CAN SW Design Document

Designed by N Krishnamurthy under the supervision of Brad Petrus Bosch Pittsburgh RTC	Iteration 1.0, January 14, 2007

Background:

The Prism middleware (MW) jointly developed by University of Southern California and BOSCH RTC Group, is an architectural software design with abstractions that facilitate distributed software development. The core features provides interfaces for creating components, ports and connectors. The overall composition of a subsystem is specified by the architecture interface which wires up the different constituents.

In addition to providing an event based framework with worker threads responsible for scheduling, dispatching and routing of events between components. Currently the Prism extensions provide OS abstractions of TCP/UDP sockets, and serial communication capability. The objective of this project is to add communication over the CAN bus, and develop test cases as a proof of concept of the functionality of the Prism MW in using the CAN bus.

Design Requirements:

1. To use the USB adaptors developed by PEAK-Systems (http://www.gridconnect.com) and their driver API's to develop a multithreaded higher layer.
2. The higher layer (HL) must take care of fragmentation and assembly of the [0-8] bytes of the data link frames provided by the CAN bus. The HL must be able to detect incomplete segments after assembly (in the event the bus failure) and discard them.
3. The HL must be able to multiplex/de-multiplex parallel connections to the CAN bus.

Higher layer Header design:

The frame format was thus chosen to address priority, multicasting, muxing/demuxing of received fragments, detection of out of sequence fragments and last fragment in a segment.

The middle frame in figure 1.0 is the total data link (DL) frame of a CAN bus. The 11 bit arbitration field is used to provide explicit priority based arbitration when two or more can controllers access the bus simultaneously. 0 bit is dominant over 1 which is recessive thus the choice of the arbitration field enforces an implicit priority.

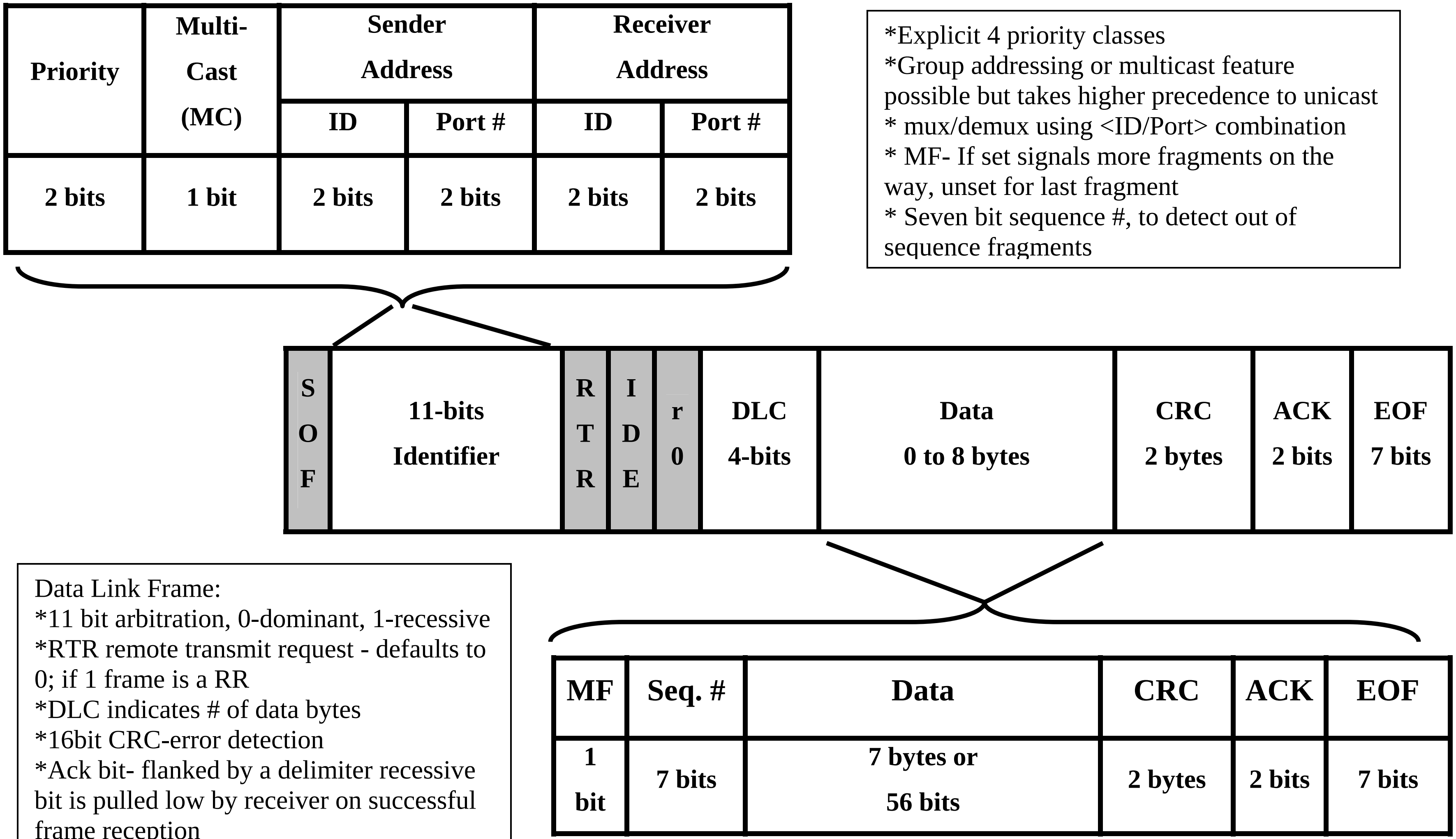
The frame on top of the DL frame is our design of the allocation of the 11 bit identifier. Explicit priority of 4 classes is enforced by allocating the first 2 bits. Two groups are currently addressed by the third MC-multicast bit, with unicast addressing given lower priority over group addressing. Note the sender address/port precedes the receiver address/port thus giving transmitters a higher priority.

