value back to Pixhawk
sendto(lights_udp_socket, &ledControlValue, sizeof(ledControlValue), 0, (struct sockaddr*)
```

```
&server_address, sizeof(server_address));

close(lights_udp_socket);
}

void lights_init()
{
// Create socket
lights_udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
if (lights_udp_socket == -1) {
perror("Socket creation failed!");
exit(-1);
}

// Set server address
memset(&server_address, 0, sizeof(server_address));
server_address.sin_family = AF_INET;
server_address.sin_port = htons(SERVER_PORT);
if (inet_pton(AF_INET, SERVER_IP, &(server_address.sin_addr)) <= 0)
{
perror("Invalid address!");
exit(-1);
}

// Initialize LED pins for GPIO?
// ...

// Initialize LED pins - commented because of the library used
// lightsPinMode(ledPin1, OUTPUT);
// lightsPinMode(ledPin2, OUTPUT);
// lightsDigitalWrite(ledPin1, HIGH);
// lightsDigitalWrite(ledPin2, HIGH);
}

void update_led_state(int value)
{
switch (value) // commented because of the library used
{
case 1234: // defined as such in the original code
// lightsDigitalWrite(ledPin1, LOW);
// lightsDigitalWrite(ledPin2, LOW);
break;
case 1111: // defined as such in the original code
```

```
// lightsDigitalWrite(ledPin1, HIGH);
// lightsDigitalWrite(ledPin2, HIGH);
break;
// Add more cases if needed

default:
// Handle unsupported values or errors
break;
}
}

// IMPORTANT!!! I used libgpiod library, so it has to be installed first: sudo apt-get install libgpiod-
dev
/*void lightsPinMode(int pin, int mode) {
int temp;

// Open the GPIO chip
chip = gpiod_chip_open("/dev/gpiochip0");
if (!chip) {
perror("Failure opening GPIO chip!");
exit(-1);
}

// Get the GPIO line
line = gpiod_chip_get_line(chip, pin);
if (!line) {
perror("Failure getting GPIO line!");
exit(-1);
}

// Set the pin direction
if (mode == OUTPUT) {
temp = gpiod_line_request_output(line, "my-output", 0);
} else {
temp = gpiod_line_request_input(line, "my-input");
}

if (temp < 0) {
perror("Failure setting pin mode!");
exit(-1);
}
}
```

```
void lightsDigitalWrite(int pin, int value)
{
// Set the pin state
int temp;
if (value == HIGH) {
temp = gpiod_line_set_value(line, 1);
} else {
temp = gpiod_line_set_value(line, 0);
}

if (temp < 0) {
perror("Failure setting pin value!");
exit(-1);
}
}*/
```

magnetometer.c

```
// HMC5883L SENSOR
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/fcntl.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>

// #include <px4_platform_common/px4_config.h>
// #include <px4_platform_common/log.h>
// #include <px4_platform_common/tasks.h>
// #include <drivers/drv_hrt.h>
#define I2C_SLAVE 0x0703 // Dummy value for testing

#define UDP_PORT 5000

// extern "C" __EXPORT int hmc5883l_main(int argc, char *argv[]);

int mag_i2c_file;
int mag_udp_socket;
struct sockaddr_in mag_destAddr;

void magnetometer_init();

void magnetometer_spg()
{

// Initialize HMC5883L
magnetometer_init();
// Sleep is not necessary, will be done by AADL
// px4_usleep(1000);
// sleep(1);

// Read 6 bytes of data from register(0x03)
// xMag msb, xMag lsb, zMag msb, zMag lsb, yMag msb, yMag lsb
char reg = 0x03;
```

```
char data[6] = {0};
// uint8_t reg = 0x03;
// uint8_t data[6] = {0};
if (write(mag_i2c_file, &reg, 1) != 1 || read(mag_i2c_file, data, 6) != 6) {
perror("Error reading from the I2C device");
// PX4_ERR("Error reading from the I2C device");
// return -1;
exit(1);
}

// Convert the data
// int16_t xMag
int xMag = (data[0] * 256 + data[1]);
if (xMag > 32767) {
xMag -= 65536;
}
// int16_t zMag
int zMag = (data[2] * 256 + data[3]);
if (zMag > 32767) {
zMag -= 65536;
}
// int16_t yMag
int yMag = (data[4] * 256 + data[5]);
if (yMag > 32767) {
yMag -= 65536;
}

