CANopen

high-level protocol for CAN-bus

H. Boterenbrood

NIKHEF, Amsterdam
March 20, 2000

Version 3.0

Contents

Version History		
Version	**Date**	**Comments**
1.x	1998	First draft versions
2.0	1999	Change of title; change in chapter numbering; addition of CAN message syntax details
2.0a	7 April 1999	Corrected error in table 7 and 8 captions
3.0	20 March 2000	Several additions in accordance with CiA DS301 CANopen Communication Profile Version 4.0 (update from 3.0)

1 Introduction

Fieldbus networks from the OSI network model point-of-view usually only have the layers 1 (Physical Layer), 2 (Datalink Layer) and 7 (Application Layer) implemented. The intermediate layers are not needed because a fieldbus network usually consists of a single network segment only (no need for Transport and Network Layer, layer 3 and 4) and has no notion of 'sessions' (layer 5) or a need for different internal data 'presentation' (layer 6).

The **CAN** (*Controller Area Network*) fieldbus defines only the layers 1 and 2 (ISO11898); in practice these are completely handled by the CAN hardware, significantly reducing the implementation effort for developers of the fieldbus node firmware.

However, a high level protocol is necessary in order to define how the CAN message frame's 11-bit identifier and 8 data bytes are used. Building CAN-based industrial automation systems guaranteeing interoperability and interchangeability of devices of different manufacturers requires a standardized **application layer** *and* '**profiles**', standardizing the communication system, the device functionality and the system administration:

- The **application layer** provides a set of services and protocols useful to every device on the network imaginable.
- The **communication profile** provides the means to configure devices and communication data and defines how data is communicated between devices.
- **Device profiles** add device-specific behaviour for (classes of) devices (e.g. digital I/O, analog I/O, motion controllers, encoders, etc.).

The following sections describe the **CAL** application layer for CAN networks on which the *CANopen* standard is based and subsequently *CANopen* itself and the **profiles** that define it.

The relation between OSI network model, CAN-bus standards and CANopen profiles is illustrated in Figure 1.

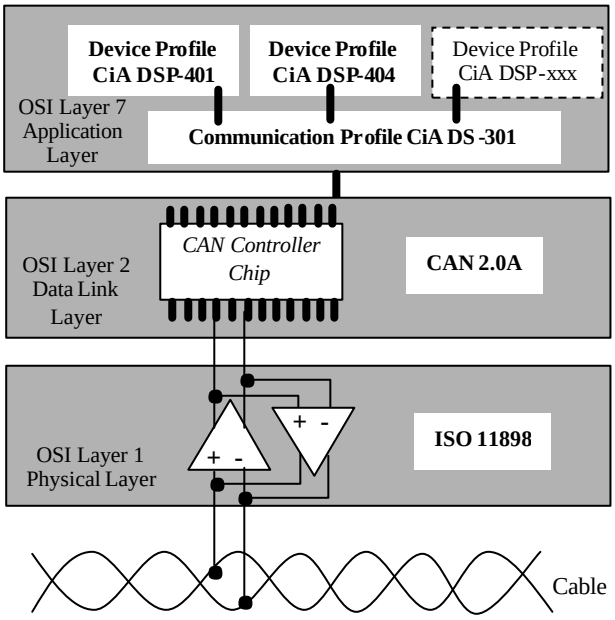

Figure 1. Schematic overview of CAN and CANopen standards in the OSI network model.

2 CAL

One of the existing higher layer communication protocols for CAN-based networks –developed by Philips Medical Systems– is **CAL** (*CAN Application Layer*). It was adopted by the independent CAN users and manufacturer group **CAN in Automation** (CiA), developed further and published in a series of standards [1].

CAL provides 4 application layer service elements:

1. **CMS** (*CAN-based Message Specification*): offers objects of type Variable, Event and Domain to design and specify how the functionality of a device (a node) can be accessed through its CAN interface (e.g. how to up- or download a set of data ('domain') exceeding the 8 bytes maximum data content of a CAN-message, including an 'abort transfer' feature).

CMS derives from MMS (*Manufacturing Message Specification*), which is an OSI application layer protocol designed for the remote control and monitoring of industrial devices.

2. **NMT** (*Network ManagemenT*):
   offers services to support network management, e.g. to initialize, start or stop nodes, detect node failures; this service is implemented according to a master-slave concept (there is one NMT master).

3. **DBT** (*DistriBuTor*):
   offers a dynamic distribution of CAN identifiers (officially called **COB-ID**, Communication Object Identifier) to the nodes on the network; this service is implemented according to a master-slave concept (there is one DBT master).

4. **LMT** (*Layer ManagemenT*):
   offers the ability to change layer parameters e.g. change the NMT-address of a node,
   or change bit-timing and baud-rate of the CAN-interface.

CMS defines 8 priority levels in its messages, each having 220 COB-IDs, occupying COB-IDs 1 to 1760; the remaining identifiers (0, 1761-2031) are reserved for NMT, DBT and LMT. See Table 1.
In a CAN-network the lower the value of the COB-ID, the higher the priority of the corresponding message on the network is.

Note that this standard assumes *CAN2.0A* (CAN Standard Message Frame) having an 11-bit identifier, allowing a range of [0, 2047], but which -for historical reasons- is limited to [0, 2031]. However using *CAN2.0B* (CAN Extended Message Frame) with 29-bit identifier doesn't change the picture: the 11-bits range in the table maps to the 11 most significant bits of the 29-bits COB-ID and the COB-ID ranges in the table will just become (much) larger.

CAN Application Layer (CAL)	
COB-ID	**Usage**
0	NMT start/stop services
1 - 220	CMS objects priority 0
221 - 440	CMS objects priority 1
441 - 660	CMS objects priority 2
661 - 880	CMS objects priority 3
881 - 1100	CMS objects priority 4
1101 - 1320	CMS objects priority 5
1321 - 1540	CMS objects priority 6
1541 - 1760	CMS objects priority 7
1761 - 2015	NMT Node Guarding
2016 - 2031	NMT, LMT, DBT services

Table 1. COB-ID (11-bit CAN-identifier) mapping to *CAL* services and objects.

3 CANopen

CAL provides all network management services and message protocols, but it does not define the contents of the CMS objects or the kind of objects being communicated (it defines *how*, not *what*).
This is where ***CANopen*** enters the picture.

CANopen is built on top of *CAL*, using a subset of *CAL* services and communication protocols; it provides an implementation of a distributed control system using the services and protocols of *CAL*.
It does this in such a way that nodes can range from simple to complex in their functionality without compromising the interoperability between the nodes in the network.

Central concept in *CANopen* is the **Device Object Dictionary** (OD), a concept used in other fieldbus systems as well (*Profibus*, *Interbus-S*).
Note that the Object Dictionary concept is not part of *CAL*, it is an implementation aspect of *CANopen*.

In the following sections we will first present the Object Dictionary, and only then the *CANopen* communication mechanisms.

3.1 CANopen Object Dictionary

The *CANopen* Object Dictionary (OD) is an ordered grouping of objects; each object is addressed using a 16-bit index. To allow individual elements of structures of data to be accessed an 8-bit subindex has been defined as well. The general layout of the *CANopen* OD is shown in Table 2.
Don't be confused by all the 'data types' located in the OD at indices below 0FFF; they're there mainly

for definition purposes only. The relevant range of a node's OD lies between 1000 and 9FFF.

For every node in the network there exists an OD. The OD contains all parameters describing the device and its network behaviour.

The OD of a node mainly exists in the form of a database described in the *EDS* (see below) or on paper. It is not necesssarily possible to 'interrogate' a node via the CAN-bus about all its parameters in its OD. It is sufficient if the node behaves exactly according to its OD description on paper. The node itself only needs to be able to present at least all mandatory OD entries (as dictated by *CANopen;* these are actually very few), and optionally any other entries that form part of the configurable functionality of the node.

