

Real Time Systems (CS G553)
RTS Project

Project Title: Automated Vehicle Control using Trampoline

Group No. :1

SAIRAJ DHARWADKAR (2023H1030025G)

SAKSHI SINGH (2023H1030046G)

SAISREE CHINTHALAPANI (2023H1030049G)

AJEET KUMAR Vishwakarma (2023H1030060G)

Introduction:

In the evolving landscape of automotive technology, the integration of smart decision making systems is an important step towards safer and efficient transportation. Our project aims to contribute to the vision of designing and implementing a robust decision making architecture for automotive vehicles by using a combination of proximity sensors , gyro sensors, pressure sensors and an OSEK/VDX based RTOS called Trampoline. The framework is built on trampoline RTOS which takes inputs from various sensors to gather critical information about the vehicle surroundings. These inputs help in generation of essential outputs - Acceleration Intensity, Brake Intensity and Steering Intensity to determine vehicle movement and direction. To have seamless exchange of information between multiple devices , we are using CAN. A CAN Driver has been implemented, the hardware communication bus is simulated using Linux Virtual CAN(vCAN).

Objective:

To create smart decision making software for automotive vehicles using Proximity Sensors, Gyro Sensors, Pressure Sensors, etc. to safely drive the car.

Motivation:

Automobile vehicles are a necessity and for a long time have been integrated with technology. We intend to learn, understand and implement a simple smart decision making architecture that will allow the car to navigate on the roads safely without human intervention.

Architecture:

- **RTOS:**

To realize the project we have used an OSEK/VDX based RTOS called *Trampoline*. (Git: <https://github.com/TrampolineRTOS/trampoline>). We have created tasks, oil files and drivers for our project based on Trampoline architecture.

- **Input Devices:**

1. An array of Proximity Sensors in the front that will sense the Distance of obstacles from the car.
2. Proximity Sensors on Left and Right of the car that will fetch the presence of obstacles on the sides of the car.
3. Sensors on left and right of the car to check for lane check.
4. Gyro Sensor to get the Rotation of the car for checking if it is going uphill or downhill.

- **Generated Output:**

1. Acceleration Intensity, a parameter (values 1 to 10 for forward movement, and -1 to -10 for reverse) that will determine the acceleration of the vehicle.
2. Break Intensity, a parameter (values 1 to 10) that will determine the breaking force.
3. Steer Intensity, a parameter (values 1 to 10 for left steer, and -1 to -10 for right steer) that will determine the angle of turn.

- **Communication:**

To establish communication between multiple devices that send information and are supposed to receive information we have utilized the CAN.

We have implemented a CAN driver for posix using CAN RAW sockets that reads data on the CAN bus and writes data to the CAN bus. Trampoline provides a basic framework for implementing CAN services, we need to implement the required protocol that will govern the usage of the CAN bus.

We have used the can module of the Linux Kernel to establish a virtual CAN bus.

Environment SetUp:

- **Setup VCAN on Linux:**

Linux has a module that needs to be initialized to create a Virtual CAN bus. Run the following commands to create a CAN bus with name vcan0 =>

```
sudo modprobe vcan
```

```
sudo ip link add dev vcan0 type vcan
```

```
sudo ip link set up vcan0
```

CAN Driver:

File: *tpl_posix_can_driver.c, tpl_posix_can_driver.h*

Path: <TrampolineRoot>/libraries/drivers/can/posix

The Trampoline Architecture contains a structure that is responsible for invoking message read and write, the functions that will contain the actual read and write operations are provided to the `tpl_can_controller_t` struct.

