Onboard Spacecraft Software

Project – Task A, B and C

Work made by:

Andrei Marius Pop Daichendt, 100508659
Luis Barbero Benítez, 100516467
Manuel Mendes Barbosa Salema Guilherme, 100506395

2023/2024 – 1st Semester, Term 1

Contents

1 Introduction

In the ever-evolving realm of satellite technology, the development of onboard software plays a pivotal role in shaping the performance and capabilities of these advanced systems. The focus of this project is to design and construct a prototype for optimized onboard software, a critical endeavor to enhance satellite functionality and efficiency.

This project centers around two fundamental components, each with a specific role and purpose:

1. **Main Computing Hardware System:** This core element acts as the central command and control hub, orchestrating the primary tasks and functions of the satellite.

2. **Micro-controller Subsystem:** This subsystem is intricately linked to the sensors and actuators, allowing for direct communication and execution of commands issued by the main computing system. The synergy between these two components is a cornerstone of the project's success.

The micro-controller subsystem, in particular, is tasked with simulating vital aspects of the satellite's operation, such as monitoring temperature variations and tracking orbital positioning. It is crucial in executing commands swiftly and providing timely responses to adapt to changing conditions. To achieve this, the micro-controller subsystem is equipped with an Arduino micro-controller, alongside additional components like a sunlight monitoring sensor and a heater control system.

This project was developed in three distinctive phases, each contributing to the overall success of the mission:

1. **Task A:** The focus of Task A is to construct a prototype for reduced onboard software that encompasses both the main computing hardware system and the micro-controller subsystem. These components will communicate seamlessly, employing a master/slave message protocol meticulously defined for this project. A detailed breakdown of this task will be developed among chapters 2 and 3.

2. **Task B:** Task B delves into the development of the micro-controller subsystem, which is essential for interacting with sensors and actuators using the versatile Arduino platform. This subsystem plays a vital role within the prototype, collaborating closely with the main computing hardware system through the established master/slave message protocol. A detailed breakdown of this task can be found in chapter 4

3. **Task C:** Task C is dedicated to constructing the main computing hardware system, responsible for overseeing and directing the core functions of the satellite. Task C is closely integrated with the micro-controller subsystem and relies on the master/slave message protocol for efficient communication. A detailed breakdown of this last task can be found in chapter 5.

These tasks represent distinct facets of a complex project, and their interdependence is crucial in achieving our ultimate goal of an efficient and effective onboard software system for the satellite.

In the upcoming sections, we will explore the intricacies of each task and the corresponding processes that have led to the project's success.

2 Arduino

2.1 Arduino Code

2.1.1 Get Temperature

This function is responsible for getting the average temperature of the satellite. In order to achieve this we start by determining the time elapsed since the last temperature computation. This time frame defines the duration during which the satellite has experienced temperature fluctuations. Regarding the power calculations, we take into account the state of the heater, whether it is ON or OFF, and also consider whether the satellite is exposed to sunlight, in which case we factor in solar heat. These two prior calculations, time elapsed and total power, play a crucial role in calculating the energy exchange.

Once we have the total energy gained or lost, we can proceed to calculate the updated temperature. The new temperature is computed 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.

```
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 == 1)
    {
        total_power += HEATER_POWER;
    }
    if (sunlight_on == 1)
    {
        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;
}
```

2.1.2 Get Position

This function aims to get the current satellite position. To do that we start by getting the current time. It begins by calculating the relative time, which is the time elapsed since the last orbit started. 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 function then proceeds to calculate the indices of the two consecutive positions in the *orbit_points* array based on the relative time. These indices help interpolate the satellite's position along its orbit path. The function then updates the coordinates of the satellite's position (x, y, z) by interpolating between the coordinates of the two consecutive positions using *offset_ratio*. This interpolation formula ensures that the satellite's position smoothly transitions along its orbital path. The ratio is given by:

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

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

```c
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;
}
```

2.1.3 Receive commands from the main system

With this function we aim to read the commands received by the main system, update the state and build the response message to be sent.

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.

```c
void exec_cmd_msg()
{
  // Initialize the response message with default values
  last_cmd_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;
    }
    next_res_msg.cmd = SET_HEAT_CMD;
    // Set the status in the response message to indicate success
    next_res_msg.status = 1;
    break;

  case READ_SUN_CMD:
    next_res_msg.cmd = 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.sunlight_on = sunlight_on;
    break;

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

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

  default:
    // This section is for NO_CMD or unknown commands
    next_res_msg.cmd = NO_CMD;
    last_cmd_msg.cmd = NO_CMD;
    response_ready = true;
    break;
  }
}
```

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 Temperature

This function is responsible for controlling the heater by comparing the current temperature with the average temperature.

