Modelling CANopen Communications According to the Socket Paradigm

Gianluca Cena  
IEIIT-CNR  
C.so Duca degli Abruzzi, 24  
10129 Torino - Italy  
gianluca.cena@polito.it

Ivan Cibrario Bertolotti  
IEIIT-CNR  
C.so Duca degli Abruzzi, 24  
10129 Torino - Italy  
ivan.cibrario@polito.it

Adriano Valenzano  
IEIIT-CNR  
C.so Duca degli Abruzzi, 24  
10129 Torino - Italy  
adriano.valenzano@polito.it

Abstract

At present, a number of platforms are becoming available for the implementation of intelligent field devices, which are based on low-cost microcontrollers and open-source real-time operating systems. To take full benefit from these platforms and to ease the task of developing and porting the application software, a standard interface has to be provided to cope with communication facilities.

In this paper an application programming interface for CANopen is introduced that relies on the well-known socket paradigm. Despite its inherent simplicity, it is very flexible and offers the programmer all the functionalities foreseen by the CANopen specification.

1. Introduction

Nowadays, there is a wide availability of inexpensive microcontrollers that, besides the main processor, embed on the same chip many other functions, such as for example volatile and read-only memory, timers, DMA and interrupt controllers, chips select logic, AD/DA converters, and so on. Thanks to synergetic effects with the automotive industry, many microcontrollers available off-the-shelf are provided natively with one or more embedded CAN controllers. In this way, setting up the hardware platform on which to build CANopen devices is really not a difficult task.

The same, however, is not completely true for what concerns software. In fact, despite the good availability of many real-time operating systems conceived explicitly for embedded devices, they are often quite expensive. As a consequence, the design of simple sensors and actuators seldom exploits the most interesting features provided by modern real-time operating systems, such as multithreading, concurrent access to shared resources, process intercommunication, prioritised scheduling, and so on. This has led to the situation where the application program many existing field devices rely on, often builds as a single-threaded process. In such systems, messages received from the network are extracted one at a time from a local queue and dealt with by means of a loop. In spite of its inherent simplicity, such an approach does not provide optimal performance concerning response times, and hardly ensures that timing constraints will always be met.

Recently, a number of open-source real-time operating systems have appeared, such as for example eCos [1], RT-Linux [2] and RTAI [3], which comply with a profile of the POSIX standard and provide a great deal of features at no cost. With no doubt, they constitute an adequate software platform which inexpensive field devices can be built on. In such a context, it can be advantageous modelling network communications by means of the socket paradigm – even though this is not strictly required. Such an approach reduces the efforts of programmers for getting skilled at building distributed applications, and makes the learning curve not as steep as using “ad-hoc” libraries.

For the above reasons, we defined an application programming interface (API) based on the well-known socket paradigm, which complies with the CANopen specification. In order to make this API really usable, great care was taken to provide a high level of flexibility – that is, the programmer/designer should be enabled to seamlessly exploit all the mechanisms and communication services foreseen by the CANopen specification.

This paper must not be considered a complete specification of the CANopen socket interface. Rather, it provides some guidelines and hints on how the above API should be defined, and also describes the way the interface can be used in real application programs. The paper is structured as follows: section 2 describes shortly the conventional socket interface, whereas the general aspects of the socket interface for CANopen are given in section 3. At last, the way remote device configuration and process data exchange can be dealt with through sockets is described in sections 4 and 5, respectively.

2. Socket basics

The socket facility, and its application programming interface, were conceived at first to enhance the interprocess communication capabilities of the Berkeley 4.2BSD operating system [4] and then incorporated into the IEEE Std 1003.1 international standard [5]. The main goal of sockets is to offer a set of interprocess communication functions that is as independent as
possible from the network protocols being used and, at the same time, is also efficient and does not depart significantly from the traditional standard input and output interface commonly used by Unix programs.

Generally speaking, in order to use this facility, a process has first to create one or more communication endpoints, known as sockets, belonging to a specific communication domain and implementing one of several communication models specified by the standard; with reference to Fig. 1, that outlines the typical temporal diagram of socket usage, this is accomplished by means of the `socket()` function. To be actively engaged in data reception, the socket must then be assigned a unique local address either explicitly through the `bind()` function or implicitly.

