This code is designed to **read in binary data** from a file containing **CAN** (**Controller Area Network**) messages, **extract specific data from those messages**, and **write the extracted data to a binary file**.

The code begins by declaring constants and a **CanMessage struct** that represents the structure of a **CAN message**. It then declares several helper functions that handle file I/O and data extraction.

The main function initializes two string variables with the names of the input and output files, reads the input file and stores the data in a vector of **CanMessage structs**, extracts the relevant data from the **CanMessage** **vector** and stores it in a **vector of bytes**, and finally **writes the bytes to the output file**.

The **readCanData** function reads the binary file containing **CAN messages** and returns a vector of **CanMessage structs**. It checks whether the file can be opened and returns an empty vector if it cannot. It reads the file contents in a loop and appends the relevant messages to the vector.

The **extractTransferData** function takes a vector of **CanMessage structs**, **extracts the data from messages** that have a **specific DONGLE\_ID and SERVICE\_ID**, and returns a vector of bytes. It checks each **CanMessage** in the input vector and appends the remaining 7 bytes of the CAN data field to the output vector if the message matches the specified criteria.

The **saveBinaryFile** function takes a filename and a vector of bytes and writes the bytes to a binary file. It checks whether the file can be created and returns without writing any data if it cannot. It writes the data to the file using the **file.write()** function.

Overall, this code provides a simple way to read and extract specific data from binary files containing **CAN messages**, which is a common format for storing and transmitting data in automotive systems.

Text

Description automatically generated

Text

Description automatically generated

**Here is a line by line explanation of the code in detail:**

Text

Description automatically generated

#include <iostream>

#include <fstream>

#include <vector>

using namespace std;

This block includes necessary headers and declares a namespace. iostream is used for input/output operations, **fstream** for file handling, and vector is a data structure that can dynamically resize itself. using namespace std is used to avoid **writing std::** before each of the standard library functions, which can be time-consuming and make the code less readable.

Text

Description automatically generated

constexpr uint32\_t DONGLE\_ID = 0x7E0;

constexpr uint32\_t ECU\_ID = 0x7E8;

constexpr uint8\_t SERVICE\_ID = 0x36;

These are constants declared using **constexpr**, which means they are evaluated at compile-time rather than run-time**. DONGLE\_ID** and **ECU\_ID** are **hexadecimal constants** that represent the unique identifiers of the devices that send and receive the data. **SERVICE\_ID** is a constant that represents the type of message being sent.

Text

Description automatically generated

struct CanMessage {

uint32\_t timestamp;

uint32\_t canId;

uint8\_t canData[8];

};

This is a struct that defines the structure of a CAN message. A CAN message contains a **timestamp**, a **CAN ID**, and a data field that is an **array of 8 bytes**.

Text

Description automatically generated

vector<CanMessage> readCanData(const string& filename);

vector<uint8\_t> extractTransferData(const vector<CanMessage>& canData);

void saveBinaryFile(const string& filename, const vector<uint8\_t>& data);

These are function declarations**. readCanData** reads a binary file containing **CAN messages** and returns a vector of **CanMessage structs**. **extractTransferData** takes a vector of **CanMessage structs**, extracts the data from messages that have a specific **DONGLE\_ID** and **SERVICE\_ID**, and returns a vector of bytes. **saveBinaryFile** takes a **filename and a vector of bytes** and writes the bytes to a binary file.

Text

Description automatically generated

int main() {

const string inputFilename = "mg1cs002-stockmapsflash.candata";

const string outputFilename = "mg1cs002-stockmapsflash.transferdata.bin";

const auto canData = readCanData(inputFilename);

cout << "Number of messages read: " << canData.size() << endl;

const auto transferData = extractTransferData(canData);

cout << "Size of extracted transfer data: " << transferData.size() << endl;

saveBinaryFile(outputFilename, transferData);

cout << "Data has been saved to: " << outputFilename << endl;

return 0;

}

This is the **main** function, the entry point of the program. It initializes two string variables with the names of the input and output files. **It calls readCanData** to read the **input file** and **store the data in a vector of CanMessage structs**. It then **calls extractTransferData** to extract the relevant data from the **CanMessage vector** and store it in a vector of bytes. Finally, it **calls saveBinaryFile** to **write the bytes to the output file**. The function then prints out the number of messages read, the size of the extracted transfer data, and the name of the output file. The function returns 0 to indicate successful completion of the program.

