



**MASTER DEGREE in EMBEDDED COMPUTING SYSTEMS**  
**A.Y. 2016 - 2017**

# **Report of VxWorks Application's Project**

## **Real-Time Systems**

**Project n. 36: VxWorks-App.**

**Implement a real-time application on VxWorks, on an embedded platform. Both the kernel and the platform will be provided by the RETIS Lab.**

### **Full Professor**

Giorgio C. Buttazzo

### **Students**

Silvio Bacci  
Andrea Baldecchi

Student ID: 507073  
Student ID: 555629

Email: ciabbi94@live.it  
Email: a.baldecchi@yahoo.it

Delivery date: 25/02/2017

# Index

## **1. General Description**

- 1.1. Introduction
- 1.2. BeagleBone board
- 1.3. Unreal Engine
- 1.4. Car application

## **2. Physical Models**

- 2.1. Model of a car
- 2.2. Sensor model
- 2.3. Automatic mode and obstacle avoidance
  - 2.3.1. Going to a specific destination
- 2.4. Escape from collision
- 2.5. Model of obstacles and finishing line

## **3. Design Choice**

- 3.1. User settable parameters
- 3.2. Car attitude data and sensors
- 3.3. Ray of the obstacles
- 3.4. Distance of the obstacle from the finishing line
- 3.5. Car's speed and direction changing
- 3.6. Car's view

## **4. Tasks**

- 4.1. Task division
- 4.2. Period of the application
- 4.3. Estimated Worst Case
- 4.4. Scheduling
- 4.5. Rate of Unreal Engine
- 4.6. Concurrency
- 4.7. Ptask library

## **5. User Interface and User Manual**

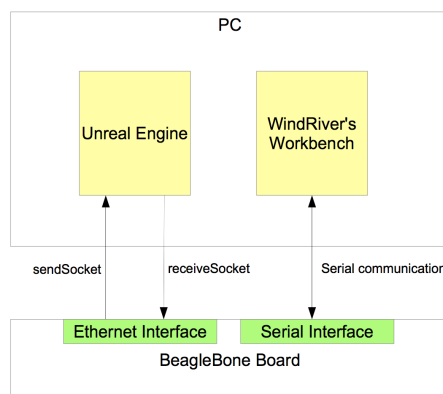
- 5.1. User Interface
- 5.2. User Manual

# Chapter 1

## General description

### 1.1 Introduction

The application we developed is a driving simulator which allows the user to drive a car, that in this case is a BMW X5, in a predefined map with some obstacles. The car can also move by itself, reaching a predefined destination point avoiding obstacles, if needed. The software system that supports the game is composed by two applications: on one side a graphical software that shows the virtual world and all the other components and on the other side a calculation software that processes the information relative to the physics aspects of the context. The graphical software is an Unreal Engine application running on a general purpose operating system mounted on laptop or on a desktop computer. The other one runs in a computing board equipped with a real-time operating system called VxWorks developed by WindRiver. These softwares have to exchange some informations by using the UDP protocol in order to make the whole system works properly. The first chapter contains a general description of system's components and their features. In the second chapter we introduce and characterize all the physical models managed by the application. Third and Fourth chapters explain the technical and software details of the given system. Finally we conclude with a simple user manual in order to make the simulator usage easier.



### 1.2 BeagleBone board

The BeagleBone Black board is an open source hardware design based on the AM335x 1GHz ARM® Cortex-A8. We used it to develop a Real-Time application that was developed by using the primitives of the operating system called VxWorks and developed by WindRiver. VxWorks is a Real-Time ticked operating system with a priority-based scheduling. The idea is to use the board BeagleBone in order to run on it our application without using real sensors. Given the lack of available drivers for the board, we decided to use only:

- Serial communication driver
- Network communication driver

Thanks to the serial driver we can send to the board some information directly from the keyboard and on the other side we can display on the terminal the output of the application. Thanks to the network driver we can send and receive packets to and from the network.

### **1.3 Unreal Engine**

In order to represent the car, we chose to use a 3D representation by using the graphic engine “Unreal Engine” that allowed us to create a virtual car in a virtual world. As we will see, we can also use this engine to simulate some sensors, like proximity sensors, directly attached to the car. The network interface is used to create a communication channel between the board and the Unreal Engine. In this way the board receives the sensory data, sending at the same time new attitude of the car.

### **1.4 Car application**

The application that manages car's behaviour allows the user to drive the car in the world where he wants (manual mode), or let the car moving by itself and going to a specific destination avoiding all the obstacles (automatic mode). According to this, the application has to manage some situations of this context such as obstacle avoidance and collisions.