After creation and binding, connection-oriented sockets pass through a connection establishment phase, mediated by the `connect()`, `listen()` and `accept()` functions; this step is unnecessary for connectionless sockets. Of the functions just mentioned, `listen()` marks a socket as willing to accept connection requests, `connect()` transmits a connection request and returns to the caller its outcome, and `accept()`, when executed on a listening socket, waits for a connection request to arrive and returns to the caller a new socket through which data transfer will take place, while the original socket remains available to wait for further connection requests.

Once a connection is established – or even immediately, for connectionless sockets – the functions `send()`, `sendto()` and `sendmsg()` allow the calling processes to transmit messages through a socket, organized either as datagrams or as a continuous data stream; each function offers a different trade-off between expressive power and interface complexity. They have their counterpart in the functions `recv()`, `recvfrom()` and `recvmsg()`, that allow a process to retrieve incoming data from a socket.

If necessary, the semantics of both sets of functions can be made non-blocking by means of a control operation on the socket they work on. In addition, it is possible to retrieve and set a number of options for each level of the protocol stack associated with a socket. This feature is used to deal with protocol-specific parameters such as, for example, the amount of receive buffer space to be associated with a socket.

At the end of a transfer, applications can (and should) discard the sockets that are no longer in use by invoking the `close()` function on them.

### 3. CANopen Socket API

CANopen [6] is an application protocol for the use in automated factory environments that relies on the well-known controller area network (CAN) technology [7] for transferring information among the nodes.

Besides the aspects concerning communication, CANopen also defines a basic device model and a means for specifying the device behaviour in a standardised way. All the process and configuration data in CANopen devices – both static and dynamic – are stored in the so-called object dictionary (OD). The OD acts as the interface between the CAN network and the application that interacts with the controlled physical system. More information on CANopen can be found in [8].

#### 3.1. Interface definition

In CANopen, as well as in many other popular fieldbuses, aspects concerning the communication over the network and those related to the behaviour of devices are tightly intertwined. This is certainly one of the most appealing features of these solutions, and overcomes many of the limitations of elder protocol stacks, which provided the user a set of application-layer services only.

The ability of specifying the behaviour of (classes of) devices – the so-called device profiles – right into the CANopen specification, in fact, permits a much higher degree of interoperability among devices. At present, it is possible to replace a CANopen device with a different one (maybe produced by a different manufacturer) without changing anything in the system configuration or in the control applications (provided that both comply with the same profile).

However, when dealing with the socket API, only the communication aspects have to be taken into account. In particular we decided not to include any aspect concerning the OD. This is due to many reasons: first we want to obey the original socket philosophy. In this way, we think it is much easier for programmers coming from the UNIX world to learn how to use the interface. Secondly and, perhaps, most important, is that the internal structure of the OD varies noticeably when different classes of devices are considered. Simple devices implement the OD with a small, static structure, whereas more complex devices may rely on dynamic memory allocation. While there are not many degrees of freedom when implementing communication (at most, it makes sense to distinguish between a complete protocol...
3.2. Socket creation

According to the general socket usage discussed in section 2, sockets are created by means of the *socket()* function. To cope with communications over CANopen, a new protocol family was introduced, namely **PF_COP**, that uniquely identifies the CANopen communication domain. After a socket has been created, *bind()* is called so as to define a connection endpoint (for either peer-to-peer or multiplex connections).

Connections in CANopen are defined statically, by setting the CAN identifiers of the frames that are used for exchanging the related COBs (COB-IDs). This information, alongside other parameters that provide details on the connection, is included in a suitable *sockaddr* data structure that is passed to *bind()*.

The *sockaddr* data structure is a variable-length, tag-length-value (TLV) string, used ubiquitously in socket implementations to represent any kind of address, and usually implemented in the C programming language as a tagged union with explicit tag and length fields. It is the common means by which addresses are exchanged both between the socket implementation and its users, and among the internal layers of the socket implementation itself. In order to cope with a new kind of address, the *sockaddr* data structure is usually specialized by introducing a new tag for it. In this case, the **AF_COP** tag was introduced, and, correspondingly, a new variant of the union was defined for it. In the following, only the COB-specific part of the *sockaddr* structure will be shown.