The HL muxing and demuxing in the use of the can bus is accomplished by using the notion of ports in addition to host addressing. An endpoint source or destination is thus not only identified by the host address but also the port on which the application issued a request or is expecting a reply.

The MF flag bit is used to signal the receiver that more fragments in a segment are on its way. The last fragment has its MF flag unset signaling the end of the segment.

The seven bit sequence numbers are used to detect out of order fragments and to signal to the driver of the receive error if any fragment in a segment is lost.

Figure 1.0 - Header design that addresses:
priority, multicasting, muxing/demuxing, out of sequence fragments and last fragment in a segment



Priority	Multi-Cast (MC)	Sender Address		Receiver Address	
		ID	Port #	ID	Port #
2 bits	1 bit	2 bits	2 bits	2 bits	2 bits

*Explicit 4 priority classes
*Group addressing or multicast feature possible but takes higher precedence to unicast
* mux/demux using <ID/Port> combination
* MF- If set signals more fragments on the way, unset for last fragment
* Seven bit sequence #, to detect out of sequence fragments

SOF	11-bits Identifier	RTR	IDE	r0	DLC 4-bits	Data 0 to 8 bytes	CRC 2 bytes	ACK 2 bits	EOF 7 bits

Data Link Frame:
*11 bit arbitration, 0-dominant, 1-recessive
*RTR remote transmit request - defaults to 0; if 1 frame is a RR
*DLC indicates # of data bytes
*16bit CRC-error detection
*Ack bit- flanked by a delimiter recessive bit is pulled low by receiver on successful frame reception

MF	Seq. #	Data	CRC	ACK	EOF
1 bit	7 bits	7 bytes or 56 bits	2 bytes	2 bits	7 bits

Driver Design Overview:

The functionality of the different modules can be abstracted into objects and further into a layering scheme as shown below.

