A socket-based interface to CAN

Ivan Cibrario Bertolotti, Gianluca Cena and Adriano Valenzano, IEIIT-CNR

This paper aims at defining an interface for the CAN data-link layer that both fits in well with the IEEE Std 1003.1 *socket* paradigm, and allows the user to access the full range of capabilities implemented by CAN interface controllers. At the same time, this paper also attempts to set up a set of specifications and guidelines for the actual implementation of the proposed interface.

1. Introduction

This paper aims at defining a process-level interface for the transfer of data at the data-link level among CAN nodes that fits in well with the well-known and understood *socket* paradigm, as specified by the IEEE Std 1003.1 [1]. At the same time it allows the user to access a range of capabilities usually implemented by modern off-the-shelf CAN controllers – such as, for example, hardware-assisted filtering and autonomous response to RTR frames – that are not usually available in other kinds of network interfaces and are not accounted for in the traditional socket paradigm.

This paper also gives a set of specifications and guidelines for the actual implementation of the proposed interface in the framework of the socket implementation found in the Berkeley 4.4BSD operating system. We believe that these specifications can readily be applied with few modifications to a wide range of other operating systems, because the Berkeley code base was used as the starting point for many other socket implementations in use nowadays, both open-source and proprietary. The above interface can be easily embedded in those real-time operating systems for advanced industrial and embedded devices that are built on a microcontroller.

The paper is organized as follows: Sections 2 and 3 briefly introduce the main characteristics of the socket Application Programming Interface (API) and CAN, respectively, that will be referred to throughout the paper. Then, Section 4 describes how the socket API must be extended to support the transfer of data at the data-link level among CAN nodes, whereas Section 5 draws some conclusions.

2. The Socket API

The socket facility, and its application programming interface, were initially designed to enhance the *interprocess communication* capabilities of the Berkeley 4.2BSD operating system [2]. Before that release, UNIX systems were generally weak in this area, leading to the offspring of several, incompatible experimental facilities which did not enjoy widespread adoption.

The interprocess-communication facility of 4.2BSD was developed with several goals in mind, the most important of which was to provide access to communication networks, such as the DARPA Internet that was just born at that time, hence the interprocess-communication and network-communication subsystems were tightly intertwined from the very beginning.

Another important goal was to overcome many of the limitations of the existing *pipe* mechanism, in order to allow multi-process programs – such as distributed databases – to be implemented in an efficient and straightforward way. In order to do this it was necessary, for example, to enable *any* pair of processes to communicate.

In summary, the socket facility was designed to support:

transparency: the communication among processes should not depend on the physical location of the communicating processes (on a single host or on multiple hosts), and should be as much independent as possible from the communication protocols being used;

efficiency: in order to obtain higher performance, it was decided to layer interprocess communication on top of network communication and not vice-versa;

compatibility: the new communication facility should not depart significantly from the traditional standard input and standard output interface commonly used by UNIX programs, so that *naive* processes using it should still be usable with no or minimal modifications in a distributed environment.

In order to use the interprocess-communication facility, a process must first create one or more communication endpoints, known as *sockets;* this is accomplished through the invocation of the **`socket()`** function. When doing this, the caller must pass three arguments, namely:

1. *a protocol family* identifier, that uniquely identifies the network *communication domain* the socket belongs to and operates within; for example, **`PF_INET`** identifies the Internet communication domain whereas **`PF_ISO`** identifies the ISO/OSI communication domain.
2. a *socket type* identifier, that specifies which communication model will be obeyed by the socket. For example, a socket of type **`SOCK_STREAM`** is connection-oriented and supports the orderly, reliable delivery of a stream of data, while a **`SOCK_DGRAM`** socket is connectionless and supports the delivery of datagrams without any guarantee of order and reliability;
3. a *protocol* identifier, that selects which specific protocol stack – among those suitable for the given protocol family and socket type – the socket will use. For example, **`IPPROTO_TCP`** selects the Transmission Control Protocol (TCP) and **`IPPROTO_ICMP`** selects the Internet Control Message Protocol (ICMP). Both protocols are defined in the Internet communication domain, so they shall be used only with the **`PF_INET`** protocol family.

The communication domain and the socket type are orthogonal one another, and together determine a (possibly empty) set of communication protocols that belong to the domain and obey the communication model the socket type calls for; then, the protocol identifier can be used to narrow the choice to a specific protocol in the set.

