UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Student: Rogoz Bogdan Dorin

Group: 30433

Table of Contents

1. Theme

The topic discussed in this paper is the CAN standard used within the automotive industry.

2. Objective

The objective is to provide an overview on CAN systems' communications protocol. After finishing the paper, the reader should be able to understand the theoretical fundamentals behind the CAN 2.0A's architecture and messaging protocol.

3. Specifications

This paper serves as an entry point on the road of building a fully-fledged CAN system, a simulator or the logic behind a CAN controller linking a certain microprocessor inside an ECU to the rest of the CAN Bus. By using the information presented, the engineer can make his way through learning the more advanced specifications of the CAN standard, in order to be able to build a fully-working system (including error checking, correction, bit stuffing etc.).

4. Abstract

A Controller Area Network (CAN) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other in applications without a host computer. It is a message-based protocol, designed originally for multiplex electrical wiring within automobiles to save on copper, but is also used in many other contexts.

5. History

The CAN 1.0 protocol was released in 1986. The first CAN controller chips, produced by Intel and Philips, came on the market in 1987.

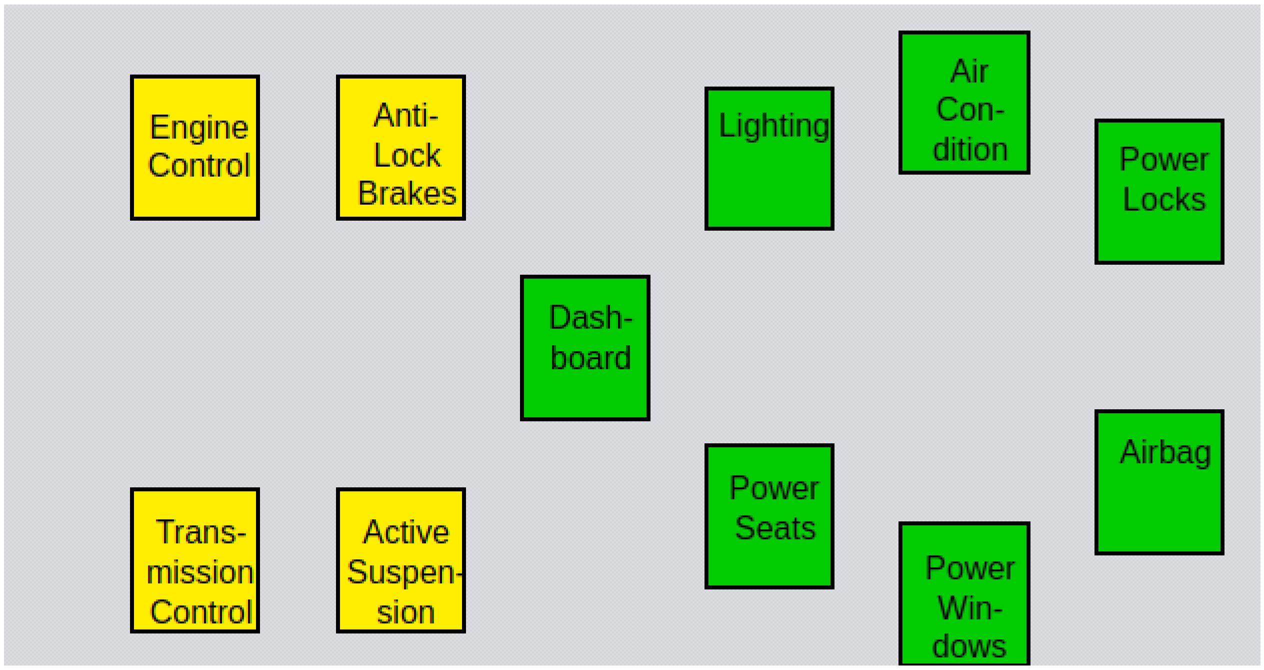
Bosch published several versions of the CAN specification and the latest is CAN 2.0 published in 1991. This specification has two parts; part A is for the standard format with an 11-bit identifier, and part B is for the extended format with a 29-bit identifier. A CAN device that uses 11-bit identifiers is commonly called CAN 2.0A and a CAN device that uses 29-bit identifiers is commonly called CAN 2.0B.

Bosch is still active in extending the CAN standards. In 2012, Bosch released CAN FD 1.0 or CAN with Flexible Data-Rate.

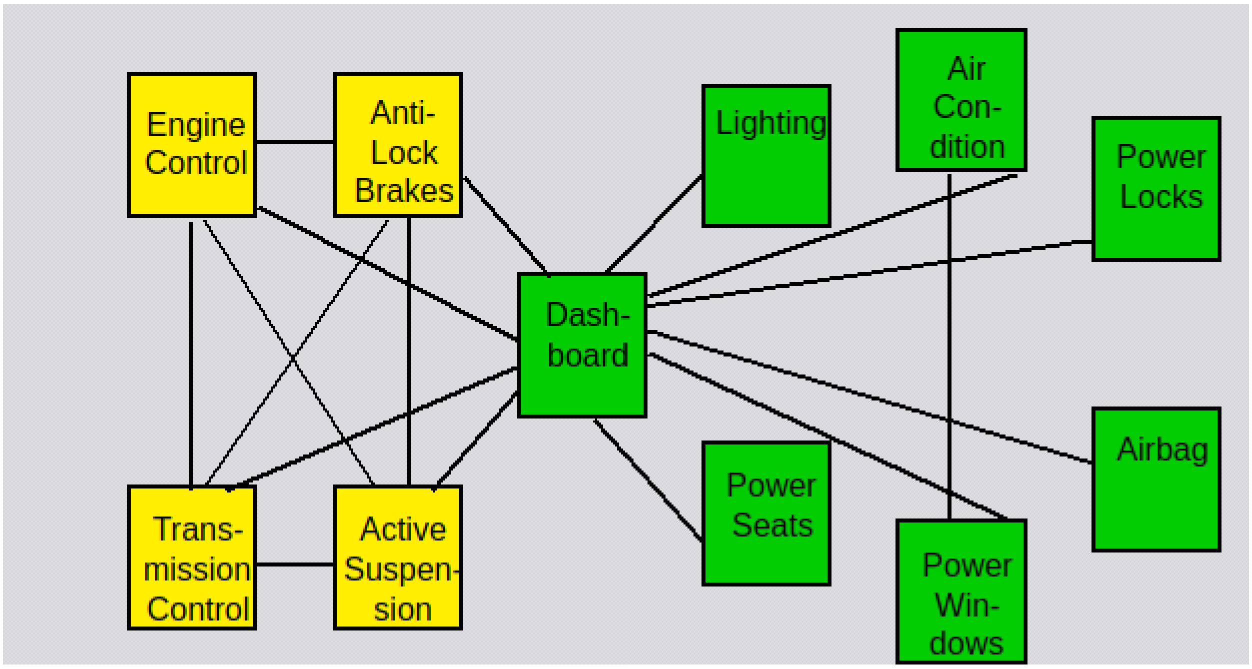
6. Applications

The modern automobile have large numbers of ECUs (Electronic Control Units) and it is essential that they are capable of intercommunicating. Before CAN systems became available, the intercommunications system occupied large amounts of space and quickly became hard to develop and test.