<table>
<tr><td colspan="8"><b>Application</b></td></tr>
<tr><td colspan="8"><b>PCAN Driver D[0-15]</b></td></tr>
<tr><td><b>D0</b></td><td><b>D1</b></td><td><b>*</b></td><td><b>*</b></td><td><b>*</b></td><td><b>*</b></td><td><b>D14</b></td><td><b>D15</b></td></tr>
<tr><td colspan="8"><b>PCAN Daemon</b></td></tr>
<tr><td colspan="8"><b>PCAN-USB API</b></td></tr>
</table>

1. The driver abstraction provides an interface to the application layer to send and receive messages over the CAN bus. Multiple instances of the driver can exist for multiple applications/processes to transmit and receive simultaneously.
2. A singleton driverDaemon abstraction allows for the multiple driver instances to register with it to use the lower level API calls of the pcan_usb.dll.
    a. The daemon implements fragmentation and reassembly of higher layer messages using the API's provided by the pcan_usb.dll. The fragmentation and reassembly can be implemented as multiple vectors, with a one to one mapping between the registered driver and the vector.
    b. The daemon is also responsible for registering and unregistering the different driver instances. The registration process initializes the vectors to be used with the associated driver.
    c. The reverse mapping of received fragments to buffers is obtained by decoding the header of the received fragment. The CAN controller filter MASK is enabled to receive broadcast – emergency message packets, and packets addressed to self.
    d. D0 is configured by default as the driver to handle broadcast messages, and any host application on power up registers at least this driver.
    e. The daemon inherits the callback feature to notify the driver on successfully assembling an entire segment.
    f. Out of sequence fragments signals a segment loss and the received segment is flushed. The transmitter can be notified using an RTR frame, for now -- to keep the implementation simple we do not support retransmission .
    g. Three threads are envisaged for simultaneous read, write and cleanup operations performed by the daemon. The cleanup thread flushes the receive vector after its individual timeout.
    h. Tx and Rx threads use individual queues to multiplex and demux the use of the underlying CAN bus by a multithreaded application. Note: The

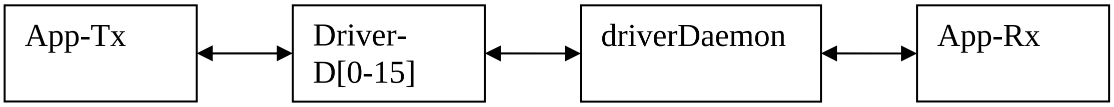
Prism MW which was originally developed in Java has been ported to C++, it has classes defined for threads, mutexes, semaphores, queues and other OS abstractions.

i. The above constructs for threads and mutexes are to be used and restrained use of Standard Template Library (STL), namely queues, lists, vectors would be used to demonstrate the use of CAN in the middleware framework.

j. The lower level API calls are wrapped with mutual exclusion constructs.

See the interface design section for the sequence of function calls between the different objects in the system