The below functions are part of the struct and have been implemented:

1. *Driver Initializer - can_posix_driver_init*: This function will initialize the CAN socket and bind it to the “vcan0” bus(virtual can bus created in Linux). The linux kernel will create a RAW CAN Socket, make an entry of it in the file descriptor table and return to us the id of that entry. This id will be used in read and write operations to access the socket.
2. *Driver Writer - can_posix_driver_transmit*: This function is responsible for writing the data on the Bus. The function provides us the payload which will be present inside the `Can_PduType` struct in Trampoline. This function performs the following tasks:
 - a. Check if the payload is greater than 8 bytes.
 - b. If yes, send the data into small chunks of 8 bytes each time using the CAN socket
 - c. Set the CAN frame ID which will be an identifier that will act as the ECU identifier as well as an indicator of priority of the message on the Bus. In general we expect the ECU that produces more necessary data to get a higher priority or lesser CAN frame ID value.
 - d. Set the payload size as 8 bytes.
 - e. For the very first frame of the Payload send extra two bytes `CAN_PT_HEAD1` and `CAN_PT_HEAD2`.
 - f. Now if there are some bytes left to be written and there is space for the TAIL bytes then put the tail bytes `CAN_PT_TAIL1` and `CAN_PT_TAIL2` and send the packet, else send the tail bytes in the next packet.
3. *Driver Reader - can_posix_driver_receive*: This function reads data from the bus and saves it back into the `Can_PduType` Struct. This function performs the following tasks:
 - a. It starts reading from the CAN socket, each time only 8 bytes are received as a packet.
 - b. While reading if a packet has the Header bytes then the payload has started. Check the CAN frame ID of this data. We need to read the packets of this CAN frame ID alone
 - c. If the Tail bytes of the packet are equal to defined tail bytes then message will end
 - d. Do not write the Head and Tail bytes in the Payload.
4. *Data Availability – can_posix_driver_is_data_available*: Check if data is present to read.

A controller “*can_posix_driver_controller*” has been defined that has reference to the above function and will be used for testing communication.

Files Created:

1. *config.oil*:

Path : <TrampolineRoot>/goil/templates/libraries/drivers/can/posix

Trampoline uses GOIL as an OIL compiler, and the Libraries that the project will require as dependencies. This is done in this config file, where the header and C file are mentioned.

2. *can_read_write.c*:

Path: <TrampolineRoot>/examples/posix/can_read_write

To test the communication on the Bus, a Task has been created that first writes some bytes to the Bus then Starts Listening for Data on the Bus.

3. *can_read_write.oil*:

Path: <TrampolineRoot>/examples/posix/can_read_write

The OIL configuration that will be used during compilation by GOIL. Mention the Library = can, so that the CAN posix driver is built and included as a dependency.

4. *mycar.oil*:

Path: <TrampolineRoot>/examples/posix/Car/mycar.oil

This OIL file contains the definition & properties (such as priority, preemptable nature of task, number of its instances that can be present in the ready queue, whether task will be in ready state at startup) of different tasks defined in mycar.c. The tasks defined here are: *Brake*, *Steer*, *Accelerate*, with Brake having the highest priority, followed by Steer and then Accelerate.

5. *mycar.c*

Path: <TrampolineRoot>/examples/posix/Car/mycar.c

This C file contains the implementation details of all the tasks defined in *mycar.oil* file.

Tasks Created for Car Functioning:

Initially, we are defining *Obstacle_threshold*(up to this unit front 5 sensors can sense), *Car_width*, *Road_width*, *Road_threshold*(road sensors can sense up to this distance unit), *min_lr_road*(minimum distance to maintain from the edge of the road), *max_braking*=10, *max_acceleration*=10, *lr_sensor_threshold*(range of sensing left or right side for obstacle), *To-steer*=0(Not to Steer), *max_left_steer*=-10, *max_right_steer*=10. And initializing some global variables: *To_steer*=0, *braking*=0, *acceleration*=10.

1. *Accelerate*:

This task is used to determine the speed intensity of the car at a particular instant by leveraging captured sensor data. First, we check whether an obstacle exists within the maximum sensing distance of the sensors. Then using this, we calculate the proximity of the nearest obstacle. If the nearest detected obstacle falls within the range of the minimum distance required for acceleration, the acceleration is set to 0. Otherwise, we calculate the acceleration based on the proximity of the nearest detected obstacle.

