Scuola Superiore Sant'Anna - Università di Pisa

Master Degree of Embedded Computing Sistems

Design of Embedded Systems course

Project Report

LeapCar

Professor

Marco Di Natale

Students

Antonio Di Guardo
Andrea Vincentini

Accademic year 2015-2016

Chapter 1

Requirements

1.1 User Requirements

`LeapCar` is a little toy designed with the following requirements:

1.1.1 System Definition

1. The system is composed by three elements: a little motorized tricycle *toy car*, a fixed component named *controller* and the *user*.

2. *toy car* must move in all directions of an orizontal plane. Motion is provided by two continuos speed step motors. The plane of motion has not to be too much smooth, in order to guarantee friction on the wheels.

3. *controller* will be fixed and implemented on a PC. There is no physical contact between *controller* and *toy car* but they must be at most 30 m far.

4. *user* is a human with at least an healthy arm and with no sight problems.

5. the *environment* temperature must be between 0 and 50 celtious degrees.

6. `LeapCar` should work in dry, indoor and bright evironment, but it will be designed in order to maximaze its robustness.

1.1.2 General Behaviour

1. The user must control *toy car* only by the use of *controller*.

2. *controller* must communicate with *toy car* in order to move it according with user commands.

1.1.3 User Interface

1. There will be no physical contact between *controller* and *user*.
2. *User* must be at most 1 m far from the *controller*.
3. The language between the *user* and the *interface* must have the same semantic of a joystick.

1.1.4 Energy

1. *Toy car* must be powered only by battery, so it must adopt energy saving policies.
2. No particular energy policy will be attached to the *controller*.

1.1.5 Safety

1. `LeapCar` is not a critical real time system.

1.2 Functional Specifications

1.2.1 User Interface

1. The user interface must be able to detect hand position and velocity (gesture recognition technology).
2. Depending to an easily identifiable point in the space named *center*, commands must be generated according with:
   - the palm position in a cartesian space having the origin in the *center* point.
   - some simply *gestures*. A *gesture* is a well-defined motion of the hand (for example the change of the palm orientation).
3. The feedback to the user is the motion of the car itself.

1.2.2 Communication

1. *Controller* and *toy car* must communicate without cables, so a wireless technology will be used.
2. The technology must guarantee that at each time, if there exist messages to be delivered, at least one is received by the car.
3. The communication protocol must use acknowledge messages.
4. In order to preserve the temporal validity of the messages, if more than one message is available, only the last one will be transmitted.

1.2.3 Real Time Constraints

1. Even though `LeapCar` is not a critical system, a very small latency must be guaranteed. In particular the system behaviour must consider four types of delay:
   - Δt_{det}: interval of time between the user gestures and the gesture detection by the *controller*. It must be less then 50 ms.
   - Δt_{el}: time between the detection of the gesture and the message sent to the *toy car*. It must be less then 20 ms.
   - Δt_{com}: time required by the comunication. It is impredictable, but it can be minimized accepting the loss of data (look at Par. 1.2.2).
   - Δt_{act}: time between the end of the comunication and the motion change. It must be less than 50 ms.
2. With the aim of making motion changes as smooth as possible, in some cases can be present an additional delay Δt_{steady} of at most 150 ms.

1.2.4 Type of motion and unit of measure

1. *Toy car* must be able to turn itself around.
2. *Toy car* must be able to go forward, go back and move with differend values of curvature radius.
3. the displacement unit of measure is always a half of the dinstance between the *toy car* wheels.
4. the unit of measure of time is always the *second*.

Chapter 2

Design

2.1 Comunication subsystem

The Technology

The communication between *toy car* and *controller* is made by a pair of `XBee serie 2`. This technology is choosen for the following reasons:

1. it can be easily attached on a PC (using a USB cable and a special board) as well as on a ARM-based board (simply using 3.3 serial cable).
2. It guarantees very small latency (it works up to 115200 baudrate).
3. It implements policy of energy saving.
4. It transmits up to 140 m far.
5. Our laboratory have already two of those.

`XBee` devices work in transparent mode, so the devices themself do not provide any service in terms of delivery. They are configurated to work in a subnet composed only by the two devices. In this way is avoided the delay due to broadcast messages. *Toy car* side is configured as the server of the subnet in order to get it always ready to receive commands without any additional delay.

Message Structure

The transport protocol is designed in order to guarantee the delivery of at least the last message. It takes the frame structure from the MIDI protocol, while the presence of the ack message and timers remembers somehow TCP.