So as to simplify socket configuration and reduce the chance of introducing flaws in the control program, the pre-defined connection set of CANopen should be taken into account. This is a standard allocation of COB-IDs for a number of essential COBs that must be implemented mandatorily by every CANopen device.

The COB-IDs foreseen by this standard allocation are obtained by concatenating a 4-bit *function code* (that specifies the type of the COB, e.g. PDO, SDO, etc.) and the *node identifier* (Node-ID, in the range from 1 to 127) encoded on 7 bits. As for the TCP/IP suite, the node identifier has to be made known to the socket layer by some other means.

4. Service Data Objects

Service data objects in CANopen are used to remotely access the object dictionary of devices. Despite OD management actually pertains to the user and application layers, the fragmentation protocol which SDO transmission relies on is a feature that is typically included in the transport layer. Indeed, it was originally conceived for domain transfer services in the CAN-based application layer (CAL). Hence, it can be dealt with adequately by means of the socket interface.

According to the CANopen specification, each device must provide (at least) one *server SDO* (SSDO), used for accessing its OD. To this extent, a pair of CAN identifiers are allocated in the pre-defined connection set (for transmitting and receiving data, respectively) which can not be deleted nor modified. Optionally, devices can have additional SSDOs, as well as *client SDOs* (CSDOs) to be used mainly by configuration tools. Besides the simpler static SDO allocation, a mechanism aimed at defining dynamic SDO connections at run-time has been included in [9].

4.1. SDO socket creation

Two protocol identifiers have been defined in the socket interface for handling SDO transfers, namely **COPPROTO_SSDO** and **COPPROTO_CSDO**. A call to *bind()* issued on the server side is used to set up an SDO connection endpoint, by defining a pair of CAN identifiers (one per direction). To make configuration simpler, a default value is supported for the first SSDO.

Adapting the socket interface to cope with all details of the SDO transfer protocol in a proper way is actually a bit tricky task, because SDOs are modelled upon a client-server scheme, where the client is always responsible for initiating any data transfer. Two kinds of transfers can take place over an SDO connection, that is *download* (data flow from the client to the server) and *upload* (data are read from the server).

In CANopen, the object exchanged by means of an SDO is always an entry of the OD, which is identified uniquely and unambiguously by means of a multiplexer made up of a 16-bit index and a 8-bit sub-index. Unlike the other data exchanges foreseen by CANopen, the object size is allowed to exceed the maximum payload of CAN frames (i.e., 8 bytes). This means that a mechanism is required, which breaks the original data in smaller chunks (the so called fragments) on the transmitter side.

![Fig. 2. Format of sockaddr for SDOs.](image-url)
and reassembles them in the receiver so as to obtain the initial object back.

From a practical point of view, a concept of temporary connection has been introduced, which is used to exchange exactly one object. We’ve called it an object (or SDO) transaction. Transactions have to be initiated on an SDO connection endpoint each time an object has to be exchanged, and are terminated automatically as soon as the object exchange ends. SDO sockets, instead, should be closed explicitly by means of close(). This also holds for the sockets returned by accept().

Each transaction begins with a suitable SDO initiate request/response message pair exchanged between the client and the server, that is used to negotiate the parameters of the transaction (direction, transfer type, multiplexer of the entry of the OD to be accessed, details on the size of object, etc.). The CANopen specification permits the sender to leave the size field unspecified in the SDO download initiate request – as well as in the SDO upload initiate response. To carry out fragments reassembly, a “more” flag is included in each SDO download segment request and SDO upload segment response to inform the receiver whether or not more fragments will follow.

To achieve maximum flexibility, the socket interface supports two different fragmentation modes, that is transparent and explicit. In the former case the socket layer manages all details of the fragmentation protocol in a transparent way, whereas the latter case is used to set up a stream-like data transfer where the user has a transparent way, whereas the latter case is used to set

layer manages all details of the fragmentation protocol in transparent mode for small objects and segmentation for the others.

Concerning these flags, we have:

- **scop_index**: index component of the object’s multiplexer;
- **scop_subid**: sub-index component of the object’s multiplexer;

Finally, the transmitter of the object is allowed to set the following values:

- **scop_len**: size (in bytes) of the object to be transferred; optionally set by sendto() and provided as return value in recvfrom();
- **scop_tflags**: a bunch of flags defining additional aspects of the SDO transaction.