// Format the sensor data as a string
char sensorData[50];
snprintf(sensorData, sizeof(sensorData), "X:%d, Y:%d, Z:%d", xMag, yMag, zMag);

// Send the sensor data over UDP
if (sendto(mag_udp_socket, sensorData, strlen(sensorData), 0, (struct sockaddr*)&mag_destAddr,
sizeof(mag_destAddr)) < 0) {
perror("Error sending data");
// PX4_ERR("Error sending data");
// return -1;
exit(1);
}

// Close the socket and I2C file
close(mag_udp_socket);
```

```
close(mag_i2c_file);
}

void magnetometer_init()
{
// Create socket
if ((mag_udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
perror("Error creating the socket!");
// PX4_ERR("Error creating socket!");
// return -1;
exit(1);
}

// Set destination address
memset(&mag_destAddr, 0, sizeof(mag_destAddr));
mag_destAddr.sin_family = AF_INET;
mag_destAddr.sin_port = htons(UDP_PORT);
if (inet_pton(AF_INET, "PIXHAWK_IP_ADDRESS", &(mag_destAddr.sin_addr)) <= 0) // Replace with
real address
{
perror("Error connecting to the address!");
// PX4_ERR("Error setting address!");
// return -1;
exit(1);
}

// Create I2C bus
char *bus = "/dev/i2c-1"; // Check this path!
if ((mag_i2c_file = open(bus, O_RDWR)) < 0)
{
perror("Error creating the bus!");
// PX4_ERR("Error creating the bus!");
// return -1;
exit(1);
}

// Get I2C device, HMC5883 I2C address is 0x1E(30)
if (ioctl(mag_i2c_file, I2C_SLAVE, 0x1E) < 0)
{
perror("Error getting the address!");
// PX4_ERR("Error getting the address!");
// return -1;
```

```
exit(1);
}

// Select Configuration register A(0x00)
// Normal measurement configuration, data rate o/p = 0.75 Hz(0x60)
char config[2] = {0x00, 0x60};
if (write(mag_i2c_file, config, 2) != 2)
{
perror("Error writing to the I2C device!");
// PX4_ERR("Error writing to the I2C device!");
// return -1;
exit(1);
}

// Select Mode register(0x02)
// Continuous measurement mode(0x00)
config[0] = 0x02;
config[1] = 0x00;
if (write(mag_i2c_file, config, 2) != 2)
{
perror("Error writing to the I2C device!");
// PX4_ERR("Error writing to the I2C device!");
// return -1;
exit(1);
}
}
```

power_sensor_module.c

```c
// BLUE ROBOTICS PSM
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening

int power_udp_socket;
struct sockaddr_in power_serverAddr;

void power_sensing_module_init();
void readPowerData(float *current, float *voltage);

void power_sensing_module_spg()
{
power_sensing_module_init();

// Read power data
float current = 0.0, voltage = 0.0;
readPowerData(&current, &voltage);

printf("Current: %.2f A\n", current);
printf("Voltage: %.2f V\n", voltage);

// Send power data to Pixhawk
char powerStr[50]; // Modify the size as per your data requirements
sprintf(powerStr, "Current: %.2f A, Voltage: %.2f V", current, voltage);

if (send(power_udp_socket, powerStr, strlen(powerStr), 0) == -1)
perror("Error sending the data!");

close(power_udp_socket);
}

void power_sensing_module_init()
{
// Create socket
```

```
if ((power_udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{ // For TCP, SOCK_STREAM
perror("socket");
exit(1);
}

// Set server address
memset(&power_serverAddr, 0, sizeof(power_serverAddr)); // V1
power_serverAddr.sin_family = AF_INET;
power_serverAddr.sin_port = htons(SERVER_PORT);
if (inet_pton(AF_INET, SERVER_IP, &(power_serverAddr.sin_addr)) <= 0)
{
perror("inet_pton");
exit(1);
}

// Connect to the server
if (connect(power_udp_socket, (struct sockaddr *)&power_serverAddr, sizeof(power_serverAddr)) == -
1)
{
perror("connect");
exit(1);
}
}

void readPowerData(float *current, float *voltage)
{
// Replace the code below with implementation to read the current and voltage values
// GPIO, maybe?
*current = 1.23;
*voltage = 12.34;
}
```

pressure.c

