1 Task Overview

The project aimed to handle incoming data stored in a file that came from various sensors, particularly focusing on processing data from a 3-axis accelerometer within an embedded system. The task was to interpret and analyze ASCII-formatted messages received from the accelerometer sensor and extract meaningful information, such as axis readings and checksum validation.

1.1 Problem Description

The accelerometer messages within the data stream were structured in a specific format: starting with the identifier #G4 and ending with \r\n. Each message had a fixed length and included data representing the X, Y, and Z axis readings of the accelerometer. Additionally, the message contained a checksum value calculated from the ASCII values of characters in the message.

For instance, a typical message looked like: #G4001FFF03DA8\r\n, where the values XXX, YYY, and ZZZ represented the X, Y, and Z axis readings respectively in hexadecimal format.

1.2 Approach

To handle this problem, a C++ program was developed using a class-based approach. The solution consisted of three main files: Acceleo.h, Acceleo.cpp, and main.c.

- **Acceleo.h** - This file contained the class definition for the Acceleo class. The class encapsulated functions and variables essential for processing accelerometer data, including methods for reading, parsing, calculating, and validating the accelerometer readings and checksum.
- **Acceleo.cpp** - This file implemented the functions defined in the Acceleo class. It contained the logic for functions such as acceloReading(), checksum(), acceleroCalculation(), writeFile(), etc., which were responsible for processing the accelerometer data according to the specified format and rules.
- **main.c** - This file handled the main functionality of the program. It read the sensor data from the "sample_data.txt" file and interacted with the Acceleo class by calling its member functions to process each data entry and extract relevant information.

1.3 Class Design

The Acceleo class included members responsible for various tasks, such as initializing variables, performing checksum validation, calculating accelerometer values, reading and processing data, setting offsets, and handling file operations.

The class members, including Z_offset, total_Nonvalid_checksum, total_Accelerometer_value, total_alert_value, maxX, maxY, and maxZ, stored relevant information during data processing and analysis.

Overall, the solution aimed to provide a structured and functional approach to handle and interpret data of the accelerometer sensor within the specified format, ensuring accurate data processing and analysis.

2 Challenges Faced and Solutions

During the implementation of the project, several challenges arose due to the constraints of using only the iostream library in compliance with the CPP 0.0x standards.

2.1 Custom Function Implementations

One of the significant challenges was the absence of external libraries. To handle tasks such as hexadecimal to decimal conversion, processing 12-bit hex values as 2's complement, and performing checksum validation, we had to implement custom functions. For instance, the `SignedVal` function was created to perform ASCII conversions, addition, and modulus operations to validate the checksum.

2.2 Processing Sensor Data

The processing of sensor data involved interpreting the XXX, YYY, and ZZZ values representing accelerometer readings. To convert these 12-bit hex values to integers and handle them as 2's complement, custom logic was implemented within the code.

2.3 Checksum Validation

The checksum validation process required meticulous effort. We converted each character to its ASCII value, added them iteratively while ensuring the sum did not exceed 255 or 256 (inclusive of zero) using modulus operations. The last two characters were compared as hex values converted to decimals with the result from the `SignedVal` function. If the checksum matched, it signified a successful validation, updating the relevant variables in the Acceleo class accordingly.

2.4 Implementing ALERT Signal

Another challenge involved appending the [ALERT] signal upon detecting consecutive Y-axis values exceeding 0.2G. This required conditional loops and comparisons. We incremented the `checkAlert` variable upon each Y-axis value surpassing 0.2G. If three consecutive values exceeded this threshold, the [ALERT] signal was appended to the output data until the Y-axis value dropped below 0.2G, resetting the `checkAlert` variable.

2.5 Handling Max Values

The approach used for tracking maximum X, Y, and Z values in each iteration of sample_data.txt utilized conditional statements to update the maximum values if the current values exceeded the recorded maximums. Additionally, adjustments for Z-axis offset were made based on a conditional check (`Z_offset`).

Overall, overcoming these challenges required creative problem-solving and custom logic implementation, ensuring compliance with CPP 0.0x standards while meeting the project guidelines without relying on external libraries.

3 main.cpp Overview

The main.cpp file is the one handling and joining all of our code together, the flow of the code is explained below to give idea of how data is aqquired processed and displayed.

1. **Initialize Variables**: Start by initializing necessary variables, such as `MAX_ENTRIES`, `MAX_ENTRY_LENGTH`, and any counters or arrays required for storing data.
2. **Open File**: Open the file `"sample_data.txt"` in read mode.
3. **Read Entries**: Read the file character by character until the end of the file (EOF) or the maximum number of entries is reached.
   - Check for the start of a new entry:
     - If the character is '#', start building a new entry.
     - If encountering 'G' consecutive after '#', continue adding characters to the current entry as this means the dataset is from accelerometer.
   - Check for the end of an entry:
     - If a newline character '\n' is encountered and the entry length is valid:
       - Store the entry in the entries array.
       - Process the entry immediately:
         - Pass the entry to functions like `acceloReading()` and `checksum()` for processing.
         - Display the processed entry using `std::cout`.
4. **Close File**: Close the file after reading and processing all entries.