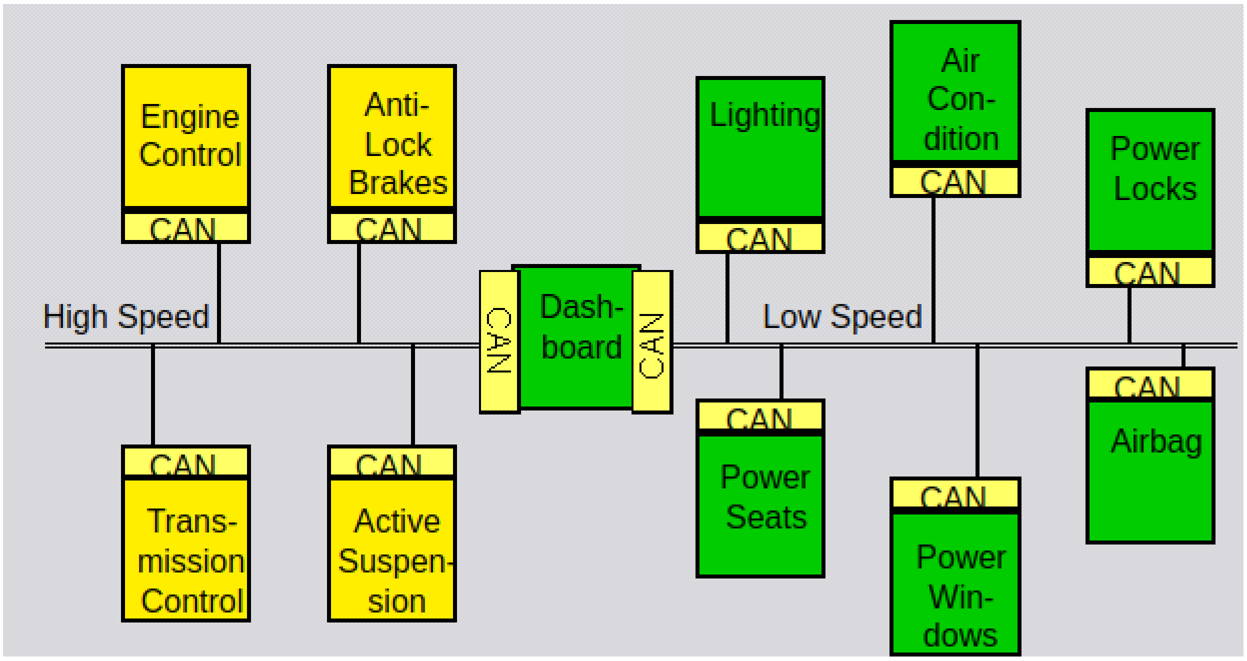
Let’s give the following example: suppose we have the following ECUs on our vehicle:



Interconnecting these ECUs in a traditional way would result in this:



Of course, such wiring (point-to-point wiring) would produce high material costs, high time delays and reliability problems. On the other hand, using a CAN architecture produces the following system:

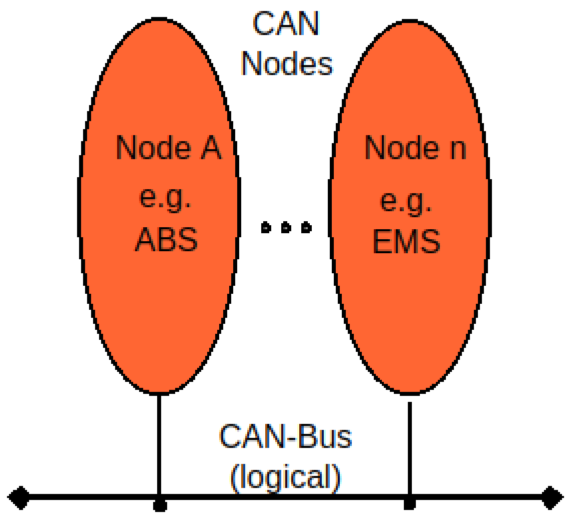


This change results in a more simplistic, cost-effective design. In this example, two CAN standards were used (the High Speed and Low Speed / Fault Tolerant), depending on the designer's preferences on each ECU's maximum data transfer latency.

As a result, different ECUs are able to intercommunicate and perform certain actions, including:

- Auto start/stop: If the car’s sensors detect that the vehicle has been stationary for a while, they can send the Engine Control ECU the signal to shut down in order to improve fuel economy and emmisions.
- Air condition: If the driver activates the air contitioner, the dashboard will send a message through the Low Speed CAN to the Air Condition ECU
- Automatic door locks: If the car is stationary for a while or the driver presses a certain button, the dashboard could send a message to the Power Locks ECU in order to lock all the doors. The same components are present in the case of unlocking the doors.

7. Architecture

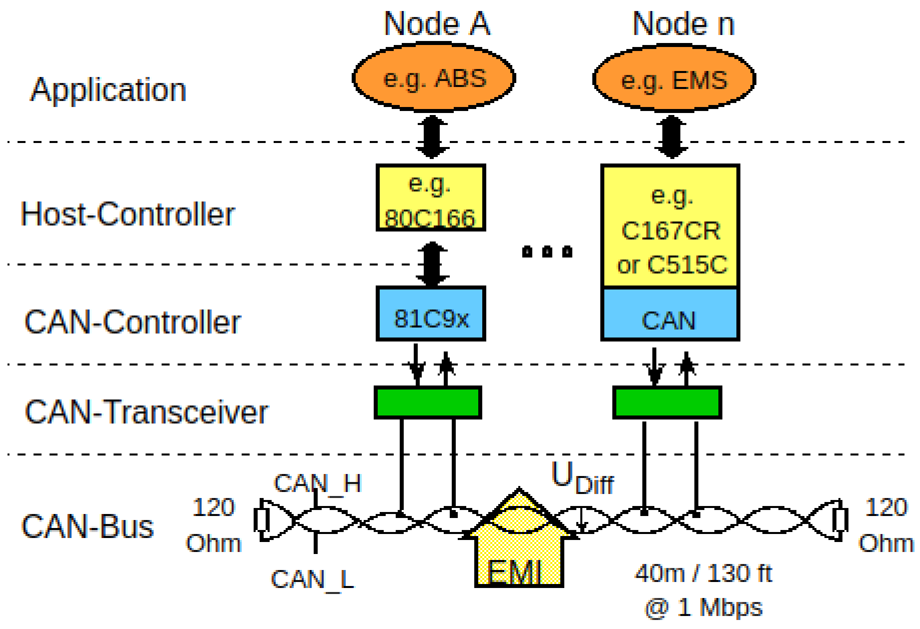


CAN is a multi-master bus with an open, linear structure with one logic bus line and equal nodes. The number of nodes is not limited by the protocol.

In the CAN protocol, the bus nodes do not have a specific address. Instead, the address information is contained in the identifiers of the transmitted messages, indicating the message content and the priority of the message.

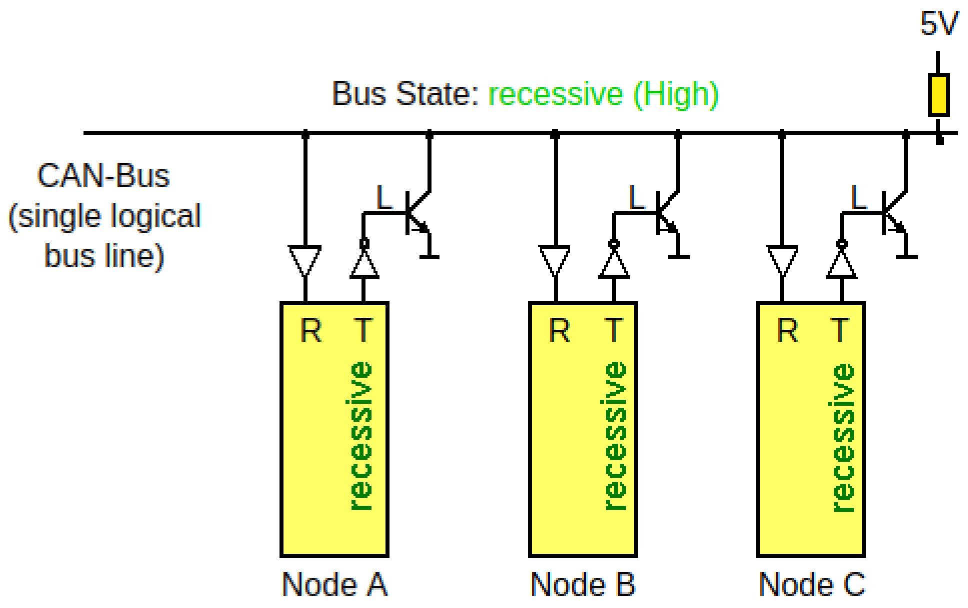
The number of nodes may be changed dynamically without disturbing the communication of the other nodes. Multicasting and Broadcasting is supported by CAN.