CANopen is defined in the form of documents describing **profiles**.

There is the **communication profile** [2] describing the general form of the OD and the objects in the OD's Communication Profile Area, the communication parameters; it also describes the *CANopen* communication objects (see next section); this profile applies to all *CANopen* devices.

Then there are the various **device profiles** (e.g. [3]) defining for a particular type of devices the OD objects; there are now about 5 different device profiles and several more are under development.

A profile describes for each OD object its function, its name, its index and sub-index, its data type, whether the object is mandatory or optional, whether the object is 'read only' or 'write only' or 'read/write', etc.

The description of a device's communication functionality and objects and its device-specific objects and their default values is proviced in the form of an **Electronic Data Sheet** (EDS), an ASCII file with a strictly defined syntax.

A description of the object configuration of an individual device is called a **Device Configuration File** (DCF) and has the same structure as an EDS.

Both file types are defined in the *CANopen* specification.

The profiles define which OD objects are mandatory and which are optional; the number of mandatory objects is kept to a minimum allowing lean implementations.

Optional features –in the communication part as well as the device-specific part– can be added as required to extend a *CANopen* device's functionality.

If more features are required than are present in the profile, there is plenty of space in the profile available for the addition of manufacturer-specific functionality.

So the part of the OD describing the communication parameters is the same for all *CANopen* devices (i.e. object placing in the OD is the same, not necessarily the value of the object…), the device-specific part of the OD is different for different (classes of) devices.

CANopen Object Dictionary	
Index	**Object**
0000	*not used*
0001 - 001F	Static Data Types (standard data types, e.g. Boolean, Integer16)
0020 - 003F	Complex Data Types (predefined structures composed of standard data types, e.g. PDOCommPar, SDOParameter)
0040 - 005F	Manufacturer Specific Complex Data Types
0060 - 007F	Device Profile Specific Static Data Types
0080 - 009F	Device Profile Specific Complex Data Types
00A0 - 0FFF	*reserved*
1000 - 1FFF	Communication Profile Area (e.g. Device Type, Error Register, Number of PDOs supported)
2000 - 5FFF	Manufacturer Specific Profile Area
6000 - 9FFF	Standardised Device Profile Area (e.g. "DSP-401 Device Profile for I/0 Modules" [3]: Read State 8 Input Lines, etc.)
A000 - FFFF	*reserved*

Table 2. General *CANopen* Object Dictionary structure ('Index' in hexadecimal notation).
The terms 'PDO' and 'SDO' denote *CANopen* communication objects, described in the next section.

Trans-Mission Type	Condition to trigger PDO (B=both needed, O=one or both)			PDO Transmission
	SYNC*	RTR*	Event*	
0	B	-	B	Sync, acyclic
1-240	O	-	-	Sync, cyclic
241-251	-	-	-	*reserved*
252	B	B	-	Sync, after RTR
253	-	O	-	Async, after RTR
254	-	O	O	Async, manufacturer specific event
255	-	O	O	Async, device profile specific event

***SYNC**= SYNC-object received.

***RTR** = Remote Transmission Request received (= a '*Remote Frame*' CAN-message).

***Event** = e.g. 'Change-of-Value' or timer-interrupt.

Table 3. Definition of PDO transmission types in *CANopen*; for types 1 to 240 the number indicates the number of SYNC objects between two PDO transmissions.

3.2 CANopen communication

Now that we have presented the concept of the Object Dictionary we now will look at the messages that are communicated in CANopen networks, their content and their function, in other words: the ***CANopen* communication model** (see [2]).

NB: be sure to make the distinction between **OD objects** (objects in the Object Dictionary) – characterized by their OD index and sub-index, as described in the previous section– and **communication objects** or **messages**, characterized by their COB-ID or CAN-identifier, described in this section.

The *CANopen* communication model defines four types of *messages* (**communication objects**):

1. **Administrative message**:
   - Layer management, network management and identifier distribution services: i.e initialisation, configuration and supervision of the network (the latter aspect includes *'node/life guarding*': see below).
   - Services and protocols are according to the **LMT**, **NMT** and **DBT** service elements of ***CAL***. These services are all based on a Master-Slave concept: in a CAN-network there is only one **LMT**-, **NMT**- or **DBT**-master and one or more slaves.

2. **Service Data Object (SDO)**:
   - An SDO provides a client access to entries (objects) of a device OD (the device is the server) using the object's OD index and sub-index, contained in the first few bytes of the CAN-message.
   - An SDO is implemented as a CMS object of type 'Multiplexed Domain' according to *CAL*, thus permitting transfer of data of any length (as data is split up over several CAN messages if necessary, which is when data occupies more than 4 bytes).
     The protocol is of the 'confirmed service' type: a reply is generated for every CAN message (one SDO requires 2 CAN-identifiers). The SDO request and reply message always contain 8 bytes (the number of non-significant bytes is shown as part of the first byte, which carries protocol information), thus communication via an SDO has a considerable overhead.

3. **Process Data Object (PDO)**:
   - Is used to transfer real-time data; data is transferred from one (and only one) producer to one or more consumers. Data transfer is limited to 1 to 8 bytes (for example: one PDO can transfer at maximum 64 digital I/O values, or 4 16-bit analogue inputs).
     It has no protocol overhead. The data content of a PDO is defined through its CAN-identifier only and this content is assumed known to sender as well as receiver(s) of the PDO.
   - Each PDO is described by 2 objects in the Object Dictionary:
     - *PDO Communication Parameter*: contains which COB-ID is used by the PDO, the transmission type, inhibit time and timer period (see below for more details).
     - *PDO Mapping Parameter*: contains a list of objects from the Object Dictionary that are mapped into the PDO, including their size in bits (the producer as well as the consumers of a PDO have to know the mapping to be able to interpret the contents of a PDO).

- Contents of the PDO message is predefined (or is configured at network start-up):

mapping of application objects into a PDO is described in the devices' OD (producer as well as consumer(s)) by the *PDO Mapping Parameter*, and is configurable using SDO messages if *'variable PDO mapping*' is supported by the device(s) (producer and consumers).

- A PDO can have a number of transmission modes:
  - ***synchronous*** (synchronization by receipt of a **SYNC** object, see next message type):
    - **acyclic** (synchronized with respect to **SYNC** message but not periodical):
      - transmission is 'pre-triggered' by a *remote transmission request* from another device,
      - transmission is 'pre-triggered' by the occurrence of an object-(device-)specific event specified in the device profile.
    - **cyclic**: transmission is triggered periodically after every 1, 2 or up to 240 **SYNC** messages.
  - **asynchronous**:
    - transmission is triggered by a *remote transmission request* (by a CAN *Remote Frame*) from another device,
    - transmission is triggered by the occurrence of an object (device) specific event specified in the device profile (e.g. an input-change-of-value or a timer event).

  Table 3 gives an overview of the different PDO transmission modes as defined by the *transmission type*, part of the *PDO Communication Parameter* object and defined as an unsigned 8-bit integer.
- A PDO can be assigned an *inhibit time*, defining the minimum time between two consecutive PDO transmissions, to prevent 'starvation' on the network. *Inhibit time* is a part of the PDO Communication Parameter object and is defined as an unsigned 16-bit integer in units of 100 µs.
- A PDO can be assigned an event timer period where PDO transmission is triggered periodically when a specified time has elapsed (without the occurrence an alternative trigger); it is defined as an unsigned 16-bit integer in units of 1 ms.
- A PDO is implemented as a CMS object of type 'Stored-Event' according to *CAL*, meaning that the data is transferred with no protocol overhead and that the message is not confirmed (one PDO requires one CAN-identifier);
  a maximum of 8 bytes (64 bits) of data can be transferred.