A message is composed by a mandatory part (1 byte) and an optional part (always a multiple of 1 byte). The mandatory part, called header byte, has the MSB equal to 1, while the MSBs of the bytes of the optional part are always equal to 0.

Therefore header byte is composed by (from the MSB to LSB):

- a bit equals to 1
- 3 bit that identify the ID of the message
- 4 bit that identify the type of message (and so the number of byte of the optional part).

In this way the protocol can use at most 16 different types of message. In this system only two types of messages are implemented:

- 0b0000: `ACK` message. No optional part
- 0b0001: `DATA_SEND` message. Requires 2 byte of optional part.

A `DATA_SEND` message is used to transmit commands. A command is an information composed at least by the velocity and the curvature radius of the car. A `DATA_SEND` message is composed by three byte structured as following:

- First Byte: header byte.
- Second Byte. It contains information about velocity Structure from MSB to LSB:
  - 1 bit equals to 0;
  - 1 bit for the sign (0 = plus);
  - 6 bits for the modulus of the velocity.
- Third Byte. It contains information about curvature radius. Structure from MSB to LSB:
  - 1 bit equals to 0;
  - 1 bit for the validity of the byte (0 = no curvature radius);
  - 1 bit for the position of the center radius (0 = left);
  - 5 bits for the modulus of the curvature radius.

Protocol Rules

In `LeapCar` this protocol is used in a very simple way:

- *toy car* starts comunication with a `DATA_SEND` message. If more than one commands are available, the most recent one is sent, while the others are simply neglected. If it does not receive an `ACK` message with the desidered ID it will not send other messages, even if they are avaliable. After a well defined time if no `ACK` is received the most recent available message is sent again. Notice that it is possible to send a message that is different from the previous one.

- *controller* waits for a well-structured `DATA_SEND` message, therefore sends back an `ACK` message having the same ID of the received message.

2.2 Controller

2.2.1 Functional Modeling

Controller is the subsystem that has the aim to recognize the movements of the hand to generate messages for the ToyCar subsystem.

As it's shown on Fig. 2.1 *Controller* subsystem is composed by 3 main

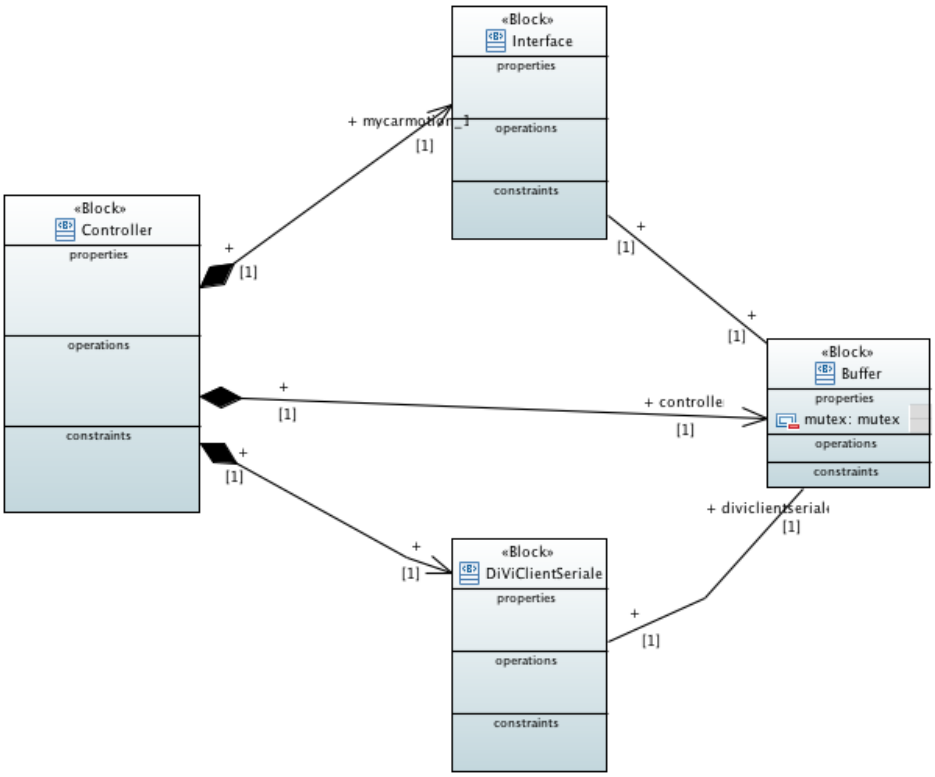

Figure 2.1: Block Diagram of *Controller* subsystem.

subsystems:

- *DiViClientSeriale* block, which deals with all concern with comunication;
- *Buffer* block, which contains messages;
- *Interface* block, which deals with all concern with gesture detection.

