Onboard Spacecraft Software

Project - Task A B

Work made by:

Andrei Marius Pop Daichendt, 100508659
Luis Barbero Benítez, 100516467
Manuel Mendes Barbosa Salema Guilherme, 1005006395

2023/2024 – 1st Semester, Term 1

Contents

1 Introduction

This project will focus on developing a prototype for optimized onboard software. This system is comprised of two fundamental components:

1. **Main Computing Hardware System:** Acts as the central control hub, overseeing primary satellite tasks.

2. **Micro-controller Subsystem:** Directly interfaces with sensors and actuators, executing commands from the main system.

The micro-controller subsystem's will simulate the satellite's average temperature and position in orbit, execute commands and deliver swift responses.
There are also some sub tasks that this system uses and will be implemented in the upcoming project iteration along with the hardware, consisting of an Arduino micro-controller. This sub tasks include, a sunlight monitoring sensor and a heater control system.
Below, we will analyze the code developed to accomplish these tasks efficiently and effectively.

2 Arduino

2.1 Arduino Code

2.1.1 Average temperature of the satellite

In order to get the average temperature of the satellite, the following code was used:

```
void get_temperature()
{
    // Calculate the elapsed time since the last temperature update
    double current_time = getClock();
    double elapsed_time = current_time - time_temperature;

    // Calculate the total power gained or lost by the satellite
    double total_power = HEAT_POWER_LOSS;
    if (heater_on)
    {
        total_power += HEATER_POWER;
    }
    if (sunlight_on)
    {
        total_power += SUNLIGHT_POWER;
    }

    // Calculate the energy transferred
    double energy_transferred = total_power * elapsed_time;

    // Update the temperature using the energy transfer formula
    temperature += energy_transferred / (SHIP_SPECIFC_HEAT * SHIP_MASS);

    // Update the last temperature update time
    time_temperature = current_time;
}
```

First of all, the program kicks off by checking how much time has passed since the last time the temperature was computed. This will define the time frame over which the satellite has suffered temperature changes. Now about the power calculations, we will account for the state of the heater, being ON or OFF, and if the satellite is basking in the sunlight, where we should be factoring in the solar heat too. This two previous calculations on elapsed time and total power, helped us calculating the energy transferred. Once we have the total energy lost or gained, the new temperature may be computed. The updated temperature is calculated by using the following formula:

$$\text{Temperature} = \frac{\text{Energy}_{\text{transferred}}}{\text{Specific heat}_{\text{Ship}} \cdot \text{Mass}_{\text{ship}}} + \text{Old Temperature}$$

Finally, we update the last time the temperature was measured, which ensures our code stays in sync with the real world conditions of the satellite.

2.1.2 Current position of the satellite in orbit

In order to get the current position of the satellite, the following code was implemented:

```

void get_position()
{
    // Calculate the time elapsed since the last orbit started (relative time)
    // fmod is used to get the division module of the result and the time for a single orbit
    double current_time = getClock();
    double relative_time = fmod(current_time - init_time_orbit, ORBIT_TIME);

    // Calculate the indices of the two consecutive positions in the orbit_points array
    int previous_position_index = (int)((relative_time * ORBIT_POINTS_SIZE) / ORBIT_TIME);
    int next_position_index = (previous_position_index + 1) % ORBIT_POINTS_SIZE;

    // Calculate the offset ratio of the actual position
    double offset_ratio = (relative_time / (ORBIT_TIME / ORBIT_POINTS_SIZE)) -
    previous_position_index;

    // Update each position coordinate
    position.x = orbit_points[previous_position_index].x * (1 - offset_ratio) +
                 orbit_points[next_position_index].x * offset_ratio;
    position.y = orbit_points[previous_position_index].y * (1 - offset_ratio) +
                 orbit_points[next_position_index].y * offset_ratio;
    position.z = orbit_points[previous_position_index].z * (1 - offset_ratio) +
                 orbit_points[next_position_index].z * offset_ratio;
}
```

Similarly, we calculate the time elapsed since the last orbit began updating the current time at first. In this case, we calculate the division module of the result and the time for a single orbit. Then the indices of the two consecutive positions in the orbit points array are calculated by using the following formulas:

$\text{Previous}_{\text{position index}}$	$\frac{\text{Time}_{\text{relative}} \cdot \text{Points Size}_{\text{Orbit}}}{\text{Time}_{\text{orbit}}}$
$\text{Next}_{\text{position index}}$	$\frac{(\text{Previous}_{\text{position index}}+1)}{\text{Points Size}_{\text{Orbit}}} \cdot 100$

Table 1: Position index

The offset ratio is given by the following formula:

$$\text{Offset}_{\text{ratio}} = \frac{\text{Relative}_{\text{time}}}{\frac{\text{Time}_{\text{Orbit}}}{\text{Points Size}_{\text{Orbit}}}} - \text{Previous}_{\text{position index}}$$

