Project Acronym: MEDIS

Project Title: A Methodology for the Formation of Highly Qualified Engineers at Masters Level in the Design and Development of Advanced Industrial Informatics Systems

Contract Number: 544490-TEMPUS-1-2013-1-ES-TEMPUS-JPCR

Deliverable Number: 2.4.6

Title of the Deliverable: AIISM teaching resources - Industrial Computers

Task/WP related to the Deliverable: AIISM teaching resources - Industrial Networks and Fieldbuses – CAN-Open

Type (Internal or Restricted or Public): Internal

Author(s): Mário de Sousa

Partner(s) Contributing: FEUP

Contractual Date of Delivery to the CEC: 30/09/2014

Actual Date of Delivery to the CEC: 30/09/2014

Project Co-ordinator

Company name :	Universitat Politècnica de València (UPV)
Name of representative :	Houcine Hassan
Address :	Camino de Vera, s/n. 46022 - Valencia (Spain)
Phone number :	+34 96 387 7578
Fax number :	+34 96 387 7579
E-mail :	husein@disca.upv.es
Project WEB site address :	http://medis.upv.es/

Context

WP 2	Design of the AIISM-PBL methodology
WPLeader	Universitat Politècnica deValència (UPV)
Task 2.4	Development of the AIISM teaching resources - – Industrial Networks and Fieldbuses – CAN-Open
Task Leader	UP
Dependencies	UPV, MDU, TUSofia, USTUTT, UP

Author(s)	Mário de Sousa
Contributor(s)	
Reviewers	

History

Version	Date	Author	Comments
0.1	22-05-14	Mario Sousa	Initial draft
0.2	19-09-14	Mario Sousa	Revised labs and mini-projects

Table of Contents

1 Executive summary

WP 2.4 details the learning materials of the Advanced Industrial Informatics Specialization Modules (AIISM) related to the Industrial Networks and Fieldbuses.

The contents of this package follows the guidelines presented in the Partner's documentation of the WP 1 (Industrial Networks and Fieldbuses)

1. The PBL methodology was presented in WP 1.1
2. The list of the module's chapters and the temporal scheduling in WP 1.2
3. The required human and material resources in WP 1.3
4. The evaluation in WP 1.4

During the development of this WP a separate document has been created for each of the chapters of the Industrial Networks and Fieldbuses Module (list of chapters in WP1.1). This document is for the sixth chapter – CAN-Open.

In this document, section 2 defines the lecture, sections 3-5 describe the laboratory work for weeks 11-13, sections 6-8 explain the seminar topics for weeks 11-13, and sections 9-11 define the requirements to fulfil for the mini-project during weeks 11-13. Section 12 lists the bibliography and the references.

2 Lecture

2.1 Introduction to CAN Open

As was already said, CAN Open is an application layer protocol that works over CAN. Similarly to the Modbus protocol (already covered in previous chapters), CAN Open specifies a set of data that each CAN Open node must maintain locally, and that may be read from (or sometimes written too) over the CAN network. This is known as the CAN Open Object Dictionary. It is called a *Dictionary* because it represents the data stored locally at each node, and that may be looked up over the network. It is called an *Object* Dictionary, because the data in the dictionary is organised as data structures (or objects), with each object capable of maintaining several data sub-entries. This is unlike the Modbus data model, where all data is simply organised as arrays of boolean or word (16 bit) variables.

CAN Open specifies a large set of dictionary entries, as well as a large set of interaction protocols. Nevertheless, most of the functionality and dictionary entries are optional, and a large number of CAN Open devices limit themselves to implementing very little besides the limited mandatory features.

In Modbus each manufacturer is free to decide where to place the application specific data inside the (1 bit or 16 bit) data arrays, and a client must know the mapping of each Modbus server node. This has the drawback that it is not possible to exchange one failed server for another server of a competing manufacturer, unless the source code in the Modbus client that reads the data off the server is also changed accordingly. CAN Open solves this problem by expecting each CAN Open device to follow one of several device profiles. A device profile will define where in the Object Dictionary the application data must be stored, as well as some related communication configuration parameters. One device profile exists for each type of device (e.g. motor speed drive, simple Input/Output, closed loop controllers, encoders, etc.). As long as all communication with a node of a specific device profile is limited to the capabilities

defined in the device profile itself, it should be possible (from a communication network point of view) to exchange a node device to one of another manufacturer or model that follows the same device profile.