The return value of **`socket()`** is a small integer, known as *socket descriptor,* which uniquely represents the socket and has to be passed to all the other functions referencing the socket itself. The semantics of the **`close()`** function, already defined to close a *file descriptor*, has been overloaded to also destroy a socket, given a socket descriptor.

In order to be actively engaged in data reception, a socket must have a unique local address; the **`bind()`** function allows the caller to associate a specific local address to a socket, or to let the system choose one automatically.

The **`connect()`** function has two different semantics for connection-oriented and connectionless sockets, namely:

- when invoked on a connection-oriented socket, **`connect()`** sets out a connection request directed towards the destination address specified in the call;
- when invoked on a connectionless socket, **`connect()`** simply associates a destination address to the socket itself, so that, in the future, it will be possible to use it with data-transmission functions which do not indicate explicitly the destination address, like **`send()`**; the same function also limits the remote sender address for any subsequent message reception from the socket.

When invoked on a connection-oriented socket, **`listen()`** marks the socket as willing to accept connection requests; it has no meaning for a connectionless socket.

The **`accept()`** functions waits for a connection request to arrive on a given socket, accepts the connection and *clones* the socket so that the *new* socket is connected to the originator of the connection request, and the *old* one is still available to wait for further connection requests; then, it returns to the caller the identifier of the new socket. It has no meaning for a connectionless socket.

The functions **`send()`**, **`sendto()`**, and **`sendmsg()`** allow the caller to send data through a socket, with different levels of expressive power and interface complexity. For example, **`send()`** lacks the ability to explicitly specify the destination address, which is provided by **`sendto()`** instead. The function **`sendmsg()`** is the most powerful one, and also supports data gathering as well as the specification of a set of *ancillary data* along with the data transfer request.

Conversely, the functions **`recv()`**, **`recvfrom()`** and **`recvmsg()`** allow a process to wait for and retrieve incoming data from a socket. Like their transmit-side counterparts, they have different levels of expressive power. The most powerful interface, **`recvmsg()`** returns to the caller the transmitter address and the ancillary data (if any); in addition, it performs data scattering.

The semantics of both sets of functions can be made non-blocking by setting the **`O_NONBLOCK`** flag in the file descriptor by means of the **`fcntl()`** function.

Last, the functions **`getsockopt()`** and **`setsockopt()`** allow the caller to retrieve and set, respectively, a set of *options* supported by either the socket itself or by each level of the protocol stack associated with it. Both of them take as arguments a socket identifier, an identifier that specifies the protocol level at which the option resides, the name of the option to get or set, and a buffer used to store or retrieve the value of the option.

3. CAN Data-link Protocol and Services

The most important, user-visible difference between CAN and the other kinds of network the socket interface was traditionally focused on, is the adoption of a *message-oriented* (instead of *node-oriented*) addressing scheme at the data-link level. Roughly speaking, in a message-oriented addressing scheme, a data-link message does not convey any explicit indication of the identity of its originating and target nodes – that is, the so-called source and destination addresses.

Instead, the contents of the message itself are tagged with a unique message identifier and the broadcasting capability of the communication media is leveraged to transmit the message to all nodes in the network; at this point, each node checks the identifier of the received message against one or more filters to determine whether it is interested in the message itself or not (and, consequently, to either process it further or discard it).

CAN data-link services [3] provide for real-time data transfer among CAN nodes. They are modelled after a producer/consumer relationship and are implemented with minimal protocol overhead. There are two different methods for data transfer, and each of them is implemented by means of its own protocol:

- the method based on *data frame* transfer adheres to the *push* model and is unconfirmed. According to this method, the producer simply sends the data in a CAN data frame and each consumer is notified of its reception.
- the method based on *remote frames* operates according to the *pull* model and is somehow confirmed. It allows a consumer to initiate the transmission of the data it is interested in by sending a remote frame on the CAN bus. The producer replies with the related CAN data frame and all consumers are notified of its reception.

4. CAN-specific Extensions to the Socket API

4.1 Addressing scheme

The socket API makes a clear-cut distinction between the source and target address of a message and its design was oriented towards unicast communication; broadcasting and multicasting are supported as well, with broadcast and multicast addresses being handled as a special form of destination address, but none of these features is either assumed or required at the data-link level.

Instead, as pointed out in Section 3, the CAN data-link protocol has a very different addressing scheme, which is based on message identifiers and heavily relies on the broadcasting capabilities of the communication media. Hence, when dealing with data-link-level protocols, the distinction between the source and target address of a message is moot on CAN.