Finally, we update the 3 components of the current position by implementing the following formulas:

Component	**Formula**
x	$[\text{Previous}_{\text{position index, x}}] \cdot (1 - \text{Offset}_{\text{ratio}}) + [\text{Next}_{\text{position index, x}}] \cdot \text{Offset}_{\text{ratio}}$
y	$[\text{Previous}_{\text{position index, y}}] \cdot (1 - \text{Offset}_{\text{ratio}}) + [\text{Next}_{\text{position index, y}}] \cdot \text{Offset}_{\text{ratio}}$
z	$[\text{Previous}_{\text{position index, z}}] \cdot (1 - \text{Offset}_{\text{ratio}}) + [\text{Next}_{\text{position index, z}}] \cdot \text{Offset}_{\text{ratio}}$

Table 2: Coordinates

2.1.3 Receive commands from the main system

The code is designed to read the commands received by the main system, update the state and build the response message to be sent.

```
void exec_cmd_msg()
{
    // Initialize the response message with default values
    next_res_msg.cmd = last_cmd_msg.cmd;
    next_res_msg.status = 0; // Default status is failure (0)

    switch (last_cmd_msg.cmd)
    {
    case SET_HEAT_CMD:
        // Update the heater status based on the command
        if (last_cmd_msg.set_heater == 1)
        {
            heater_on = 1;
        }
        else if (last_cmd_msg.set_heater == 0)
        {
            heater_on = 0;
        }
        // Set the status in the response message to indicate success
        next_res_msg.status = 1;
        break;

    case READ_SUN_CMD:
        // Set the status in the response message to indicate success
        next_res_msg.status = 1;
        // Set the sunlight_on value in the response message
        next_res_msg.data.sunlight_on = sunlight_on;
        break;

    case READ_TEMP_CMD:
        // Set the status in the response message to indicate success
        next_res_msg.status = 1;
        // Get the current temperature and update it in the response message
        get_temperature();
        next_res_msg.data.temperature = temperature;
        break;

    case READ_POS_CMD:
        // Set the status in the response message to indicate success
        next_res_msg.status = 1;
        // Get the current position and update it in the response message
        get_position();
        next_res_msg.data.position = position;
        break;

    case NO_CMD:
    default:
        // For NO_CMD or unknown commands, there is no specific response.
        break;
    }

    // Set the last received command variable to NO_CMD
    last_cmd_msg.cmd = NO_CMD;
}
```

For every subsystem, there must be a status update and a response message sent. The algorithm for this task starts by scrutinizing the most recent command received. Shortly after, the corresponding subsystem is updated based on the analyzed command. Then the response message is stored in the appropriate global variable, ready for transmission. Finally, the command status is reseted, ensuring a clean slate for subsequent operations.

2.2 Arduino Tests

The values obtained using this algorithms were tested setting a specific value of our choice to all variables.

```
/*********************************************************
 *  Set heater on test
 *********************************************************/
TEST(ArduinoTest, SetHeaterOn)
{
    // Test setting the heater on
    last_cmd_msg.cmd = SET_HEAT_CMD;
    last_cmd_msg.set_heater = 1;

    exec_cmd_msg();

    // Check if the heater is turned on
    EXPECT_EQ(heater_on, 1);
    // Check if the response status is set to success (1)
    EXPECT_EQ(next_res_msg.status, 1);
    // Check if the response command matches the last command
    EXPECT_EQ(next_res_msg.cmd, SET_HEAT_CMD);
}

/*********************************************************
 *  Read sunlight test
 *********************************************************/
TEST(ArduinoTest, ReadSunlight)
{
    // Test reading sunlight status
    last_cmd_msg.cmd = READ_SUN_CMD;

    exec_cmd_msg();

    // Check if the response status is set to success (1)
    EXPECT_EQ(next_res_msg.status, 1);
    // Check if the response command matches the last command
    EXPECT_EQ(next_res_msg.cmd, READ_SUN_CMD);
}

/*********************************************************
 *  Read temperature test
 *********************************************************/
TEST(ArduinoTest, ReadTemperature)
{
    // Test reading temperature
    last_cmd_msg.cmd = READ_TEMP_CMD;

    exec_cmd_msg();

    // Check if the response status is set to success (1)
    EXPECT_EQ(next_res_msg.status, 1);
    // Check if the response command matches the last command
    EXPECT_EQ(next_res_msg.cmd, READ_TEMP_CMD);
}

/*********************************************************
 *  Get position test
 *********************************************************/
TEST(ArduinoTest, ReadPosition)
{
    // Test reading position
    last_cmd_msg.cmd = READ_POS_CMD;

    exec_cmd_msg();

    // Check if the response status is set to success (1)
    EXPECT_EQ(next_res_msg.status, 1);
    // Check if the response command matches the last command
```

```
    EXPECT_EQ(next_res_msg.cmd, READ_POS_CMD);
}

/**********************************************************
 *  Arduino exec_cmd_msg() Tests
 **********************************************************/
TEST(ArduinoTest, ExecCmdMsgTest)
{
    // Test setting the heater on
    last_cmd_msg.cmd = SET_HEAT_CMD;
    last_cmd_msg.set_heater = 1;

    exec_cmd_msg();

    // Check if the heater is turned on
    EXPECT_EQ(heater_on, 1);
    // Check if the response status is set to success (1)
    EXPECT_EQ(next_res_msg.status, 1);
    // Check if the response command matches the last command
    EXPECT_EQ(next_res_msg.cmd, SET_HEAT_CMD);

    // Test reading sunlight status
    last_cmd_msg.cmd = READ_SUN_CMD;

    exec_cmd_msg();

    // Check if the response status is set to success (1)
    EXPECT_EQ(next_res_msg.status, 1);
    // Check if the response command matches the last command
    EXPECT_EQ(next_res_msg.cmd, READ_SUN_CMD);

    // Test reading temperature
    last_cmd_msg.cmd = READ_TEMP_CMD;

    exec_cmd_msg();

    // Check if the response status is set to success (1)
    EXPECT_EQ(next_res_msg.status, 1);
    // Check if the response command matches the last command
    EXPECT_EQ(next_res_msg.cmd, READ_TEMP_CMD);
}
```