Each node in a CAN Open network has a unique 7 bit identifier, called the Node ID, which may therefore range from 1 to 127. This may seem contradictory as CAN is based on unique identifiers for each data message (with each node capable of transmitting several distinctly identified data messages) and does not therefore support the use of node identifiers. Nevertheless, CAN Open implements the Node IDs by simply mandating that every message sent by a specific node be integrated must have the 7 bit Node ID of the transmitting node inserted into the 11 bit CAN message ID. This leaves 4 free bits in the message ID, which implies that each CAN Open node may send up to 16 distinct message IDs. Each of these 16 possible messages that each CAN Open node may transmit will each have a special function within the CAN Open protocol, which will be explained further on in this chapter.

Unlike CAN that works based on the producer/subscriber communication model, CAN Open networks work both in the client/server and the producer/consumer models. In a CAN Open network that is fully configured some nodes will automatically publish some objects in their object dictionary when the data changes(the publish/subscribe). However, these same nodes will also be listening on the network for requests coming from clients, and will therefore also reply to those requests (the client/server model).

2.2 The Object Dictionary

All the data in the Object Dictionary (OD) may be accessible over the CAN Open network. This dictionary, besides containing the data related to the application specific (industrial) process under control, also contains data used for the configuration of the node itself, as well as a description of the node. The configuration data means that the node may be configured remotely (over the network). The functionality description data means that the node may be automatically detected and discovered by other nodes on the network, and its functionality and capabilities is made available to all interested parties also over the network itself.In other words, when a new node is connected to a CAN Open network, the other nodes can determine what this new node does, and how it operates.

A very simple example of a CAN Open node would be a water level sensor in a water tank. The node will determine the water level using a physically connected water level sensor, and store that data in its local OD. By reading from the object dictionary of this node, other nodes on the CAN Open network may obtain the current water level. Another example would be a node connected to a valve actuator, that physically moves the valve's position to whatever value is stored in a specific position of its local OD. By writing to this position of this node's OD, other nodes on the network can change the position of the valve.

The object dictionary may be viewed as a sequence or array of objects, with each object being identified by its 16 bit Index. This results in a maximum of 65536 entries. Each object (or entry) in the object dictionary may be viewed as a structured variable, with up to 256 values usually known as sub-entries. Each sub-entry is identified by its Subindex, an 8 bit value. Object Indexes and Subentry Indexes are commonly written in hexadecimal form, and for that reason we shall follow that convention too, appending 'h' to hexadecimal values to clearly distinguish them from decimal values (e.g. 22 will be written 16h in hexadecimal).

Every object has at least one element. For objects with only one element, this is stored in one Subentry with the index 00h. However, for objects with two or more elements, the first Subentries at Subindex 00h is used to store the number of elements in the object, and the elements themselves will be stored in the Subentries starting off at Subindex 01h. For example,

an Object in the OD with 4 data elements will use up 5 Subentries. The first Subentry at Subindex 00h will contain the value 4, while the data elements will be stored in the Subindexes 01h to 04h.

When an Object in the OD has only one element (and therefore one Subindex), it is common to refer to the object merely by its Object Index – the fact that no Subindex is mentioned implies a single Subentry.

Each Subentry can be of one of the several elementary datatypes defined by CAN Open: a 1 bit boolean (BOOLEAN), signed or unsigned integer with a bit length of 8, 16, 24, 32, 40, 48, 56, 64 (INTERGER8, ..., INTERGER64, UNSIGNED8, ... UNSIGNED64), a 32 or 64 bit real (REAL32, REAL64), an ASCII character string (VISIBLE_STRING), an array of 8 or 16 bit unsigned integers (OCTET_STRING, UNICODE_STRING), a 48 bit value for time (TIME_DIFFERENCE), or a block of data (DOMAIN).

Each Object in the Object dictionary with more than one element is equivalent to a data structure in C. Each Subentry of the Object may be of a distinct elementary data-type. For example, an Object in the OD representing the state of a battery has one entry for the voltage, another for the current, and yet another for the state of charge. The voltage and charge are stored in UNSIGNED8, whereas the current is stored as SIGNED8 to distinguish between charging (positive) and discharging (negative) currents.