2.2.2 Architecture Exploration

The Controller wil be implemented using two threads using with different periods:

- 15 ms: thread used to manage serial comunication (*thread_tick)*;
- 50 ms: thread used to detect a hand gesture and create a new message (*thread_hand*).

Threads will be implemented using Posix library and they will be scheduled with SCHED_OTHER policy.

2.2.3 Component and Behavior Modeling

Interface

Interface (shown in the figure 2.4) has the aim to detect the gestures made by hand and translate these gestures in the speed and rotation angle.
Depending on the state machine (in Figure 2.2) this block will be able to transform the movement of the hand from the origin of the axes (reference axes shown in figure 2.3) in the speed and/or rotation angle.

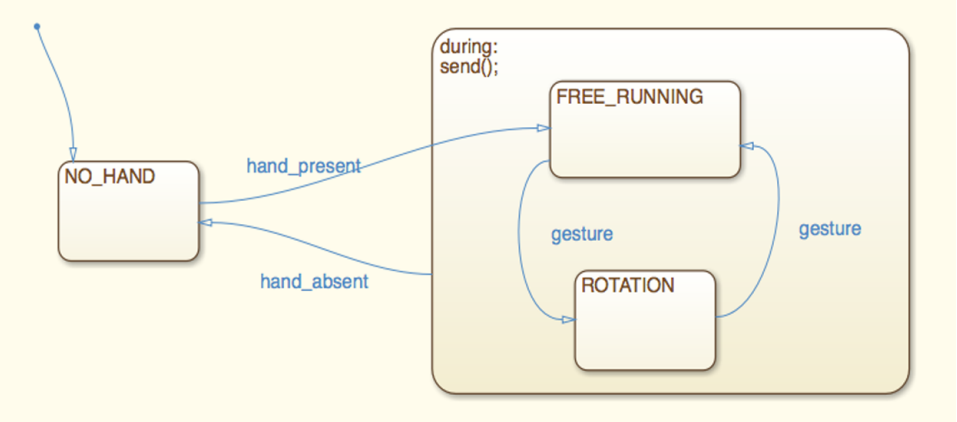

Figure 2.2: State Machine of *Interface*.

In order to recognize a hand motion it will use a device called Leap Motion. This device will allow to detect movements with an accuracy of 0.01 mm.
The actions required to generate speed and rotation angle are similar in both states *FREE_RUNNING* and *ROTATION* and coincide with a movement of the hand along the x and z axes like using a joystick.
In *FREE_RUNNING* mode the module of state speed and rotation angle will be calculated as follows:

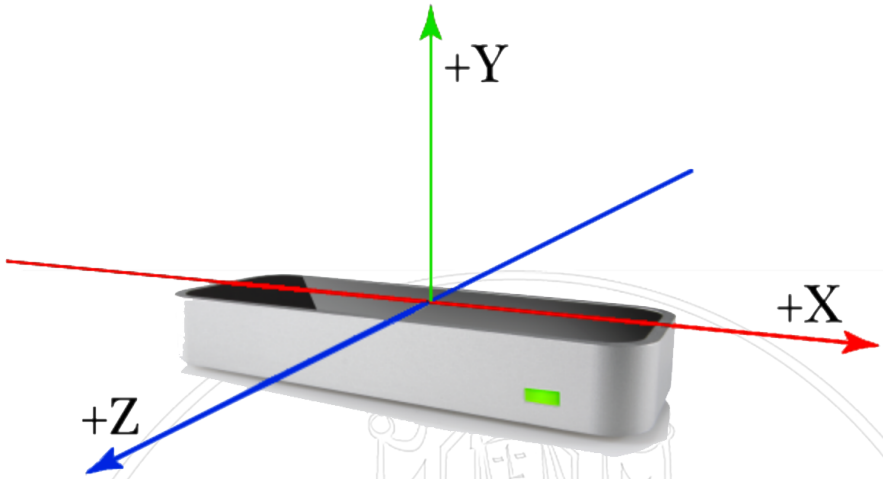

Figure 2.3: Coordinates system.