# Chapter 2

## Physical Models

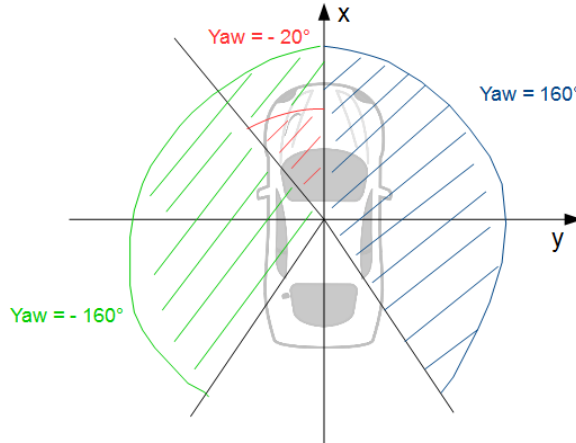
### 2.1 Model of a car

As we said in the previous chapter the car is represented in a 3D world but all the logic that represents the behaviour of the car is executed and stored only in the board. At the beginning, we have to explain how car's position and orientation are updated according to the changes imposed by the user or by the external events.

We have to specify that the car moves on a 2D surface, so we do not need the Z component in order to represent its position. The car is described by the following vector:

$$[x, y, yaw, speed]$$

The first pair of variables describe car's location in the map with meters as unit of measure. The "yaw" variable contains the information relative to the orientation of the car expressed in degrees. Finally the last variable describe the current velocity of the car in m/s. From now on, we will refer to *i-th* element of the vector as *element-name<sub>car</sub>*. Yaw is simply an angle, used by the Unreal Engine as in the following image:



Every time we want to update the position we need to use the previous position and then we need to change it by using the motion hour law, that is:

$$x = x_0 + v * t$$

Using the following formula we compute the module of movement produced in a given interval of time:

$$\lambda = speed_{car} * \Delta t$$

where:

- $\lambda$  is the movement produced
- $\Delta t$  is the interval of time between two consecutive updates

Using the motion hour law, we compute new car's position by splitting the movement produced into the 2 axis with the following system:

$$\begin{cases} x_{\text{car}} = x_{\text{car}} + \lambda * \cos(\text{yaw}_{\text{car}}) \\ y_{\text{car}} = y_{\text{car}} + \lambda * \sin(\text{yaw}_{\text{car}}) \end{cases}$$

## 2.2 Sensor Model

In Unreal Engine we simulate a sensor as a segment from the car to a certain distance. Formally we can define a sensor as a bidimensional vector:

$$[l, \alpha]$$

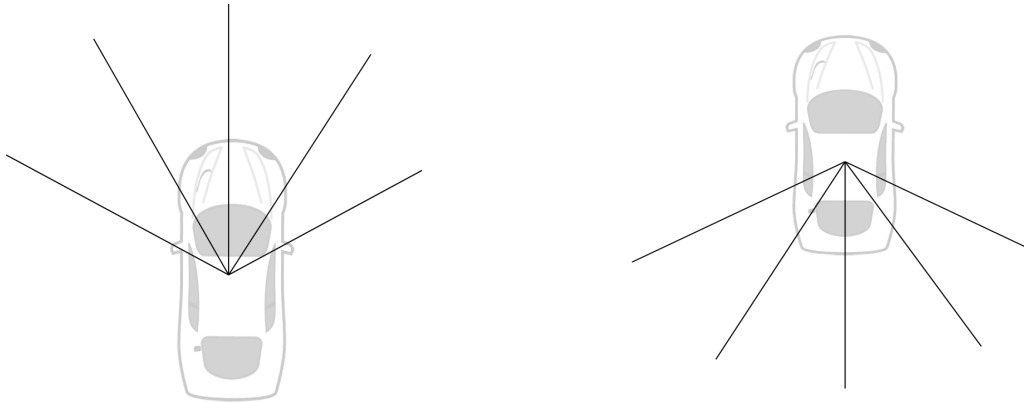
where:

- $l$  is sensor's length expressed in meters and computed starting from the car's extremity
- $\alpha$  is the orientation angle of the sensor starting from car's direction (note that  $\alpha$  is an integer multiple of  $30^\circ$ )

In our case, sensors' set is the following:

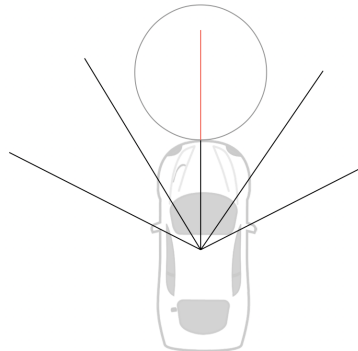
$$\begin{bmatrix} 5, & -60 \\ 5, & -30 \\ 5, & 0 \\ 5, & 30 \\ 5, & 60 \end{bmatrix}$$

We used also a second specular set of sensors that are attached to the back of the car. In any instant, we use only a sensors' set according to the direction of the car by disabling the other set. In the following images we show how the sensors are orientated:



We used a sensor to obtain a floating point number that indicate the distance between the extremity of the car and the nearest obstacle standing on its trajectory. Considering this distance, ray's orientation and car's location we can easily compute the exact location of the obstacle in the map.