The list of the data-types of each element is considered a new Complex data-type. Each Object with more than one element must be of a specific pre-defined complex data-type. The CAN Open standard pre-defines 4 complex datatypes, but each manufacturer is free to define other complex datatypes.

The definition of the datatypes is itself stored in the Object Dictionary. Entries from 0001h to 000Fh define the elementary datatypes. These Object Dictionary entries have a single Subentry of UNSIGNED8 data-type, where the size in bits of the respective data-type is stored. For example the Index 0003h, corresponding to the INTERGER16 data-type, will store the value 16.

The 4 pre-defined complex datatypes are defined in the Indexes 0020h to 0023h. Other complex data-types may be defined in the Indexes 0040h to 025Fh (table 2). Each Object entry in the OD defines one complex data-type, and consists of one Subentry for each Subentry of the complex data-type. As usual, the first Subentry (00h) stores the number of Subentries in the Object, whereas the remaining Subentries will store the data-type of each Subentry. For example, the data-type of the Object used to describe the state of the battery consisting of two UNSIGNED8 followed by a SIGNED8, would be described in an Object at Index 0040h (corresponding to a Manufacturer Complex Data-type) with Subentry 00h set to 3 (3 elements in the Object), Subentries 01h and 02h set to 5 (corresponding to UNSIGNED8), and Subentry 03h set to 2 (corresponding to INTERGER8).

2.3 Structure of the Object Dictionary

The Object Dictionary is divided into 4 main areas:

Index	Description
0000h	Reserved
0001h – 0FFFh	Data-Types
1000h – 1FFFh	Communication Entries
2000h – 5FFFh	Manufacturer Specific

Index	Description
6000h - 9FFFh	Device Profile Parameters
A000h - FFFFh	Reserved

The Manufacturer Specific entries are at the disposal of the device manufacturer to use as it desires. No limits are imposed on these objects, other than the fact that they must follow the standard Object Dictionary rules of object formatting. Although the manufacturer is free to place application specific data in this location, for compatibility purposes it is advisable that it be placed in the location specified by the device profile it follows. Each device profile specifies a distinct mapping for the Indexes in the range 6000h-9FFFh.

The area reserved for data-type definition is sub-divided into:

Index	Description
0001h - 001Fh	Standard Data-Types
0020h – 0023h	Pre-Defined Complex Data-Types
0024h – 003Fh	Reserved
0040h – 005Fh	Manufacturer Complex Data-Types
0060H - 007Fh	Device Profile Standard Data-Types
0080h - 009Fh	Device Profile Complex Data-Types
00A0h - 025Fh	Multiple Device Modules Data-Types
0260h - 0FFFh	Reserved

The Communication Entries describe the device itself, its capabilities, as well as its current configuration. By writing to some of these Objects, it becomes possible to remotely configure the device itself, and how the device will behave in the CAN Open network.

Index	Size (bits)	Description
1000h	32	Device Type
1001h	8	Error Register
1002h		Manufacturer Status Register
1003h		Pre-Defined Error Field
1004h		
1005h		COB ID SYNC
1006h		Communication Cycle Period
1007h		Synchronous Window Length
1008h		Manufacturer Device Name
1009h		Manufacturer Hardware Version
100Ah		Manufacturer Software Version
100Bh		

Index	Size (bits)	Description
100Ch	16	Guard Time
100Dh	8	Life Time Factor
100Eh		
100Fh		
1010h		Store Parameters
1011h		Restore Default Parameters
1012h		COB ID Time
1013h		High Resolution Time Stamp
1014h		COB ID EMCY
1015h		Inhibit Time EMCY
1016h		Consumer Heartbeat Time
1017h		Producer Heartbeat Time
1018h		Identity Object
1200h - 127Fh		Server SDO Parameters
1280h - 12FFh		Client SDO Parameters
1400h - 15FFh		RxPDO Communication Parameters
1600h - 17FFh		RxPDO Mapping Parameters
1800h - 19FFh		TxPDO Communication Parameters
1A00h - 1BFFh		TxPDO Mapping Parameters