2. *Brake:*

This task is used to determine the braking intensity of the car at a particular instant by leveraging captured sensor data. Initially, we evaluate whether an obstacle is present within the maximum sensing distance of the sensors, and also calculate the proximity of the nearest obstacle. If the nearest detected obstacle is within the range of the maximum distance required for braking, full braking is applied. Otherwise, we calculate the braking intensity based on the proximity of the nearest detected obstacle.

3. *Steer:*

This task gives by which intensity the car to steer and in which direction. To steer, we are checking the `sensor_data` array of 9 elements. From the first 5 elements, we create an array `sensor_data` to reflect the coordinates of obstacles in front of the car, then we steer according to that. And successive 2 elements notify if any car is on the left or the right respectively and successive 2 elements notify if the car is going offroad. We take care of all possible cases if there is any obstacle in front of the car according to the sensor data we created.

Cases like:- When obstacles are on either the left or right front of the car, some of the critical cases are when obstacles are in the mid-front of the car, while taking care of the running car on the left or right car as we are going to steer that side. At last, we take care of the Road edge so the car will not go off-road.

Testing Utilities Used:

The `can-utils` provides great tools for performing operations related to CAN => `sudo apt-get install can-utils`

To check the network => `sudo ip link ls`

To continuously read all data on the bus => `candump vcan0`

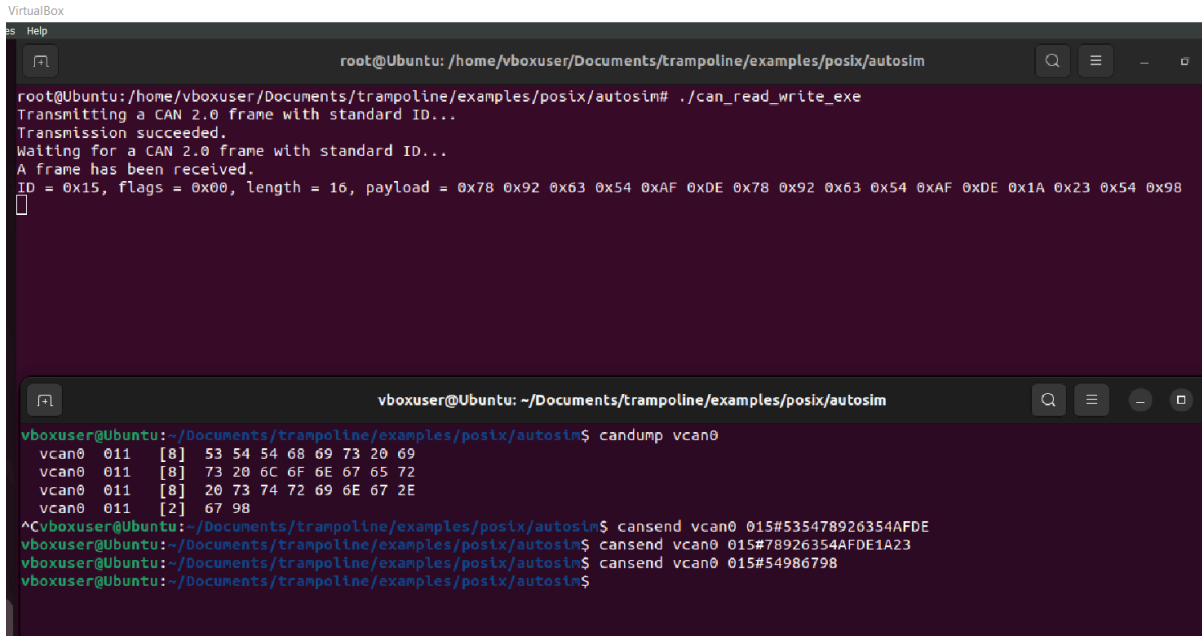
To send a message on the can bus => `cansend vcan0 <id num>#<msg>`