Among the flags in scop_tflags, there are:

- **SCOP_TF_SI**: size indicator (specified or unspecified), set by the object transmitter through sendto() tells the receiver if the scop_len field has valid contents (useful in stream-like transfers);
- **SCOP_TF_DIR**: direction of data transfer (either download or upload); actually, this flag is set automatically by the client (and not by the transmitter) and is made available to the server through accept();

4.2. Initiating an object transaction

A problem when dealing with SDO exchanges is that the application program running on the server must be informed about both the direction of the transfer and the object that is being exchanged. The former requirement depends on the fact that the server should be able to accept both download and upload transfer requests coming from the client. Knowing the object multiplexer, instead, can be advantageous if large objects have to be downloaded to very simple devices, that are usually provided with small amounts of memory. In this case, the received object can be stored into the right entry of the server’s OD directly.

As depicted in the upper part of Fig. 3, accept() can be used on the server side to admit an incoming SDO transaction. As soon as a SDO download/upload initiate request is received, accept() returns the relevant information about the transaction (multiplexer

![Fig. 3. Normal/expedited downloads.](image-url)
and direction) by means of the sockaddr structure. At this point, it is up to the server application program to invoke either recv() or send() with the right receive/transmit buffer, to deal with download or upload operations, respectively.

In a similar way, the first call issued on the client side is connect(), which sets the multiplexer of the object to be exchanged. It is worth noting that connect() does not send anything over the network. It is up to the first recv() or send() call issued on the client side to set the right direction of the transfer and send the related SDO download/upload initiate request over the network.

4.3. SDO download

The simplest way to accomplish SDO downloads is as shown in Fig. 3. In this case the client, after invoking connect(), issues a send() (or sendto()) call for transferring the actual object data. In the same way recv() is invoked on the server side after accept() to receive the object.

As said before, other than the scheme depicted above, where the object is transferred as a whole (from the user’s viewpoint), the CANopen socket interface foresees an additional transfer mode where the user is made aware of the fragmentation protocol. To select such a kind of behaviour, the size of the transmit buffer in the first send() should be set to 0. Using sendto() could be advantageous, in that it permits providing the receiver with the size of the object. As an alternative, setsockopt() can be used.

As depicted in Fig. 4, this mechanism enables a sort of stream-like data transfer, in which the sender application produces its data one segment at a time by invoking send() repeatedly. The MSG_EOR flag is used on the last send() call to notify the socket layer (and, consequently, the receiver) that the transfer is finished and the transaction has to be terminated.

A similar transfer mode can be enabled independently on the receiver side, too. Once explicit fragmentation is enabled (by passing a zero-sized buffer to the first receive call), each subsequent call to recv() confirms the reception of the last exchanged segment and triggers the transmission of a new segment. Its return value can be used to determine whether or not there are other segments to be read. In particular, 0 means that the object transfer is completed and the transaction has been terminated.

4.4. SDO upload

Modelling upload services is a bit more difficult. In this case, in fact, the originator (that coincides always with the client) is not the same as the transmitter of data (i.e., the server). A satisfactory arrangement for them is shown in Fig. 5.

In some cases, the client application might be interested in knowing the size of the object (which is known by the server only) before the exchange is effectively carried out (in order to allocate memory, for example). This can be accomplished either by setting to 0 the size of the receive buffer in the first recvfrom() or by means of getsockopt(). This call returns the required information to the client through the sockaddr structure. If an expedited transfer was issued by the object transmitter, data are made available the next time recv() is invoked. In doing so, explicit fragmentation is enabled on the client side for the current transaction.

Fig. 4. Explicit fragmentation in downloads.

Fig. 5. Normal/expedited uploads.

Fig. 6. Getting object size in uploads.
5. Process Data Objects

Process data objects are used in CANopen to exchange real-time data involved in controlling the physical system. Each PDO is mapped on exactly one CAN frame, and no protocol control information is included in the data field. This means that up to 8 bytes are available to encode process data. All the information concerning a PDO are embedded right into the identifier field of the CAN frame. This means that, in order to have devices exchanging meaningful information through PDOS, they must agree in advance on which piece of data is assigned to each COB-ID. It is possible to distinguish between transmit PDOS (TPDOs) and receive PDOS (RPDOs), where the direction of the data transfer is as seen by the device viewpoint. Hence, devices produce TPDOs and consume RPDOs.