Simulating collision involves using proximity sensors and checking whether if the distance detected by the sensors is less or equal than 0. In that case we collided an obstacle and then we have to stop the car. From that moment on we can only go back, of course. On the other side if we collided while we were going back we can go only forward. An example of a collision is the following one:

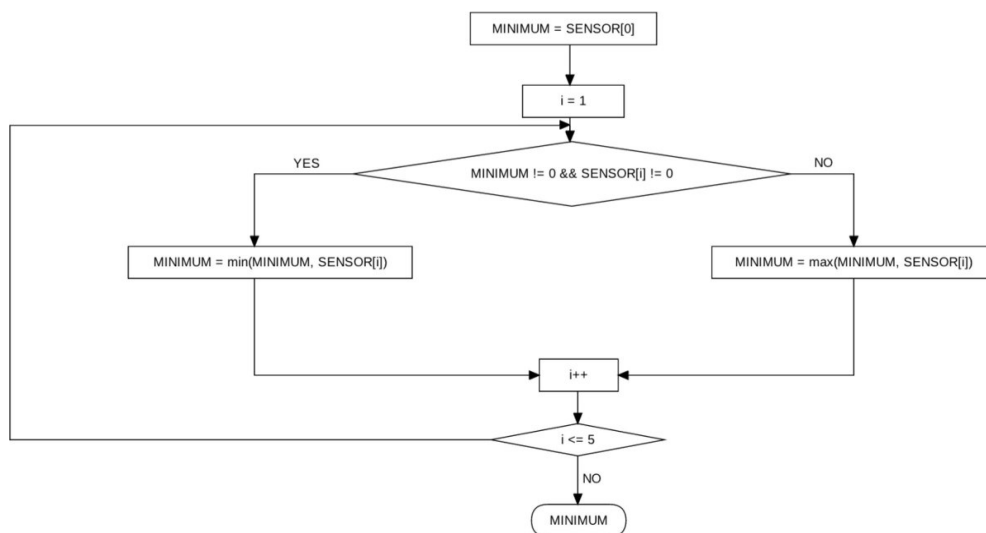


### 2.3 Automatic mode and obstacle avoidance

In the automatic mode we need also to avoid obstacles. We can use the sensory data in order to know the actual distance from obstacles. If an obstacle is detected we have to choose in which direction we want to escape. For instance, if an obstacle is detected on the left side of the car we want to turn on the right.

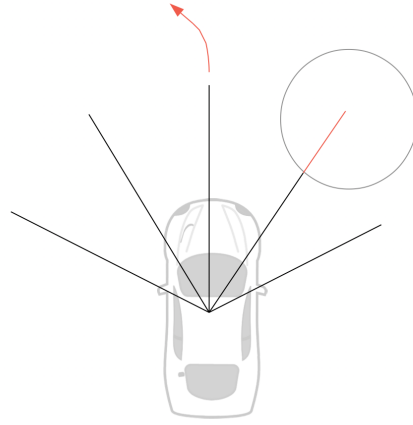
In order to choose the direction, every time we detect an obstacle we need to:

- Compute which sensor has the minimum detected distance as in the following diagram:



- If the chosen sensor is the central one:
  - We need to recompute the minimum distance among the others by using the previous diagram without considering the central sensor which index is 2.
- If the chosen minimum is on the left side we need to turn on the right
- If the chosen minimum is on the right side we need to turn on the left

In order to explain our logic, you can see the following example:



### 2.3.1 Going to a specific destination

In the automatic mode we need to go to a specific destination. In order to do it, every step we need to:

- Transform the yaw angle, which can be either positive or negative, in to an angle between 0 and 360 degrees that we called  $\theta$ .
- Compute car's trajectory in according to position and orientation of the car. The trajectory is given by the following straight line equation:

$$x = x_{\text{car}} + m * (y - y_{\text{car}})$$

where:

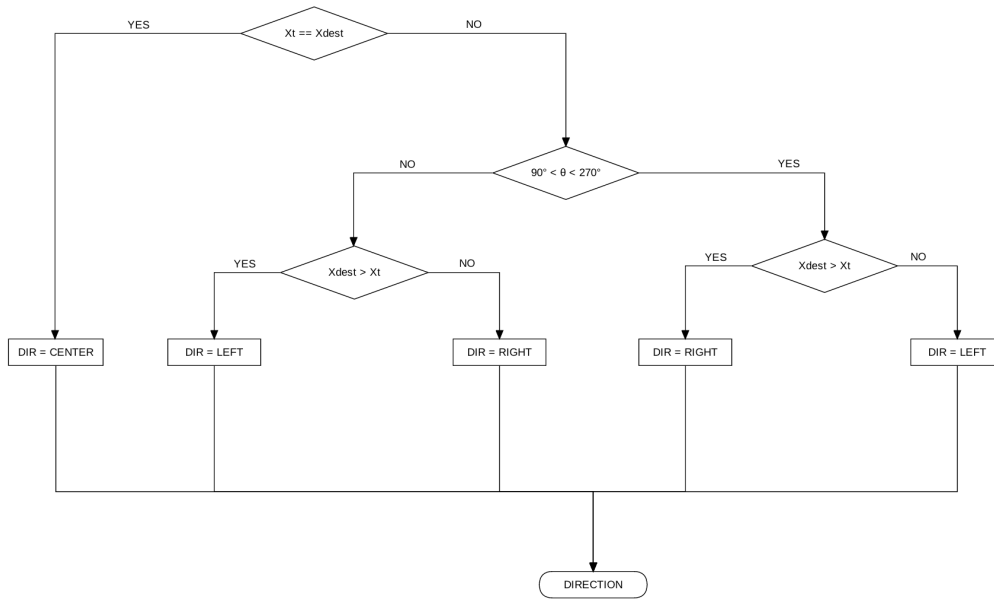
$$m = \tan(\theta)$$

- Compute the current destination point  $(x_t, y_t)$  by substituting the y coordinate of the destination point  $(x_{\text{dest}}, y_{\text{dest}})$  as in the following formula:

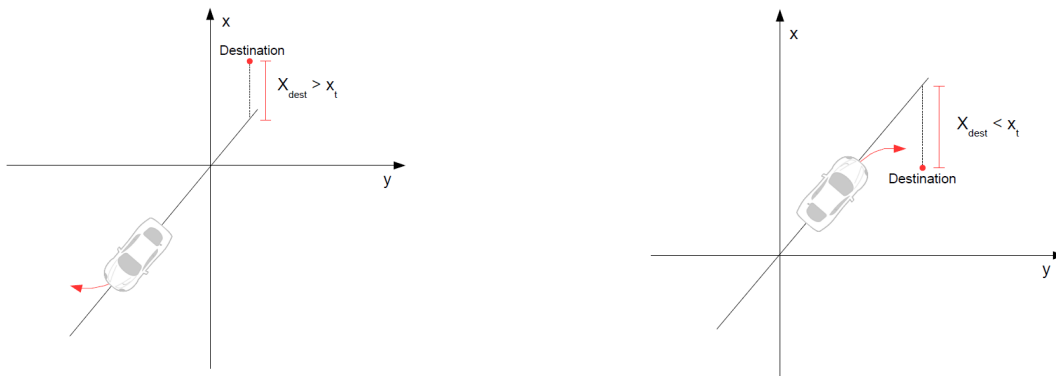
$$\begin{cases} x_t = x_{\text{car}} + m * (y_{\text{dest}} - y_{\text{car}}) \\ y_t = y_{\text{dest}} \end{cases}$$



- Choose turning direction according to the logic described by the following diagram:



In order to explain our logic, you can see the following example:



## 2.4 Escape from collision

When a collision is detected in the automatic mode, we stop the car and after that we move it in the opposite direction for a fixed safety distance checking and avoiding obstacles. After that we re-enable the automatic driving mode. If a collision happens in manual mode, we stop the car and we disable all the movement commands except the only one allows us to escape from the obstacle.

## 2.5 Model of obstacles and finishing line

In order to represent an obstacle in Unreal Engine we used cylinders. In fact the application leads us to choose some particular 3D figures. The cylinder is also a figure that our sensors can detect. In the following chapter we will see the minimum ray of the cylinder and in particular why we need a minimum ray.

# Chapter 3

## Design Choice

In this chapter we will explain the design choices we chose during the development of the application.

### 3.1 User settable parameters

In the manual modality the user can drive the car by choosing the speed (by pressing the “UP” and “DOWN” arrows on the keyboard) and the direction (by using the “RIGHT” and “LEFT” arrows). The user can choose also when he wants to change the modality from manual to automatic and vice versa, by pressing the “m” key on the keyboard. The user can reset the play by pressing the “r” key. The user can also change view by pressing the “v” button. Besides the user can choose to show or hide the sensors by pressing the button “s”. Finally, we can change tasks' period in response to button “o” or “p” pressure. Respectively, “o” to decrease the period and “p” to increase it.

### 3.2 Car attitude data and sensors

As we explained before the car has some proximity sensors around the car. We have exactly 5 sensors if we are moving forward and 5 different sensors if we are moving back. Totally we need 10 sensors.