4. **Predefined messages** or **Special Function Objects**:
   - Synchronization (**SYNC**)
     - Used to synchronize tasks network-wide (particularly relevant for drive applications): actual values of inputs are stored quasi-simultaneously network-wide and subsequently transmitted (if required), output values are updated according to messages received after the previous SYNC.
     - Master-slave concept: SYNC master issues the periodic SYNC object, SYNC slaves carry out their synchronous tasks on reception.
     - Transmission of a synchronous PDO is within a given time window with respect to the transmission of the SYNC message.
     - Implemented as a CMS object of type 'Basic Variable'.
     - *CANopen* suggests a COB-ID in the highest priority group to ensure a regular synchronization signal; to keep the message as short as possible no data bytes are transferred.

   - Time Stamp
     - Provides application devices a common time frame reference.
     - Implemented as a CMS object of type 'Stored Event'.

   - Emergency
     - Is triggered by the occurrence of a device internal error.
     - Implemented as a CMS object of type 'Stored Event'.

   - Node/Life Guarding
     - Master-slave concept.
     - The NMT master monitors the state of the nodes: this is called *node guarding*.
     - Nodes optionally monitor the state of the NMT master: this is called *life guarding*; it starts on the NMT slave after it has received the first Node Guard message from the NMT master.
     - Detects errors in the network interfaces of the devices, *not* failures in the device itself: these are reported by means of the *Emergency*.
     - Implemented according to the NMT node guarding protocol:
       a *Remote Transmission Request* from the NMT master to a particular node triggers a reply containing the node's state.

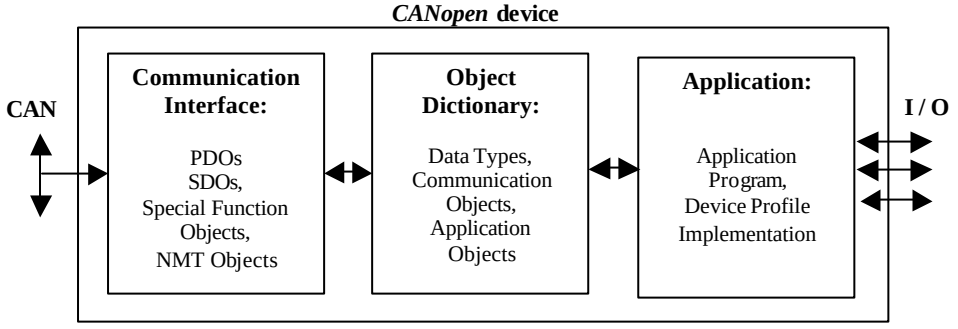

Figure 2. Schematic relationship between CAN-bus communication, Object Dictionary and application software on a *CANopen* device.

- Boot-up
  - Master-slave concept.
  - By sending this message the NMT slave indicates to the NMT master that is has transitioned from state *Initialising* to state *Preoperational* (see section 3.5).

So two of the above-mentioned types of communication objects are meant for data transfer. They implement two different mechanisms of data transfer:

- **SDO** is used for (large,) low-priority data transfer between devices, typically used for configuring the devices on a CANopen network.
- **PDO** is used for fast data transfer of 8 bytes of data or less without protocol overhead (the meaning of the data content has been defined beforehand).

A *CANopen* device has to support a number of the network management services (administrative messages) and needs a minimum of one SDO. Each *CANopen* device that produces or consumes *process data* should have at least one PDO. All other communication objects are optional.

For more details about the various *CANopen* communication objects see [2]. See also section 3.6.

The relation between CAN communication, the Object Dictionary and application software on a *CANopen* device is schematically illustrated in Figure 2.

3.3 CANopen Predefined Connection Set

In order to reduce configuration effort for simple networks a mandatory default CAN-identifier allocation scheme is defined. These identifiers are available in the *Pre-Operational* state (see section 3.5 ***CANopen boot-up***) immediately following initialisation and may be modified by means of dynamic distribution. A device has to provide the corresponding identifiers only for the supported communication objects.

The allocation scheme is based on the division of the 11-bit CAN-identifier into a 4-bit function code part and a 7-bit node-identifier (Node-ID) part as shown in Figure 3.

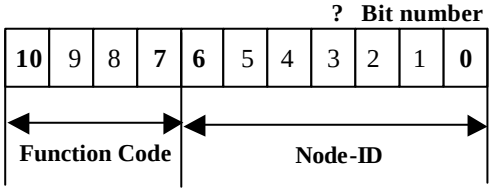

Figure 3. Structure of the 11-bit CAN-identifier in the *CANopen* Predefined Master / Slave Connection Set.

The Node-ID is defined by the system integrator, for example by setting DIP switches on the device. The Node-ID has to be in the range from 1 to 127 (0 (zero) not allowed).

The predefined connection set defines 4 Receive PDOs, 4 Transmit PDOs, 1 SDO (occupying 2 CAN-identifiers), 1 Emergency Object and 1 Node-Error-Control Identifier.

It also supports the broadcasting of non-confirmed NMT-Module-Control services, SYNC- and Time Stamp-objects.

The resulting CAN-identifier allocation scheme is shown in Table 4.

The identifier distribution corresponds to a **master/slave** connection set because all peer-to-peer identifiers are different so that in fact only a master device that knows all connected Node-IDs can communicate to each individual connected slave node (up to 127 nodes) in a peer-to-peer fashion. Two connected slaves would not be able to communicate because they don't know eachother's Node-ID.

Broadcast objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID**	**Communication parameters at OD index**
NMT Module Control	0000	000h	–
SYNC	0001	080h	1005h, 1006h, 1007h
TIME STAMP	0010	100h	1012h, 1013h

Peer-to-Peer objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID** *	**Communication parameters at OD index**
EMERGENCY	0001	081h - 0FFh	1024h, 1015h
PDO 1 (transmit)	0011	181h - 1FFh	1800h
PDO 1 (receive)	0100	201h - 27Fh	1400h
PDO 2 (transmit)	0101	281h - 2FFh	1801h
PDO 2 (receive)	0110	301h - 37Fh	1401h
PDO 3 (transmit)	0111	381h - 3FFh	1802h
PDO 3 (receive)	1000	401h - 47Fh	1402h
PDO 4 (transmit)	1001	481h - 4FFh	1803h
PDO 4 (receive)	1010	501h - 57Fh	1403h
SDO (transmit/server)	1011	581h - 5FFh	1200h
SDO (receive/client)	1100	601h - 67Fh	1200h
NMT Error Control	1110	701h - 77Fh	1016h, 1017h

Table 4. Assignment of CAN-identifiers in the *CANopen* Predefined Master/Slave Connection Set. ("PDO/SDO transmit/receive" is from the (slave) CAN-node point of view). *NMT Error Control* includes *Node Guarding*, *Heartbeat* and *Boot-up* protocols.

Comparing the identifier mapping of the default *CANopen* set in Table 4 to the mapping of *CAL* in Table 1 clearly shows how *CANopen* objects with specific defined functions map to the general CMS objects of *CAL*, illustrating how *CANopen* implements a system using the more general *CAL* facilities.

3.4 CANopen identifier distribution

The allocation of CAN-identifiers (or COB-IDs) in *CANopen* can take place in 3 different ways:

- using the Predefined Master/Slave Connection Set (see previous section):
  allocation of identifiers is default, so no configuration is needed; configuration of PDO data contents (socalled *PDO mapping*) is possible if the node supports it.

- modifying the PDO identifiers after power-up (when the node is in *Pre-Operational* state; see next section), using the (predefined) SDO to write new values to the appropriate locations in the node's Object Dictionary.

- using the *CAL* DBT (DistriBuTor) services: nodes or slaves connected to a *CANopen* network are initially identified by their configured Node-ID. The Node-ID may be configured by setting DIPswitches on the device or by use of *CAL* Layer Management services (LMT).
  When the network initializes and boots the network master initially establishes a dialog with each connected slave by means of a 'Connect_Remote_Node' telegram (a *CAL* NMT service). Once this dialog has been established the CAN identifiers for communication of SDOs and PDOs are allocated to the node using *CAL* Distribution services (DBT); it requires the node to support *extended boot-up* (see next section).