2.4 EDS and DCF files

When configuring the devices that are to be connected to a CAN Open network, manufacturer specific (and proprietary) software applications are often used. For example, a PLC controller needs to communicate over CAN Open to obtain the current temperature in a tank. To do this, it is necessary to configure the PLC using the PLC manufacturer's proprietary configuration software. However, the PLC will be communicating with a CAN Open device (that reads the temperature) from a different manufacturer. It is therefore understandable that the PLC configuration software does not come prepared with the details of the temperature reading device, but will somehow need to import these details in order to configure the PLC appropriately. In order to allow the easy exchange of CAN Open related configuration data regarding a CAN Open device, the CAN Open standard defines a standard way of describing this configuration data in EDS (Electronic DataSheet) or DCF (Device configuration File) files.

Another tool that can make use of EDS files are protocol analysers. These tools listen to all the traffic on a CAN Open bus, and interpret the messages to present them in a user friendly manner.

Basically, an EDS file describes which object dictionary entries a specific CAN Open device supports. It is simply a description of the object dictionary in an electronically readable format. The file itself is a text file that follows a format similar to the windows '.ini' files, which means that it may be edited and changed using a standard text file editor.

An excerpt from an EDS file follows. Note that this file was cut in the locations indicated as '(cut)', and that these lines do not belong in the EDS file itself.

[FileInfo]

CreatedBy=ROM

ModifiedBy=ROM

CreationTime=09:42AM

CreationDate=06-06-2014

ModificationTime=09:42AM

ModificationDate=06-06-2014

FileName=XXXXXXXXXXXXX.eds

FileVersion=1

FileRevision=0

EDSVersion=4.0

[DeviceInfo]

VendorName=SD

ProductName=MIxx

ProductNumber=1

RevisionNumber=0

BaudRate_10=0

BaudRate_20=0

BaudRate_50=1

BaudRate_125=1

BaudRate_250=1

BaudRate_500=1

BaudRate_800=0

BaudRate_1000=1

SimpleBootUpMaster=0

SimpleBootUpSlave=1

Granularity=0

DynamicChannelsSupported=0

CompactPDO=0

GroupMessaging=0

NrOfRXPDO=1

NrOfTXPDO=2
LSS_Supported=0

[MandatoryObjects]
SupportedObjects=2
1=0x1000
2=0x1001

--------- (cut) ----------

[1018]
ParameterName=Identity object
ObjectType=0x8
SubNumber=5

[1018sub0]
ParameterName=Highest sub-index supported
ObjectType=0x7
DataType=0x0005
AccessType=ro
DefaultValue=4
PDOMapping=0

[1018sub1]
ParameterName=Vendor-ID
ObjectType=0x7
DataType=0x0007
AccessType=ro
DefaultValue=19
PDOMapping=0

[1018sub2]
ParameterName=Product code
ObjectType=0x7
DataType=0x0007

AccessType=ro

DefaultValue=0

PDOMapping=0

[1018sub3]

ParameterName=Revision number

ObjectType=0x7

DataType=0x0007

AccessType=ro

DefaultValue=0

PDOMapping=0

[1018sub4]

ParameterName=Serial number

ObjectType=0x7

DataType=0x0007

AccessType=ro

DefaultValue=0

PDOMapping=0

Editing an EDS file can also be achieved using a specific EDS editor. These editors simplify the task of editing EDS files by using menus and forms to interact with the user, and generate syntactically correct EDS files. Typically these editors are capable of inserting standard object dictionary entries with a click of the mouse.

Once an EDS file has been edited by hand, it is advisable to check whether it follows the correct syntax. This can be done by using the free software EDSchecker, available for download from http://www.can-cia.org.

DCF (Device Configuration Files) files follow a syntax very similar to that of EDS files. Although the data contained in each file (a description of an Object Dictionary) is similar, EDS and DCF files have celar and different objectives. An EDS file describes the object dictionary entries of all devices of a specific model/make. A DCF file describes the configuration of a specific instance/device (eventually of the same model/make).

An EDS file, besides the object entries, may contain the possible values (minimum, maximum) an object entry may contain. On the other hand, a DCF file will contain the value of each object entry that a specific device contains or should contain. A PLC configuration tool can use an EDS file to become acquainted with the object dictionary entries in a device, and can store the values that should be configured in each object entry in a specific node inside a DCF file.