- the speed is calculated based on the distance from the origin of the hand according to a movement in the z axis

$$vel = abs(z) - (abs(z)\%5) - Z\_MM\_MIN$$

- the rotation angle is calculated based on the distance from the origin of the hand according to a movement in the x axis

$$angle = MAX\_ABS\_ANG - (abs(x) + 1 - X\_MM\_MIN + 1)/3$$

where MAX_ABS_ANG and Z_MM_MIN, X_MM_MIN represent thresholds of maximum and minimum values.
In *ROTATION* mode only the module of speed will be calculated as follows:

$$vel = abs(x) - (abs(x)\%5) - Z\_MM\_MIN$$

while rotation angle will be always equals to 0.

Buffer

Buffer is a block that manages all the operations of insertion, reading and removing from the buffer structure.
In order to guarantee the consistency of the data all these operations will be conducted in mutual exclusion through the use of a mutex.
All properties and methods related to Buffer are shown in figure 2.5.

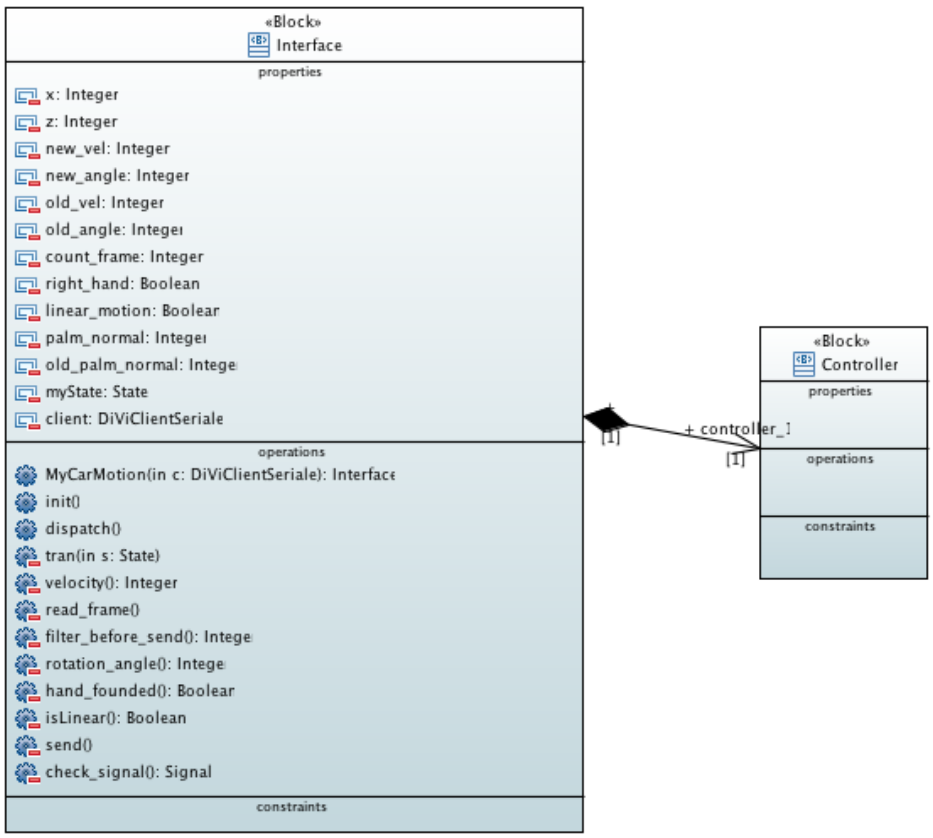

Figure 2.4: Block Diagram of *Interface*.

DiViClientSeriale

DiViClientSeriale (show in figure 2.6)is the block that handles the communication on the controller side.
Using this block, the controller system will be able to send a message to ToyCar system or to receive from this a confirmation ACK according to the operation mode expressed in Fig. 2.7.

2.3 Toy Car

2.3.1 Functional Modeling

Toy car is a subsystem composed by a little tricycle, as it was stated in the requirements part. Continuos speed servo motors are used as actuators. Their characteristic curves were esimated using two encoders and a matlab script.

