Exercise #05 (Vehicle)

Description

Your program should read data from a software module in a car. The software module is currently not able to connect to hardware so we're using faked data.

In the data, as read using the `get_bus_data` function you'll get a lot of data out of which only one, `engine_speed` is interesting to us.

Your task is to:

- read data, using the `get_bus_data` function, at least two times per second. It is quite possible you will get data, or no data, or some error occurs - it is thus important to check the return value of the function.
- every second you should printout the mean value of the last 10 (valid) data read. You are recommended to do this by storing the 10 last read data in an array.

Not obligatory

- When starting up you do not have 10 valid data (how could you?). Make sure you calculate the mean value only from valid data.
- What shall happen with the mean value calculation if you get incorrect data? No coding needed, just think about it.

Note: the name `magic-bus` is inspired by the The Who song, https://en.wikipedia.org/wiki/Magic_Bus_(song)

Syntax

```
magic-bus
```

Try the program in the shape it is in

The program is in a very basic shape - your job is to extend it - and you can try it out to get a feeling.

Compile and execute with no log or debug

```
make
```

```
./magic-bus
```

Note: `make clean` will be needed if the last compilation was done with either `make debug` or `make log`

Compile and execute with log but no debug

```
make clean log
```

```
./magic-bus
```

Note: clean is not needed as long as you continue doing `make log`

Compile and execute with log and debug

```
make clean debug
```

```
./magic-bus
```

Note: clean is not needed as long as you continue doing `make debug`

Note: You don't know the shape I am in.

Your coding

This exercise/handin contains two directories, `windows` and `unix`. Windows users should use the code in the `windows` directory and BSD (including MacOS) and GNU/Linux users shall use the `unix` directory.

You can do all the coding in the file `main.c`. The other file are basically there to fake data and provide log/debug functionality for you.

In the file `main.c` you will find instructions/hints on how to proceed with the exercise.

How to develop

We recommend small increments - add a small piece of code, compile and execute.

Collecting data

You'll get the `main.c` together with this exercise. In the main function you will find a loop together with recommendations on how to proceed.

API description

Struct vehicle_info

You will find the definition of this structure in the file `vehicle/bus.h`. In this struct we're only interested in the `engine_speed` variable.

Function `get_bus_data`

```
SYNTAX:

   int get_bus_data(vehicle_info *vi)

DESCRIPTION:

   Stores the latest data from the vehicle module in the passed
   vehicle_info pointer.

RETURN VALUES:

   0 (BUS_OK) - if data was found and stored in the passed
   vehicle_info struct.

   1 (BUS_NO_DATA) - if no data was found. No data is stored in the
   passed vehicle_info struct. This means that the data queue in the
   vehicle is empty and getting this every now and then is good since
   it shows you're reading data quicker than the data is produced.

   2 (BUS_BAD_INDATA) - if the passed pointer to vehicle_info struct
   was NULL.

   2 (BUS_BAD_DATA) - if for different reasons the data could not be
   retrieved. Reasons could be electrical failure.
```

Presenting mean value

In each turn in the the loop together in the main function you can check if one second has passed since the last printout. If one second has passed, print again.

Hint: print the first time, store the current time. Next turn in the loop you check the time and see if one second has passed ....

Collecting data - storing 10 items in an array

You can do this in several ways. We list two strategies here.

Store items ordered and starting from 0 (easier)

In short you always store the latest incoming data in the last position (10) in the array. To do this you need to move the 9th element to the 8th position etc.

We given an example here with an array of size 3.

Initial array:

```
array: [ a | b | c ]
```

We shall now store a new element, `d`. To do this we can discard the oldest one, which is `a`, so move all the elements left and finally store d on the last position.

```
array: [ a | b | c ]
```

Move elements on step left - we do this by copying so we'll have two c's for a while:

```
array[0] = array[1] ;
array[1] = array[2] ;

The array now looks like this:
array: [ b | c | c ]
```

And finally store `d` in the last position:

```
array[2] = d;

The array now looks like this:
array: [ b | c | d ]
```

When the array is bigger than 3 you most likely want to use a for loop instead of adressing the array as above.

Calculating the mean value is done by looping through the array and sum the elements and divide ... well, you get it.

Store items ordered and starting from an index (a bit harder)

We still use an array of size 10. But now we use an index to keep track of the where to store the next item. If the index reaches 10 we can zero the index (i e assign it `0`).

Let's go for an example:

Initial array:

```
The array and index now looks like this:
array: [ a | b | - ]
index: 2
```

We shall now store a new element, `c`. To do this we can discard the element at position `index` (in this case `2`) which currently is undefined (not set). We also need to increase the index - since the index now is 3 (size of the array) we zero it.

```
index++;
if (index==3) { index=0; }

The array and index now looks like this:
array: [ a | b | c ]
index: 0
```

We shall now store a new element, `d`. To do this we can discard the element at position `index` (in this case `0`) which is `a`. We also need to increase the index.

```
index++;
if (index==3) { index=0; }

The array and index now looks like this:
array: [ d | b | c ]
index: 1
```

Makefile

We have written a Makefile for you:

- `make magic-bus` - produces a program, `magic-bus`, with no debug or log printouts
- `make all` - same as make `magic-bus`

- `make debug` - produces a program, `magic-bus`, with debug and log printouts
- `make log` - produces a program, `magic-bus`, with log printouts (no debug printouts)
- `make clean` - cleans up your source from generated files (such as .o)

Log and debug

In `log.h`, which is already included in `main.c` you'll find functions with which you can have conditional printouts - printouts that are made only if a certain condition is true.

debug

Use it like you would use printf. But you need to add an extra parenthesis:

```
debug(("Wowie the variable ret now has the value: %d", ret));
```

Note: there are two parenthesises.

To trigger the debug printouts to actually occur you need to compile the code with `DEBUG` defined. The easiest way to do this is to type:

`make clean debug`

Note: enabling debug printouts also triggers log printouts.

log

Use it like you would use printf. But you need to add an extra parenthesis:

```
log(("Wowie the variable ret now has the value: %d", ret));
```

Note: there are two parenthesises

To trigger the log printouts to actually occur you need to compile the code with `LOG_TO_FILE` defined. The easiest way to do this is to type:

`make clean log`

Test

Run the program.

Code structure

You only need to write code in `main.c`. Make sure no function, including main, is more than 20 lines of code. If a function is longer than 20 lines you are encouraged to split it into functions and invoke calls to these instead. This will make your code easier to read, review and maintain.