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

Temperature	Heater
Below Average	ON
Above Average	OFF

Table 3: Temperature Control

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

3.1.2 Send Command Message

This function was designed with the intention to prepare and send a command message to the Arduino system, by specifying the command type.

```c
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)
    {
        next_cmd_msg.set_heater = heater_on; // set the heater
    }
}
```

3.1.3 Receive Response Message

This function is designed to receive and process a response message from the Arduino system. It updates the state of the subsystem based on the information contained in the received response message.

We start reading the command type from the *last_res_msg* structure, which indicates the type of command associated with the received response, based on this received command type, the function updates the state of the subsystem 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

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

    // update the state of the subsystems
    if (cmd == SET_HEAT_CMD)
    {
        heater_on = last_res_msg.status; // update the state of the heater

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

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

3.2 i386 Tests

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

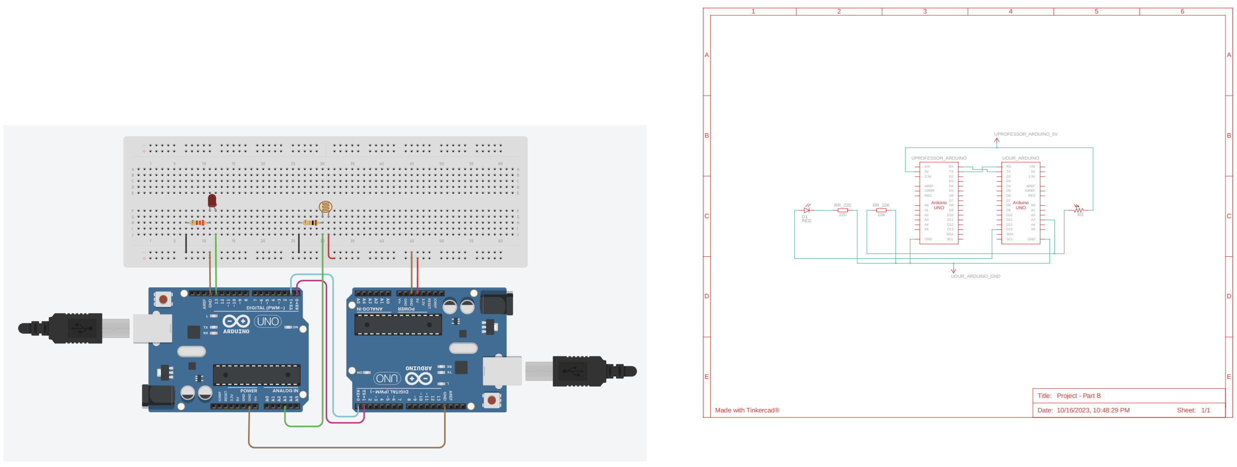
```
/**********************************************************
 *  Test: control_temperature -> basic
 **********************************************************/
TEST(test_control_temperature, basic)
{
    // test 1
    temperature = 20;
    control_temperature();
    ASSERT_EQ(1, heater_on);

    // 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



(a) Circuit View **(b)** Schematic View

Figure 1: TinkerCad Subsystem

4.1 Arduino Code

4.1.1 Read Sun Sensor

This function is responsible for determining the status of sunlight exposure on the satellite. This is achieved by reading the data from a Light Resistor Sensor (*LDR*) connected to the circuit, as seen in figure 1. The function reads an analog value from the *LDR* pin (A3) compares it to a predetermined threshold value, and sets the value for *sunlight_on* variable to 1 if the light value exceed the threshold, which indicates sunlight exposure, or 0 in case it doesn't.

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

   // Define a threshold value
   int sunlightThreshold = 500;

   // 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

This function is responsible for controlling the satellite's heater by turning it on and off. The heater control is implemented by toggling an LED connected to digital pin 13 on the breadboard. To achieve this, we started by defining the *letPin* which is the digital pin to which the *LED*, which is representing the heater, is connected to, in this case is 13 as seen in figure 1. If the *heater_on* is equal to 1, indicating that the heater should be on, then the function will turn on the *LED* by setting the *digitalWrite(ledPin, 1)* to *HIGH*, if the *heater_on* is 0, indicating that the heater should be turned off, the function turns off the *LED* by setting the *digitalWrite(ledPin, 0)* to *LOW*.