Text

Description automatically generated

vector<CanMessage> readCanData(const string& filename) {

vector<CanMessage> canMessages;

ifstream file(filename, ios::in | ios::binary);

if (!file){

cerr << “Error opening file: “ << filename << endl;

}

CanMessage msg;

while (file.read(reinterpret\_cast<char\*>(&msg.timestamp), sizeof(msg.timestamp)) &&

file.read(reinterpret\_cast<char\*>(&msg.canId), sizeof(msg.canId)) &&

file.read(reinterpret\_cast<char\*>(&msg.canData), sizeof(msg.canData))) {

if (msg.canId == ECU\_ID || msg.canId == DONGLE\_ID) {

canMessages.push\_back(msg);

}

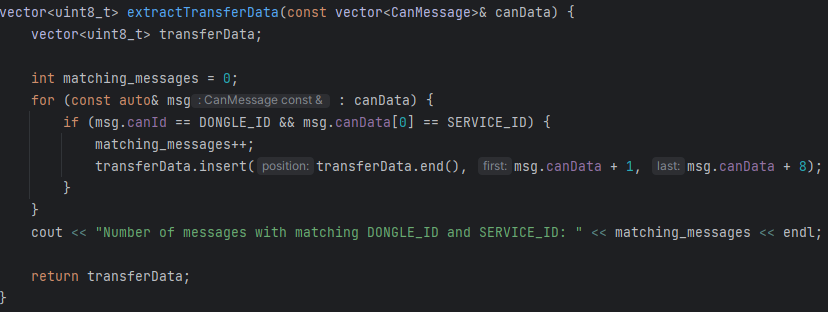
}

return canMessages;

}

This is the implementation of the **readCanData function**. It takes a filename as input and returns a vector of **CanMessage structs**. It first creates an empty vector called **canMessages**. It then creates an input file stream object called file and opens the input file in binary mode. If the file cannot be opened, it prints an error message and returns an empty vector.

The function reads the file contents in a loop using the **file.read() function**. It reads the **timestamp**, the **CAN ID**, and the **CAN data fields** one by one using the **reinterpret\_cast** function to convert the pointer type. The function then checks whether the **CAN ID** of the message is equal to the **DONGLE\_ID** or **ECU\_ID**. If it is, the function appends the message to the vector **canMessages**. The function then returns the vector of **CanMessage structs**.



vector<uint8\_t> extractTransferData(const vector<CanMessage>& canData) {

vector<uint8\_t> transferData;

int matching\_messages = 0;

for (const auto& msg : canData) {

if (msg.canId == DONGLE\_ID && msg.canData[0] == SERVICE\_ID) {

matching\_messages++;

transferData.insert(transferData.end(), msg.canData + 1, msg.canData + 8);

}

}

cout << "Number of messages with matching DONGLE\_ID and SERVICE\_ID: " << matching\_messages << endl;

return transferData;

}

This is the implementation of the **extractTransferData** function. It takes a vector of **CanMessage structs** as input and returns a vector of bytes. It first creates an empty vector called **transferData**. It then initializes an integer called **matching\_messages** to 0.

The function iterates over each CanMessage in the input vector using a range-based for loop. It checks whether the CAN ID of the message is equal to the DONGLE\_ID and whether the first byte of the CAN data field is equal to the SERVICE\_ID. If both conditions are true, it increments matching\_messages by 1 and appends the remaining 7 bytes of the CAN data field to the transferData vector using the insert() function.

Finally, the function prints out the number of messages that matched the DONGLE\_ID and SERVICE\_ID conditions. It then returns the transferData vector.

Text

Description automatically generated

void saveBinaryFile(const string& filename, const vector<uint8\_t>& data) {

ofstream file(filename, ios::out | ios::binary);

if (!file) {

cerr << "Error creating output file: " << filename << endl;

return;

}

file.write(reinterpret\_cast<const char\*>(data.data()), data.size());

}

This is the implementation of the **saveBinaryFile** function. It takes a filename and a vector of bytes as input and writes the bytes to a binary file. It first creates an output file stream object called file and opens the output file in binary mode. If the file cannot be opened, it prints an error message and returns without writing any data.

The function then uses **the file.write()** function to write the data to the file. It takes two arguments: a **const char\*** pointer to the data to be written (which is obtained by casting the pointer to the first byte in the data vector using **data.data()**), and the size of the data to be written (which is obtained using **data.size()**).

The function then closes the file and returns. Note that the data is written to the file as-is, without any additional headers or formatting. It is assumed that the file will be read back into a program that knows the structure of the data.