As it is shown in Fig. 2.8, *toy car* subsystem is composed by 3 main

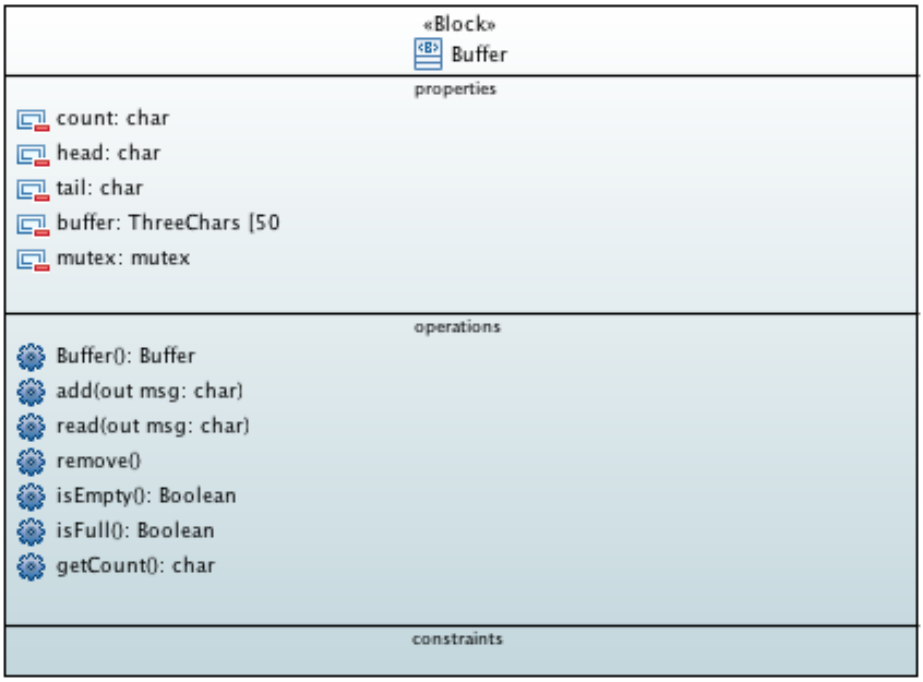

Figure 2.5: Block Diagram of *Buffer*.

subsystems:

- *DiViServerSerial* block, which deals with communication;
- *DecoderMsg* block, which is able to command motors according with the request messages;
- *CSServo* blocks (one for each servo), which virtualize the control of the motors and smoth velocity changes.

The software part will run on a board with a Amtel SAM3X8E ARM Cortex M3 core (Arduino 2).

Pin 1 and 2 of the board will be used as serial port.

2.3.2 Architecture Exploration

With the aim of achieve the timing requirements expressed in Par. 1.2.3, the software system will be implemented using two threads with different periods:

- 10 ms: thread used to manage serial comunication;
- 40 ms: thread used to decode a new message (if it exists) and manage the two servo.

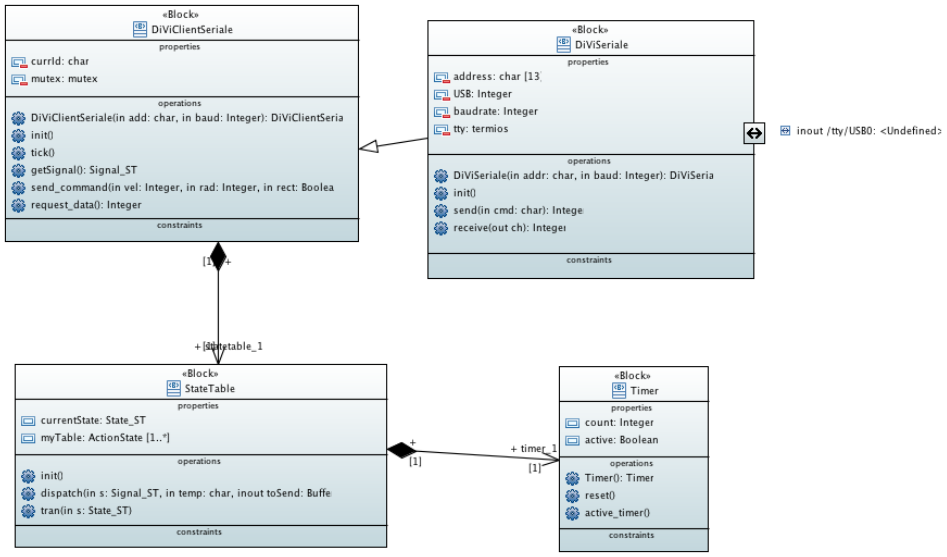

Figure 2.6: Block Diagram of DiViClientSeriale

Threads will be scheduled according with Rate Monotonic policy by the Embedded Real Time OS `ARTE`. When mutual exclusion is needed (coerence of *DiViServerSerial* internal buffer) no-preemption is forced using `ARTE` primitives.

