2024

"Code Cruise" Hackathon

"Code Cruise" Hackathon

Navigate the Future of Automotive with Embedded C

Welcome aboard, gearheads and code wizards! Buckle up for Code Cruise Hackathon! Here, your passion for cars meets the power of programming. Get ready to dive into the exciting world of embedded C, where your coding skills will steer the future of automotive innovation. This hackathon is your chance to collaborate, create, and push the boundaries of what's possible. So, fire up your engines, fuel your creativity, and let's hit the road together!

Ahoy mates! Here's the lowdown for smooth sailing:

No hardware this time: Unlike a traditional embedded system, we'll be simulating hardware actions. We'll provide the tools you need to navigate this virtual world and translate your code into car magic!

Beyond coding: This hackathon tests more than just coding prowess. We'll throw some curveballs (test cases) your way to see how your code handles the pressure. But that's not all! We'll also be impressed by your problem-solving skills and how you tackle new challenges on the fly. Get ready to collaborate with your team and show off your teamwork, adaptability, and out-of-the-box thinking when a Visteon expert joins your coding pitstop to assess your communication and problem-solving approach.

How we score your voyage: Buckle up, because we'll be evaluating your journey using these key metrics:

(a) Test Case Results (60%): Did your code conquer the challenges we threw your way?
(b) Technical Approach and Correctness (20%): Is your code a well-oiled machine, efficient and accurate?
(c) Behavioral Aspects (20%): Did your crew work together seamlessly, adapt to new situations, and impress the Visteon expert with your communication skills?

Ready to set sail and navigate the future of automotive with Code Cruise?

Let's go!

You are required to build a system that consists of the following sub-systems:

(a) Lane Departure Warning System
(b) Tyre Pressure Monitor
(c) Blind Spot Detection System
(d) Airbag Actuator System
(e) Reverse Camera Categorization System
(f) Driving Range Estimator System

The working of these different subsystems is discussed below. Organization of the skeleton code and expectations are discussed in a later section.

Lane Departure Warning System

All major highways in the world are equipped with different lanes to ensure the vehicles can segregate themselves into lanes based on their speeds. It's important to always stay in a particular lane and not slide into neighboring lanes for the safety of oneself as well as fellow riders on the road. Cars are installed with lane departure warning systems that alert the driver when a car deviates from its original lane to neighboring lanes. In the real world, this is calculated by observing the deviation of the car from its mean position as shown below:

Adjusted threshold = (width of the lane - width of the car) / 2
Deviation from mean position = abs (distance of left sensor from lane boundary - distance of right sensor from lane boundary)

A deviation higher than the adjusted threshold is indicative of the car sliding from its designated lane to another lane.

A file "LaneDeparture.txt" contains 25 records. Each record consists of 4 values, viz lane width, car width, distance of left sensor from lane boundary, distance of right sensor from lane boundary. All values are in meters.

For each of the records, your program should call the function below to determine if the car is within its designated lane or deviating from it.

Incorporate the logic for lane departure waring system in a function with the following details:

Function: **determineLaneDeparture( )**
Return value: an integer value 0, 1 or -1, with 0 meaning the car is within its lane and 1 meaning the car is deviating from it, and -1 meaning the input is wrong.

Tyre Pressure Monitor

In modern cars, a TPMS sensor is used to determine the air pressure in each individual tyre. These sensors are typically installed on the inside of the valve stem on each wheel. They measure the air pressure directly and transmit the data wirelessly to a receiver in the car's computer system. This allows for real-time monitoring of the pressure in each tyre and provides individual pressure reading for each wheel.

A text file "TPMonitor.txt" contains 25 records. Each record contains three numerical values as per the following details:

First numerical value - 1 byte long
Second numerical value - 1 byte long
Third numerical value - 4 bytes long