Accordingly, the semantics of all socket functions dealing with the source and target address of a message has been changed to deal with its message identifier in this case. However, we believe that this twist does not lead to any confusion, because the intended meaning of those functions to the programmer is preserved.

4.2 Data-link Services and Protocol Family

In order to support the CAN data-link services we need to specify both a new protocol family identifier and a new address family, along with the address structure associated with it. In particular:

- the symbolic protocol family identifier **`PF_CANDL`** identifies the set of protocols for data transfer at the data-link level on a CAN network, and can be used in an invocation of **`socket()`** to signify that the socket to be created belongs to the CAN communication domain for data-link data transfer;
- the symbolic address family identifier **`AF_CANDL`** identifies the addressing mode used by the CAN data-link layer and can be used, for example, in an invocation of **`sendto()`** on a socket belonging to the **`PF_CANDL`** domain. Accordingly, the corresponding address template structure shown in Figure 1 represents all possible ways of addressing a communication endpoint for data transfer at the data-link level in CAN.

Referring to Figure 1, the portion of address specific to CAN (i.e., the **`scandl_data`** field) is by itself a structure that holds the various component of the address.

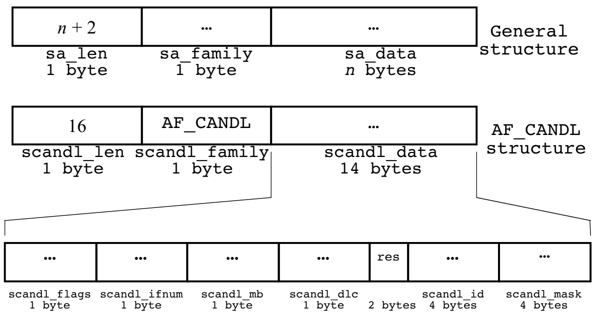

Figure 1: The AF_CANDL address template

Flag	Description
`SCANDL_F_EXT`	The message identifier and its mask (if any) are in extended (29-bit) rather than standard (11-bit) format.
`SCANDL_F_MASK`	The message identifier mask of the address structure has valid contents; setting this flags also requests incoming message filtering to be performed on the corresponding socket.
`SCANDL_F_DLC`	The DLC field of the address structure has valid contents; setting this flags also activates the check of outgoing message lengths against the specified DLC for consistency, and enables the transmission of maximum-size data frames with a DLC greater than 8.
`SCANDL_F_DEFER`	Do not send the output data frame immediately.
`SCANDL_F_ONRTR`	Enable automatic transmission of the data frame in response to an incoming remote frame with the same message identifier.

Table 1: Valid flags in AF_CANDL

This hierarchical way of specifying an address type proves to be very useful to accommodate a wide range of address types in the socket paradigm while, at the same time, minimizing the amount of knowledge that the upper-level portions of the socket implementation must have on the internal structure of addresses belonging to the same family. The components of an **`AF_CANDL`** address are:

`scandl_flags`: holds a set of flags that specify what parts of the address are valid, and how they shall be interpreted; moreover, some flags provide for alternate, nonstandard semantics of send operations and will be described in more detail in Section 4.5. Table 1 lists the allowed flags and summarizes their meaning.

`scandl_ifnum`: contains a positive interface number, used to distinguish between multiple CAN interfaces connected to the same node; the special value zero denotes the default CAN interface.

`scandl_mb`: contains a positive integer that represents a message buffer on the CAN interface and enables the user to exercise a finer control on the interface-level resources that are allocated to the socket; the special value zero leaves this burden to the interface driver.

`scandl_dlc`: this field is valid only if flag **`SCANDL_F_DLC`** is set, and contains the Data Length Code (DLC) to be used for outgoing data frames. Its presence has two purposes: first, it allows the socket implementation to check the actual length of outgoing messages for consistency when their expected length is fixed and known in advance; moreover, by specifying a value greater than 8 here, the user is enabled to send data frames with a payload of 8 bytes but a DLC in the range from 9 to 15, a possibility explicitly allowed by the most recent version of the CAN specifications [3]. If not specified, the DLC is calculated by the socket implementation on a frame-by-frame basis from the data length.

`scandl_id`: contains either a standard (11 bit) or an extended (29-bit) CAN message identifier, depending on the setting of flag **`SCANDL_F_EXT`**. In both cases, the identifier must be left-aligned, so as to place the most significant bit of the identifier into the most significant bit of the field; in this way, the mutual alignment between standard and extended identifiers will conform with their arbitration weight.

