Universidade de Aveiro

DETI

Report for the second practical project

Implementation of a C module to process commands received via UART, Unit testing, GCC, Make and C modularity

Course of SETR at UAveiro
April of 2025

Teacher:

Professor Paulo Pedreiras

Autors:

Pedro Ramos
p.ramos@ua.pt
n.º 107348
Masters in Robotics
and Intelligent Systems

Rafael Morgado
rafa.morgado@ua.pt
n.º 104277
Masters in Robotics
and Intelligent Systems

First Project

1. Introduction

This project focuses on the implementation of a C module that processes commands received via the UART protocol. The objective of this project is to build upon the knowledge gained in the first assignment and further enhance our skills in developing C modules and in the usage of tools such as GCC, Make, Doxygen, Git and Github.

Additionally, this project introduces the implementation of unit tests on our codebase via the Unity test tool, which helps us to ensure the accuracy and reliability of the codebase. The project will also serve as an opportunity to improve debugging skills and explore unit testing methodologies to validate the functionality of the code.

2. Project links

Github: https://github.com/rafaelmorgado6/SETR_Proj2

3. How to execute the project

Download and compile the project:

```
$ https://github.com/rafaelmorgado6/SETR_Proj2
$ cd SETR_Proj2/src/build
$ cmake ..
$ make
```

Run the test and the main executable:

```
$ cd $PROJECT_ROOT/src/build
$ ./tests
$ ./main
```

Generate and examine the doxygen documentation:

```
$ cd $PROJECT_ROOT
$ doxygen Doxyfile
$ firefox docs/html/index.html
```

Assuming that the user is inside the 'PROJECT_ROOT', all compiled files are generated inside the 'src/build' folder and all documentation is generated inside the 'docs' folder.

4. Main Features

The main features of the program are:

- The program can process various commands:
  - **A**: Request data from all sensors (Temperature, Humidity, and CO2);
  - **P**: Request data from a single sensor (Temperature, Humidity, or CO2);
  - **L**: Retrieve and display the last 20 readings from each sensor;
  - **R**: Reset all buffers, clearing any stored data.
- The system simulates readings from three sensors:
  - **Temperature**: -50 to 60 ^{o}C
  - **Humidity**: 0 to 100 %;
  - **CO2**: 400 to 20000ppm;
- The program calculates a checksum for each command to ensure data integrity;
- The program stores and retrieves historical sensor data, keeping the last 20 readings for each sensor;
- The program dynamically generates responses based on the sensor values, using predefined responses stored in arrays;
- The program includes logic to handle invalid commands, ensuring robustness by checking for missing or incorrect data in the commands and returning appropriate error messages;
- The program formats the sensor data in a specific format (e.g., #pt+10112!) and appends the calculated checksum before sending the response.

5. Protocol overview

The implemented protocol is similar to the one provided in the beginning of this project.

All received messages (from the point of view of the sensor module), come as an array of characters. All messages have the starting byte '#' and the ending byte '!'. Immediately after the starting byte, one or two bytes representing the command selected are received, for example, "#Pt196!" represents a request for command 'P' and sensor 't'. Some commands do not require a second command byte.

Afterwards, three bytes represent the checksum of the data from the message, which is calculated from all bytes between '#' and the three checksum bytes, not including the starting, ending and checksum bytes.

The sent messages are similar, but have more complex data, for example, the response to the previous temperature command may be '#pt+10112!'. The 'p' command byte is the response to command 'P', the 't' byte tells us which sensor the data belongs to and the "+10" is the sensor value. Finally, the "112" is the checksum of "pt+10".

The bytes 't', 'h' and 'c' tell the user which sensor data is being sent, and each type has their own set number of characters, for example, "#at+30h020c02500183!" should be decoded like:

- 'a': Response to command 'A';
- 't': Start of temperature data;
- '+30': Temperature reading of $+30^{o}$C. Temperature readings always consist of 3 bytes (signal + 2 digits);
- 'h': Start of humidity data;
- '020': Humidity reading of 20 %. Humidity readings always consist of 3 bytes (3 digits);
- 'c': Start of CO2 data;
- '02500': CO2 reading of 2500ppm. CO2 readings always consist of 5 bytes (5 digits);
- '183': Checksum of the whole message, excluding the starting, ending and checksum bytes.

6. Additional features

When the L command is invoked to retrieve historical data, the system now sends the data in multiple messages, each with a 20-character limit. This ensures that if the dataset exceeds the 20-character limit defined for every UART message, it is split into multiple parts and sent sequentially, avoiding buffer overflow issues.

The way this process was implemented is simple, the user first requests the number of "sub-messages" needed in order to receive all the information of the command, and then requests the message number 1, 2, 3, etc, until all the parts of the whole message are received. Each of these "sub-messages" has its own checksum and follows the same format as all other messages.

7. Test program

The featured test program tests the following data workflows:

- Test the processing of the **temperature command**;
- Test the processing of the **humidity command**;
- Test the processing of the **CO2 command**;
- Test the processing of the **'all sensors' command**, which requests data from all sensors (temperature, humidity, CO2);
- Test the **history command**, which retrieves the last 20 readings for each sensor;
- Test the **reset command**, which clears all buffers and resets the system's state;
- Test how the system handles an **invalid command**;
- Test if the **checksum** calculation is correct for **valid data**;
- Test if the **checksum** calculation works correctly for **invalid data**;
- Test if the **buffers** (RX and TX) are **correctly reset** using the resetTxBuffer() and resetRxBuffer() functions;
- Test how the system handles an **incomplete command** (e;g;, missing characters);
- Test the system's ability to handle **RX buffer overflow**;
- Test the system's ability to handle **TX buffer overflow**;
- Test if the system handles **invalid checksums** correctly;
- Test how the system handles **missing the EOF symbol (!)**;
- Test if the system rejects **lowercase commands**.

These tests were created in order to test as much of the implemented functionality as possible, especially some edge-cases that might be harder to replicate manually. The test file also provides a quick way of telling if a change negatively affected the functionality of our module.

The tests were chosen in such a way as to test every single possible outcome of every function. Some simulate a correct workflow and expect a correct response, while others test the robustness of the code based on how well it responds to introduced faults.

8. Updates made after delivery

After the initial delivery of this project, we changed part of the main CMakeLists file and added a secondary CMakeLists file inside the produced module in order to separate the compilation of each .c file. This allows faster compilation as only the changed files are compiled again, rather than compiling the whole project every time.