The significance of bits in these numerical values is shown in Figure 1.1

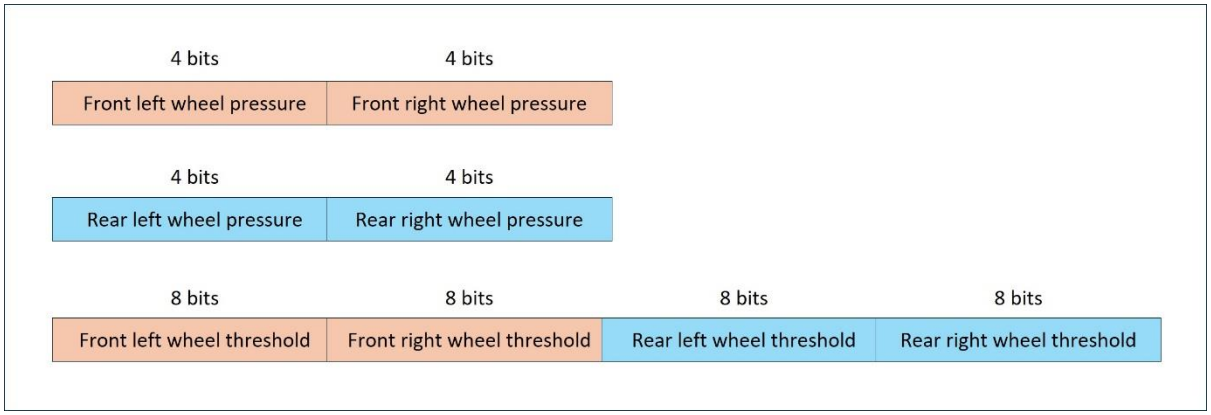

Figure 1.1

Your program should generate a random number between 1 and 25. Based on the random number generated, it should read that record from the file "TPMonitor.txt". So, if the random number generated is 4, it should read 4th record from the file. Once the record is read, it should check each wheel's pressure against the threshold value.

If the tyre pressure is lower than the threshold value for that tyre, then a flag should be set up in a **tyrePressureFlags** variable. If more than one tyre's pressure is below the threshold value, then the warning flag should be set up in the same variable. The bitwise positions of these flags are shown in Figure 1.2.

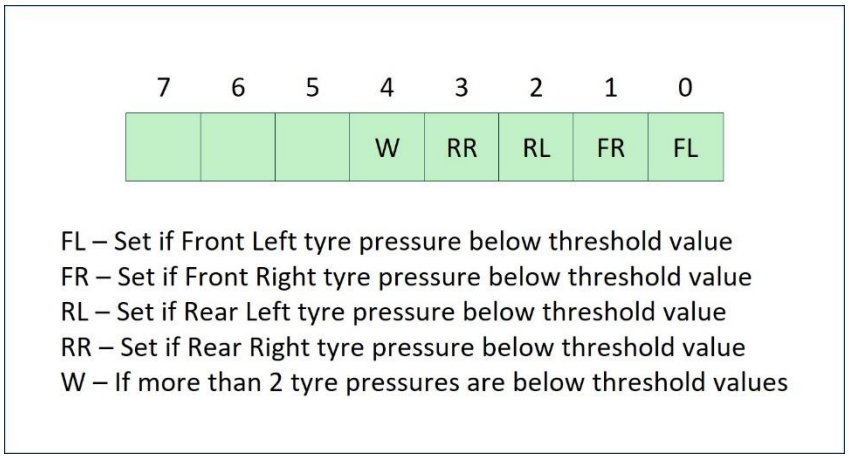

Figure 1.2

Incorporate the logic for tyre pressure monitoring in a function with the following details:

Function name: **monitorTyrePressure( )**
Return value: 8-bit integer set as per the distribution shown in Figure 1.2. The remaining bits of the return value should be set to zero.

Blind Spot Detection System

Cars today are fitted with a blind spot detection system that alerts the drivers of static or dynamic objects present in the vehicle's blind spot on either side of the vehicle at any time. Radar sensors are installed on both sides of the vehicle that emit radio waves and detect their reflection off nearby objects. Based on the time taken to receive a response, the sensor calculates the distance of the object from the vehicle.

A text file "BSDetection.txt" contains 25 records each with an integer value is provided. Each record contains a single number as per the following convention:

(a) The LSB indicates the side of the vehicle which is being checked. LSB is 0 => Left side, LSB is 1 => Right side.

(b) The remaining bits of the number represent time taken in nanoseconds for the radar sensor to receive responses from objects (both stationary and mobile). This is indicated in Figure 1.3.

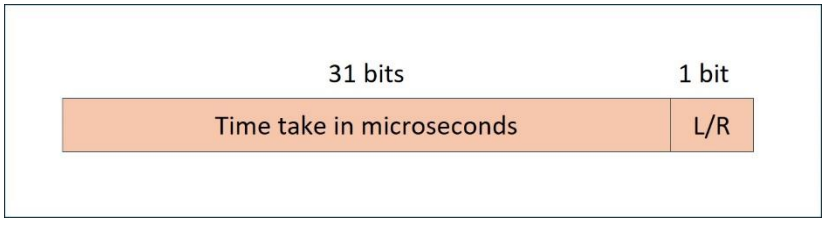

Figure 1.3

Distance between objects is calculated using the following formula:

Distance = Speed of radio waves in air * Time taken for reflection to reach sensor / 2

The speed of radio waves is approximately 3,00,000 km / second. For each record, your program should call the function given below and return which side's blind spot should be activated.