In order to improve efficiency, CANopen permits several application data (that are stored in specific entries of the OD) to be gathered in the same PDO. This is know as PDO mapping. Simple devices only support static mapping, whereas the most complex ones also provide a means to deal with dynamic mapping. Furthermore, device profiles often define a default mapping, so that a device can, in theory, be connected to the network and operated on with no prior configuration. Even though interesting, PDO mapping is not dealt with directly by the socket interface, in that this would involve accessing the entries of the OD, which is outside the communication context.

As PDO exchanges are quite simple, one might wonder whether or not it does bring any advantages to model them according to the socket paradigm. Besides the unified way to access communication facilities, CANopen sockets handle implicitly aspects such as synchronous transmission modes, inhibit and event times, and so on. So they can unburden the programmer from most details concerning communication.

A very important kind of entry in the OD, which is devoted to configure PDOS, is the PDO communication parameter. It includes all of the aspects related to the exchange of PDOS over the network. Besides the COB-ID used at the data-link level, it defines the transmission type parameter, which specifies the triggering type to be used by the PDO. The transmission type (in the range from 0 to 255) is encoded on one byte and concerns not only communications, but also the operations carried out at the application level (i.e., actuation and sampling).

As we decided not to include anything involving the application in the socket interface, the meaning of the transmission type here covers only the communication aspects. Hence, the available choices are not exactly the same as those foreseen in the CANopen specification.

5.1. PDO socket creation

When the socket is created by means of the socket() call, its producer/consumer role is decided (either COPPROTO_TPDO or COPPROTO_RPDO). The first bind() invoked on the socket will set the actual behaviour of the PDO, which is described by means of the sockaddr structure. Besides the transmission type, this structure also defines (optionally) the inhibit time and the event timer, the period and initial phase for synchronous transmission, and so on. The sockaddr structure to be used for PDO sockets is depicted in Fig. 8. Among the fields included in the structure, we have:

- **scop_id**: the COB-ID used for the PDO;
- **scop_num**: the PDO number; for PDOS 1 to 4 the COB-ID is pre-configured according to the pre-defined connection set, but can be changed;
- **scop_tt**: transmission type, to be selected according to the rules described below;
- **scop_inhib**: inhibit time, e.g., the minimum time which must elapse between two consecutive transmissions of the same PDO;
- **scop_event**: event timer, e.g., the maximum time between any two subsequent asynchronous transmission; whenever exceeded, the socket layer will send the PDO autonomously;
- **scop_period**: transmission period for a cyclic PDO, expressed as an integral number of communication cycles;

<table>
<thead>
<tr>
<th>COB-ID</th>
<th>PDO number</th>
<th>Transm. type</th>
<th>Event timer</th>
<th>Inhibit time</th>
<th>Sync period</th>
<th>Sync offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>1 byte</td>
<td>1 byte</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>1 byte</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

Fig. 8. Format of sockaddr for PDOS.
5.2. Transmit PDOs

When the socket is set up as PDO producer, the following transmission types can be selected:

- **SCOP_TTASY**, used for modelling asynchronous transmissions (CANopen transmission types 254 and 255): the message is sent as soon as `send()` is invoked, as depicted in Fig. 9.a;
- **SCOP_TT_RTR**, for RTR-only transmissions: `send()` updates a local buffer on the producer side, and data will be sent only when the related RTR is received from the network (Fig. 9.b);
- **SCOP_TT_SYNCYC**, for synchronous cyclic transmissions: `send()` is used to update a local buffer, and the message will be effectively exchanged when the SYNC message is received from the network (Fig. 9.c); it is up to the socket layer handling the (optional) PDO period and offset;
- **SCOP_TT_SYNRTR**, for synchronous acyclic transmissions: similar to the previous case, but each piece of data is sent only once, i.e., the message is not transmitted again unless a new value is set by means of `send()`.