2.3.3 Component and Behavior Modeling

DiViServerSerial

DiViServerSerial block is a very complex subsystem used to manage the communication. It's composed by

- a state machine (look at Fig. 2.9) that implements the protocol already explaned in Par. 2.1);
- a buffer able to store received messages;
- a part that manages the physical serial interface with the XBee device.

According with the state machine behavior, at least three periods are needed such that a message is acquired.

DecoderMsg

DecoderMsg is a block that is able to convert a command in terms of velocity v and curvature radius c_r in the velocities of the motors (look at Fig. 2.10).

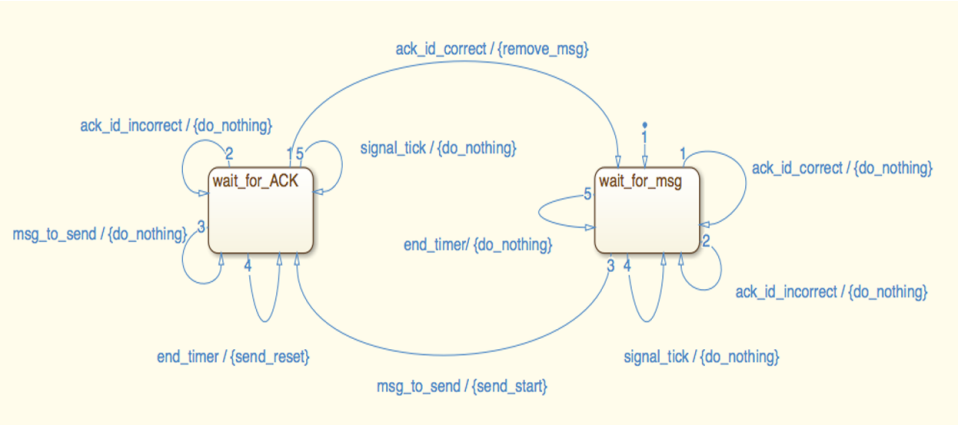

Figure 2.7: State Machine of DiViClientSeriale

The idea is that if the curvature radius is zero, the speed of the motors must be the same and equal to v, otherwise they must be different and equal to:

$$v_{right} = v/c_r$$

$$v_{left} = v/c_r$$

Remember that displacements are measured using *half of distance between wheels* as unity of measure.

CSServo

CSServo is a class that simply virtualizes the physical motors (look at Fig. 2.11). An instance of this class must contain the parameters of the characteristic curve of the motor, in order to map speeds into values (closed to 1.5ms) which correspond to the *pulse width* currently used to manage the motor behavior.

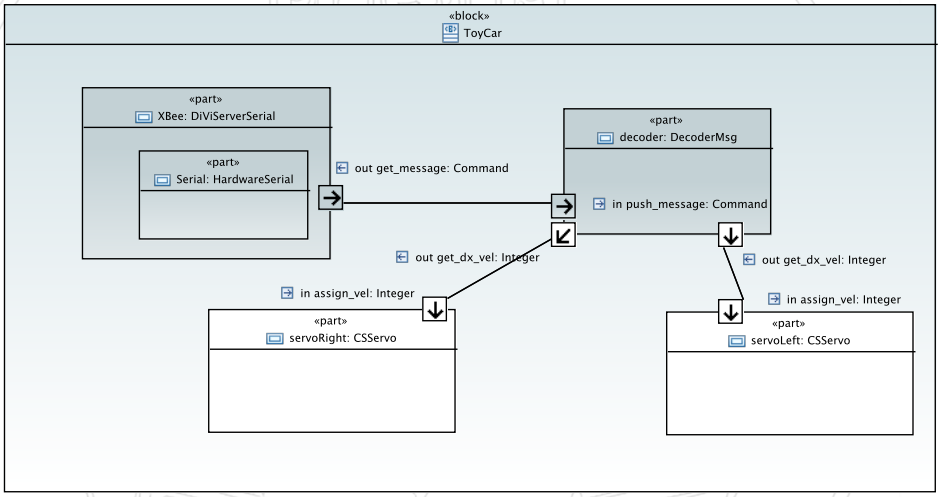

Figure 2.8: Internal Block Diagram of *toy car* subsystem.