A device configuration tool may upload (over the CAN Open network) the configuration of a device on the network, and store the current state of that device in a DCF file. This state, if necessary, may later be restored by downloading to the device the data in the same DCF file.

2.5 Accessing Objects in the OD using SDO

To read and write to objects in the object dictionary, the SDO (Service Data Objects) protocol is used. This protocol works using the client/server interaction model, where the client requests to read from or to write to a specific object in the object dictionary of the server. Note that both the client and the server are both CAN Open devices.

The SDO request sent by the client is sent using a CAN message with a CAN message ID that consists of the server's CAN Open node ID (7 bits), plus the fixed value 600h. For example: a client that wishes to send an SDO request to the can Open node with ID 04h will send a CAN message with a CAN message ID of 604h.

Likewise, the SDO reply sent by the server is sent using the server's CAN Open node ID, added to the fixed value 580h. In the previous example, the server with ID 04h would reply with a CAN message ID of 584h.

Basically, one says that the SDO(rx) messages, those sent by the client to be received by the servers, use the function code Ch, which may also be written as 1100b (in binary - function codes are a 4 bit number). This function code is sent as the most significant bits of the CAN message ID. Since CAN message IDs use only 11 bits, this function code will occupy bits 8 to 11 of the CAN message ID.

11	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	→ Bit nº
1	1	0	0									→ Function code (Ch)
				0	0	0	0	0	1	0	0	→ CAN Open node ID (04h)
1	1	0	0	0	0	0	0	0	1	0	0	→ resulting CAN message ID (604h)

Similarly, the SDO(tx) messages, those sent by the servers back to the client, uses the function code Bh, corresponding to 1011b.

In CAN Open, the CAN message ID resulting from the addition of the function code to the CAN Open node ID is called the COB-ID (Communication Object Identifier). As mentioned previously, this is nothing other than the ID of the CAN message, nevertheless, we will from now on use the CAN Open nomenclature for this identifier.

Note that the above scheme means that only one client can exist on a CAN Open network. This is a consequence of the medium access control algorithm used in the underlying CAN bus that assumes that no two nodes may send messages with the exact same CAN message ID. Since all client requests to a CAN Open server node are sent using the same CAN message ID, two clients on the same network is simply not supported. Due to this limitation, the client/server interaction model reduces to a master/slave model (similar to a client/server, but with a single client). We will therefore from now call the client a master node, and the servers slave nodes.

The are three possible modes of data transfer using the SDO protocol: expedited transfer, segmented transfer, and block transfer. The expedited transfer is used when transferring objects that are 4 bytes or less in length, which means that they will fit inside a single CAN message (the other 4 bytes will be used for the SDO message headers). The segmented transfer is used for larger objects, whose data needs to be broken up into several segments of 8 bytes each, before being transmitted over the CAN bus. Block transfer has the same objective as segmented transfers (i.e. send larger sized objects) in smaller pieces, but it is more optimized than

segmented transfer, and only implemented in more recent CAN Open devices. We will therefore limit ourselves to the expedited and segmented transfer modes.

A client sending a request to write to an object on a server will send a CAN message with a COB-ID determined using by adding 600h to the CAN Open node ID, containing a data payload of 8 bytes, organised as follows:

Byte 0					Byte 1..3	Byte 4..7
Bit 7..5	Bit 4	Bit 3..2	Bit 1	Bit 0		
ccs = 1	x = 0	n	e	s	Index/ Subindex	Expedited Data

ccs – Client command specifier. Always set to 1.

e – expedited transfer = 1 (=> data in bytes 4..7); segmented transfer = 0

s – set to 1 if data size is indicated

n – if e=1, and s=1, the number of data bytes (4..7) that do not contain data

x – reserved

Byte 1..3 contains the Index and Subindex of the object in the object dictionary of the server that will be written to.

If the expedited data bit is set, bytes 4 to 7 will contain the data to be written to the indicated object. The n will indicate how many of these bytes are real data, and how many are merely padding to fill up the packet. If the expedited bit is not set, then the data bytes will not contain nay data, and should be ignored by the server.