Incorporate the logic for blind spot detection system in a function with the following details:

Function: **activateBlindSpot( )**
Return value: 8-bit unsigned integer set as per the distribution shown in Figure 1.4. The remaining bits of the variable should be set to zero.

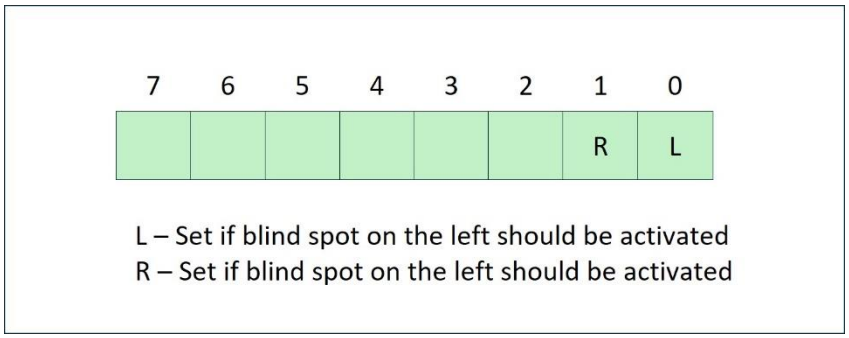

Figure 1.4

Airbag Actuator

Most cars today are fitted with airbags that inflate in front of the driver and the front passenger for extra safety in case the car meets with an accident. In real world systems, this decision is made by calculating the g-force value. g-force is the acceleration of a vehicle in terms of acceleration due to gravity (g). If the g-force value exceeds a threshold, then the airbags are activated, otherwise not.

A text file "ABActuator.txt" contains 25 records. Each record contains 5 values as per the following details:

First value - 4 bytes long (start time in seconds)
Second value – 4 bytes long (initial speed in kmph)
Third value – 4 bytes long (end time in seconds)
Fourth value – 4 bytes long (final speed in kmph)

Fifth value – 4 bytes long (threshold g-force value)

The g-force should be calculated as per the following formula:

g-force = (abs (final speed - initial speed) / time) / g

where time is the difference between start and end times.

For each record, your program should call the function given below to determine whether to deploy airbags or not.

Incorporate the logic for airbag actuator in a function with the following details:

Function: **activateAirbags( )**
Return value: an integer that indicates the status of the airbags.

Note: Remember to convert all values to SI systems for accuracy and correctness.

Reverse Camera Categorization System

A reverse camera is useful for a driver to navigate the car while reversing. This camera also indicates the distance of an object from the back of the car. The measured distance is categorized into multiple levels, for example, red, yellow, green, depending on the distance of the object from the car.

RED Category: 0 < distance <= 2 feet
YELLOW Category: 2 feet < distance <= 5 feet
GREEN Category: 5 feet < distance <= 10 feet

These categories are present in a lookup table represented in a file 'lookup_table.txt' containing 3 records as shown below:

Red 0 2

Yellow 2 5
Green 5 10

If the object does not belong to any of these categories, then it is categorized into a **White** category representing "None of These".

A sample file "ReverseData.txt" contains 25 records. Each record consists of a single number and contains the time taken by the reverse camera sensor to receive a response from an object, in nanoseconds.

The speed of radio waves in air can be assumed to be 3,00,000 km/sec.
Distance of object from car = (speed of radio waves * time taken) / 2

For each record, your program should call the function given below to classify objects into their appropriate categories.

Function: **classifyObjectsWhileReversing( )**
Return value: a string that specifies the category to which the object belongs. The returned string should be in Pascal casing. In case of wrong inputs, return an empty string.

Caution: Remember to convert all values to compatible units for performing valid calculations.

Driving Range Estimator

The purpose of this estimator is to determine how many kilometers a vehicle can travel for a given fuel level. The estimation is done by the Engine Control Unit or ECU based on the data that it receives from various sensors described below:

(a) Fuel Level Sensor: This sensor, typically a capacitive or ultrasonic sensor, measures the amount of fuel remaining in the fuel tank. It sends a signal to the vehicle's computer (Engine Control Unit or ECU)

indicating the fuel level percentage. Five possible values for this input are 10%, 25%, 50%, 75%, 100%.

(b) Mileage Sensor: This sensor measures the historical mileage of the car in kmpl.

(c) Engine RPM (revolutions per minute): This value indicates the engine's rotational speed. Based on this, there is a consumption multiplier calculated as follows:

RPM < 2000: 1
RPM >= 2000 and RPM < 3200: 1.2
RPM > 3200: 1.5

Based on these inputs the usable fuel and the estimated driving range can be calculated using the following formulae:

usableFuel = fuelLevel * fuelTankCapacity / 100
estimatedDrivingRange = usableFuel * mileage / consumptionMultiplier

Your program should read a set of 25 sets of values from a text file "DREstimator.txt". Each line will contain values in the following sequence:

Fuel tank capacity
Fuel level
Mileage
Engine RPM

For each set of values, your program should call the function given below and return the estimated driving range.