As we said the attitude of the car is described by the X and Y components of the position, by the speed and by the yaw angle. These are the parameters we need to send to Unreal Engine in order to change the attitude of the car. On the other side Unreal Engine sends us some informations as:

- Distances detected by the sensors
- If a collision is detected or not
- If an obstacle is detected or not

In general we send 2 different structures that are “Car” and “SensorData” that are reported in the following lines:

```
struct Car {
    float X;           // Car's position on X axis in meters
    float Y;           // Car's position on Y axis in meters

    float yaw;         // Car's direction in radians

    int speed;         // Car's speed in m/s
    int view;          // Current car's view
    int sensorDirection; // Direction in which sensors are attached
};

struct SensorData {
    int collisionDetected; // Returns 1 if a collision was detected

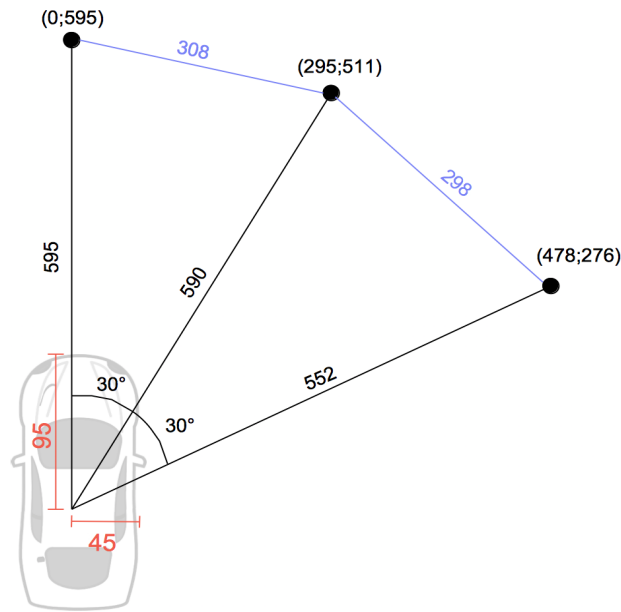
    int obstacleDetected; // Returns 1 if an obstacle was detected

    float distanceFromSensor0; // Distance from the sensor to the left
    float distanceFromSensor1;
    float distanceFromSensor2; // Distance from the central sensor
    float distanceFromSensor3;
    float distanceFromSensor4; // Distance from the sensor to the right
};
```

### 3.3 Ray of the obstacles

In Unreal Engine every sensor is composed by a vector of a given length of 5 meters starting from car's extremities. According to this situation we need to be sure that an obstacle can't be between 2 sensors without being detected. So we can compute the distance between the sensors and we can choose a minimum ray for the cylinder in order to avoid this kind of situation.

The minimum ray we could use is of 2 meters but according to graphical motivations we chose to use cylinders with 3.5 meters of ray. With this choice we are sure that no obstacles can't be undetected. The computation is in the following picture:



### 3.4 Distance of the obstacle from the finishing line

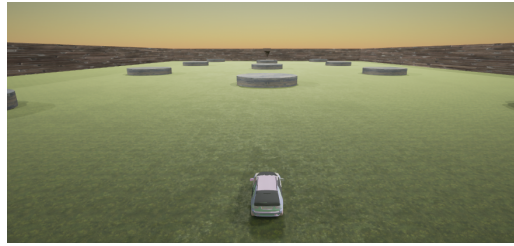
In order to reach our destination we need to put all the obstacles far from the destination otherwise the car will continue to avoid the obstacles without reaching the finishing line. In order to avoid this situation we need to put the obstacles at least 3 meters far from the destination. We need to underline that 3 meters is the actual sensor length. This length is settable by the programmer so when we set a new length we must also change the distance of the obstacles from the finishing line. Finally we have to note that it is better to talk about a finishing area. In fact, around the destination point we consider a circular area with a fixed ray of 1.5 meters to solve problems caused by numerical approximations. Without this expedient we could turn around the finishing line without reaching it never.

### 3.5 Car's speed and direction changing

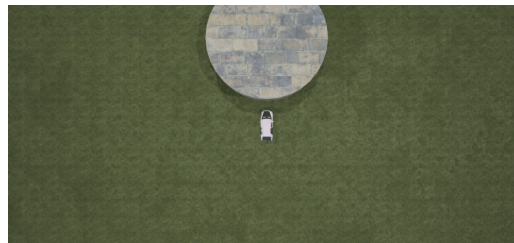
Every time we try to increase the speed what we do is to increase the actual speed of 1 m/s. Of course we have a maximum speed that can be reached and then we can't increase the speed again. The same thing we do for the driving back but in that case the maximum speed is less than the other one. Every time we try to turn the car we change the Yaw angle by increasing or decreasing it by 1 degree.