`scandl_mask`: holds a mask to be applied to the message identifier to perform incoming message filtering; it is valid only if flag **`SCANDL_F_MASK`** is set.

The socket interface allows an address to be *underspecified* under several circumstances, by leaving one or more of its components unspecified; for example, in the TCP/IP address family, either the IP address or the TCP port number can be left unspecified when binding a socket, with the result of asking the system to use the default IP address of the host, and to allocate a fresh TCP port number.

Identifier	Description
`CANPROTO_DATA`	Plain, bidirectional data transfer.
`CANPROTO_RAW`	Raw access to CAN interface.

Table 2: Valid protocol identifiers for the PF_CANDL communication domain

Another possible use of underspecified addresses is the definition of an input filter, to be further described in Section 4.4: in the **`AF_CANDL`** address family, a set of bits in the message identifier can be marked as "don't care" by specifying an additional *bit-mask* in the address structure. The mask has one bit for each message identifier bit; the mask bit can be either one or zero to denote that the corresponding bit of the message identifier should or should not be taken into account, respectively, when matching incoming messages to sockets.

4.3 Protocol Identifiers

When creating a socket, it is possible to specify *a protocol identifier,* to explicitly indicate which protocol stack must be used with the new socket. In the Internet communication domain, this argument is often left unspecified, because there are only few valid protocol stacks available and the combination of the communication domain and socket type passed to the **`socket()`** function is restrictive enough to uniquely select one of them.

In the CAN communication domain all data-link protocols for data transfer are based on the concept of datagram, hence the socket type is always set to **`SOCK_DGRAM`** and is not of any help in locating an appropriate protocol stack for the socket.

As a consequence, the protocol identifier must be actively used to select the right protocol for the socket; Table 2 lists the new protocol identifiers we defined.

4.4 Plain Data Transfer Protocol

This is the simplest service foreseen by the CAN data-link level, and supports the bidirectional exchange of data and remote frames among CAN nodes; except for the severe constraints on the payload size, its behaviour resembles the other datagram-based protocols such as, for example, the UDP protocol in the Internet domain.

In order to use this service, both the communicating agents should create a new communication endpoint by means of the **`socket()`** function, and select the **`CANPROTO_DATA`** protocol. This function merely creates the socket, but does not allocate any interface-level resource to it, because those resources are often in scarce supply.

Then, a successful invocation of **`bind()`** by the consumer assigns a message identifier and possibly (if flag **`CANDL_F_MASK`** is set in the address structure) an input filter to the socket, and enables the reception of messages that either match the given message identifier or satisfy the input filter. These actions may, and usually do, require the allocation of interface-level resources such as, for example, a message buffer in the CAN controller.

After a successful **`bind( )`**, the consumer can wait for the arrival of a data frame on a socket by means of the usual **`recv()`**, **`recvfrom()`** or **`recvmsg()`** functions. The last two functions also return the actual identifier of the received message; this information may be useful if the socket has an input filter.

To send data, the producer can use either the **`sendto()`** or **`sendmsg()`** functions; both of them allow the caller to explicitly indicate the message identifier to be used for the outgoing data frame. Those functions should allocate temporarily the interface-level resources needed for transmission, request the transmission and then wait until either a positive acknowledge is received from the CAN bus or a transmission error occurs. At this point, they release any interface-level resource they allocated and return to the caller.

According to the socket API, both sets of functions can be made non-blocking by setting the **`O_NONBLOCK`** flag in the file descriptor by means of the **`fcntl()`** function, and this capability has been maintained in the CAN communication domain. Furthermore, the **`select()`** and **`poll()`** functions allow the caller to perform polling and timed waits on a set of sockets to implement, for example, synchronous I/O multiplexing by a single agent.

As an optional step, the producer can associate permanently a message identifier to be used for data transmission and permanently allocate the interface-level resources needed for transmission to a socket by means of **`connect()`**, thus choosing a different tradeoff between efficiency of transmission and resource consumption. After a successful **`connect()`**, **`send()`** can be used for data transmission, too, because the message identifier of the outgoing message can thereafter be determined implicitly.

In order to solicit the transmission of data it is waiting for, and carrying the message identifier specified in the **`bind()`** function, the consumer can also transmit a remote frame by invoking **`send()`** with a dummy data buffer and the **`MSG_OOB`** flag set; the data are then received as before.

The **`MSG_OOB`** flag was chosen because of the loose analogy between a remote transmit request and out-of-band data transmission (otherwise unused in the CAN communication domain). The same flag can also be used in conjunction with **`sendto()`** and **`sendmsg()`** but, if the message identifier passed to those functions is not the same as the identifier passed to **`bind()`**, some implementations may rebind the socket to the new identifier.