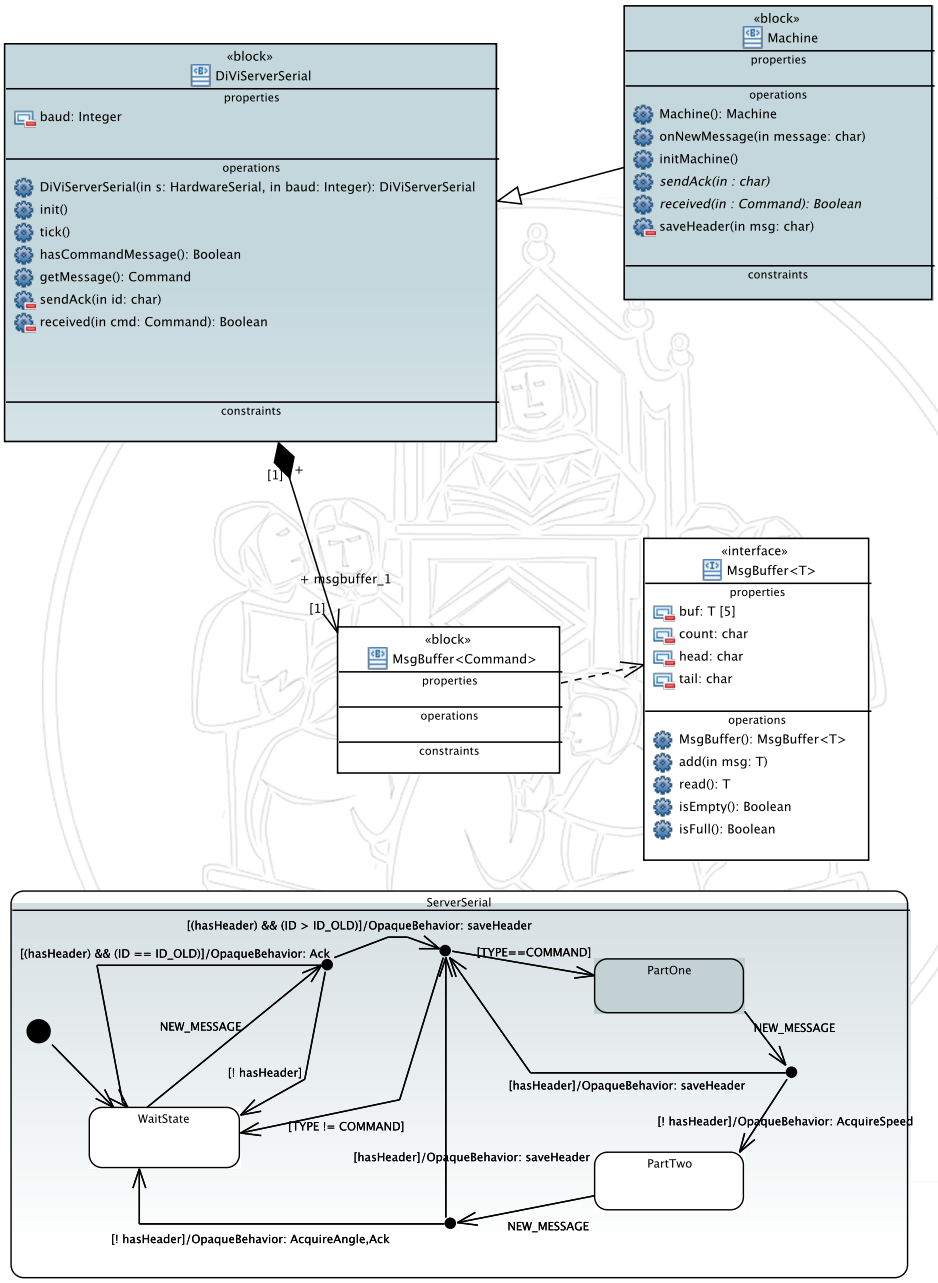

Figure 2.9: From top to bottom: Block Definition Diagram of *DiViServerSerial* part; state machine implemented by *Machine*.