### 3.6 Car's view

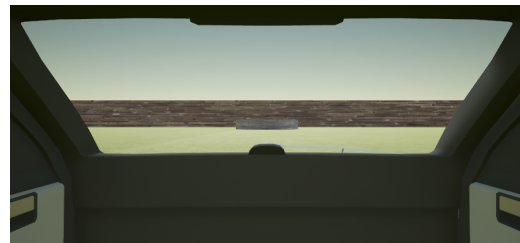
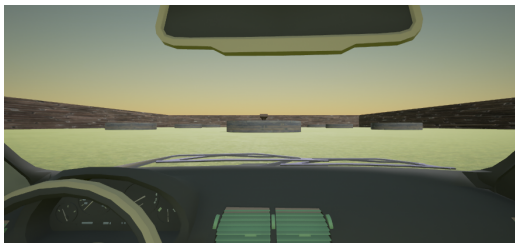
The default point of view is situated over the car in such a manner to see it entirely from the backside. In the following picture we can see it:



Another view is a simple car's overview as you can see in the following picture:



Finally, we developed an on board view that changes according to car movement (front and back movement). The following pictures explain it better:



# Chapter 4

## Tasks

### 4.1 Task division

In order to realize this application we chose to divide it in 4 tasks:

- Receiver task
- Keyboard task
- Engine/Car task
- Sender task

The receiver task has to receive the structure composed by the informations of the sensors from Unreal Engine. The keyboard task has to check if a key was pressed and in that case it has to understand which key was pressed. The engine task has to use the informations arrived from Unreal Engine and from the keyboard in order to update the current attitude of the car if it is necessary. Finally the sender task has to send the attitude of the car to Unreal Engine in order to update the image.

### 4.2 Period of the application

Usually a good game with a 3D representation needs an update frequency of the image that is about 30 fps. In order to reach this frequency we need to be sure to finish all the tasks 30 times in one seconds. The main problem is that VxWorks' operative system uses a system's tick. Every time that the tick arrives a routine is executed and it has to do everything that is necessary (of course this routine is waken up also if it has to do nothing). Every task can be delayed only by using the number of ticks as unit of measure. The maximum rate of the tick is 300 Hz, that is about 3.3 ms for every tick. This means that the minimum period for a task is 3.3 ms and this is a problem if we need to be sure that in one second all the 4 tasks execute 30 times. In fact if we want 30 fps we need to finish all the 4 tasks in 33 ms. So we have about 10 ticks to finish all the four tasks.

### 4.3 Estimated Worst Case

In order to estimate the worst case execution time of each task we tried to execute in a while loop every task with different inputs for 1000 times. At the end we saved the worst case execution time and the average execution time. For each task we applied this method 3 times. The results are in the following table:

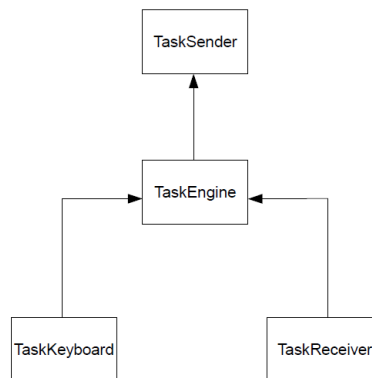
<b>WORST CASE</b>	<b>TaskSender</b>	<b>TaskKeyboard</b>	<b>TaskEngine</b>	<b>TaskReceiver</b>
<b>FIRST EXECUTION</b>	1	1	1	1
<b>SECOND EXECUTION</b>	1	1	1	1
<b>THIRD EXECUTION</b>	1	1	1	1

## 4.4 Scheduling

We need to execute the tasks in the following order:

- Receiver task (PRIORITY: 101)
- Keyboard task (PRIORITY: 102)
- Engine/Car task (PRIORITY: 103)
- Sender task (PRIORITY: 104)

In order to do it we decided to use a FIFO scheduling in which all the tasks have a different priority (the highest one is the receiver task), same arrival times and periods. To better explain precedence's constraints we can see the following precedence's graph:

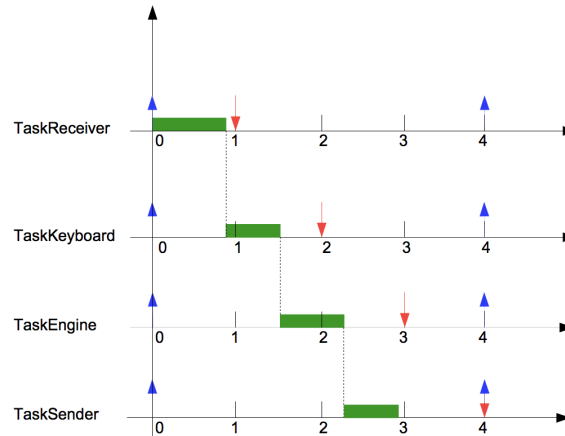


Note that even if the receiver task and keyboard task can execute in any order we decided to choose the previous order to simplify the code (otherwise we have simply to check which is the first task in execution by using a boolean variable).