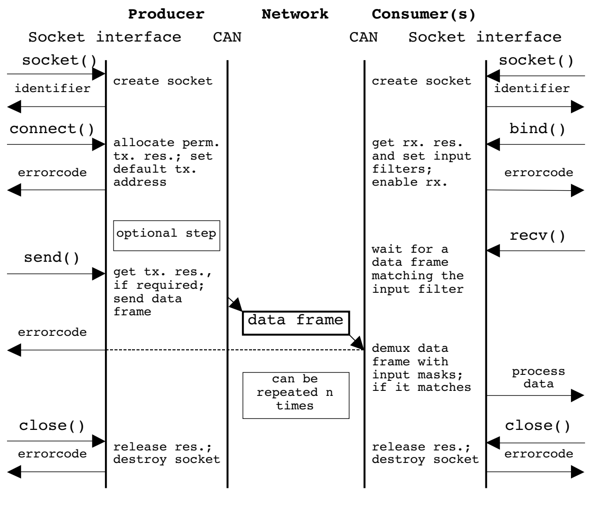
Producer
Network
Consumer(s)
Socket interface
CAN
CAN
Socket interface
socket()
identifier
create socket
create socket
socket()
identifier
connect()
errorcode
allocate perm. tx. res.; set default tx. address
get rx. res. and set input filters; enable rx.
bind()
errorcode
optional step
recv()
send()
get tx. res., if required; send data frame
wait for a data frame matching the input filter
data frame
errorcode
demux data frame with input masks; if it matches
can be repeated n times
process data
close()
errorcode
release res.; destroy socket
release res.; destroy socket
close()
errorcode

Figure 2: Socket setup and data exchange with the plain data transfer protocol

Any agent can destroy a socket by means of **close()**, with the side effect of releasing any interface-level resource allocated to it as soon as possible.

It should be noted that, due to the inherent symmetry of this protocol, a single socket can be used to send and receive data, as is commonly done for other datagram-based protocol, such as UDP; therefore, a single agent can act as both a producer and a consumer at the same time.

Figure 2 shows an example of socket setup and data transfer using this protocol, assuming that the data transmission functions are blocking (their default behaviour), and that no error occurs.

4.5 Raw Access to the CAN Interface

In order to support a more fine-grained control on the allocation of interface-level resources, like CAN message buffers, and a one-to-one mapping between sockets and such resources, the protocol **CANPROTO_RAW** allows the user to access the CAN interface in a more direct way.

As for the plain data transfer, all communicating agents should create a new communication endpoint by means of the **socket()** function. Then, a successful invocation of either **connect()** or **bind()**, performed by each producer and consumer, respectively, assigns a data transfer direction to the socket and allocates any interface-level resource needed by the socket itself. Unlike the **CANPROTO_DATA** protocol described in Section 4.4, a single agent can alternate between the producer and consumer roles, but cannot act as both simultaneously.

For consumers, **bind()** enables the reception of data frames matching the input filter, if any; on the other hand, it is not possible to enable the automatic response to remote frames on the producer side as soon as the producer invokes **connect()**, because the response data have not been set yet.

After a successful **bind()** each consumer can transmit a remote frame (carrying the identifier specified in the **bind()** call) by invoking **send()** with the **MSG_OOB** flag set; the data buffer argument will be ignored in this case.

The **send()** function returns when either the successful transmission of the remote frame has been acknowledged by the CAN controller, or an error has occurred. It should be noted that this step is optional, because a consumer can choose to passively wait for the arrival of a data frame, instead of actively solicit its transmission.

Like before, the consumer can also use **sendto()** or **sendmsg()** instead of **send()** to send a remote frame but, in this case, the socket could be rebound as a side effect.

To wait for the arrival of a data frame any consumer can invoke **recv()**. This function waits for the arrival of a data frame that holds the identifier assigned to the socket, and returns any data it contains to the caller.

On the producer side, the response to remote frames can often be carried out automatically by a hardware-assisted mechanism implemented at the CAN interface level with no or minimal software intervention. The only action the producer must perform is to set up the contents of the data frame to be sent back, and this can readily be accomplished by means of **send()**, **sendto()** or **sendmsg()**.