Even though interesting, the synchronous and RTR-only modes depicted above are not the same as those foreseen by CANopen. In that case, in fact, the reception of the SYNC or RTR messages triggers sampling, and only then the PDO is sent over the network (hence, the term “synchronous” does not refer to communication only). This behaviour can be achieved quite easily by means of the following transmission types:

- **SCOP_TT_SYN**, for dealing with synchronous transmissions (0-240): the application program on the producer first waits for the SYNC message, then performs sampling and finally sends the acquired value explicitly by means of an asynchronous transmission, as shown in Fig. 9.d. Synchronisation on incoming SYNC messages is accomplished by means of `recv()`, `select()` or `poll()`, and can be carried out either by opening a special socket for accepting SYNC messages, or by performing a call for receiving OOB messages on the same socket used to send TPDOs (this is permitted only when this transmission type is selected). The latter choice is the most advantageous, in that it is possible to specify period and offset of synchronous transmissions on a per-socket basis.
- **SCOP_TT_SYNRTR** for CANopen synchronous RTR-only transmissions (252): it is similar to the previous case, however the TPDO is not exchanged immediately, but buffered locally and sent only when the related RTR is received from the network.
- **SCOP_TT_ASYRTR** handles the CANopen asynchronous RTR-only transmission mode (253): a means is provided so that the reception of a RTR for a given TPDO can be notified to the producer application. By selecting this transmission type the user is enabled to wait for an OOB message on the socket. In this way it can be awakened on every incoming RTR targeted at the intended TPDO. Then, as depicted in Fig. 9.e, data is sampled, after which an usual `send()` is invoked to send it over the network.

5.3. Receive PDOs

RPDO sockets are used by process data consumers. The following transmission types are foreseen:

- **SCOP_TTASY**, used for carrying out reception of asynchronous data (254, 255): in this case data are made available to the control application through `recv()` as soon as the related message has been read from network (Fig. 10.a);
- **SCOP_TT_ASYRTR**, used for asynchronous RTR-only exchanges (253): it is implemented in the same way as the previous case, however the consumer is enabled to request data from the producer by sending an RTR; this is achieved by transmitting an empty OOB message on the same socket used to read incoming RPDOs (see Fig. 10.b);
- **SCOP_TT_SYN**, used to model both cyclic and acyclic synchronous exchanges (0-240): messages are buffered in the socket layer and made available to the application only when the SYNC message is received from the network, as depicted in Fig. 10.c;
- **SCOP_TT_SYNRTR**. for handling reception according to the synchronous RTR-only scheme (252): similar to the previous case, but the consumer is allowed to send RTR messages to request data;

![Fig. 9. Transmission modes for TPDOs.](image-url)
Sometimes, in the case of synchronous RPDOs, it is preferable to make data available to the application as soon as they are received; this gives the application a way to carry out pre-processing on the incoming data, which will be eventually actuated when the SYNC message is received. In this case the socket should be set as `SCOP_TT_SYNIMM`, and `recv()`, `select()` or `poll()` must be invoked to wait for the SYNC message (modelled as an OOB message), as shown in Fig. 10.d.

In the case pre-processing involves several data, it is possible to exploit the synchronous window concept: after a given time (the so-called synchronous window length) has elapsed after the last SYNC, it is possible to assume that all the relevant RPDOs have been collected and the application can be notified of the related values. Fig. 10.e depicts this arrangement, which is selected through the `SCOP_TT_SYNDEL` transmission type.

6. Conclusions

At present there is wide availability of low-cost microcontrollers provided with a reasonable processing power, as well as full-fledged open-source real-time operating systems. Because of their interesting performance, an ever-increasing number of field devices will be likely based on such platforms.

In this paper a socket-based API has been introduced whose aim is to access the communication facilities of CANopen. Such an interface is very simple and intuitive. In fact, sockets are well known and understood by programmers that are used to write applications for UNIX-like environments. In addition, it supports all the functions foreseen by the CANopen specification. A further advantage of using a standard well-agreed interface to CANopen communications is portability. As long as the CANopen API is provided by the operating system, applications can be easily moved from one platform to another with minimum effort. Another example of socket interface for CAN is given in [10].

The CANopen socket interface can be readily and easily embedded in most real-time operating systems, so as to offer field-device designers an inexpensive platform for developing their products. This is a main requirement because, with the advent of high-performance industrial Ethernet solutions, in the near future the CANopen technology will be likely targeted at cheap (intelligent) devices.

References