Independently of whether the expedited bit is set or not, the server will reply with a CAN message with the following contents, and using a COB-ID of 580h added to the server's CAN Open node ID:

Byte 0		Byte 1..3	Byte 4..7
Bit 7..5	Bit 4..0		
ccs = 3	x = 0	Index/ Subindex	Reserved

ccs – Client command specifier. Always set to 3.

x – reserved.

If the object to be written is larger than 4 bytes, then the first write request will indicate a segmented write (expedited bit e=0). In this case, the client will continue sending messages until all the data has been transmitted. These subsequent messages are organised as follows:

Byte 0				Byte 1..7
Bit 7..5	Bit 4	Bit 3..1	Bit 0	
ccs = 0	t	n	c	Data

ccs – Client command specifier. Always set to 0.

c – set to 1 if it is the last segment/message of the write request.

t – toggle bit. Set to 0 in first segment, toggled in each subsequent request.

n – number of data bytes in bytes 1 to 7 that do not contain data.

To each of the segmented write messages, the server will reply with a message containing:

Byte 0			Byte 1..7
Bit 7..5	Bit 4	Bit 3..0	
ccs = 1	t	x	Reserved

ccs – Client command specifier. Always set to 1.

t – toggle bit. Set to 0 in first segment, toggled in each subsequent request.

x – reserved.

When a client wishes to read an object in a server, it will send the following request:

Byte 0		Byte 1..3	Byte 4..7
Bit 7..5	Bit 4..0		
ccs = 2	x = 0	Index/ Subindex	Reserved

ccs – Client command specifier. Always set to 2.

x – reserved.

The protocol to transmit the data in the reverse direction, from server to client, will use a similar protocol as that to download data. Therefore, in reply to the above request, the server will send the following 'initiate upload response':

Byte 0					Byte 1..3	Byte 4..7
Bit 7..5	Bit 4	Bit 3..2	Bit 1	Bit 0		
ccs = 2	x = 0	n	e	s	Index/ Subindex	Expedited Data

ccs – Client command specifier. Always set to 2.

e – expedited transfer = 1 (=> data in bytes 4..7); segmented transfer = 0

s – set to 1 if data size is indicated

n – if e=1, and s=1, the number of data bytes (4..7) that do not contain data

x – reserved

If the above message indicates a request for a segmented transfer (e=0), then the server does not take the initiative to start sending the data. The client must first request that each segment be sent, by sending to the server the message:

Byte 0			Byte 1..7
Bit 7..5	Bit 4	Bit 3..0	

ccs = 3	t	x	Reserved

ccs – Client command specifier. Always set to 3.

t – toggle bit. Set to 0 in first segment, toggled in each subsequent request.

x – reserved.

To which the server will reply with the data message:

Byte 0				Byte 1..7
Bit 7..5	Bit 4	Bit 3..1	Bit 0	
ccs = 0	t	n	c	Data

ccs – Client command specifier. Always set to 0.

c – set to 1 if it is the last segment/message of the write request.

t – toggle bit. Set to 0 in first segment, toggled in each subsequent request.

n – number of data bytes in bytes 1 to 7 that do not contain data.

At any time during the above mentioned data transfers, both the client and the server are free to abort the transaction by sending an error message.

Byte 0		Byte 1..3	Byte 4..7
Bit 7..5	Bit 4..0		
ccs = 4	x = 0	Index/ Subindex	Error Code

ccs – Client command specifier. Always set to 4.

x – reserved.

Examples of errors that may occur and may need to be identified are an attempt to write/read from an index that does not exist in the server's object dictionary, or a write to an object whise data length does not match the number of bytes ent in the write request.

2.6 Accessing Objects in the OD using PDO

The SDO protocol is relatively slow, as it requires that the client request the data to be read from/written to, as well as an explicit confirmation message from the server. If an object of only 4 bytes length is being accessed using the expedited SDO method, two CAN messages will be transmitted over the CAN bus. This results in an effective use of the available bandwidth for transmitting data of only 25% (4 data bytes out of 16 transmitted in both messages). Even in the segmented mode, bandwidth utilization is always smaller then 44% (7 data bytes out of 16) even if we disregard the first two messages to set up the data transfer.

In contrast, the PDO protocol allows a better bus utilization, as well as data transmission with lower delays (i.e. lower latency). Whereas the SDO protocol implements a client/server interaction protocol, the PDO (Process Data Objects) protocol is based on a publish/subscribe paradigm.