```
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
   {
      digitalWrite(ledPin, 0); // Turn the LED off (heater) if heater_on is false
   }
}
```

4.1.3 Arduino Loop and Setup

The *setup* function initializes the necessary components and settings for the satellite system.

We start by initialising the serial monitor for communication, we then proceed to to configure the digital pin as an output in order to control the *LED*.

The loop function serves as the main execution loop for the satellite system. It cyclically executes a series of tasks to manage and monitor the satellite's state.

These tasks are performed in a cyclic manner, and a 100 milliseconds delay is inserted at the end of each iteration, helping to regulate the overall system timing.

```
void setup()
{
   // Setup Serial Monitor
   Serial.begin(9600);
   pinMode(13, OUTPUT);
}

void loop()
{
   comm_server();
   exec_cmd_msg();
   get_temperature();
   get_position();
   read_sun_sensor();
   set_heater();
   delay(100);
}
```

5 RTEMS

5.1 i386 Code

5.1.1 Print State Function

This function is responsible for displaying the current state of the system, such as: its position, temperature, sunlight exposure, and heater status on the screen. It does this by formatting and printing the relevant information to the console. The terminal will then display this information by printing the satellite position in three-dimensional space, the temperature, whether the sunlight is *ON* or *OFF* and whether the heater is *ON* or *OFF*.

```c
void print_state()
{
    printf("Position: (%.2f, %.2f, %.2f)\n", position.x, position.y, position.z);
    printf("Temperature: %.2f\n", temperature);
    printf("Sunlight: %s\n", (sunlight_on ? "ON" : "OFF"));
    printf("Heater: %s\n", (heater_on ? "ON" : "OFF"));
}
```

5.1.2 Controller Function

This function, implements a cyclic scheduler for managing a set of tasks. The scheduler handles the execution of various tasks, each with its specified period and execution time requirements. The tasks include reading the sunlight status, controlling temperature, reading temperature, setting the heater, reading sunlight again, and reading the position.

The scheduler was accomplished by starting by tracking the current by resourcing to the *getClock* function, by doing this we determine which task is ready to be executed based on their predefined periods, it then prints the state of the system after executing the specified tasks and it calculates the time taken by the tasks and sleeps to maintain the defined periods. Finally we update the *current_time* variable to manage task execution periods effectively

With the implementation of this cyclic scheduler we ensure that each task is executed at its specified intervals and manages the timing and execution requirements for the various system operations.

Before implementing the scheduler we first need to specify the *Period* and *Execution Time* for each task:

```c
// Define task periods and execution times
#define TASK_A_PERIOD 5000
#define TASK_B_PERIOD 2000
#define TASK_C_PERIOD 2000
#define TASK_D_PERIOD 2000
#define TASK_E_PERIOD 4000
#define TASK_F_PERIOD 4000

#define TASK_A_EXECUTION_TIME 10
#define TASK_B_EXECUTION_TIME 10
#define TASK_C_EXECUTION_TIME 400
#define TASK_D_EXECUTION_TIME 400
#define TASK_E_EXECUTION_TIME 400
#define TASK_F_EXECUTION_TIME 400
```

These definitions specify the periods and execution times for each task, providing the necessary parameters for the scheduler.

With the task periods and execution times defined, we proceeded to develop the cyclic scheduler, as described below:

```c
void *controller(void *arg)
{
    unsigned long current_time = 0;

    while (1)
    {
        unsigned long start_time = getClock(); // Get current time in milliseconds

        // Check which tasks are ready to execute based on their periods
        if (current_time % TASK_A_PERIOD == 0)
        {
            execute_cmd(READ_SUN_CMD);
        }
        if (current_time % TASK_B_PERIOD == 0)
        {
            control_temperature();
        }
        if (current_time % TASK_C_PERIOD == 0)
        {
            execute_cmd(READ_TEMP_CMD);
            usleep(TASK_C_EXECUTION_TIME);
        }
        if (current_time % TASK_D_PERIOD == 0)
        {
            execute_cmd(SET_HEAT_CMD);
            usleep(TASK_D_EXECUTION_TIME);
        }
        if (current_time % TASK_E_PERIOD == 0)
        {
            execute_cmd(READ_SUN_CMD);
            usleep(TASK_E_EXECUTION_TIME);
        }
        if (current_time % TASK_F_PERIOD == 0)
        {
            execute_cmd(READ_POS_CMD);
            usleep(TASK_F_EXECUTION_TIME);
        }

        print_state();

        // Calculate time taken by tasks and sleep to maintain the period
        unsigned long end_time = getClock();
        unsigned long execution_time = end_time - start_time;
        if (execution_time < TASK_A_PERIOD)
        {
            usleep(TASK_A_PERIOD - execution_time);
        }

        // Update current_time
        current_time += TASK_A_PERIOD;
    }
}
```