According to the previous computation, the period of the tasks we choose in order to obtain at least 30 fps in Unreal Engine project, is the following one:

Task	Period	Relative Deadline	Task run time
TaskReceiver	4	1	1
TaskKeyboard	4	2	1
TaskEngine	4	3	1
TaskSender	4	4	1

In the following picture we show the typical and desired execution of the task set:



In order to compute the relative deadlines we start from the tasks' period divided by the number of the tasks and then we multiply the result by the index of the task (the index is associated to the execution's order). So we do not need to store it in the parameters because we can compute it easily and in this way we avoid to share the parameters' structure among all the tasks (note that when we change the period we should change all the deadline in the structure).

According to the fact that we have integer periods and integer ticks' number every time we want to check if a deadline was missed we need to use integer values, so we decided to use only periods multiple of the default period of 4 ticks. In fact when we test a deadline we check if the difference between the current tick's number and the tasks' arrival tick's number (that is an integer value) is greater than the relative deadline (that should be integer).

## 4.5 Rate of Unreal Engine

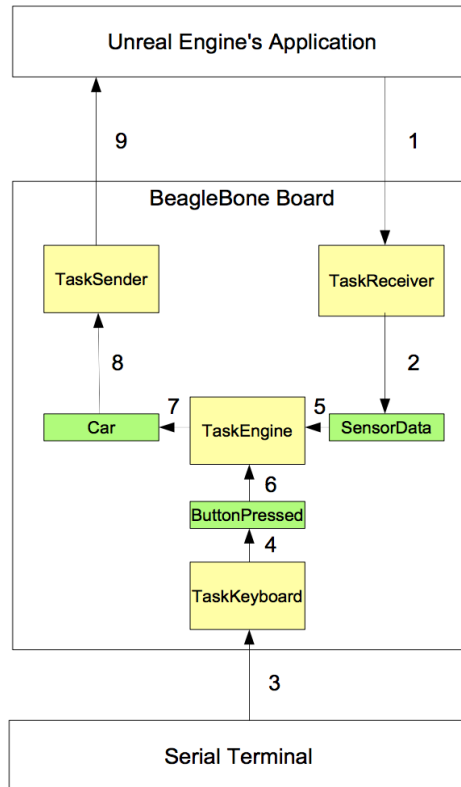
As we saw, in normal conditions all the tasks are executed in 4 ticks, that is about 13.3 ms. This means that the rate of Unreal Engine should be at least equal to the rate of the entire task's set. So Unreal Engine's rate should be 75 Hz, that is exactly 300 Hz (rate of the system's tick) divided by 4 (number of ticks that we need). When we increment the period of the task we can easily understand that the receiving rate increments proportionally with the period. For example if the period is 8 ticks we will receive 2 packets, and so on. Every time a packet arrives, it is saved in the buffer, and so when the receiver task runs we should drop all the packets except the last one which contains the most current information. So what we need to do is to execute more than once the receive operation (note that we use a not-blocking operation). In order to be sure that we have the most current information, every time we read the receiver's buffer once in addition (we are also sure that no deadline is missed).

## 4.6 Concurrency

In the application the shared structures are:

- **Pressed button**
- **Car's information**
- **Sensors' data**

In the following picture we can see which structures are used by which task (structures in green and tasks in yellow):

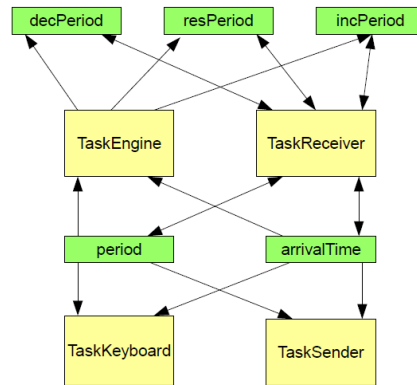


In the previous picture we showed the common structures useful for the all the computations of the application. There are also other shared variables that are useful to manage the scheduling and the periodic execution in general. These variables are:

- **period**: tasks' period that is equal for all tasks
- **arrivalTime**: arrival time of the first task that is the same of all the tasks
- **decPeriod**: boolean variable useful to say to the first task to decrement the period
- **incPeriod**: boolean variable useful to say to the first task to increment the period
- **resPeriod**: boolean variable useful to say to the first task to reset the period



In the following picture we can see the relation between tasks and shared variables (variables in green and tasks in yellow):



In general we do not have particular synchronization constraints, but we have to ensure the mutual exclusion access to shared data. In order to achieve this we used a semaphore for each shared variable. Semaphores are taken from the *semLib* library provided by VxWorks. All of them are managed by using the priority inversion protocol in order to be sure that FIFO order is guaranteed.

#### 4.7 Ptask library

In order to manage all the parameters related to the scheduling we provided a library in which we represent all the data we need and the functions to manage them.