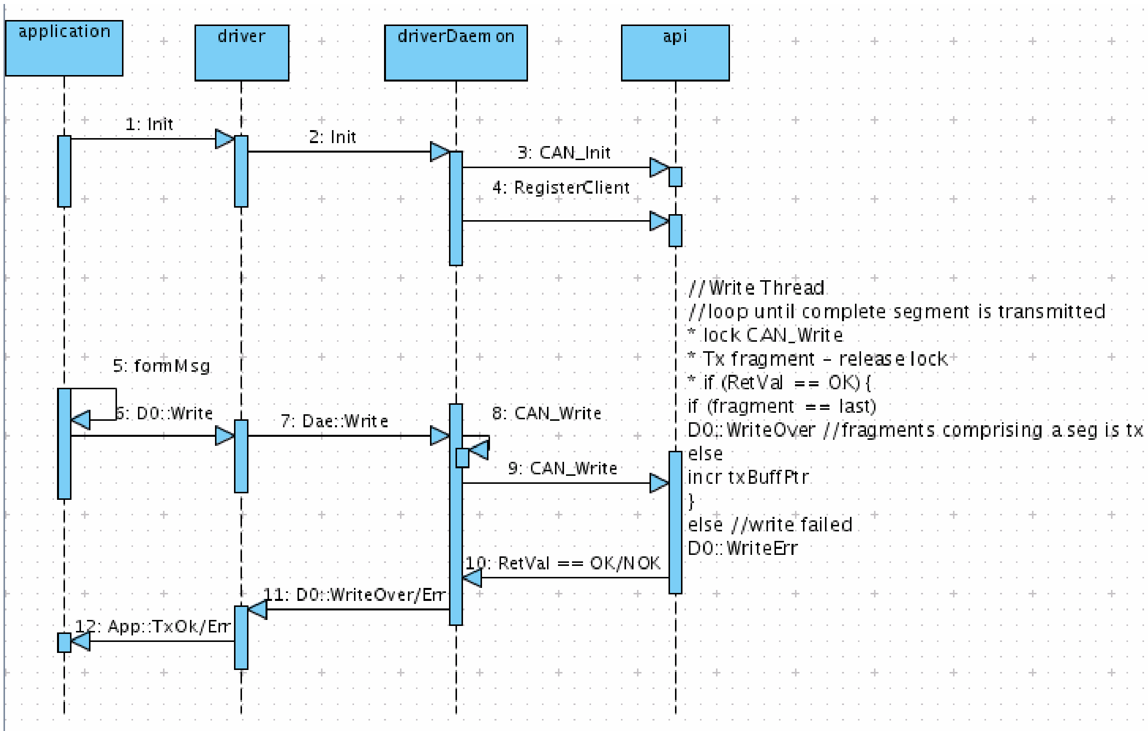
Interacting Objects



The class hierarchy of application and driver for now is kept simple, to a single class to prevent over-objectification. If sufficient properties and behavior distinction of receiver and transmitter is observed upon implementation. A class hierarchy with the common features being exported to an application/driver super class, with specifics of transmission/reception being the extended to respective subclasses can be enforced.