8. Data transmission

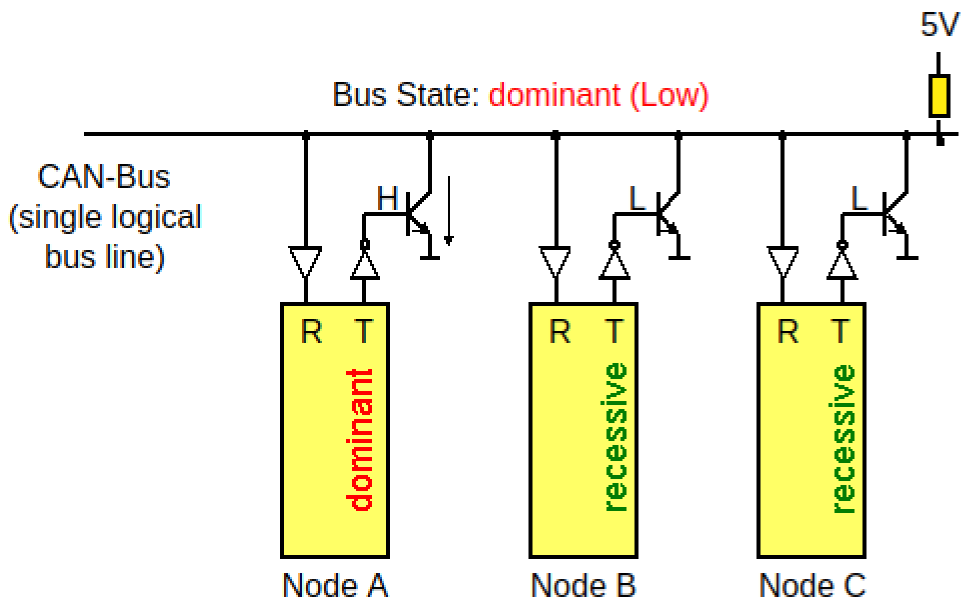


CAN provides sophisticated error-detection and error handling mechanisms such as CRC check, and high immunity against electromagnetic interference. Erroneous messages are automatically retransmitted. Temporary errors are recovered. Permanent errors are followed by automatic switch-off of defective nodes. There is guaranteed system-wide data consistency. The CAN protocol uses Non-Return-to-Zero or NRZ bit coding. For synchronization purposes, Bit Stuffing is used. Message length is short with a maximum of 8 data bytes per message and there is a low latency between transmission request and start of transmission.

There are two bus states, called "dominant" and "recessive". The bus logic uses a "Wired-AND" mechanism, that is, "dominant bits" (equivalent to the logic level "Zero") overwrite the "recessive" bits (equivalent to the logic level "One").

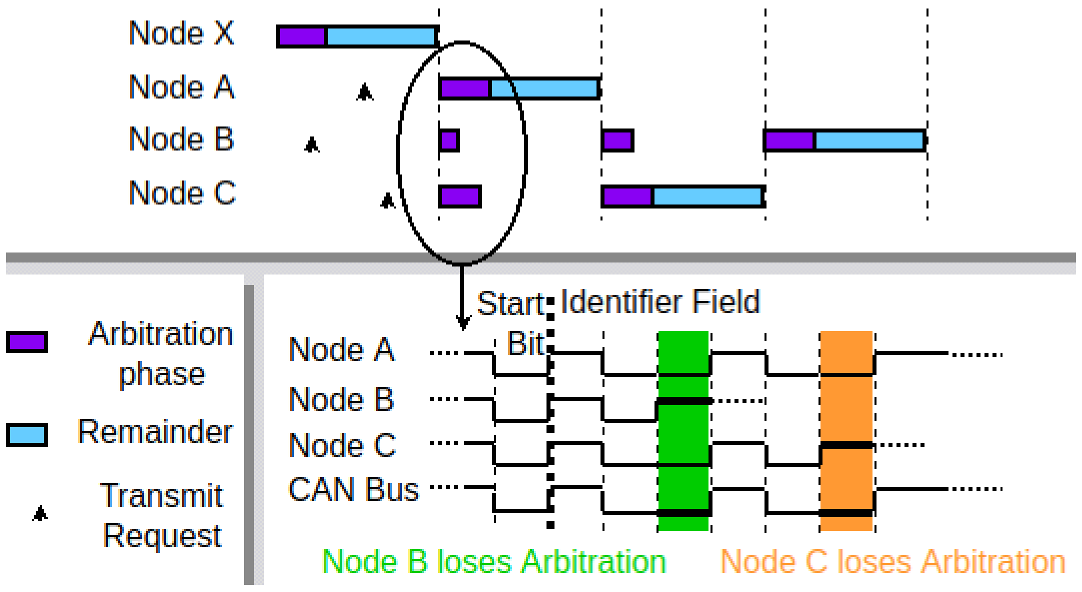


Only if all nodes transmit recessive bits (ones), the Bus is in the recessive state.



As soon as one node transmits a dominant bit (zero), the bus is in the dominant state.

8.1. Collisions



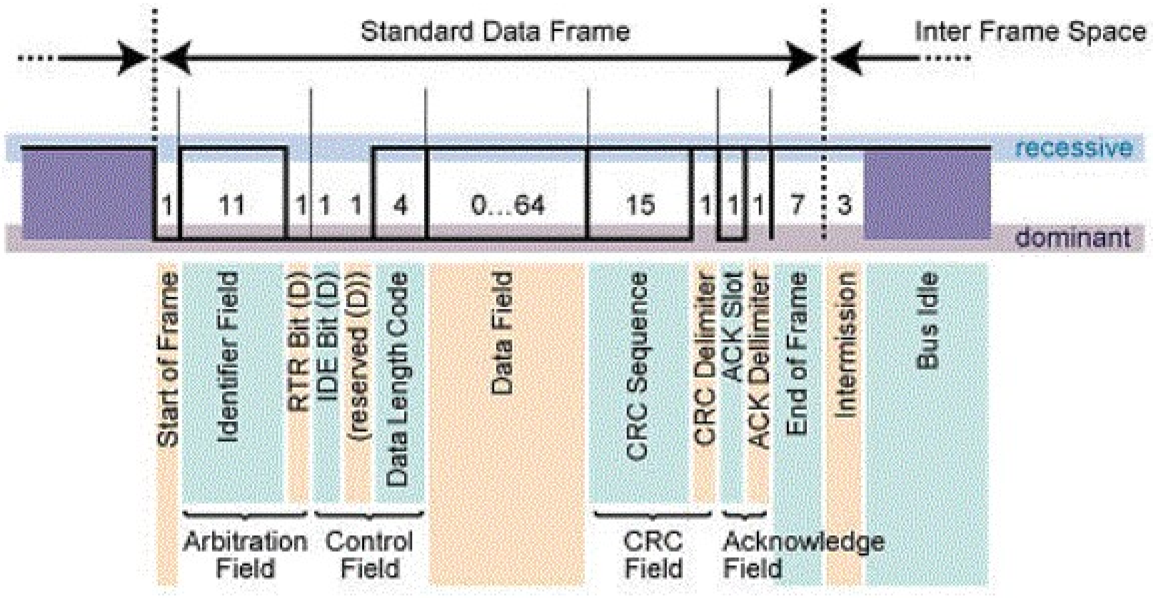
If two or more bus nodes start their transmission at the same time after having found the bus to be idle, collision of the messages is avoided by bitwise arbitration. Each node sends the bits of its message identifier and monitors the bus level.