As we said before when we increment or decrement the period we do it by using only periods that are multiple of 4. In general we assume also that 4 ticks is not only the default period but also the minimum one and 20 ticks is the maximum period (a period greater than this will compromise the performances too much).

Finally we want to underline that we are sure that no deadlines will be missed because of our small computational load. After lots of test we never get a missed deadline.

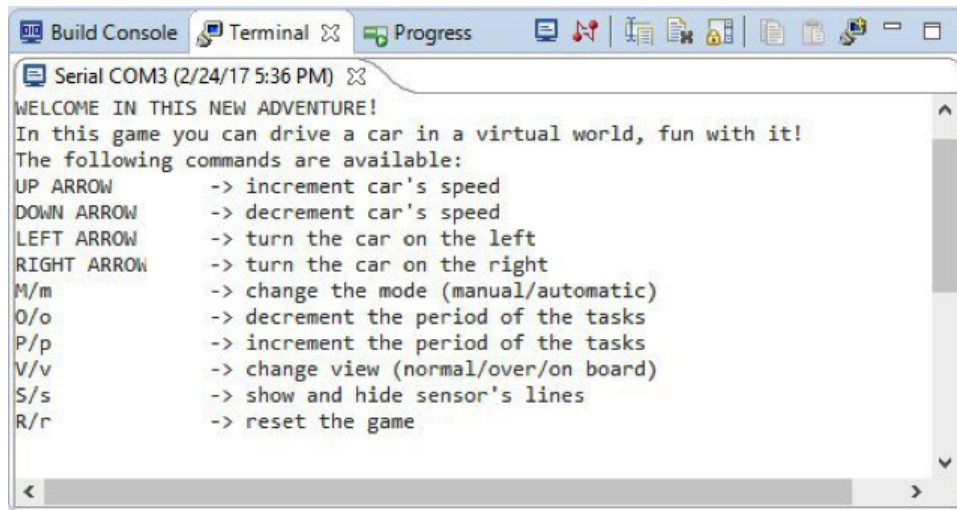
# Chapter 5

## User Interface and User Manual

### 5.1 User Interface

The user interface is composed by a VxWorks terminal from which we can communicate with the board by using only the keyboard. On the other side we can see the car by using Unreal Engine and then we can see the effects of our choices directly in Unreal Engine.

At the beginning of the game our application prints on the terminal all the available commands with the associated descriptions. In the following picture we show the example:



```
Build Console | Terminal | Progress
Serial COM3 (2/24/17 5:36 PM)
WELCOME IN THIS NEW ADVENTURE!
In this game you can drive a car in a virtual world, fun with it!
The following commands are available:
UP ARROW      -> increment car's speed
DOWN ARROW    -> decrement car's speed
LEFT ARROW    -> turn the car on the left
RIGHT ARROW   -> turn the car on the right
M/m          -> change the mode (manual/automatic)
O/o          -> decrement the period of the tasks
P/p          -> increment the period of the tasks
V/v          -> change view (normal/over/on board)
S/s          -> show and hide sensor's lines
R/r          -> reset the game
```

In general on the terminal we print the new period, if the user changed it, and, eventually, some error messages.

### 5.2 User Manual

In order to play the game we need to do some steps in the following order:

- 1. Connect to Retis Lab' VPN**

For this purpose we use OpenVPN that allows you to connect our machine to the VPN. In fact the license released by WindRiver is given to the Retis Lab network. Without this step you are not able to open WindRiver's Workbench.

- 2. Open WindRiver's Workbench**

This workbench is an Eclipse based workbench developed by WindRiver.

- 3. Build Project**

This step could be avoided but rebuilding the project guarantees you an automatic license check with positive response.

- 4. Connect the board**

First of all you have to connect the board to the machine using serial cable and Ethernet cable. After that you have to connect the board via software. For this purpose you have to create a new connection in the workbench and connect it using the specific button.

### **5. Open terminal**

Now you have to open a serial terminal typing “exit” on it. If you do not do it, you will communicate with the workbench's terminal instead of the serial terminal.

### **6. Flash project**

Now you can upload the application on the board. At this point you should see command's list in the terminal.

### **7. Run Unreal Engine**

Now you are ready to play and you can open Unreal Engine's application. Note that could be done when you want, obviously we will see the car fixed in the world.

### **8. Optional: Open servers in the middle**

If you want to run Unreal Engine and WindRiver's workbench on different machines you must create 2 servers conceptually placed between the machines. The IP addresses of the board and of Ethernet machine's interface are static. In fact this is the only way to be sure that the packets go from and to the board. If Unreal Engine is running in a different machine we need to send all the packets to the IP address of the machine with the workbench. At this moment the server will forward them to the board (the IP address of the board is visible only at machine connected to it). Of course, we need also a second server to forward packets from the board to the machine that is running Unreal Engine's application.