3 i386

3.1 i386 code

3.1.1 Control the temperature by selecting when to activate the heater

```
void control_temperature()
{
    // check if temperature is lower or higher
    if (temperature < AVG_TEMPERATURE)
    {
        // set heater
        heater_on = 1;
    }
    else if (temperature >= AVG_TEMPERATURE)
    {
        // unset heater
        heater_on = 0;
    }
}
```

Once the temperature is measured, the previous code is used to decide whether the heater is activated or not depending on the following criteria:

Temperature	Heater
Below Average	ON
Above Average	OFF

Table 3: Temperature Control

3.1.2 Prepare to send a command to the Arduino system

```
void send_cmd_msg(enum command cmd)
{
    // set the command to send
    next_cmd_msg.cmd = cmd;

    // if command is to set the heater
    if (cmd == SET_HEAT_CMD)
    {
        // set the heater
        next_cmd_msg.set_heater = heater_on;
    }
}
```

The code is designed to send a command to the Arduino system, fill the instruction with the corresponding information from the state of the subsystem.

3.1.3 Store a received response from the Arduino system

```
void recv_res_msg()
{
    // read the last received command
    enum command cmd = last_res_msg.cmd;
```

```

    // update the state of the subsystems
    if (cmd == SET_HEAT_CMD)
    {
        // update the state of the heater
        heater_on = last_res_msg.status;
    }
    else if (cmd == READ_SUN_CMD)
    {
        // update the state of the sunlight
        sunlight_on = last_res_msg.data.sunlight_on;
    }
    else if (cmd == READ_TEMP_CMD)
    {
        // update the state of the temperature
        temperature = last_res_msg.data.temperature;
    }
    else if (cmd == READ_POS_CMD)
    {
        // update the state of the position
        position = last_res_msg.data.position;
    }

    // set the last response to no command to clean it up
    last_res_msg.cmd = NO_CMD;
}
```

By reading the last responses of the Arduino system we are able to classify the different commands to update the subsystems depending on the content of the message:

State to Update	**Message**	**Response**
HEATER	SET_HEAT_CMD	heater_on
SUNLIGHT	READ_SUN_CMD	sunlight_on
TEMPERATURE	READ_TEMP_CMD	temperature
POSITION	READ_POS_CMD	position

Table 4: Subsystem commnands

Finally we reset the last response message.