```
        // SENSOR SKU237545
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>
#include <po_hi_time.h>
#include <termios.h>
#include <fcntl.h>

// Define pin for connecting the data wire of the pressure sensor
#define PRESSURE_PIN 12

#define SERVER_PORT 1234 // Replace with the actual port number
#define SERVER_IP_ADDRESS "127.0.0.1" // Replace with the actual IP address

int pressure_udp_socket;
struct sockaddr_in pressure_destAddr;

void pressure_init();
// int analogRead(int);
// int readPressure();

void pressure_spg()
{
pressure_init();

// Read pressure from the SKU237545 pressure sensor
int pressure = 0;
// pressure = readPressure(); // gcc -o your_program pressure.c -lgpiod !!!!IMPORTANT!!!

// Print pressure value to the screen
// printf("Pressure = %d\n", pressure);

// Prepare the sensor data packet
char dataPacket[64]; // Adjust the packet size as needed
sprintf(dataPacket, "Pressure = %d", pressure);

// Send the data packet to Pixhawk
if (send(pressure_udp_socket, dataPacket, strlen(dataPacket), 0) < 0)
```

```
{
perror("Error sending data to Pixhawk!");
exit(1);
}

close(pressure_udp_socket);
}

void pressure_init() {
// Create socket
pressure_udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
if (pressure_udp_socket < 0)
{
perror("Error creating the socket!");
exit(1);
}

// Set server address
memset(&pressure_destAddr, 0, sizeof(pressure_destAddr));
pressure_destAddr.sin_family = AF_INET;
pressure_destAddr.sin_port = htons(SERVER_PORT); // Replace SERVER_PORT with the actual port
number
if (inet_aton(SERVER_IP_ADDRESS, &pressure_destAddr.sin_addr) == 0)
{
perror("Error setting the address!");
exit(1);
}

// Connect to the server
if (connect(pressure_udp_socket, (struct sockaddr *)&pressure_destAddr, sizeof(pressure_destAddr)) <
0)
{
perror("Error connecting to the server!");
exit(1);
}
}

// Function to read analog value from a pin
/*int readPressure()
{
// Initialize GPIO library
struct gpiod_chip* chip = gpiod_chip_open("/dev/gpiochip0"); // Change if necessary
if (!chip) {
```

```
perror("Failed to open GPIO chip!");
exit(1);
}

// Set pin direction to output
struct gpiod_line* line = gpiod_chip_get_line(chip, PRESSURE_PIN);
if (!line) {
perror("Failed to get GPIO line!");
exit(1);
}
if (gpiod_line_request_output(line, "PressureSensor", GPIOD_LINE_ACTIVE_STATE_DEFAULT) < 0) { //
Change if necessary
perror("Failed to set GPIO line direction!");
exit(1);
}

// Send start signal to the sensor
gpiod_line_set_value(line, 0);
usleep(2000); // Wait for 2 milliseconds

// Set pin direction to input
if (gpiod_line_request_input(line, "PressureSensor") < 0) {
perror("Failed to set GPIO line direction!");
exit(1);
}

// Wait for sensor response
int response = 0;
while (gpiod_line_get_value(line) == 1) {
usleep(1);
response++;
if (response > 100) {
perror("Sensor failed to respond!");
exit(1);
}
}
// Read analog value from the sensor
int analogValue = analogRead(PRESSURE_PIN); // Replace with the appropriate function to read
analog value

// Calculate pressure value based on analog reading
// Assuming offset = 94 and maxReading = 920 (as written in documentation)
float pressure = ((analogValue - 94) * 1.2) / (920 - 94);
```

```
// Cleanup GPIO resources
gpiod_chip_close(chip);

return (int)(pressure * 1000); // Return pressure in millipascals (mPa)
}

// Function to read analog value from a pin
int analogRead(int pin) {
// TODO: ADC conversion
int analogValue = 512;



return analogValue;
}*/
```

propulsion.c

```c
// T200
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <time.h>
#include <po_hi_time.h>

#define ESC_SERVO_PIN 17 // Defined as such in the original code

// Macros for easier code reading
#define DRIVE_INPUT 0
#define INPUT 0
#define DRIVE_OUTPUT 1
#define OUTPUT 1