4 Acceleo.h Overview

The `Acceleo.h` file contains the class definition for the `Acceleo` class, encapsulating various functionalities to process accelerometer data within an embedded system.

4.1 Class Structure

- The `Acceleo` class is structured as follows, containing essential functions and variables for handling accelerometer data.
- **Public Functions**:
  - `Acceleo()` - Constructor function initializing the class.
  - `initialization()` - Initializes necessary variables.
  - `checkSumValidation()` - Validates the checksum of the message.
  - `accelerometerValue()` - Processes and retrieves accelerometer values.
  - `acceloReading()` - Reads and processes accelerometer data from the message.
  - `writeFile()` - Handles writing data to an output file.
  - `combineFloatsToString()` - Combines float values into a string for output.
  - `setZoffset()` - Sets the Z-axis offset.
  - `SignedVal()` - Converts hexadecimal values to integers for signed representation.
  - `checksum()` - Performs checksum validation.
- **Public Variables**:
  - `Z_offset` - Boolean variable representing the Z-axis offset status.
  - `total_Nonvalid_checksum` - Counter for the total number of invalid checksums.
  - `total_Accelerometer_value` - Counter for the total accelerometer readings processed.
  - `total_alert_value` - Counter for the total occurrences of an alert signal.
  - `maxX`, `maxY`, `maxZ` - Variables storing maximum X, Y, and Z axis values respectively.

The `Acceleo` class in `Acceleo.h` provides necessary functions and variables to manage, process, and analyze data received from the 3-axis accelerometer sensor within the specified embedded system constraints.

5 Acceleo.cpp Overview

The code in the Acceleo.h file defines functions that handle processing accelerometer data. Let's understand the flow of the code:

5.1 Constructor Function

The `Acceleo()` constructor initializes a `char` array (`data[]`) with a placeholder timestamp string and then invokes the `writeFile()` function with this timestamp.

5.2 Acceleo Reading Function

- The `acceloReading()` function processes the accelerometer data contained in the input message.
- It parses the X, Y, and Z axis values from the message and converts them to float values (`FinalAcc_X`, `FinalAcc_Y`, `FinalAcc_Z`).
- Each axis value is checked for the maximum value and updated accordingly (`maxX`, `maxY`, `maxZ`).
- An alert check is performed on the Y-axis data. If it exceeds a threshold for three consecutive readings, an alert is appended to the data string.
- The resulting data string is then written to the output file using the `writeFile()` function.

5.3 Other Functions

- The `setZoffset()` function sets the Z-axis offset based on a boolean input.
- `writeFile()` appends data to an output file by finding the last non-empty line and writing the provided string after it.
- `combineFloatsToString()` formats three float values into a single string, appending an "[ALERT]" signal if specified.
- The `SignedVal()` function converts a hexadecimal string to an integer, handling 12-bit 2's complement representation.
- `checksum()` calculates the checksum from the input string and validates it against the calculated value.
- `initialization()` initializes counters and variables used in the class to zero or default values.

This code flow demonstrates how various functions in the `Acceleo.h` file work together to process and manage accelerometer data within the specified constraints.

6 Conclusion

The processing of a large data file containing over 5000 lines initially took around 2 minutes to complete. To optimize this duration, the code was refined to perform data reading and processing simultaneously, thereby reducing the overall processing time significantly to less than 30seconds.

In addition to optimizing performance, clarity in the debugging process was emphasized. Each step of data extraction, parsing, and validation was methodically explained within the output. Debugging outputs were added at each step of processing to increase understanding and help in building/testing process.

The entire data processing flow, starting from data extraction to checksum validation and calculation of XXX, YYY, and ZZZ acceleration values, meets to the problem guidelines. Detailed statistics showing the count of alerts, successfully processed values, failed checksum validations, and maximum acceleration values (X, Y, and Z) throughout the data file were presented to ensure all project requirements were met.

Although explicit timestamps indicating the processing time were not possible due to not being able to use libraries, a timestamp such as chrono or ctime but a static header was added in each iteration of the sample_data file. This static timestamp marks each new iteration of code, appending data to the existing file.

Overall, the proposed solution addressed the objectives within the mentioned constraints. The optimizations improved the processing efficiency, and the debugging outputs enhanced the code's understanding, ensuring that project specifications are met along the necessary data processing criteria.