At a certain time nodes A and C send a dominant identifier bit. Node B sends a recessive identifier bit but reads back a dominant one. Node B loses bus arbitration and switches to receive mode. Some bits later node C loses arbitration against node A. This means that the message identifier of node A has a lower binary value and therefore a higher priority than the messages of nodes B and C. In this way, the bus node with the highest priority message wins arbitration without losing time by having to repeat the message.

Nodes B and C automatically try to repeat their transmission once the bus returns to the idle state. Node B loses against node C, so the message of node C is transmitted next, followed by node B’s message.

It is not permitted for different nodes to send messages with the same identifier as arbitration could fail leading to collisions and errors.

8.2. Data Frame



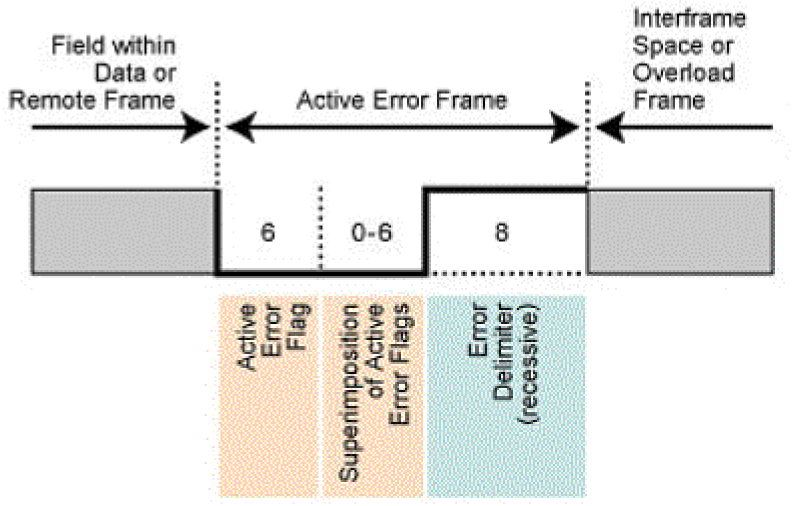
A "Data Frame" is generated by a CAN node when the node wishes to transmit data. The Standard CAN Data Frame is shown above. The frame begins with a dominant Start Of Frame bit for hard synchronization of all nodes.

The Arbitration Field consists of 12 bits: The 11-bit Identifier, which reflects the contents and priority of the message, and the Remote Transmission Request bit. The Remote transmission request bit is used to distinguish a Data Frame (RTR = dominant) from a Remote Frame (RTR = recessive).

The Data Length Code (DLC) specifies the number of bytes of data contained in the message (0 - 8 bytes). The data being sent follows in the Data Field which is of the length defined by the DLC above (0, 8, 16, ..., 56 or 64 bits).

Seven recessive bits (End of Frame) end the Data Frame.

8.3. Error Frame


Field within Data or Remote Frame
Active Error Frame
Interframe Space or Overload Frame
6
0-6
8
Active Error Flag
Superimposition of Active Error Flags
Error Delimiter (recessive)

An Error Frame is generated by any node that detects a bus error. The Error Frame consists of 2 fields, an Error Flag field followed by an Error Delimiter field. The Error Delimiter consists of 8 recessive bits and allows the bus nodes to restart bus communications cleanly after an error.

8.4. Error checking and handling

The CAN protocol provides sophisticated error detection mechanisms. These mechanisms are:

- Cyclic Redundancy Check: the transmitter calculates a check sum for the bit sequence from the start of frame bit until the end of the Data Field.
- Acknowledge Check: checks in the Acknowledge Field of a message to determine if the Acknowledge Slot contains a dominant bit.
- Frame Check: If a transmitter detects a dominant bit in the CRC Delimiter, Acknowledge Delimiter, End of Frame or Interframe Space, then a Form Error has occurred.
- Bit Check: if a transmitter either sends a dominant bit but detects a recessive bit on the bus line or sends a recessive bit but detects a dominant bit on the bus line, then a Bit Error occurred.
- Bit Stuffing Check: if six consecutive bits with the same polarity are detected between Start of Frame and the CRC Delimiter, the bit stuffing rule has been violated.

Detected errors are made public to all other nodes via Error Frames. The transmission of the erroneous message is aborted and the frame is repeated as soon as possible.

9. Design

9.1. Functional requirements

The user should be able to:

- Add CAN controllers without affecting the other controllers' functionalities
- Remove CAN controllers without affecting the other controllers' functionalities
- Define the behavior of his controllers based on a template
- Load simulations
- Save simulations
- View the bus data and the timestamps

9.2. Non-functional requirements

The following requirements should be satisfied:

- The bus and each controller should represent a different thread, in order to simulate the concurrency of the components
- The Observable pattern should be used, in order to simplify data transmission

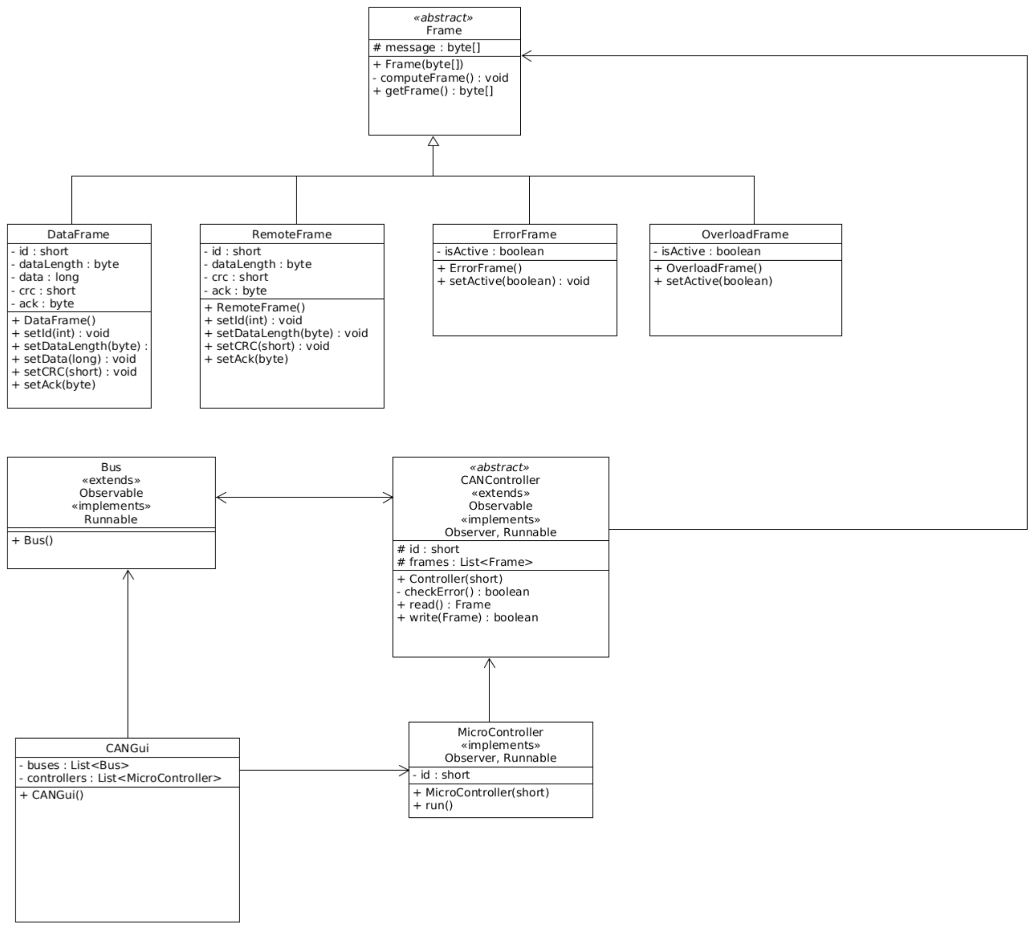
9.3. Models

The following models are implemented:

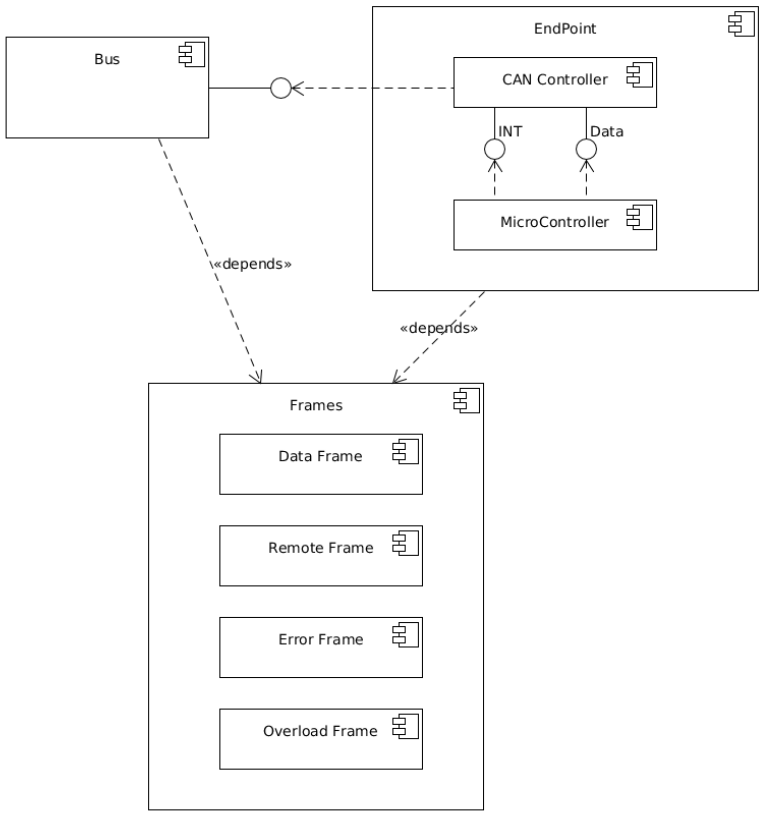
- Frame : represents a “builder” object which transforms a set of parameters into valid CAN frames; a frame can be of multiple types
- DataFrame : represents the data frame used by the CAN protocol. It builds a message based on the fields' values; it is used for sending data, usually as a response to a request
- RemoteFrame : represents the remote frame used by the CAN protocol; it is used for sending requests to remote controllers
- ErrorFrame : represents the error frame used by the CAN protocol; it is instatiated when an error is detected by the CAN controller
- OverloadFrame : represents the overload frame used by the CAN protocol.

- Bus : represents a CAN bus; multiple instances are possible
- CANController : represents an endpoint of the CAN bus; it reads from and writes to the CAN bus; it is also responsible for checking if the received messages are errorneous
- MicroController : represents a microcontroller which does a certain job; it communicates with the CAN bus through the CAN controller

9.4. UML Class Diagram



9.5. Component diagram

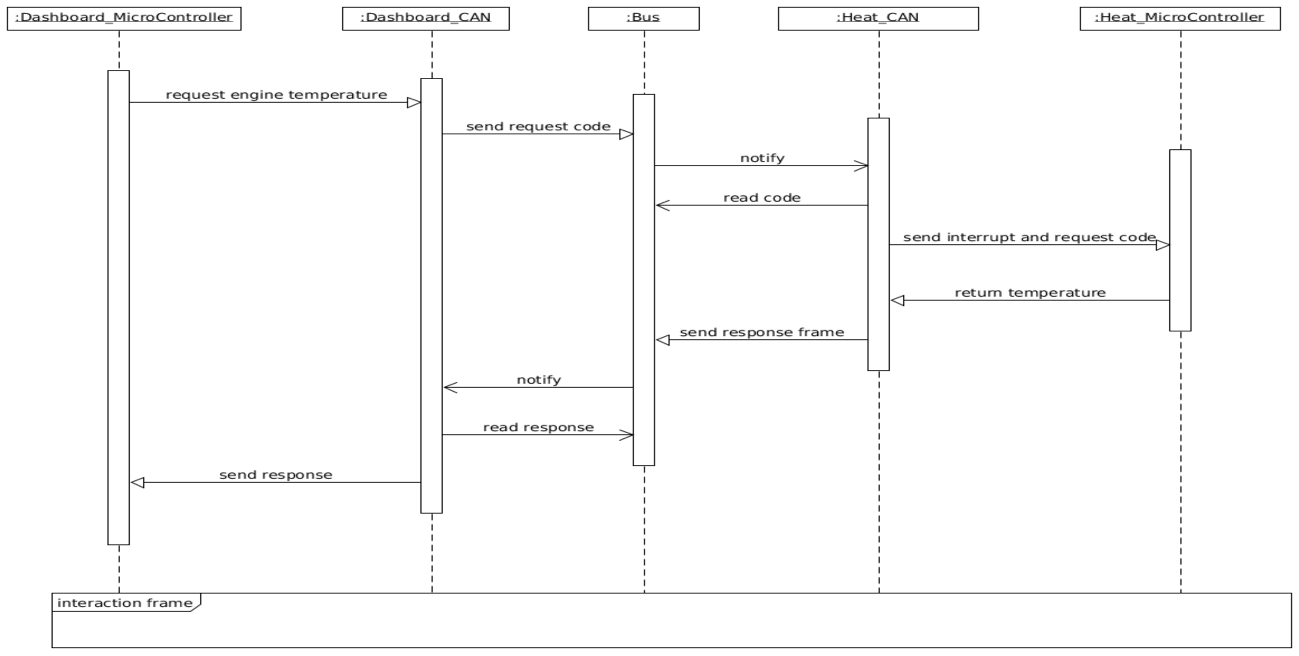


9.6. Scenarios

An example scenario would be consisting of the following components:

- One CAN bus
- 2 microcontrollers, each one having 1 CAN controller: a microcontroller for an engine heat sensor and the dashboard

At one point, the dashboard sends a message to the CAN bus, asking for the temperature measured by the heat sensor. The heat sensor’s CAN controller receives the message, sends the interrupt to its corresponding microcontroller, then sends the response back to the dashboard.



9.7. Graphical User Interface

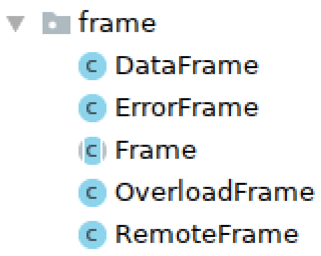
The GUI should be simple, intuitive and responsive. The following elements are considered to be some basic ones:

- Load design button
- Save design button
- Design view, containing all the active components
- Component pane, containing all the user defined components
- Load example design menu, containing a list of predefined designs
- Create component button
- A message history pane, containing a list of all messages transmitted through the bus and their status (SUCCESS, ERROR etc.)

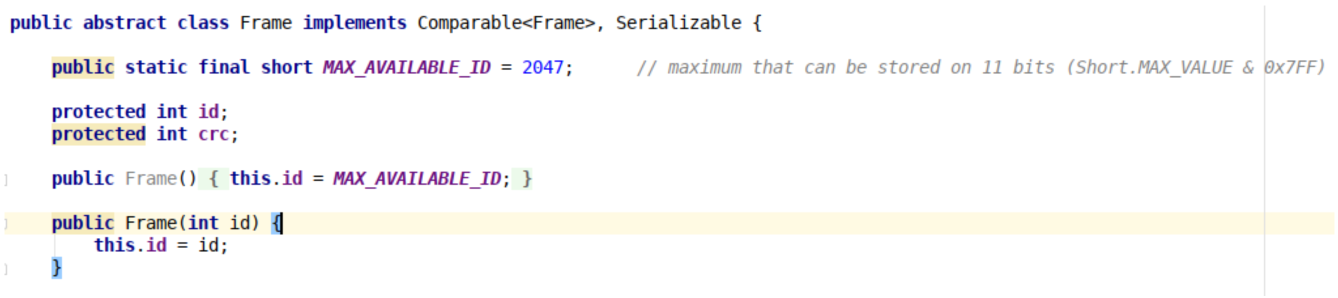
10. Implementation

10.1. Frame package

The frame package contains the frame classes, used by the Controller instances for communication. Its contents are:

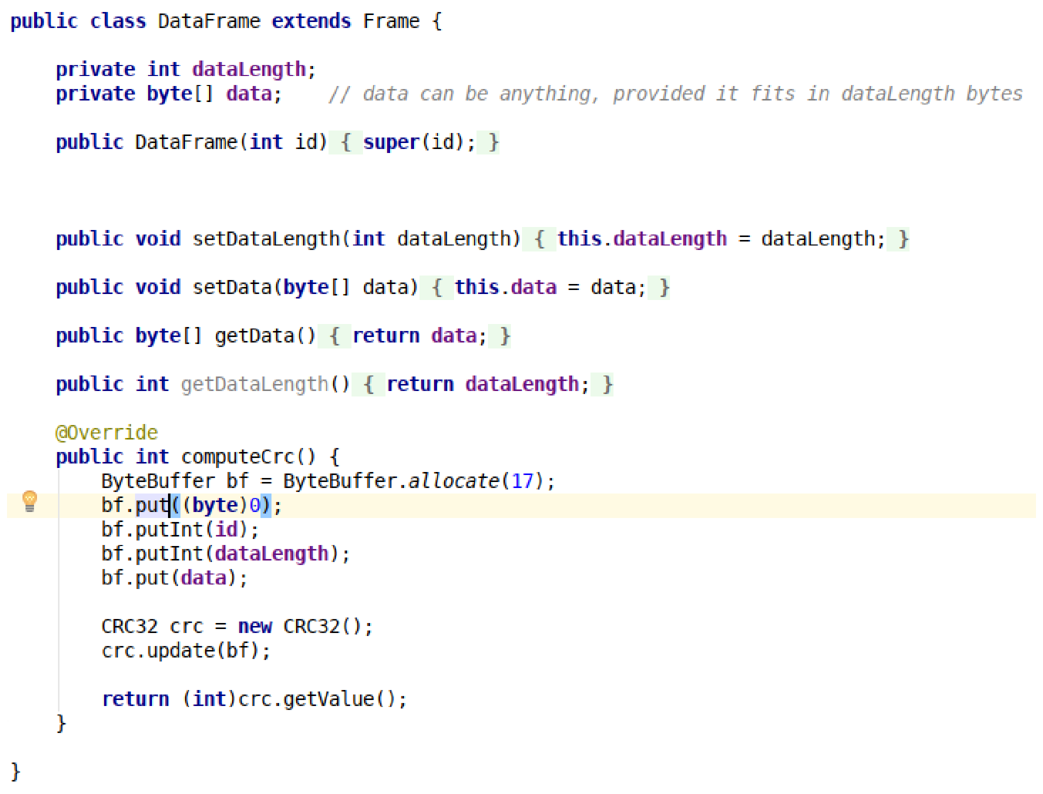


The Frame class is abstract and acts as a base for the other (concrete) classes.



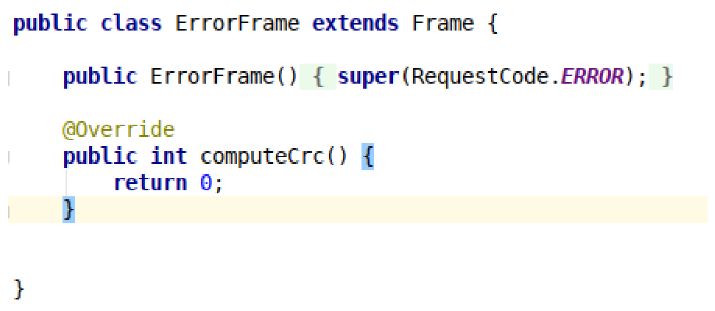
```
public abstract class Frame implements Comparable<Frame>, Serializable {

    public static final short MAX_AVAILABLE_ID = 2047;      // maximum that can be stored on 11 bits (Short.MAX_VALUE & 0x7FF)

    protected int id;
    protected int crc;

    public Frame() { this.id = MAX_AVAILABLE_ID; }

    public Frame(int id) {
        this.id = id;
    }
```

Since a frame can transmit 11 bits long Ids, the maximum value possible is 2047. When a frame is created, if the ID is not specified, the lowest priority is assigned (0 = highest priority, reserved for errors, 1 for overloads, 2047 = lowest priority).



```java
public class DataFrame extends Frame {

    private int dataLength;
    private byte[] data;    // data can be anything, provided it fits in dataLength bytes

    public DataFrame(int id) { super(id); }


    public void setDataLength(int dataLength) { this.dataLength = dataLength; }

    public void setData(byte[] data) { this.data = data; }

    public byte[] getData() { return data; }

    public int getDataLength() { return dataLength; }

    @Override
    public int computeCrc() {
        ByteBuffer bf = ByteBuffer.allocate(17);
        bf.put((byte)0);
        bf.putInt(id);
        bf.putInt(dataLength);
        bf.put(data);

        CRC32 crc = new CRC32();
        crc.update(bf);

        return (int)crc.getValue();
    }

}
```

DataFrame is a concrete implementation of the Frame class. It is differentiated from the other classes by the presence of the *dataLength* field, which can have values in range [0...8]. Thus, it must also have its own implementation of the *computeCrc()* method, which is used for error detection.



```java
public class ErrorFrame extends Frame {

    public ErrorFrame() { super(RequestCode.ERROR); }

    @Override
    public int computeCrc() {
        return 0;
    }


}
```

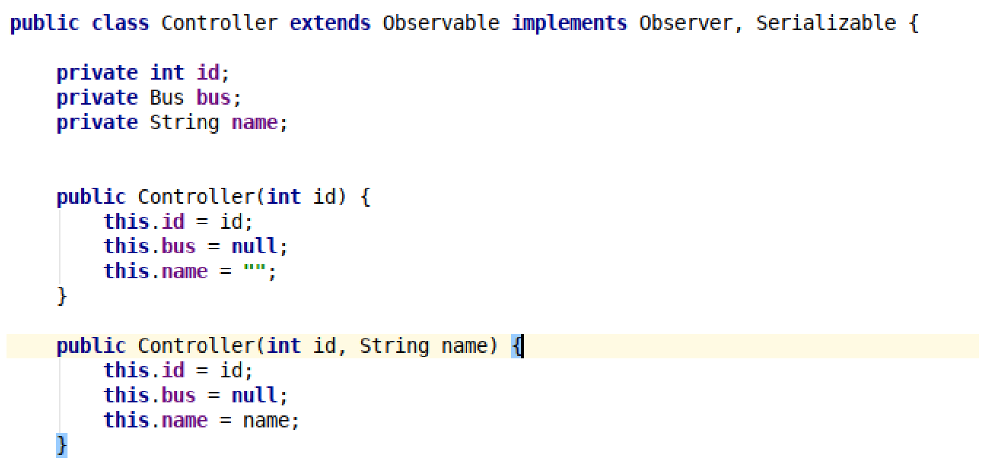
An error frame doesn't transmit any data (except dominant and recessive bits at hardware level), so the *computeCRC()* method isn't useful. Important to note is the *RequestCode.ERROR* id, which has the highest priority.