// T200 Thruster Pins
#define THRUSTER_1_PIN 2
#define THRUSTER_2_PIN 3
#define THRUSTER_3_PIN 5
#define THRUSTER_4_PIN 6
#define THRUSTER_5_PIN 7

int signal = 1700; // Defined as such in the original code

int thruster;
int speed;

void esc_init(int pin, int mode);
void escWriteMicroseconds(int pin, int value);
void escPinMode(uint8_t pin, uint8_t mode);

void thrusters_init();

void esc_spg()
{
// Configure the servo pin
esc_init(ESC_SERVO_PIN, OUTPUT);
// Send stop signal to ESC
escWriteMicroseconds(ESC_SERVO_PIN, 1500);
```

```
// Define the value of the signal, ranging from 1100 to 1900; 1700 by default
signal = 1700;

// Send the signal to the ESC
escWriteMicroseconds(ESC_SERVO_PIN, signal);

}

void esc_init(int pin, int mode)
{
if (mode == DRIVE_INPUT)
{
// Implementation specific to the hardware for input mode
// Configure the pin as an input for receiving PWM signal from Pixhawk
// Initialize the necessary communication protocol for receiving PWM signals
// Set up the appropriate interrupt or polling mechanism to read the PWM signal
// Configure the pin as an input
escPinMode(pin, INPUT);

// Initialize the necessary communication protocol for receiving PWM signals
// For example, if you are using a microcontroller with built-in PWM module, you would configure it here

// Set up the appropriate interrupt or polling mechanism to read the PWM signal
// Here, you would define the necessary interrupt handlers or polling logic to capture the PWM signal changes
}
else if (mode == DRIVE_OUTPUT) {
// Implementation specific to the hardware for output mode
// Configure the pin as an output for sending PWM signal to ESC
// Initialize the necessary communication protocol for sending PWM signals
// Set up the appropriate timing and duty cycle to generate the PWM signal
// Connect the output pin to the ESC control input
// Configure the pin as an output
escPinMode(pin, OUTPUT);

// Initialize the necessary communication protocol for sending PWM signals
// For example, if you are using a microcontroller with built-in PWM module, you would configure it here

// Set up the appropriate timing and duty cycle to generate the PWM signal
// You would define the necessary code to set the desired timing and duty cycle for the PWM signal
// Connect the output pin to the ESC control input
```

```
// Here, you would connect the output pin to the appropriate ESC control input based on your
hardware configuration
}
}

// In this version, the Servo library is replaced with "escWriteMicroseconds" to control the servo
motor.
void escWriteMicroseconds(int pin, int value) {
// Write code to send a signal specific to the hardware
}

// Implementation of escPinMode function
void escPinMode(uint8_t pin, uint8_t mode) {
// Implementation specific to the platform or library
// Configure the pin mode based on the provided parameters
// This implementation assumes the use of the Arduino framework
// Example implementation using the Arduino framework
// Here, the pin mode is set using the escPinMode function provided by the Arduino library
// The implementation would vary based on the specific platform or library being used
}

void thrusters_spg()
{
thrusters_init();

// Set the speed of the specified thruster
switch (thruster) {
case 1:
escWriteMicroseconds(THRUSTER_1_PIN, speed);
break;
case 2:
escWriteMicroseconds(THRUSTER_2_PIN, speed);
break;
case 3:
escWriteMicroseconds(THRUSTER_3_PIN, speed);
break;
case 4:
escWriteMicroseconds(THRUSTER_4_PIN, speed);
break;
case 5:
escWriteMicroseconds(THRUSTER_5_PIN, speed);
break;
```

```
default:
// Handle unsupported thruster number
break;
}
}

void thrusters_init()
{
// Initialize the ESC pins and modes for each thruster
esc_init(THRUSTER_1_PIN, DRIVE_OUTPUT);
esc_init(THRUSTER_2_PIN, DRIVE_OUTPUT);
esc_init(THRUSTER_3_PIN, DRIVE_OUTPUT);
esc_init(THRUSTER_4_PIN, DRIVE_OUTPUT);
esc_init(THRUSTER_5_PIN, DRIVE_OUTPUT);
}
```

temperature.c