e.g.: `cansend vcan0 012#53546152AFC66798`

To send random auto generated messages on the can bus => `cangen vcan0`

Output Screenshot

- *For CAN Driver:*

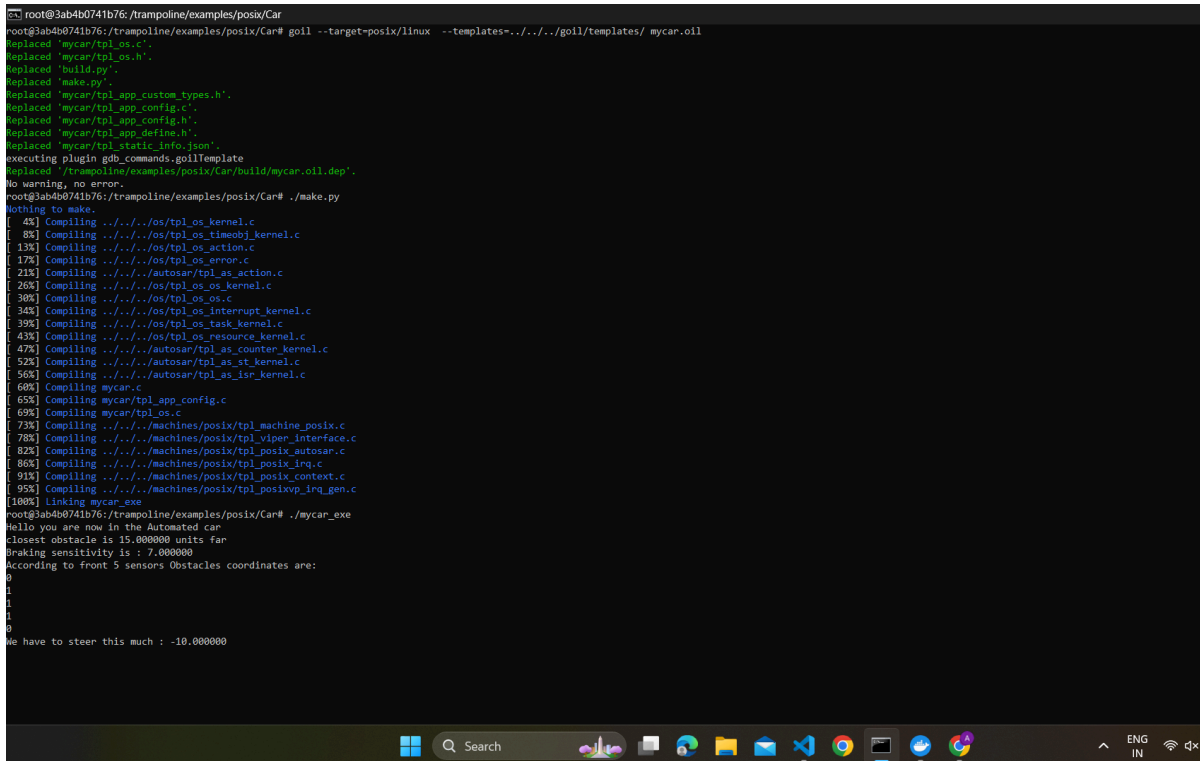


```
VirtualBox
root@Ubuntu: /home/vboxuser/Documents/trampoline/examples/posix/autosim

root@Ubuntu: /home/vboxuser/Documents/trampoline/examples/posix/autosim# ./can_read_write_exe
Transmitting a CAN 2.0 frame with standard ID...
Transmission succeeded.
Waiting for a CAN 2.0 frame with standard ID...
A frame has been received.
ID = 0x15, flags = 0x00, length = 16, payload = 0x78 0x92 0x63 0x54 0xAF 0xDE 0x78 0x92 0x63 0x54 0xAF 0xDE 0x1A 0x23 0x54 0x98

vboxuser@Ubuntu: ~/Documents/trampoline/examples/posix/autosim$ candump vcan0
vcan0 011 [8] 53 54 54 68 69 73 20 69
vcan0 011 [8] 73 20 6C 6F 6E 67 65 72
vcan0 011 [8] 20 73 74 72 69 6E 67 2E
vcan0 011 [2] 67 98
^Cvboxuser@Ubuntu: ~/Documents/trampoline/examples/posix/autosim$ cansend vcan0 015#535478926354AFDE
vboxuser@Ubuntu: ~/Documents/trampoline/examples/posix/autosim$ cansend vcan0 015#78926354AFDE1A23
vboxuser@Ubuntu: ~/Documents/trampoline/examples/posix/autosim$ cansend vcan0 015#54986798
vboxuser@Ubuntu: ~/Documents/trampoline/examples/posix/autosim$
```