* The COB-ID range covers the allowed *Node-ID* range from 1 to 127.

3.5 CANopen boot-up process

In the network initialisation process *CANopen* supports a socalled **extended boot-up** as well as a socalled **minimum boot-up** process.

Extended boot-up is optional, *minimum boot-up* has to be supported by all *CANopen* devices or nodes; both types of nodes can exist side-by-side on the same network.

A node has to support the *extended boot-up* process if the identifier distribution is by means of *CAL* DBT services (see previous section).

Both initialisation processes can be represented by a device or node state-transition diagram as shown in **Figure 4** for a *minimum boot-up* node. The *extended boot-up* state diagram has a few more states between the *Pre-Operational* and the *Operational* state.

NMT services are used to bring all or selected nodes into various operating states at any time.

The CAN-message carrying this NMT service consists of the CAN-header (with COB-ID=0) and two data bytes, 1 byte containing the requested service ('NMT command specifier') and the second byte the Node-ID, or zero which addresses all nodes simultaneously.

There is one node the CAN-network that acts as the NMT-master. It issues the NMT messages and controls the node initialization process.

CANopen devices supporting only the minimum boot-up are also called *minimum capability* devices. Minimum capability devices enter the *Pre-Operational* state automatically after finishing the device initialization. In this state device parameterization and COB-ID-allocation via SDO is possible.

By switching a device into the *Prepared* state it is forced to stop communication altogether, except NMT services, and node guarding, if supported and active. ('*Stopped*' would be a better name to describe this state).

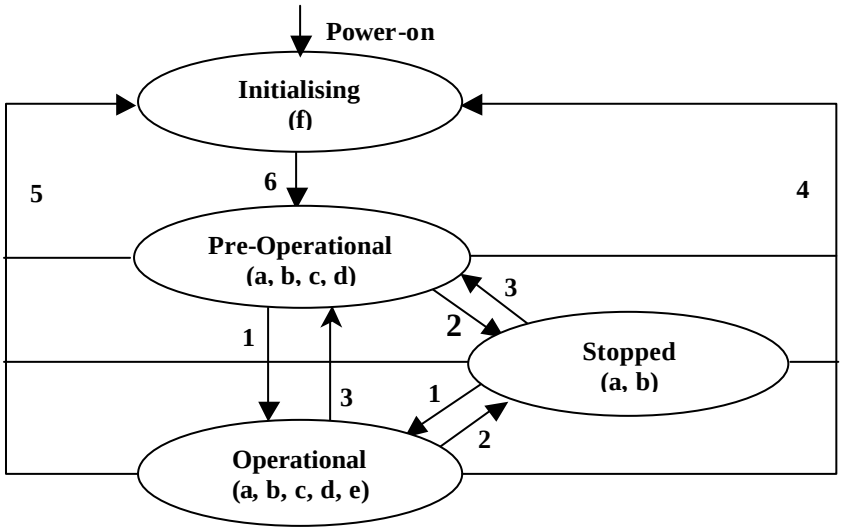

Figure 4. State transition diagram for a *CANopen minimum boot-up* node. The letters in brackets show which communication object types are allowed inside the different states:
a. NMT**, b.** Node Guard**, c.** SDO**, d.** Emergency**, e.** PDO**, f.** Boot-up

State transitions (1-5: initiated by ***NMT*** services) and the NMT command specifier (in brackets):

1: *Start_Remote_Node (0x01)*
2: *Stop_Remote_Node (0x02)*
3: *Enter_Pre-Operational_State (0x80)*
4: *Reset_Node (0x81)*
5: *Reset_Communication (0x82)*
6: Device initialisation finished,
enter *Pre-Operational* state automatically, send *Boot-up* message

3.6 Details of CANopen message syntax

In the following sections the **COB-ID**s used are as defined in the *CANopen Predefined Connection Set*.

3.6.1 NMT Module Control

Only the NMT-Master issues NMT Module Control messages. All slaves must support the NMT Module Control services. There is no response to an NMT Module Control message. The NMT message has the following format:

NMT-Master Þ **NMT-Slave(s)**

COB-ID	Byte 0	Byte 1
0x000	*CS*	Node-ID

With **Node-ID**=0 (zero) all connected NMT slaves are addressed. **CS** is the Command Specifier, which can have the following values (also see Figure 4):

Command Specifier	NMT Service
1	Start Remote Node
2	Stop Remote Node
128	Enter Pre-operational State
129	Reset Node
130	Reset Communication

3.6.2 NMT Node Guarding

Using Node Guarding the NMT-Master can check on the current state of individual nodes, which is especially useful when these nodes are not polled for data on a regular basis.

The NMT-Master sends a CAN *remote frame* (no data bytes) as follows:

NMT-Master Þ **NMT-Slave**

COB-ID
0x700 + Node_ID

The NMT-Slave replies with the following message:

NMT-Master Ü **NMT-Slave**

COB-ID	Byte 0
0x700 + Node_ID	bit 7: *toggle*, bit 6-0: *state*

The data byte contains a **toggle** bit (bit 7) that should alternate between '0' and '1' for every Node Guard request (first time = 0).

Bits 0 to 6 denote the node's **state** which can be one of the following (compare to Figure 4):

Value	State
0	Initialising
1	Disconnected *
2	Connecting *
3	Preparing *
4	Stopped
5	Operational
127	Pre-operational

States marked with ***** are present only in nodes that support *extended boot-up* (see previous section). Note that state 0 never occurs in a Node Guard reply because a node does not respond to Node Guard messages when in this state.

Alternatively a node can be configured to produce a periodical so-called Heartbeat message:

HEARTBEAT Producer Þ **Consumer(s)**

COB-ID	Byte 0
0x700 + Node_ID	*state*

with *state* any of the following:

state	Meaning
0	Boot-up
4	Stopped
5	Operational
127	Pre-operational

When a node with a Heartbeat boots up its *Boot-up* message (see next section) is its first Hearbeat message. The Heartbeat consumer typically is the NMT-master which should have a time-out for each node with a Heartbeat and takes appropriate action if a time-out occurs.

A node is not allowed to support both Node Guarding and Heartbeat protocols at the same time.

3.6.3 NMT Boot-up

An NMT-Slave issues the Boot-up message to indicate to the NMT-Master that it has entered the state *Pre-operational* from state *Initialising*:

NMT-Master Ü **NMT-Slave**

COB-ID	Byte 0
0x700 + Node_ID	0

3.6.4 PDO

As an example lets assume the mapping of the second *transmit-PDO* is as follows (in *CANopen* described by Object Dictionary entry 0x1A01):

Object 0x1A01: 2nd transmit PDO mapping		
Subindex	**Value**	**Meaning**
0	2	2 objects are mapped into the PDO
1	0x60000208	Object 0x6000, subindex 0x02, consisting of 8 bits
2	0x64010110	Object 0x6401, subindex 0x01, consisting of 16 bits

In the definitions of the *CANopen* Device Profile for I/O modules (CiA DSP-401, [3]) Object 0x6000 sub 0x02 is the second set of 8 bits of digital inputs on the node and Object 0x6401 sub 0x01 is the first 16-bit analog input on the node.

This PDO, if it is sent (triggered by a change of input, a timer interrupt, a remote transmission request, etc., in accordance with the PDO's *transmission type* (see Table 3), to be found in Object 0x1801 subindex 2) thus consists of a CAN-message with 3 data bytes and looks like this:

PDO-producer Þ PDO-consumer(s)

COB-ID	**Byte 0**	**Byte 1**	**Byte 2**
0x280 + Node_ID	8-bits digital in	LSB 16-bit analog in	MSB 16-bit analog in

By changing the contents of Object 0x1A01 the contents of the PDO can be changed (if the node supports this (so-called ***variable PDO mapping***)).

