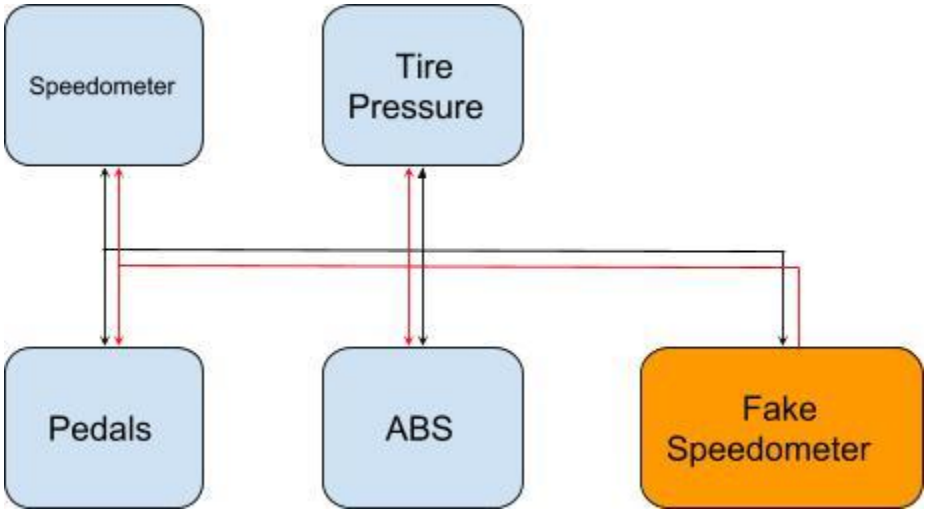
Intro

Cars nowadays are a collection of many computers (or electronic control units - ECUs) interconnected by a network. Each such unit has its own function and is connected to some component in a car: fuel pump, door lock, speed sensor, brakes,etc. For simplicity of this exercise, these components use a dedicated protocol to send messages between them over a network with a bus topology (all messages are broadcast to everyone) called SBP (Simple Bus Protocol).

The SBP protocol was not designed with security in mind. Numerous attacks can be performed by a rogue component that is connected to a car's network bus. Such a component can transmit messages that will directly influence other legitimate components: speedometer can show abnormal speeds, car stereo can be set to max volume, and even brakes and steering wheel can be manipulated.

Below is the diagram to illustrate the above:



SBP Protocol Description

- Each ECU transmits messages that are received by all the other units.
- Each unit transmits messages with some frequency all the time
- Messages can be sent at the same time.
- Each message has an ECU ID (uint32), value1 and value2 (each uint32)
- Values are always sent, but not necessarily used (data may be ignored)
- All values have a range of validity

Our car network has the following ECUs connected to it :

Unit Name	ID (32bit)	Value1 (32bit)	Value2 (32bit)
Speedometer	0x100	0	CURRENT_SPEED_VALUE (0-300)
Pedals	0x200	ACCELERATION_VALUE (0-100)	BRAKE_VALUE (0-100)
ABS	0x400	0	INACTIVE / ACTIVE (0-1)
TIRE_PRESSURE	0x800	0	PRESSURE_PERCENTAGE (0-100)

The Task

Your task is to develop a component that will be connected to the car's SBP network, monitor all the messages transmitted over it and report abnormal behavior.

Your component will receive a textual file with messages broadcast on the network and will detect various abnormal behaviors.

- Each message will start on a new line (maximum line length is 100 chars)
- Each message consists of:
    - A message id which is line number (starts from 1)
    - A timestamp in millisec (always starts from 0)
    - An ECU id
    - Value1
    - Value2
- The above fields in each message are separated by space
- All of the above fields are uint32

For example :

1 0000000001 0x100 0 80

2 0000000323 0x200 0 0

3 0000050001 0x100 0 82

4 0000100323 0x200 53 0

5 0000100323 0x400 0 0

Your mission is to detect anomalies which will tell us that either there is a rogue component pretending to be someone else or one of the existing components was hijacked by a hacker. Below are the descriptions of such anomalies, each will require you to implement an interface. The interfaces for all 3 tasks are similar and will require you to report the first 1000 anomalous messages (or less if there were less). Ids should be reported in increasing order.

Please read all the instructions till the end (including Q&A at the end) and also look at provided files before starting to solve this exercise

1. Timing Anomaly - Detect abnormal frequency. Each ECU transmits messages constantly with frequency that is not higher than the one defined below. There can be messages that are missing, but there can not be message that is sent with timestamp less than the frequency defined below. For example, there will be no more then one speedometer message every 50 millisec
    - Speedometer: transmission frequency - one message every 50 millisec.
    - Pedals: transmission frequency - one message every 5 millisec.
    - ABS: transmission frequency - one message every 10 millisec.
    - Tire Pressure: transmission frequency - one message every 100 millisec.

```
int detect_timing_anomalies(const char* file_path,
        unsigned int *anomalies_ids)
```

2. Behavioral Anomaly - Detect abnormal values. Physical limitations impose that:
    - Make sure all the values are within the given range
    - The minimum time a pedal can be pressed (value>0) is 10 milliseconds
    - The gas and brake pedal cannot be pressed simultaneously
    - The car's speed may not change faster than 5 kmh within 50 milliseconds
        i. An exception to this is a car crash, in which the speed will change to 0. After that, it should stay 0 for the rest of the recording. Any positive speed after there was a crash should be treated as an anomaly.

```
int detect_behavioral_anomalies(const char* file_path,
        unsigned int *anomalies_ids)
```