Function: **calculateDrivingRange( )**
Return value: Integer indicating estimated driving range. In case of wrong inputs, return -1.

Code Organization and Expectations

It is expected that you will strictly follow the instructions mentioned below while developing your solution. Note that if there is a typographical mistake in function name or its parameters, even though your solution is correct, it will **NOT** pass the tests.

(a) The "hackathonapp.c" file should contain the **main( )** function, which acts as the starting point. It should run an infinite/control loop, where each iteration of the loop does the following:

- Execute all the systems once (except tyre pressure monitoring system) by reading the next record from the corresponding input file for every system.

- For the tyre pressure monitoring system, a record needs to be read at random as specified in the problem statement.

- If the FILE pointer reaches the end of the file, and there are no records to read in the current iteration, then close the file and open it again to read from the beginning in the next iteration.

- The execution of the infinite loop should stop when a key is pressed on the keyboard. To achieve this, call the **kbhit( )** function defined in the "kbhit.c" file. This file should be used as is. **DO NOT** change the function definition or the file contents.

(b) All the input data files are present in the "files" directory, with appropriate filenames.

(c) All the function prototypes and corresponding structures needed are declared in the "hackathonapp.h" file for your reference. **DO NOT** modify "hackathonapp.h".

(d) The code needs to be organized in different files, one for each system described in the problem statement.

(e) A complete list of functions that you must implement for different systems is as follows:

- Lane Departure Warning System (laneDeparture.c):
  - `FILE* readLaneDepartureFile(FILE*, float* lane_width, float* car_width, float* left_sensor_distance, float* right_sensor_distance);`

    The file pointer returned is the position of the file pointer after reading the current record. All the other float pointers are for reading the record.
  - `int determineLaneDeparture(float lane_width, float car_width, float left_sensor_distance, float right_sensor_distance);`
- Tyre Pressure Monitoring System (tyrePressureMonitor.c):
  - `struct Tyre readTyreMonitorRecord(const char* filename, int record_num);`
  - `unsigned char monitorTyrePressure(struct Tyre ty);`

    `struct Tyre` is declared in the "hackathonapp.h" file.
- Blind Spot Detection System (blindSpotDetection.c):
  - `FILE* readBlindSpotDetectionFile(FILE* fp, unsigned long int* time_side, float* threshold);`

    The file pointer returned is the position of the file pointer after reading the current record. All the other pointers are for reading the record.

- `uint8_t activateBlindSpot(unsigned long int time_side, float threshold);`

- Airbag Actuator System (airbagActuator.c):

  - `FILE* readAirbagActuatorFile(FILE*, struct Airbag*);`

    The file pointer returned is the position of the file pointer after reading the current record. The `struct Airbag` pointer is for reading the record.

  - `int activateAirbags(struct Airbag);`

    `struct Airbag` is declared in the "hackathonapp.h" file.

- Reverse Camera Categorization System (reverseCameraCategorization.c)

  - `int readLookupTable(const char* filename, struct Category* ct);`

    The returned value specifies the number of categories present in the lookup table. The pointer to first category from the array present in the lookup table file should be present in the `ct` variable.

  - `FILE* readReverseCategoryFile(FILE* fp, unsigned long int* time_taken);`

    The file pointer returned is the position of the file pointer after reading the current record. The `unsigned long int` pointer is for reading the record.

  - `const char* classifyObjectsWhileReversing(struct Category* ct, int cat_count, unsigned long int time_taken);`

    `cat_count` is the total number of categories read from the lookup table.

`struct Category` is declared in the "hackathonapp.h" file.

- Driving Range Estimator System (drivingRangeEstimator.c):

  - `FILE* readDrivingRangeEstimatorFile(FILE* fp, struct CarFuel* cf);`

    The file pointer returned is the position of the file pointer after reading the current record. The `struct CarFuel` pointer is for reading the record.

  - `int calculateDrivingRange(struct CarFuel cf);`

    `struct CarFuel` is declared in the "hackathonapp.h" file.

(f) The complete application can be executed by running the "run_app.sh" script.

(g) Working of each system can be validated by running the program against some "given" tests. These tests can be executed by running the "run_given_tests.sh" script.

(h) **DO NOT** modify any file in "giventests" folder.

(i) Your solution will be evaluated against hidden test cases and checked for correctness.