```c
        // DS18B20 SENSOR
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening

int temp_udp_socket;
struct sockaddr_in temp_serverAddr;

int temperature_init();

void temperature_spg()
{
if (temperature_init() == -1)
printf("Initialization failed.\n");
FILE *tempfile = fopen("/sys/bus/w1/devices/28-0000027a4334/w1_slave", "r");
char temptext[100];
fgets(temptext, 100, tempfile);
fclose(tempfile);

char *tempdata = strtok(temptext, "\n");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, "=");

float temperature = atof(tempdata + 2) / 1000.0;
printf("%f\n", temperature);

// Send temperature data to Pixhawk
char temperatureStr[20]; // Is this good enough for ARK?
```

```
sprintf(temperatureStr, "%.2f", temperature);

if (send(temp_udp_socket, temperatureStr, strlen(temperatureStr), 0) == -1)
perror("Error sending the data!");

// sleep(1); // Delay for 1 second
// Will be done in AADL

close(temp_udp_socket); // Close the socket

}

int temperature_init()
{
// Create socket
if ((temp_udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) == -1) // For TCP, SOCK_STREAM
{
perror("socket");
return -1;
}

// Set server address
memset(&temp_serverAddr, 0, sizeof(temp_serverAddr)); // V1
temp_serverAddr.sin_family = AF_INET;
temp_serverAddr.sin_port = htons(SERVER_PORT);
// temp_serverAddr.sin_addr.s_addr = inet_addr(SERVER_IP); // For IPv4
// memset(temp_serverAddr.sin_zero, '\0', sizeof(temp_serverAddr.sin_zero)); // V2
if (inet_pton(AF_INET, SERVER_IP, &(temp_serverAddr.sin_addr)) <= 0)
{
perror("inet_pton");
return -1;
}

// Connect to the server
if (connect(temp_udp_socket, (struct sockaddr *)&temp_serverAddr, sizeof(temp_serverAddr)) == -1)
{
perror("connect");
return -1;
}

return 0;
}
```

tube_sensor.c

```c
// DS18B20 SENSOR
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#include <arpa/inet.h>
#include <sys/socket.h>

#define SERVER_IP "192.168.0.100" // IP address of the Pixhawk
#define SERVER_PORT 5000 // Port number on which Pixhawk is listening

int temp_udp_socket;
struct sockaddr_in temp_serverAddr;

int temperature_init();

void temperature_spg()
{
if (temperature_init() == -1)
printf("Initialization failed.\n");
FILE *tempfile = fopen("/sys/bus/w1/devices/28-0000027a4334/w1_slave", "r");
char temptext[100];
fgets(temptext, 100, tempfile);
fclose(tempfile);

char *tempdata = strtok(temptext, "\n");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, " ");
tempdata = strtok(NULL, "=");

float temperature = atof(tempdata + 2) / 1000.0;
printf("%f\n", temperature);

// Send temperature data to Pixhawk
char temperatureStr[20]; // Is this good enough for ARK?
```

```c
sprintf(temperatureStr, "%.2f", temperature);

if (send(temp_udp_socket, temperatureStr, strlen(temperatureStr), 0) == -1)
perror("Error sending the data!");

// sleep(1); // Delay for 1 second
// Will be done in AADL

close(temp_udp_socket); // Close the socket

}

int temperature_init()
{
// Create socket
if ((temp_udp_socket = socket(AF_INET, SOCK_DGRAM, 0)) == -1) // For TCP, SOCK_STREAM
{
perror("socket");
return -1;
}

// Set server address
memset(&temp_serverAddr, 0, sizeof(temp_serverAddr)); // V1
temp_serverAddr.sin_family = AF_INET;
temp_serverAddr.sin_port = htons(SERVER_PORT);
// temp_serverAddr.sin_addr.s_addr = inet_addr(SERVER_IP); // For IPv4
// memset(temp_serverAddr.sin_zero, '\0', sizeof(temp_serverAddr.sin_zero)); // V2
if (inet_pton(AF_INET, SERVER_IP, &(temp_serverAddr.sin_addr)) <= 0)
{
perror("inet_pton");
return -1;
}

// Connect to the server
if (connect(temp_udp_socket, (struct sockaddr *)&temp_serverAddr, sizeof(temp_serverAddr)) == -1)
{
perror("connect");
return -1;
}

return 0;
}
```