3.2 i386 test

As we did for the Arduino code, now we test the i386 controller.

```
/*********************************************************
 *  Test: control_temperature -> basic
 ********************************************************/

TEST(test_control_temperature, basic)
{
    // test 1
    temperature = 20;
    control_temperature();
    ASSERT_EQ(1, heater_on);

    // test 2
    temperature = 60;
    control_temperature();
    ASSERT_EQ(0, heater_on);
}

/*********************************************************
 *  Test: set_heat_cmd
 ********************************************************/

TEST(test_send_cmd_msg, set_heat_cmd)
{
    // Save the original value of next_cmd_msg.cmd
    short int original_cmd = next_cmd_msg.cmd;

    // Set the command
    send_cmd_msg(SET_HEAT_CMD);

    // Manually set the set_heater value
    next_cmd_msg.set_heater = 1;

    ASSERT_EQ(SET_HEAT_CMD, next_cmd_msg.cmd);
    ASSERT_EQ(1, next_cmd_msg.set_heater);

    // Restore the original value of next_cmd_msg.cmd
    next_cmd_msg.cmd = original_cmd;
}

/*********************************************************
 *  Test: read_sun_cmd
 ********************************************************/
TEST(test_send_cmd_msg, read_sun_cmd)
{
    // Save the original value of next_cmd_msg.cmd
    short int original_cmd = next_cmd_msg.cmd;

    // Set the command
    send_cmd_msg(READ_SUN_CMD);

    ASSERT_EQ(READ_SUN_CMD, next_cmd_msg.cmd);

    // Restore the original value of next_cmd_msg.cmd
    next_cmd_msg.cmd = original_cmd;
}

/*********************************************************
 *  Test: read_temp_cmd
 ********************************************************/
TEST(test_send_cmd_msg, read_temp_cmd)
{
    // Save the original value of next_cmd_msg.cmd
    short int original_cmd = next_cmd_msg.cmd;

    // Set the command
```

```
    send_cmd_msg(READ_TEMP_CMD);

    ASSERT_EQ(READ_TEMP_CMD, next_cmd_msg.cmd);

    // Restore the original value of next_cmd_msg.cmd
    next_cmd_msg.cmd = original_cmd;
}

/***********************************************************
 *  Test: read_pos_cmd
 **********************************************************/
TEST(test_send_cmd_msg, read_pos_cmd)
{
    // Save the original value of next_cmd_msg.cmd
    short int original_cmd = next_cmd_msg.cmd;

    // Set the command
    send_cmd_msg(READ_POS_CMD);

    ASSERT_EQ(READ_POS_CMD, next_cmd_msg.cmd);

    // Restore the original value of next_cmd_msg.cmd
    next_cmd_msg.cmd = original_cmd;
}
```

4 TinkerCad

For this second part of the project, the objective was to build a microcontroller subsystem that had a direct access to the sensors and actuators resourcing to the use of Arduino.

This subsystem is part of a prototype for reduced onboard software for a satellite, together with a main computing hardware system that controls the main tasks. Both subsystems communicate using a master/slave message protocol defined for this project.

With resource to the TinkerCad website, we were able to build the following subsystem

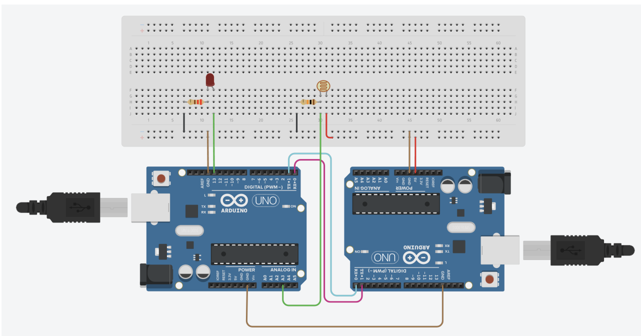

Figure 1: TinkerCad Subsystem

4.1 Arduino Code

4.1.1 Read Sun Sensor

In order to develop this functionality the following C code was created

```
/*********************************************************
 *  Function: read_sun_sensor
 *********************************************************/
void read_sun_sensor()
{
   // Read the analog value from the LDR connected to A3
   int lightValue = analogRead(A3);

   // Define a threshold value to determine if it's considered as sunlight
   int sunlightThreshold = 500; // Adjust this value as needed

   // Check if the light value exceeds the threshold
   if (lightValue > sunlightThreshold)
   {
      sunlight_on = 1; // Set sunlight status to ON
   }
   else
   {
      sunlight_on = 0; // Set sunlight status to OFF
   }

}
```

4.1.2 Set Heater ON/OFF

```
/*********************************************************
 *  Function: set_heater
 *********************************************************/
void set_heater()
{
   // LED connected to digital pin 13
   int ledPin = 13;
   // Turn the LED on (heater) if heater_on is true
   if (heater_on == 1)
   {
      digitalWrite(ledPin, 1);
   }
   else
   {
      // Turn the LED off (heater) if heater_on is false
      digitalWrite(ledPin, 0);
   }
}
```

4.1.3 Arduino Setup and Loop functions

```
// --------------------------------------
// Function: setup
// --------------------------------------
void setup()
{
   // Setup Serial Monitor
   Serial.begin(9600);
   pinMode(13, OUTPUT);
   pinMode(A3, INPUT);
}

// --------------------------------------
// Function: loop
// --------------------------------------
void loop()
{
   comm_server();
   exec_cmd_msg();
```

```
    get_temperature();
    get_position();
    read_sun_sensor();
    set_heater();
    delay(100);
}
```