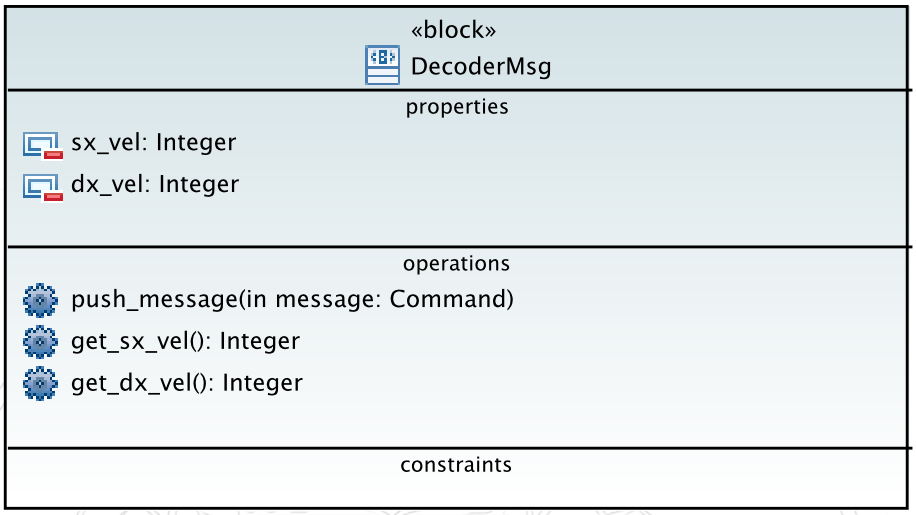

Figure 2.10: *DecoderMsg* attributes and methods.

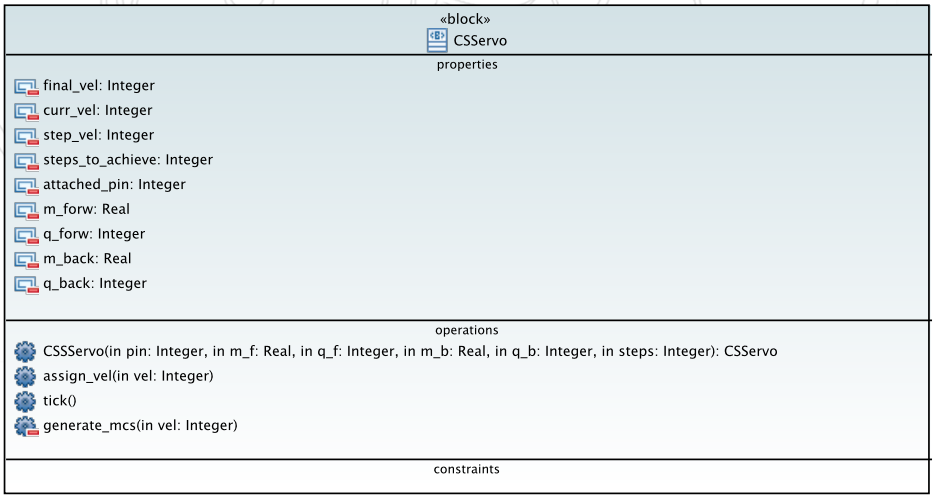

Figure 2.11: *CSServo* attributes and methods.

Chapter 3

Coding and Testing

3.1 Coding

3.1.1 Development environment

Controller and ToyCar have been developed on 2 different environments.
The controller was developed on a Linux Ubuntu system while ToyCar has been developed on a system using the Arduino real time system ARTE (Arduino Real Time exstension) provided by Retis Lab.

3.1.2 Implementations of State Machine

As shown in the *Design* chapter the behavior of the two system will be characterized using state machines.
The three state machines will be implemented using three different methods.
In the side of the *Controller* state machine that controls the operation mode wil be implemented through ***nested switch*** while the state machine that manages the states of the communication will be implemented through ***state table***.
In the side of the *ToyCar* the state machine that manages communication will be implemented using ***C ++ state pattern***.

3.2 Testing

3.2.1 C Unit

The behavior of the State Table in the figure .. was verified with the use of C Unit framework.
The test consists of a "*Suite_StateTable*" called suites containing four tests:

- **Test_Action_StateTable** to test if actions implemented for each entry of the table correspond to those required;

- **Test_Transiction_StateTable** to test if transiction implemented for each entry of the table correspond to those required;
- **Test_getSignal** to verify if the *getsignal()* is able to generate the correct signal that has to be send as input to the SM;
- **Test_dispatch** to check the full implementation of the *dispatch()* function simulating a real case

Tests are executed with the Basic Mode interface.

3.2.2 gcov

The tests performed by Unit C will be compiled with the "–coverage" option to generate additional information needed by gcov.