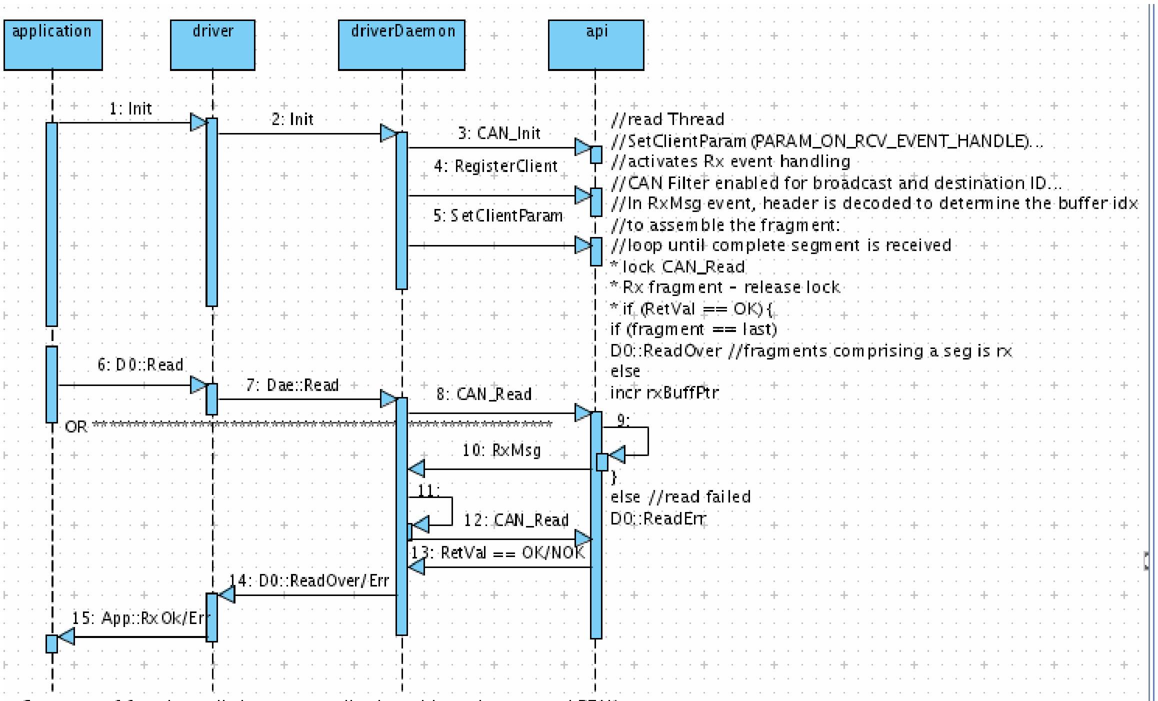
Interface Design – Sequence Diagrams

The sequence diagram is made to expose the interfaces of the implementation in terms of function calls. Note: Interfaces have to be clearly identified before actual implementation.



```
//Write Thread
//loop until complete segment is transmitted
* lock CAN_Write
* Tx fragment - release lock
* if (RetVal == OK) {
if (fragment == last)
D0::WriteOver //fragments comprising a seg is tx
else
incr txBuffPtr
}
else //write failed
D0::WriteErr
```

Sequence of function calls between application, driver, daemon and PEAK API, in the envisaged multithreaded CAN support in Prism middleware --- transmit message or write operation



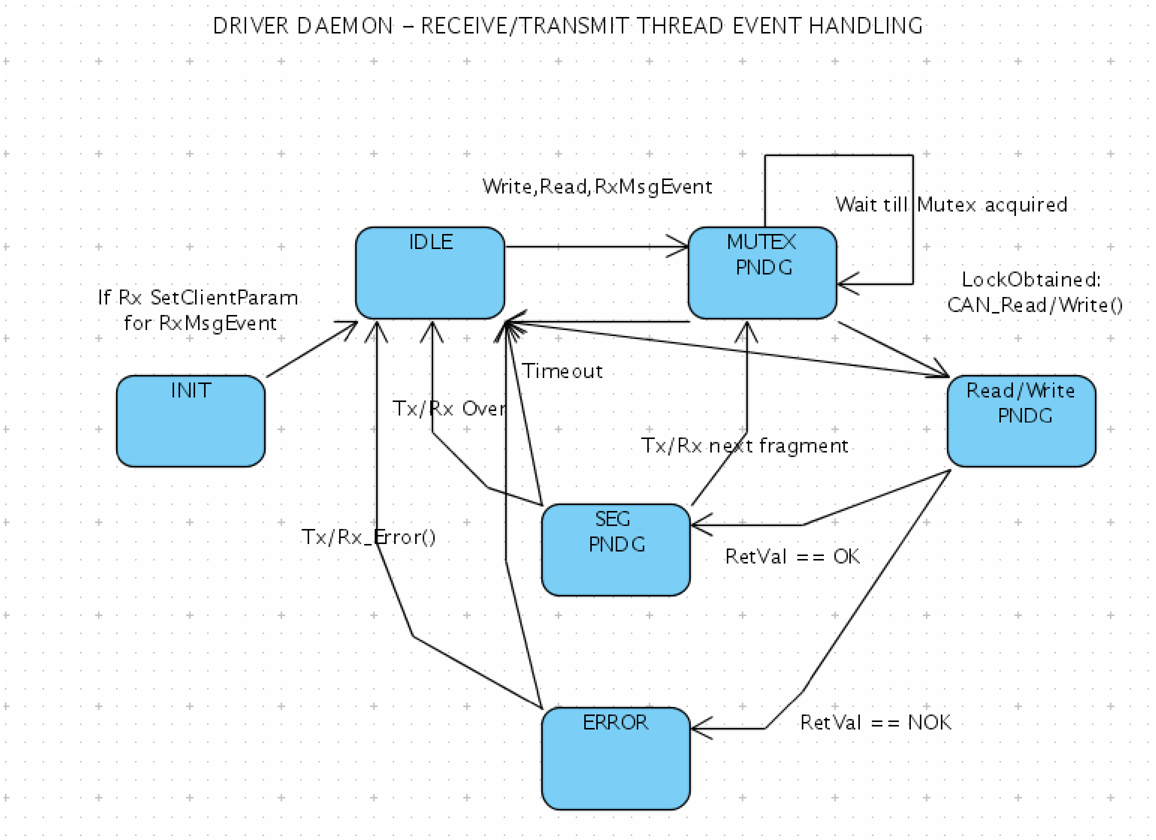
```
//read Thread
//SetClientParam (PARAM_ON_RCV_EVENT_HANDLE)...
//activates Rx event handling
//CAN Filter enabled for broadcast and destination ID...
//In RxMsg event, header is decoded to determine the buffer idx
//to assemble the fragment:
//loop until complete segment is received
* lock CAN_Read
* Rx fragment - release lock
* if (RetVal == OK) {
if (fragment == last)
D0::ReadOver //fragments comprising a seg is rx
else
incr rxBuffPtr
}
else //read failed
D0::ReadErr
```