Note that in *CANopen* multi-byte parameters are always sent LSB first (so-called ***little endian***).

There can never be more than 8 bytes of data in total mapped to a particular PDO.

In [5] a so-called **multiplexor PDO** (**MPDO**) is defined, enabling a single PDO to be used for transferring a large number of variables by including in its message bytes the source or destination node-id and Object Dictionary index and subindex.
If for example a node has 64 16-bit analog channels it would otherwise need 16 different transmit-PDOs to send its data if such a mechanism would not be available.

3.6.5 SDO

The Service Data Object (SDO) is used to access the Object Dictionary of a device. The requester of the OD access is called the *Client* and the CANopen device, whose OD is accessed and services the request, is called the *Server*. The Client CAN-message as well as the reply Server CAN-message always contain 8 bytes (although not all bytes necessarily contain meaningful data). A Client request is always confirmed by a reply from the Server.
There are 2 mechanisms for SDO transfer:

- **Expedited transfer**: used for data objects up to 4 bytes in length.
- **Segmented transfer**: for objects with length > 4 bytes.

The basic structure of an SDO is:

Client Þ Server / Server Þ Client

Byte 0	**Byte 1-2**	**Byte 3**	**Byte 4-7**
SDO Command Specifier	Object Index	Object Subindex	**

(** up to 4 bytes data (**expedited transfer**)
or a 4 bytes byte-counter (**segmented transfer**)
or parameters regarding **block transfer**
(new SDO transfer mechanism, see below))

or

Client Þ Server / Server Þ Client

Byte 0	**Byte 1-7**
SDO Command Specifier	up to 7 bytes of data (**segmented transfer**)

The *SDO Command Specifier* contains the following information:

- download / upload
- request / response
- segmented / expedited transfer
- number of data bytes in this CAN-frame
- alternating toggle bit for each subsequent segment

There are 5 request/response protocols implemented in SDO: **Initiate Domain Download, Download Domain Segment, Initiate Domain Upload, Upload Domain Segment** and **Abort Domain Transfer.**

In the latest version of the CANopen communication profile a new SDO transfer mechanism is introduced:

- **Block transfer**: in which multiple segments are confirmed by only 1 confirm message (from Server in case of download, from Client in case of upload) in order to increase bus throughput for objects with length > 4 bytes, with associated protocols: **Initiate Block Download, Download Block Segment, End Block Download, Initiate Block Upload, Upload Block Segment** and **End Block Upload.**

'Download' means ***writing*** to the Object Dictionary and **'Upload'** means ***reading*** from the Object Dictionary.

The SDO Command Specifier (first data byte of an SDO CAN-message) syntax and details for each of these protocols is shown in the tables below ("–" stands for: *don't care*, should be zero).

Initiate Domain Download								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	0	0	1	–	**n**		**e**	**s**
Ü Server	0	1	1	–	–	–	–	–

n : valid if e=1 and s=1, otherwise 0; indicates the number of bytes that do not contain data (bytes 8-n to 7 do not contain data).
e : 0 = normal transfer, 1 = expedited transfer.
s : size indicator, 0 = data set size not indicated, 1= data set size indicated.
e=0, **s**=0 : data bytes reserved for further use by CiA
e=0, **s**=1 : data bytes contain byte-counter, byte 4: LSB, byte 7: MSB
e=1 : data bytes contain data to be downloaded.

Download Domain Segment								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	0	0	0	**t**	**n**			**c**
Ü Server	0	0	1	**t**	–	–	–	–

n : indicates the number of bytes that do not contain data (bytes 8-n to 7 do not contain data); zero if no segment size is indicated.
c : 0 = more segments to be downloaded, 1 = last segment.
t : toggle bit, must alternate for each subsequent segment (first time = 0; equal for request / response).

Initiate Domain Upload								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	0	1	0	–	–	–	–	–
Ü Server	0	1	0	–	**n**		**e**	**s**

n, e, s: as for *Initiate Domain Download*.

Upload Domain Segment								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	0	1	1	**t**	–	–	–	–
Ü Server	0	0	0	**t**	**n**			**c**

n, c, t: as for *Download Domain Segment*.

SDO Client or Server can abort an SDO transfer by sending a message with the following SDO Command Specifier:

Abort Domain Transfer								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
C Þ / Ü S	1	0	0	–	–	–	–	–

In case of an *Abort Domain Transfer*, data bytes 0 and 1 contain the Object index, byte 2 the Object sub-index and data bytes 4 to 7 contains a 32-bit abort code describing the reason of the abort.

Table 7 lists a number of abort codes and descriptions as described in [2].

Initiate Block Download								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	1	1	0	–	–	**cc**	**s**	0
Ü Server	1	0	1	–	–	**sc**	–	0

cc : client CRC support on data, 0 = no, 1 = yes.
sc : server CRC support on data, 0 = no, 1 = yes.
s : size indicator, 0 = data set size not indicated, 1= data set size indicated.
s=0 : data bytes reserved for further use by CiA
s=1 : data bytes contain byte-counter, byte 4: LSB, byte 7: MSB

Server byte 4 contains **blksize**, the number of segments per block, with 0<**blksize**<128.

Download Block Segment								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	**c**	0						
Client Þ	**c**	1						
…etc…	**c**	**seqno**						
Ü **Server**	1	0	1	–	–	–	1	0

c : more segments to download, 0=yes, 1=no.
seqno : sequence number of segment, 0<seqno<128.

Client data bytes contain at most 7 bytes of segment data to be downloaded.

Server byte 1 contains the sequence number of the last segment that received successfully; if set to 0 it indicates that the segment with sequence number 1 was not received correctly and all segments have to be retransmitted.

Server byte 2 contains **blksize**, the number of segments per block that the client has to use for the next block download, with 0<**blksize**<128.

End Block Download								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	1	1	0	**n**			–	1
Ü **Server**	1	0	1	–	–	–	–	1

n : indicates the number of bytes in the last segment of the last block that do not contain data (bytes 8-n to 7 do not contain data).

Client bytes 1+2 contain the 16-bit CRC for the whole data set; the CRC is only valid if in *Initiate Block Download* **cc** and **sc** were both set to 1.

Initiate Block Upload								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Client Þ	1	0	1	–	–	**cc**	0	0
Ü **Server**	1	1	0	–	–	**sc**	**s**	0
Client Þ	1	0	1	–	–	–	1	1

cc : client CRC support on data, 0 = no, 1 = yes.
sc : server CRC support on data, 0 = no, 1 = yes.
s : size indicator, 0 = data set size not indicated, 1= data set size indicated.
s=0 : data bytes reserved for further use by CiA
s=1 : data bytes contain byte-counter, byte 4: LSB, byte 7: MSB

Client byte 4 contains **blksize**, the number of segments per block, with 0<**blksize**<128.

Client byte 5 contains **pst**, Protocol Switch Threshold in bytes to change the SDO transfer protocol, 0 = change of transfer protocol not allowed, 1 = if the size of the data in bytes that has to be uploaded is less or equal **pst** the Server can optionally switch to *Upload Domain* protocol by responding with the *Initiate Domain Upload* protocol.

Download Block Segment								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Ü **Server**	**c**	0						
Ü **Server**	**c**	1						
…etc…	**c**	**seqno**						
Client Þ	1	1	0	–	–	–	1	0

c : more segments to upload, 0 = yes, 1 = no.
seqno : sequence number of segment, 0<seqno<128.

Server data bytes contain at most 7 bytes of segment data to be downloaded.

Client byte 1 contains the sequence number of the last segment that received successfully; if set to 0 it indicates that the segment with sequence number 1 was not received correctly and all segments have to be retransmitted.

Client byte 2 contains **blksize**, the number of segments per block that the Server has to use for the next block upload, with 0<**blksize**<128.