- *For Car Functions:*



```
root@3ab4b0741b76:/trampoline/examples/posix/Car#
root@3ab4b0741b76:/trampoline/examples/posix/Car# goll --target-posix/linux --templates=../../goll/templates/ mycar.oil
Replaced 'mycar/tpl_os.c'.
Replaced 'mycar/tpl_os.h'.
Replaced 'build.py'.
Replaced 'make.py'.
Replaced 'mycar/tpl_app_custom_types.h'.
Replaced 'mycar/tpl_app_config.c'.
Replaced 'mycar/tpl_app_config.h'.
Replaced 'mycar/tpl_app_define.h'.
Replaced 'mycar/tpl_static_info.json'.
Executing plugin gdb_commands.golltemplate
Replaced '/trampoline/examples/posix/Car/build/mycar.oil.dep'.
No warning, no error.
root@3ab4b0741b76:/trampoline/examples/posix/Car# ./make.py
Nothing to make.
[ 4%] Compiling ../../../../os/tpl_os_kernel.c
[ 8%] Compiling ../../../../os/tpl_os_timeobj_kernel.c
[13%] Compiling ../../../../os/tpl_os_action.c
[17%] Compiling ../../../../os/tpl_os_error.c
[21%] Compiling ../../../../autosar/tpl_as_action.c
[26%] Compiling ../../../../os/tpl_os_os_kernel.c
[30%] Compiling ../../../../os/tpl_os_os.c
[34%] Compiling ../../../../os/tpl_os_interrupt_kernel.c
[39%] Compiling ../../../../os/tpl_os_task_kernel.c
[43%] Compiling ../../../../os/tpl_os_resource_kernel.c
[47%] Compiling ../../../../autosar/tpl_as_counter_kernel.c
[52%] Compiling ../../../../autosar/tpl_as_st_kernel.c
[56%] Compiling ../../../../autosar/tpl_as_isr_kernel.c
[60%] Compiling mycar.c
[65%] Compiling mycar/tpl_app_config.c
[69%] Compiling mycar/tpl_os.c
[73%] Compiling ../../../../machines/posix/tpl_machine_posix.c
[78%] Compiling ../../../../machines/posix/tpl_viper_interface.c
[82%] Compiling ../../../../machines/posix/tpl_posix_autosar.c
[86%] Compiling ../../../../machines/posix/tpl_posix_irq.c
[91%] Compiling ../../../../machines/posix/tpl_posix_context.c
[95%] Compiling ../../../../machines/posix/tpl_posixvp_irq_gen.c
[100%] Linking mycar_exe
root@3ab4b0741b76:/trampoline/examples/posix/Car# ./mycar_exe
Hello you are now in the Automated car
Closest obstacle is 15.000000 units far
Braking sensitivity is : 7.000000
According to front 5 sensors Obstacles coordinates are:
0
1
1
0
0
We have to steer this much : -10.000000
```

First compiled the mycar.oil file and then ./make.py then execute the mycar_exe file taking the Case: sensor_data={50,30,45,15,60,3,1,3,3} means obstacle_sensor={0,1,1,1,0} refers that obstacle is in front of car and minimum obstacle distance is 15 and there is car on the right side and left lane is free to move and both the side road is available so no off-road. Result: as we can go to either left or right but there is a car on right so to steer left. To_steer=-10. Some of the critical cases are: {0,0,1,0,0}, {0,1,0,1,0}, {0,1,1,1,0}

Contribution

Ajeet Kumar Vishwakarma- Implementation of Car functionalities, testing and debugging

Sairaj Dharwadkar- Implementation of CAN driver

Saisree Chinthalapani- Implementation of CAN driver

Sakshi Singh- Implementation of Car functionalities, testing and debugging