10.2. CAN Controller and ECU

There are 2 types of controllers implemented:

- CAN Controller (*Controller* class)
- ECU (*MicroController* abstract class and its concrete implementations)

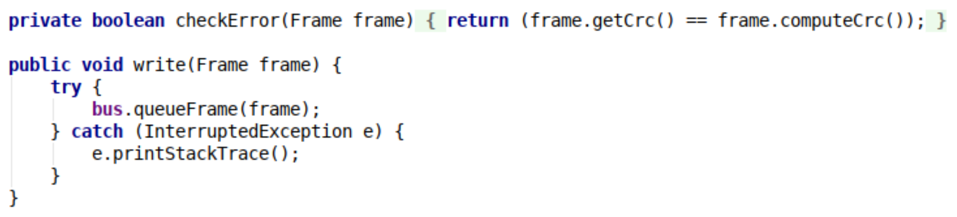
10.2.1. CAN Controller



```java
public class Controller extends Observable implements Observer, Serializable {

    private int id;
    private Bus bus;
    private String name;


    public Controller(int id) {
        this.id = id;
        this.bus = null;
        this.name = "";
    }

    public Controller(int id, String name) {
        this.id = id;
        this.bus = null;
        this.name = name;
    }
```

A CAN Controller contains 3 fields:

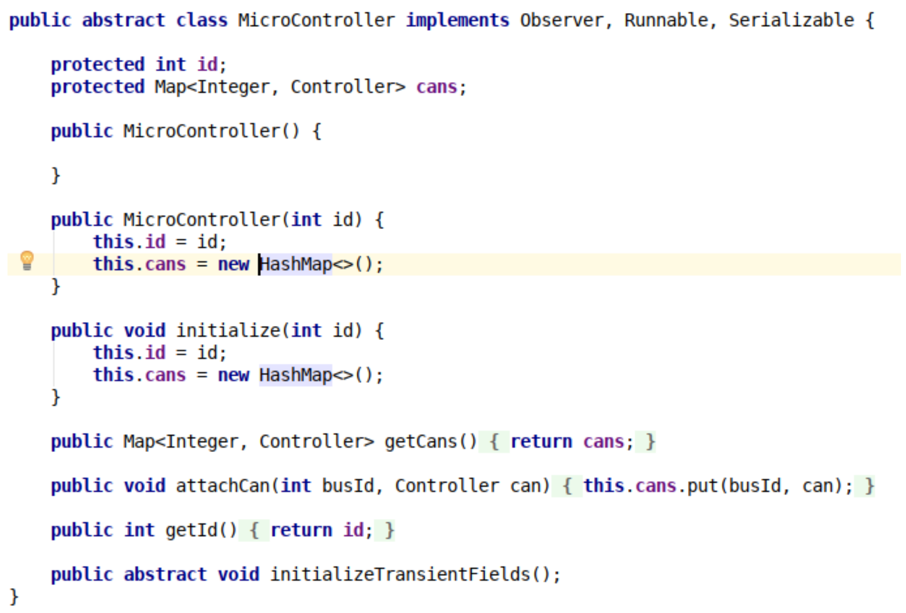
- ID : the id of the CAN Controller, auto-generated
- bus : the bus associated to the CAN Controller
- name : the name of the CAN Controller



```java
private boolean checkError(Frame frame) { return (frame.getCrc() == frame.computeCrc()); }

public void write(Frame frame) {
    try {
        bus.queueFrame(frame);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

The *checkError()* method checks if there have been errors in transmitting a certain frame, by recomputing the frame’s CRC and comparing the returned value to the CRC field inside the frame. The *write()* method sends the frame to the bus.

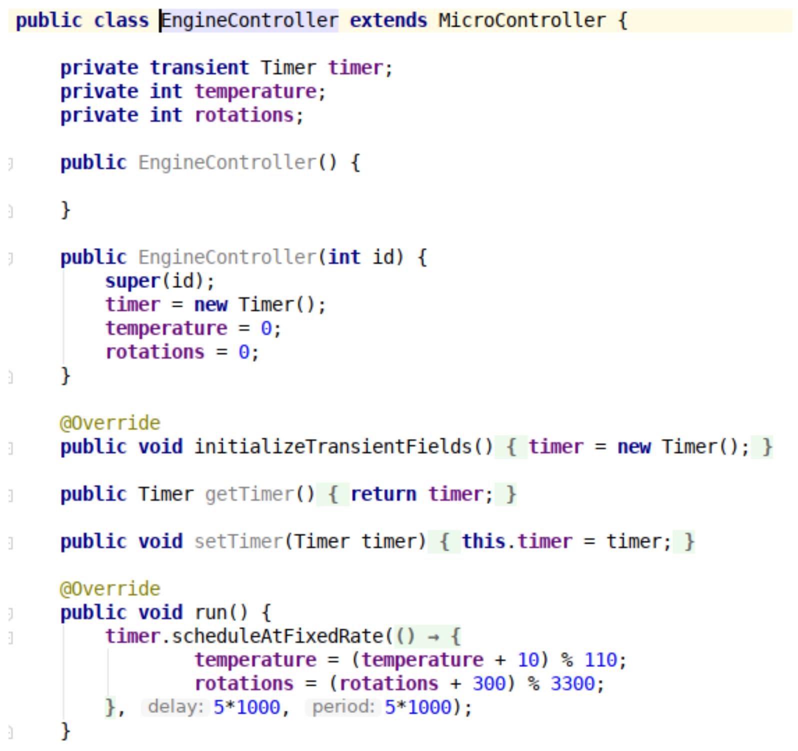
10.2.2. ECU



```java
public abstract class MicroController implements Observer, Runnable, Serializable {

    protected int id;
    protected Map<Integer, Controller> cans;

    public MicroController() {

    }

    public MicroController(int id) {
        this.id = id;
        this.cans = new HashMap<>();
    }

    public void initialize(int id) {
        this.id = id;
        this.cans = new HashMap<>();
    }

    public Map<Integer, Controller> getCans() { return cans; }

    public void attachCan(int busId, Controller can) { this.cans.put(busId, can); }

    public int getId() { return id; }

    public abstract void initializeTransientFields();
}
```

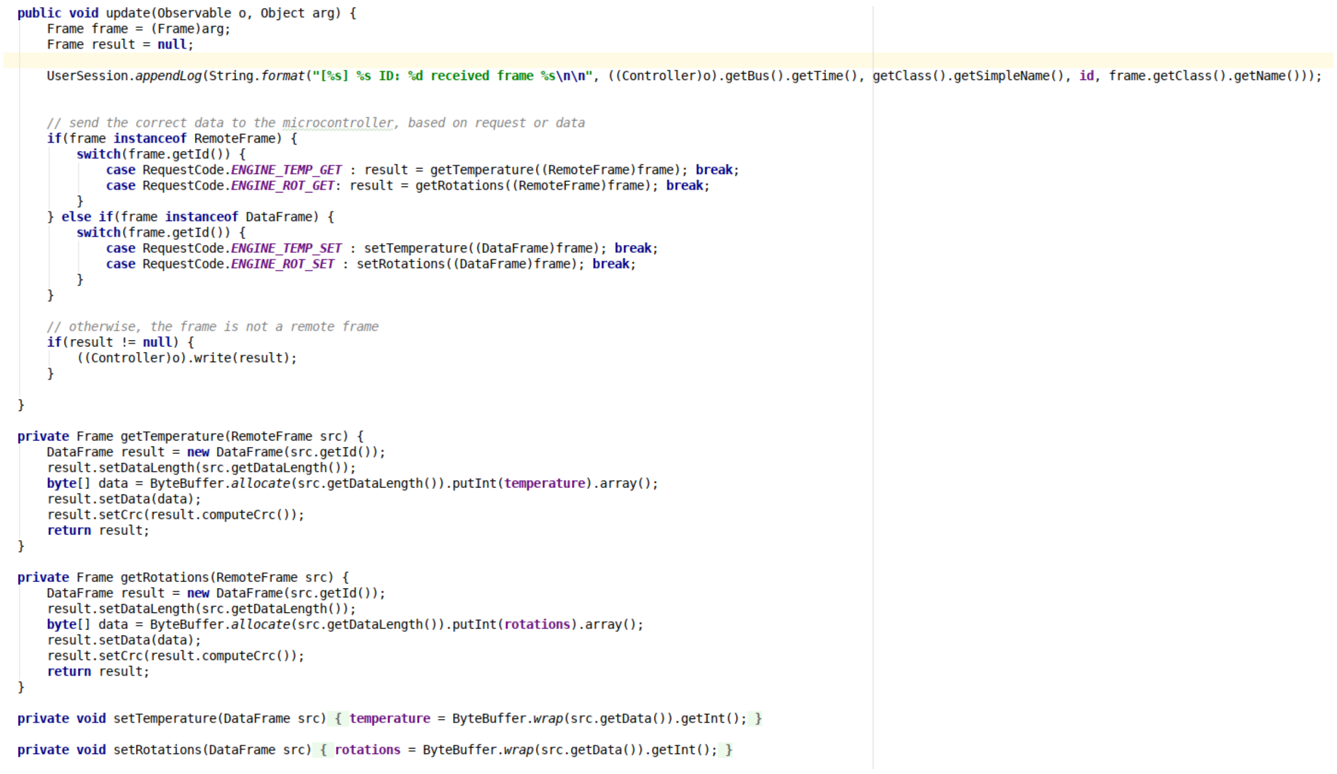
The MicroController class serves as a base for implementing specific microcontroller classes, such as an Engine class or a Dashboard class. An ECU can be connected to multiple buses, therefore it needs to store all the Controller references in a map.

An example concrete implementation is presented below:



```java
public class EngineController extends MicroController {

    private transient Timer timer;
    private int temperature;
    private int rotations;

    public EngineController() {

    }

    public EngineController(int id) {
        super(id);
        timer = new Timer();
        temperature = 0;
        rotations = 0;
    }

    @Override
    public void initializeTransientFields() { timer = new Timer(); }

    public Timer getTimer() { return timer; }

    public void setTimer(Timer timer) { this.timer = timer; }

    @Override
    public void run() {
        timer.scheduleAtFixedRate(() → {
                temperature = (temperature + 10) % 110;
                rotations = (rotations + 300) % 3300;
        },  delay: 5*1000,  period: 5*1000);
    }
```

In this example, an engine is represented by its 2 properties: the *temperature* and *rotations* passed. Since every MicroController instance must run in a separate thread, they must implement the *run()* method, where the state evolution is described.

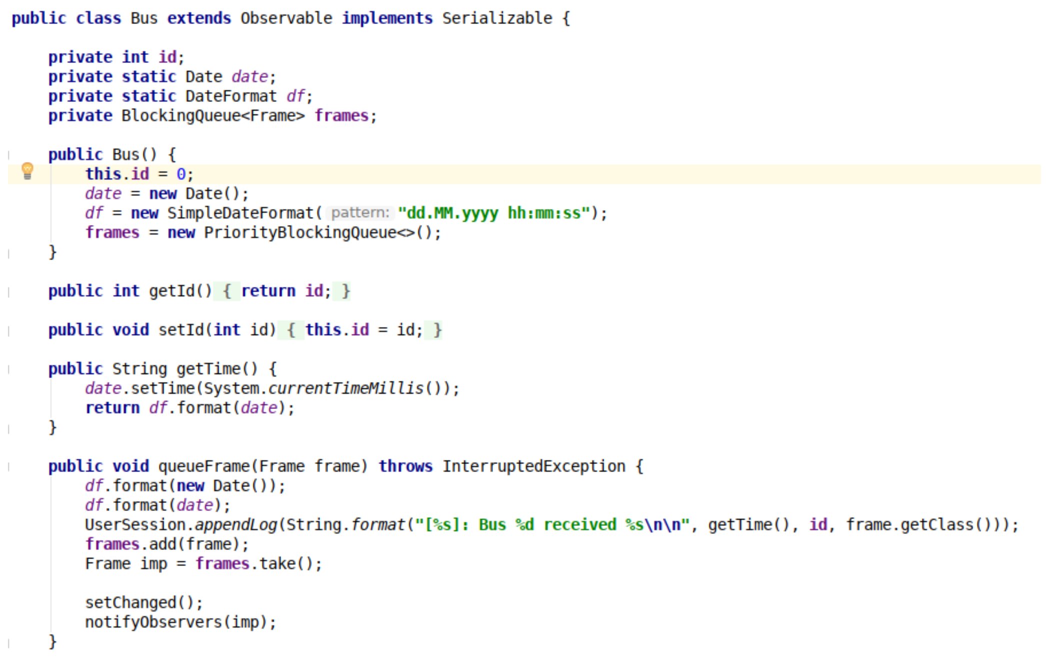


```java
public void update(Observable o, Object arg) {
    Frame frame = (Frame)arg;
    Frame result = null;

    UserSession.appendLog(String.format("[%s] %s ID: %d received frame %s\n\n", ((Controller)o).getBus().getTime(), getClass().getSimpleName(), id, frame.getClass().getName()));


    // send the correct data to the microcontroller, based on request or data
    if(frame instanceof RemoteFrame) {
        switch(frame.getId()) {
            case RequestCode.ENGINE_TEMP_GET : result = getTemperature((RemoteFrame)frame); break;
            case RequestCode.ENGINE_ROT_GET: result = getRotations((RemoteFrame)frame); break;
        }
    } else if(frame instanceof DataFrame) {
        switch(frame.getId()) {
            case RequestCode.ENGINE_TEMP_SET : setTemperature((DataFrame)frame); break;
            case RequestCode.ENGINE_ROT_SET : setRotations((DataFrame)frame); break;
        }
    }

    // otherwise, the frame is not a remote frame
    if(result != null) {
        ((Controller)o).write(result);
    }

}

private Frame getTemperature(RemoteFrame src) {
    DataFrame result = new DataFrame(src.getId());
    result.setDataLength(src.getDataLength());
    byte[] data = ByteBuffer.allocate(src.getDataLength()).putInt(temperature).array();
    result.setData(data);
    result.setCrc(result.computeCrc());
    return result;
}

private Frame getRotations(RemoteFrame src) {
    DataFrame result = new DataFrame(src.getId());
    result.setDataLength(src.getDataLength());
    byte[] data = ByteBuffer.allocate(src.getDataLength()).putInt(rotations).array();
    result.setData(data);
    result.setCrc(result.computeCrc());
    return result;
}

private void setTemperature(DataFrame src) { temperature = ByteBuffer.wrap(src.getData()).getInt(); }

private void setRotations(DataFrame src) { rotations = ByteBuffer.wrap(src.getData()).getInt(); }
```

The *update()* method is called when one of the attached CAN Controllers receives a message. Depending on the received message, the MicroController calls the appropriate method and, if necessary, sends the result back to the Bus.

10.3. Bus

The Bus class tries to mimmick the behaviour of a real CAN message bus.



```java
public class Bus extends Observable implements Serializable {

    private int id;
    private static Date date;
    private static DateFormat df;
    private BlockingQueue<Frame> frames;

    public Bus() {
        this.id = 0;
        date = new Date();
        df = new SimpleDateFormat( pattern: "dd.MM.yyyy hh:mm:ss");
        frames = new PriorityBlockingQueue<>();
    }

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    public String getTime() {
        date.setTime(System.currentTimeMillis());
        return df.format(date);
    }

    public void queueFrame(Frame frame) throws InterruptedException {
        df.format(new Date());
        df.format(date);
        UserSession.appendLog(String.format("[%s]: Bus %d received %s\n\n", getTime(), id, frame.getClass()));
        frames.add(frame);
        Frame imp = frames.take();

        setChanged();
        notifyObservers(imp);
    }
```

At any moment, it two or more CAN controllers send a message to the bus, the one having the highest priority is retained and sent to to all controllers. This is implemented by using a *PriorityBlockingQueue*, so the message having the lowest ID is always the first one to be extracted and, in case of multiple parallel accesses, the data is guaranteed to be valid.

11. Testing

Multiple functionalities have been tested using the graphical user interface, including:

- Creating multiple controller instances having the same ID
- Checking the timestamps printed in the user events log
- Deleting a certain Bus also removes all associated CAN Controllers

12. Bibliography

http://microcontroller.com/learn-embedded/CAN1_sie/CAN1big.htm

http://www.can-wiki.info/doku.php