End Block Upload								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Ü **Server**	1	1	0	**n**			–	1
Client Þ	1	0	1	–	–	–	–	1

n : indicates the number of bytes in the last segment of the last block that do not contain data (bytes 8-n to 7 do not contain data).

Server bytes 1+2 contain the 16-bit CRC for the whole data set; the CRC is only valid if in *Initiate Block Upload* **cc** and **sc** were both set to 1.

Following below are a few examples to demonstrate the use of the SDO to access a node's Object Dictionary.

With the following SDO messages value 0x3FE is written to Object Dictionary index 0x1801 sub-index 3, of a node with Node-ID=2, using the *Initiate Domain Download* protocol with *expedited transfer* (2 bytes of data):

Client Þ Server (node #2)

COB-ID	Byte						
	0	1	2	3	4	5	6-7
602	2B	01	18	03	FE	03	–

Client Ü Server (node #2)

582	60	01	18	03	–	–	–

With the following SDO messages the same Object Dictionary index 0x1801 sub-index 3 is read back from the node, using the *Initiate Domain Upload* protocol where the server replies with an *expedited transfer* (2 bytes of data):

Client Þ Server (node #2)

COB-ID	Byte						
	0	1	2	3	4	5	6-7
602	40	01	18	03	–	–	–

Client Ü Server (node #2)

582	4B	01	18	03	FE	03	–

3.6.6 Emergency Object

Emergency messages are triggered by the occurrence of a device internal (fatal) error situation and are transmitted from the concerned application device to the other devices with the highest priority. They can be used for interrupt type error alerts.

An Emergency messages consists of 8 bytes and has the following format:

Emergency-sender Þ Emergency-receiver(s)

COB-ID	Byte 0-1	Byte 2	Byte 3-7
0x080 + Node_ID	**Emergency Error Code**	**Error Register** (Object 0x1001)	Manufacturer specific error field

A list of hexadecimal **Emergency Error Code**s is shown in Table 5. The 'xx' part of the codes in this list is defined by the appropriate device profile.

Emergency Error Code	Meaning
00xx	Error Reset or No Error
10xx	Generic Error
20xx	Current
21xx	current, device input side
22xx	current, inside the device
23xx	current, device output side
30xx	Voltage
31xx	mains voltage
32xx	voltage inside the device
33xx	output voltage
40xx	Temperature
41xx	ambient temperature
42xx	device temperature
50xx	Device hardware
60xx	Device software
61xx	internal software
62xx	user software
63xx	data set
70xx	Additional modules
80xx	Monitoring
81xx	Communication
8110	CAN overrun
8120	Error Passive
8130	Life Guard Error or Heartbeat Error
8140	Recovered from Bus-Off
82xx	Protocol Error
8210	PDO not processed due to length error
8220	Length exceeded
90xx	External error
F0xx	Additional functions
FFxx	Device specific

Table 5. Emergency Error Codes (hexadecimal; 'xx' is device-profile dependent part).

The **Error Register** is contained in a device's Object Dictionary (index 0x1001). The device can map internal errors in this status byte, offering a quick overview of errors currently present. The meaning of the bits is shown in Table 6.

The *Manufacturer specific error field* may contain any other device-dependent additional information about the error.

Bit	Error type
0	generic
1	current
2	voltage
3	temperature
4	communication
5	device profile specific
6	*reserved (=0)*
7	manufacturer specific

Table 6. Bit definition of the 8-bit Error Register (Object 0x1001 in the *CANopen* Object Dictionary).

Abort Code	Description
0503 0000	Toggle bit not alternated
0504 0000	SDO protocol timed out
0504 0001	Client/Server command specifier not valid or unknown
0504 0002	Invalid block size (Block Transfer mode only)
0504 0003	Invalid sequence number (Block Transfer mode only)
0503 0004	CRC error (Block Transfer mode only)
0503 0005	Out of memory
0601 0000	Unsupported access to an object
0601 0001	Attempt to read a write-only object
0601 0002	Attempt to write a read-only object
0602 0000	Object does not exist in the Object Dictionary
0604 0041	Object can not be mapped to the PDO
0604 0042	The number and length of the objects to be mapped would exceed PDO length
0604 0043	General parameter incompatibility reason
0604 0047	General internal incompatibility in the device
0606 0000	Object access failed due to a hardware error
0606 0010	Data type does not match, lengh of service parameter does not match
0606 0012	Data type does not match, lengh of service parameter is too high
0606 0013	Data type does not match, lengh of service parameter is too low
0609 0011	Sub-index does not exist
0609 0030	Value range of parameter exceeded (only for write access)
0609 0031	Value of parameter written too high
0609 0032	Value of parameter written too low
0609 0036	Maximum value is less than minimum value
0800 0000	General error
0800 0020	Data can not be transferred or stored to the application
0800 0021	Data can not be transferred or stored to the application because of local control
0800 0022	Data can not be transferred or stored to the application because of the present device state
0800 0023	Object Dictionary dynamic generation fails or no Object Dictionary is present (e.g. OD is generated from file and generation fails because of a file error)

Table 7. SDO *Abort Domain Transfer*: descriptions of hexadecimal abort codes (in byte 4-7).

4 Summary

In summary the *CANopen* standard for communication on CAN-bus based networks has the following features:

- Standardized description of device functionality be means of a Device Object Dictionary.
- Standardized description of a device and its configuration in the form of ASCII files: Electronic Data Sheet (EDS) and Device Configuration File (DCF).
- Data exchange and system administration based on CAL CMS.
- Standardized system boot-up and node guarding based on CAL NMT.
- Definition of system-wide synchronous operations.
- Definition of node-specific emergency messages.

By conforming to the guidelines contained in the *CANopen* communication profile and the appropriate *CANopen* device profile two independent manufacturers can produce standardised devices, which can be incorporated seemlessly into the same *CANopen* CAN network.

The following 3 levels of compatibility can be distinguished (in increasing order of compatibility):

- **Conformance**:
  the device can be connected to a *CANopen* network without disturbing the communication of the other devices: application layer compatibility.
- **Interoperability**:
  the device can exchange data with other nodes in the network: communication profile compatibility.
- **Interchangeability**:
  the device can substitute another one: device profile compatibility.

A *CANopen* device requires at least (*minimum capability device*):

- a node-id,
- an object dictionary (contents depending on the device functionality),
- one SDO supporting the mandatory OD entries (read-only),
- support of the following NMT slave services: *Reset_Node, Enter_Preoperational_State, Start_Remote_Node, Stop_Remote_Node, Reset_Communication*,
- default profile ID-allocation.

5 Example of a *CANopen* Object Dictionary for Devices with CS5525 ADCs

5.1 Introduction

The CS5525 ADC ([4]) is a highly integraded A/D converter containing an instrumentation amplifier, a programmable gain amplifier, a digital filter with programmable output update rate, calibration circuitry and 4 output latch pins which can be used to control an external multiplexer to select any of up to 16 inputs.

In the case of the *CRYSTAL-CAN* ([7]) and SPICAN ([8]) hardware, there even can be 8, 16 or 32 inputs to every ADC and up to 24 ADCs per CAN-node (for a total maximum number of 192 input channels).

The *CANopen Device Profile for I/O Modules* (CiA DSP-401, [3]) is thus the natural starting point for designing an Object Dictionary for our applications. The programmability of several ADC features requires us (if we want to be able to use them) to include some custom objects in the library. This is not a problem since there is room for manufacturer specific extensions in the profile.

The large number of inputs per CAN node makes us want something called 'multiplexed PDOs', found in another device profile, the *CANopen Device Profile for Measuring Devices and Closed-loop Controllers* (CiA DSP-404, [4]). Although this is not an officially approved mechanism, if we would not have it, we would need many different PDOs per node to monitor all analogue inputs in an efficient way (as mentioned earlier).