Sequence of function calls between application, driver, daemon and PEAK API, in the envisaged multithreaded CAN support in Prism middleware, to receive a message(read/RxMsg event processing)

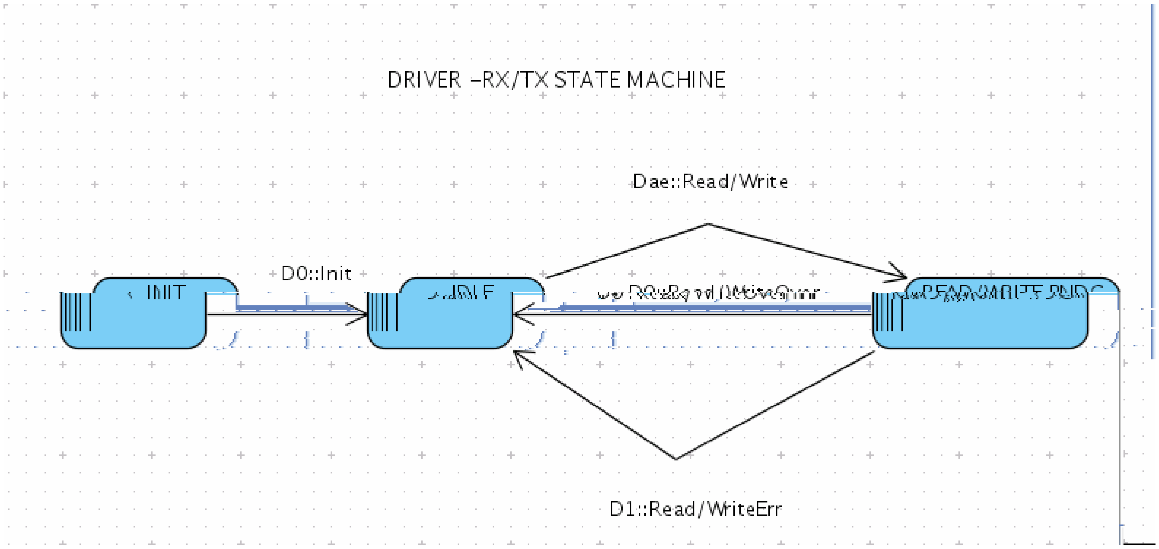
State Machine of TX/RX threads in daemon:

A vector/List of bufferState structure(struct) is used for every transmit/receive buffer. The buffer to driver mapping is obtained (using registration info for Tx or header decoding for Rx fragment), the bufferState struct is passed by reference to the transmit and receive state machines thus pre-empting duplication of state machine code.

```
typedef struct {
char * pBuff;
int offset;          //used by Read/Write thread to update
time creation;  //used by cleanup thread to time out and flush segment
int state
} bufferState;
```



State Machine of TX/RX driver:



Reference:

[1] BOSCH CAN Standard 2.a and 2.b
[2] TCP/IP Illustrated Volume 1 W. Richard Stevens
[3] Whitepapers, Tutorials on CAN Bus : http://www.kvaser.com/can/
[4] PEAK USB CAN driver/developer API – http://www.gridconnect.com