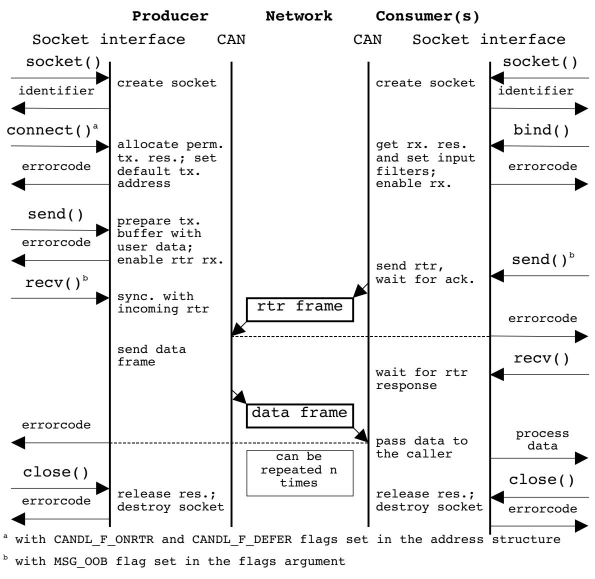

Figure 3: Socket setup and data exchange using remote transmission requests

On the other hand, the producer has two more choices, namely:

- to transmit data once, unconditionally, and then not to react to the reception of remote frames, like in the transfer of data frames;
- to transmit data once, unconditionally, and then keep the same data to be transmitted again, when a remote frame will be received for them.

Therefore, in this case, the behaviour of **`send()`** is controlled by the flags **`CANDL_F_DEFER`** and **`CANDL_F_ONRTR`** located in the address structure passed to **`connect()`**; instead, **`sendto()`** and **`sendmsg()`** can be controlled on a frame-by-frame basis since they have an address structure as a parameter. Table 1 summarizes the allowed flag settings.

In any case, the first successful invocation of **`send()`** (or **`sendto()`**) with the **`CANDL_F_ONRTR`** flag set enables the reception of remote transmission requests related to the message identifier assigned to the socket, while any subsequent invocation with the flag still set merely changes the payload; the reception can be disabled by performing a **`send()`** with the flag **`CANDL_F_ONRTR`** reset (and the flag **`CANDL_F_DEFER`** set).

Optionally, the producer can also synchronize with remote frame reception and answerback, by means of the **`recv()`** function, invoked with the **`MSG_OOB`** flag set; this function does not return any useful information in the data buffer, but waits for the arrival of a remote frame and for the subsequent (automatic) response; then, it reports the outcome of the transmission to the caller. By this mechanism, the producer can gain an idea of how many remote frames were received and when.

The send and receive steps outlined above can be repeated as many times as appropriate. At end, the function **`close()`** should be used to close the sockets and release any resource associated to them.

Figure 3 shows an example of socket setup and data transfer (triggered by a remote frame) using this protocol, assuming that no error occurs.

5. Conclusions

The socket interface is currently the de facto standard for accessing communication services in both internet and intranet distributed environments.

In this paper it has been shown how this scheme can be easily adopted – with proper extensions – also for the industrial and embedded systems that rely on the CAN protocol for communication.

Despite being able to model all the functionalities and aspects peculiar to the communication over a CAN network in a proper way, this interface achieves efficient and streamlined implementations.

References

[1] IEEE Std 1003.1-2001. The Open Group Base Specifications Issue 6, *The IEEE and The Open Group,* 2001.

[2] Marshall Kirk McKusick et al., Internal structure of the 4.4BSD operating system, *Addison-Wesley Longman,* 1996.

[3] ISO 11898-1, Road vehicles – Controller area network – Part 1: Data link layer and physical signalling, *International Standard Organisation*, 2003.

Ivan Cibrario Bertolotti
IEIIT-CNR
C.so Duca degli Abruzzi, 24
10129 Torino - Italy
Phone: +39 011 564 5426
Fax: +39 011 564 5429
E-mail: ivan.cibrario@polito.it
Web: http://is.ieiit.polito.it/staff/cibrario

Gianluca Cena
IEIIT-CNR
C.so Duca degli Abruzzi, 24
10129 Torino - Italy
Phone: +39 011 564 5424
Fax: +39 011 564 5429
E-mail: gianluca.cena@polito.it
Web: http://is.ieiit.polito.it/staff/cena

Adriano Valenzano
IEIIT-CNR
C.so Duca degli Abruzzi, 24
10129 Torino - Italy
Phone: +39 011 564 5410
Fax: +39 011 564 5429
E-mail: adriano.valenzano@polito.it
Web: http://is.ieiit.polito.it/staff/valenzano