Note: this mechanism has been superseded by an approved **multiplexor PDO** mechanism, as described in [5].

5.2 ADC Read-out

If the user has the choice of multiple ADCs per node and the number of (multiplexed) channels per ADC is variable it has to be decided how to number all the input channels present on a node. It has been decided to reserve per ADC the maximum number of channels an ADC can have (=32) and only permit to read the valid channels. This means there will be gaps in the numbering of channels from one ADC to the next, but that should not pose a problem. So if a node has three ADCs with 10 multiplexed input channels each they would number 1 to 10, 32 to 42 and 64 to 74.

(Another option would be to introduce one multiplexor per connected ADC and assign a PDO for each ADC; a reason to do something like this could be the fact that all ADCs can in fact perform a conversion simultaneously, so that in principle new data is always available from every individual ADC).

The transmission of the PDO(s) with their contents of ADC value(s) can be triggered in different ways, e.g. at regular intervals using an on-board timer, or after the transmission of a SYNC object or RTR (*Remote Transmission Request*) by the 'master' network-node, or by a combination of both events, as shown earlier. The way a PDO transmission is triggered should either be fixed (and documented in the application documentation) or configurable via the appropriate PDO Communication Parameter object in the node's Object Dictionary (this object contains among other things an 'inhibit time' with which the on-board timer interval could be set).

One should realise that a conversion on this type of ADC can take quite a long time (conversion rate ranges from ca. 220 Hz down to around as low as 4 Hz), so that if a PDO is requested by SYNC or by RTR (*CAN Remote Frame*) and the conversion is started only then, the requester will have to wait for the result for a longer time.

An option is to let the module continuously scan its input channels, store the values locally and when a PDO is requested send the last conversion value of the appropriate input channel immediately. The frequency of scanning the input channels of an ADC could be matched to the above mentioned 'inhibit time'.

NB: in this way the frequency is connected to a PDO, so if only one PDO per node is used all the node's ADCs will be scanned with the same frequency.

It is also possible to initiate a conversion of a particular channel and get the result using an SDO. This is done by reading the appropriate analog input channel from the node's Object Dictionary (using an SDO message), which we will choose to map under OD-index 6404H with 'channel number' as the sub-index. These are the indices according to DSP-401 for reading manufacturer specific analog input (we require only 24-bit: 16-bit (or 20-bit if the CS5526 ADC is used) data plus 4-bit status).
Reading the inputs from the OD by means of SDO, by the way, is a slow way of reading out the ADCs, creating a lot of overhead on the CAN-bus.

5.3 ADC Configuration and Calibration

To enable configuration (e.g. number of inputs, input voltage range, etc) and calibration of individual ADCs we have introduced a number of objects: an *ADC-configuration* object, an *ADC-calibration-configuration* object and an *ADC-reset-and-calibrate* object; one each of these objects should be present for every ADC connected to the node. The objects are specific to our applications so we will place them in the Manufacturer Specific Profile Area (OD-index 2000H to 5FFF).

The *ADC-reset-and-calibrate* objects serve to generate a reset and a calibration sequence on the corresponding ADC, when written to. The ADC-calibration-configuration object determines the parameters of the calibration sequence (see [4]).

The *ADC-configuration* object contains (see [4] for details):

- the number of input channels multiplexed to this ADC
- the offset register contents
- the gain register contents
- the conversion word rate
- the input voltage range
- unipolar or bipolar measurement mode
- power save mode

The *ADC-calibration-configuration* object contains (see [4] for details):

- conversion word rate during calibration
- offset (zero-scale) calibration type (system or self)
- input channel number in case of system offset calibration
- gain (full-scale) calibration type (system oself)
- input channel number in case of system gain calibration
- offset (offset register content after calibration)
- gain (gain register content after calibration)

In some applications with CS5525 ADCs, these objects might be completely predefined and unchangeable; therefore they need not be readable OD-objects on the node; it would be sufficient when the application documentation lists the objects and their values.

5.4 The Object Dictionary

The following tables show in detail an Object Dictionary (OD) for a device with multiple CS5525 ADCs each with multiple analogue input channels.

Note that the OD described here is shown purely as an example and has not been implemented as such.

There are three tables: one for each of the following OD parts: a **communication**, a **device-profile** and a **device-specific** (manufacturer-specific) part.

The column '**Attr**' shows the access rights attribute of an object: RO=read-only (value can change), RW=read-or-write, WO=write-only.

This OD is based on the CiA device profile for I/O modules DSP-401 ([3]) and borrows some stuff from DSP-404 ([4], superseded now…). The analogue inputs are mapped onto the second transmit PDO as in DSP-401.

Communication Profile Area						
Index (hex)	**Sub Index**	**Name**	**Data/ Object**	**Attr**	**Default (hex)**	**Comment**
1000	-	Device type	U32	RO	00040191	Meaning: DSP-401, analogue inputs on device
1001	-	Error register	U8	RO	0	Error bits according to DS-301 (error status overview)
1002	-	Manufacturer status reg *	U32	RO	0	ADC errors/timeouts, etc.
1003		Predefined error field	Array			Contains list of recent errors
1004		#PDOs supported	Array			
	0	Total #PDOs supported	U32	RO	00000001	0 receive, 1 transmit PDO
	1	#PDOs sync	U32	RO	00000000	PDO after SYNC
	2	#PDOs async	U32	RO	00000001	PDO after RTR or 'event'
1008	-	Manufacturer device name	VisStr	RO	"xxxx"	E.g. "CCTS" (Crystal-Can with Temperature Sensors"
1009	-	Manufacturer hardware version	VisStr	RO	"xxxx"	4-byte ASCII string
100A	-	Manufacturer software version	VisStr	RO	"xx.x"	A version number as a 4-byte ASCII string, e.g, " 1.0"
100B	-	Node identifier	U32	RO		
100F	-	#SDOs supported	U32	RO	00000001	0 client, 1 server SDO
1801		2nd transmit PDO parameters	Record			Data type = PDOCommPar
	0	Number of entries	U8	RO	3	
	1	COB-ID used by PDO	U32	RO	280+ Node-ID	According to CANopen Predefined Connection Set
	2	Transmission type ♣	U8	RW	FD	253 decimal
	3	Inhibit time ♣ (in units of 100 µs)	U16	RW	1000	If >0 node scans inputs with corresponding frequency (per ADC) Limitation: 0.2 Hz <= frequency <= 25 Hz (50000 >= inhibit time >= 400)
1A01		2nd transmit PDO mapping	Record			Data type = PDOMapping
	0	Number of entries	U8	RO	2	
	1	Multiplexor 1	U32	RO	6F100108	OD-index 6F10, sub-index 1: Multiplexor 1 (see DSP-404); Size = 8 bits
	2	24-bit analogue input	U32	RO	6404FD18	OD-index 6404, sub-index 253: Analogue input, via multiplexor; Size = 24 bits

Table 8. Communication Profile Area of the CANopen Object Dictionary for a device with CS5525 ADCs.

* See text for the layout of the Manufacturer Status Register.

♣ if *inhibit time* = 0: transmission type 254,253 => one ADC conversion and PDO transmission after an RTR
transmission type 1 => one ADC conversion and PDO transmission after a SYNC
if *inhibit time* > 0: transmission type 254 => scan ADC(s), a PDO transmission after every conversion
transmission type 253 => scan ADC(s), one PDO transmission after an RTR
transmission type 1 => scan ADC(s), one PDO transmission after each SYNC

The status of every CS5525 ADC channel can be read from the conversion status, which is part of the channel read-out value. Other types of errors we log in the Manufacturer Status Register (Object Dictionary index 0x1002), a 32-bit object, thus providing 4 bits per ADC if we allow a maximum of 8 ADCs per node:

Bits	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
ADC	#7	#6	#5	#4	#3	#2	#1	#0

We propose the following definition of bits:

Bit 3	**Bit 2**	**Bit 1**	**Bit 0**
(not used)	Calibration error: - error during calibration procedure	Conversion error: - timeout waiting for conversion-ready	Reset error: - reset bit not set and/or - error in default register contents

In case digital inputs are present objects 1800H (1st transmit PDO communication parameters) and 1A00H (1st transmit PDO mapping) should be added to the communication part of the Object Dictionary.