3. Correlation Anomaly - Detect abnormal correlations between various units:
    - When acceleration is active (> 0), speed should not be decreasing. In this case only wrong speed is considered an anomaly

- When brakes are active (> 0), speed should not be increasing. In this case only wrong speed is considered an anomaly
- When brake pedal is pressed hard (80+), ABS will activate (value2 is 1). It will remain active as long as the brake pedal is pressed hard. 0 value for ABS in this case is considered an anomaly
- When tire pressure is below 30, speed can't be above 50 kmh. Speed above 50 kmh reported after tire was below 30 is considered an anomaly. Tire pressure below 30, when speed was above 50 is also considered an anomaly.
- Tire pressure can't increase while the car is moving. Abnormal tire pressure is considered an anomaly.

```
int detect_correlation_anomalies(const char* file_path,
        unsigned int *anomalies_ids)
```

Additional instructions:

- The implementation should be written in c
- The implementation should be compiling properly and working under linux or be OS agnostic. We strongly advise running and testing it in Linux environment. Among others this mean avoid using non standard headers (i.e. conio.h). If you use scanf/printf, we would also recommend using format strings that work properly with uint32 types
- You have received exercise.h and exercise.c. The header file contains the mentioned above interfaces. You need to implement them inside exercise.c. Current implementation is default: (return -2). Please read the function descriptions in header file for more info on arguments and return values.
- Each subtask is completely independent and does not depend on other tasks. You can also assume each interface is supposed to deal only with anomaly it is supposed to detect and the log file that will be provided to it can only have anomalies of that type For example detect_behavioral_anomaly will never be invoked on log file with timing anomalies or correlation anomalies
- If there is more than one issue in some message, its id will be only reported once
- If some message is anomalous, in addition to reporting it, you still need to take it into account when looking at anomalies in other messages. For example if brake and gas were pressed simultaneously, and it happened for less than 10 millisec, you will report 2 anomalies, one in pedal message that has both gas and break and the other one in the next pedal message that has 0 in either gas or break (or both) within less than 10 millisec.
- You can assume that the log file is in correct format (as specified above)
- You can assume that there were no events before the start of the log

- You can assume that the initial value of each ECU is 0, except for Tire pressure that has the initial value of 100.
- The value of each ECU remains the same till next message from that ECU is received For example if there are the following messages in the log :

  0000000300 0x200 0 0
  0000000305 0x200 0 70
  0000000310 0x200 0 55

  It means that initially brake had the value 0, it remained the same till 300, it was 0 till 305, at 305 it was activated with value 70, it stayed the same till 310, then at 310 it was activated with value 55 and will stay with the same value till next message from brakes
- You have received several example files with the task you can use them to test your solution. Please take into account that your solution will be tested automatically (and also reviewed manually) with additional inputs, some of them may be very large. Space / Time complexity of the solution will also be evaluated.
- The example files also show what is the proper way to understand task requirements, in case some of them are not clear or precise. Please make sure your solution provides the result exactly as expected in our examples. It will be tested with those examples (among others) by automatic tests !
- You need to provide only exercise.c with your implementation. Any additional files will be ignored. Please do not provide main() inside exercise.c, otherwise you will break compilation and testing on our side and your solution will fail, we will compile your implementation with our own main() with tests.
- Please write your code in good style, document where needed. You can expand exercise.c with additional functions, please refrain from adding additional files, and from changing the interface file (exercise.h)

Q&A from candidates:

Q. Your sample files / exercise definition / interface contain typos and errors. I assumed that something in them is wrong and solved the exercise according to my understanding.
A. Please do not assume errors and typos and base your solution upon this assumption, especially in example files. They were chosen and tested thoroughly and in fact are part of exercise definition. Should there be any conflict between exercise definition and the sample files, the sample files take priority. In any case, if you believe that there are inconsistencies, please ask. If there was indeed an error, inconsistency or ambiguosity, we will fix it and you will get an additional time to fix your solution.

Q. I've noticed from your example files that the anomalous messages are not ignored when compared to next messages in order to detect other anomalies. For example if brake and gas were pressed simultaneously, and it happened for less than 10 millisec, you will report 2 anomalies, one in pedal message that has both gas and break and the other one in the next pedal message that has 0 in either gas or break (or both) within less than 10 millisec. A. Good observation ! That exactly what example files are for

Q. What happens when 2 messages have the same time ? Which should I accept / treat first ?
A. When 2 messages have the same time, you should treat them in the order they arrive

Q. You said that each value remains the same until reported otherwise.
If this is the case, in this example the speed is changed from 4 to 6 in 1 millisec, because it was 4 in 199 :
52 150 0x100 0 4
68 200 0x100 0 6
85 250 0x100 0 9
Why don't you report this as anomaly in your examples ?

A. Good observation. To make this exercise simple, our intention was that the minimum resolution of time you should work with is, is the one defined in "Timing Anomaly" section for each ECU. This means that you should only consider values reported in 150, 200 and 250 for speedometer when looking for anomalies.

Q. In this example :
35 100 0x100 0 298
52 150 0x100 0 7
68 200 0x100 0 9

52 is an anomaly, while 68 is not. Why ? Wasn't speed changed sharply from 29?

A. Each anomaly should only be reported once. The sharp change was already detected and reported at 52. In 68 you should compare the speed with the previous report at 52. Speed is just example, the same principle applies to other anomalies as well.

Good luck !