In case digital outputs are present objects 1400H (1st receive PDO communication parameters) and 1600H (1st receive PDO mapping) should be added to the communication part of the Object Dictionary; all according to DSP-401.

Possible extensions to the device profile part of the OD could be made –for example– in connection with analogue input limit and interrupt settings, for which the DSP-401 device profile provides several OD-entries. When these capabilities are present the node can monitor inputs and only send a message e.g. when certain limits are exceeded or when an input changes more rapidly then a certain set limit.

Standardised Device Profile Area						
Index (hex)	**Sub Index**	**Name**	**Data/ Object**	**Attr**	**Default (hex)**	**Comment**
6404		Read analogue input Manufacturer-specific	Record			
	0	Number of entries	U8	RO		Variable, depending on hardware Configuration
	1	Input 1	I24	RO		1st analog input (24-bit)
	2	Input 2	I24	RO		2nd " " "
	…	…	…	…		…
	192	Input 192	I24	RO		192th " " "
	252	Multiplexor number	U8	RO	1	Defines which *mux* in the OD is used; but in this profile we won't define the *mux* itself (DSP-404)
	253	Input via multiplexor	I24	RO		Read input #<mux1> (DSP-404)

Table 9. Standardised Device Profile Area of the *CANopen* Object Dictionary for a device with CS5525 ADCs (providing 192 input channels, sufficient for six 32-channel ADCs).

Manufacturer-specific Profile Area						
Index (hex)	**Sub Index**	**Name**	**Data/ Object**	**Attr**	**Default (hex)**	**Comment**
2A00		ADC-configuration [1] ADC#0	Record			
	0	Number of entries	U8	RO	7	
	1	Number of input channels	U8	RW		In the range [0,32]
	2	Conversion Word Rate	U8	RW	0	3-bit code [2]
	3	Input Voltage Range	U8	RW	0	3-bit code [3]
	4	Unipolar/Bipolar Measurement Mode	U8	RW	0	0 = bipolar, 1 = unipolar
	5	Power Save Mode	U8	RW	0	1 = power save
	6	Offset Register	U32	RW		CS5525 Offset Register
	7	Gain Register	U32	RW		CS5525 Gain Register
2A01		ADC-configuration ADC#1	Record			
…		…	…			
2A07		ADC-configuration ADC#7	Record			Max. 8 ADCs can be connected to CRYSTAL-CAN / SPICAN
2B00		ADC-calibration-configuration ADC#0	Record			
	0	Number of entries	U8	RO	7	
	1	Conversion word rate during calibration	U8	RO	0	3-bit code [2] (always set to 15.02 Hz)
	2	Offset calibration type	U8	RW		3-bit code [4]
	3	Offset calib input channel	U8	RW		In the range [0,31]
	4	Gain calibration type	U8	RW		3-bit code [4]
	5	Gain calib input channel	U8	RW		In the range [0,31]
	6	Offset value	U32	RO		24-bits significant
	7	Gain value	U32	RO		24-bits significant
2B01		ADC-calibration-configuration ADC#1	Record			
…		…	…			
2B07		ADC-calibration-configuration ADC#7	Record			Max. 8 ADCs can be connected to CRYSTAL-CAN / SPICAN
2C00	-	ADC-reset-and-calibrate [1]	U8	WO	*n*	Reset ADC#*n* (0<=n<=7) and perform a calibration sequence
2D00	-	ADC-reset [1]	U8	WO	*n*	Reset ADC#*n* (0<=n<=7)

Table 10. Manufacturer-specific Profile Area of the *CANopen* Object Dictionary for a device with CS5525-ADCs.

[1] write access allowed **only** when ADC-input scanning not active (PDO *inhibit time* = 0)

[2] **000**: 15.02 Hz, **001**: 30.06 Hz, **010**: 60.01 Hz, **011**: 123.18 Hz,
100: 168.9 Hz, **101**: 202.27 Hz, **110**: 3.76 Hz, **111**: 7.51 Hz

[3] **000**: 100 mV, **001**: 55 mV, **010**: 25 mV, **011**: 1 V, **100**: 5 V

[4] **001**: offset self-calibration, **010**: gain self-calibration,
101: offset system-calibration, **110**: gain system-calibration

5.5 Emergency Objects

Table 11 lists the contents of the Emergency Object message for different types of internal device errors. This is a preliminary list of the error messages defined and implemented sofar.

The *Emergency Error Codes* are defined by the Communication Profile [2] and the DSP-401 Device Profile [3]; the *Error Register* bits are defined by DSP-401 [3]. See section 3.6.6.

Error Description	Emergency Error Code (byte 0-1)	Error Register (Object 1001H) (byte 2)	Manufacturer-specific Error Field (byte 3-7)
Watchdog reset	0x6000	0x01	Byte 3,4,5,6: Manufacturer Device Name (Object Dictionary index 0x1008) Byte 7: 0
CAN-controller overrun: message lost	0x8100	0x10	Byte 3: 1 Byte 4: counter (modulo 256) Byte 5: CANSTA (CAN-controller status register) Byte 6,7: 0
CAN-controller error: communication error	0x8100	0x10	Byte 3: 2 Byte 4: counter (modulo 256) Byte 5: CANSTA (CAN-controller status register) Byte 6,7: 0
Local CAN message buffer overflow: message lost	0x8100	0x10	Byte 3: 3 Byte 4: counter (modulo 256) Byte 5: CANSTA (CAN-controller status register) Byte 6,7: 0
ADC: conversion timeout	0xFF00	0x80	Byte 3: 1 Byte 4: ADC number (0..7) Byte 5,6,7: 0
ADC: reset failed	0xFF00	0x80	Byte 3: 2 Byte 4: ADC number (0..7) Byte 5,6,7: 0
ADC: offset calibration failed	0xFF00	0x80	Byte 3: 3 Byte 4: ADC number (0..7) Byte 5,6,7: 0
ADC: gain calibration failed	0xFF00	0x80	Byte 3: 4 Byte 4: ADC number (0..7) Byte 5,6,7: 0

Table 11. Emergency Objects for a CS5525-ADC *CANopen* device.

References

[1] CAN-in-Automation,
CAL, CAN Application Layer for Industrial Applications,
CiA Draft Standard DS-201 to DS-207, Version 1.1, Feb 1996.

[2] CAN-in-Automation,
CANopen, CAL-based Communication Profile for Industrial Systems,
CiA DS-301, Version 4.0, June 16 1999.

[3] CAN-in-Automation,
CANopen Device Profile for I/O Modules,
CiA DSP-401, Version 1.4, Dec 1996.

[4] CAN-in-Automation,
CANopen Device Profile for Measuring Devices and Closed-Loop Controllers,
CiA DSP-404, Revision 1.11, Nov 27 1997.

[5] CAN-in-Automation,
Framework for Programmable CANopen Devices,
CiA DSP-302, Version 2.0, Nov 27 1998.

[6] **CS5525/CS5526 16-bit / 20-bit multi-range ADC with 4-bit latch**,
data sheet, Crystal Semiconductor Corporation, Sep 1996.

[7] H.Boterenbrood,
Crystal-CAN, a CAN-bus node for monitoring multiple distributed analog signals
(prototypes for B-field and Temperature monitoring in ATLAS),
Version 1.4, NIKHEF internal documentation, Oct 24 1997

[8] H.Boterenbrood,
SPICAN CANopen I/O-system (for analog inputs), *user documentation*,
Version 2.1, NIKHEF internal documentation, Jan 14 2000