This PDO protocol, as the name implies, is expected to be used to transfer process data, i.e. data related to the current state of the industrial process under control. This is data that typically needs to be updated often, at a certain periodic frequency. For example, a CAN Open node reading the temperature of the liquid in the tank will periodically publish the current temperature to the bus. All other nodes interested in this value will simply read it off the bus whenever the publishing node decides to send it. More than one receiving node may be interested; for example, a node that runs a PID loop controlling the heater in the same tank will need to sample the current temperature. Another example could be an alarm siren, that will go off if the temperature exceeds a certain value.

Although the SDO protocol can also be used to write and read from objects in the object dictionary whose value represents the current state of the industrial process (ex. The temperature of a liquid in a tank, the position of a valve, ...), this protocol is typically used instead to read and write to objects whose function is to configure the CAN Open node itself. This configuration is usually done once at system startup, or during testing and/or debugging.

A PDO is essentially a message that is transmitted onto the bus by a CAN Open node (i.e. the publisher), and read by one or more CAN Open nodes (subscribers). The message itself is a CAN message whose payload of up to 8 bytes will contain up to 8 bytes of process data. For the node transmitting the message, this will be a TPDO (Transmit PDO). For the nodes receiving this exact same message, this same PDO will be considered a RPDO (Receive PDO).

The process data placed inside a PDO transmitted by a node will come from objects in the object dictionary of that same node. For each PDO that a node transmits, a mapping is defined as to where the data for that PDO is obtained from. This mapping is really a list of {object index, subindex} that identify which data to place in the PDO, in the same order of the list. As you may remember, each subindex of an object may be of any elementary data-type (elementary data-types are either 1, 8, 16, 24, 32, 40, 48, 56 or 64 bis n length). In principle, a mapping with any combination of subindexes of any data-type is allowed, as long as the total number of bits does not exceed 64 bits. In the extreme, all of the 64 available bits of a PDO may be mapped to 64 subindexes of 1 bit length. In practice, nodes usually limit the mapping in such a way that data-types that are 8 bits or more long must be placed in offsets in the PDO that are themselves a multiple of 8 bits. This means that the data is usually aligned on the byte boundaries, which makes the copying of the data more efficient for the micro-processor.

For example, a PDO that uses only 4 bytes may be configured to transmit in the first byte the data in the subindex 1 of object 6000h (which happens to be 8 bits in length), followed by subindex 3 of object 6001h (8 bits), itself followed by subindex 2 of object 6401h (16 bits). The first 2 subindexes could represent the state of digital inputs in the CAN Open node, whereas the last 16 bit value could represent an analog input.

Any number of nodes on the CAN Open network may be configured to receive that PDO. Each node that receives the PDO will have its own mapping of what to do with the data it receives in that PDO. The mapping will also be a list of {object index, subindex} that references objects in its own object dictionary. The mapping of the data in the receiver may be completely different to the mapping of the transmitter, and each receiver may have a distinct mapping. It is up to each receiver to decide what to do with each bit it receives from a PDO.

For example, a CAN Open node consisting of a PLC mas be configured to receive the above mentioned 4 byte long PDO. It may decide to place the first 16 bits in object 2100h, subindex 2 (which will be a 16 bit datatype entry), and the remaining 16 bits in object 2100h, subindex 3. (also 16 bits).

As was already said, each PDO message is essentially a CAN message with all data bytes allocated to the data being transmitted. This means that the only way of distinguishing one PDO from another is by the 11 bit CAN message ID used to transmit each PDO. CAN Open reserves 1024 COB-IDs (the same as a 11 bit CAN message ID) for the transmission of PDOs.

CAN Open proposes a default allocation of these 1024 available addresses to each node on the network. The 1024 addresses are evenly distributed among the 127 nodes resulting in an allocation of 8 addresses per node; 4 of these the node is expected to use to transmit PDOs, and the other 4 to be used to receive PDOs. The 4 transmit COB-IDs of each node is obtained by adding to the CAN Open node ID (a 7 bit value) the values 180h (TxPDO1), 280h (TPDO2), 380h (TPO3), and 480h (TPO4). The same occurs for the receive PDOs: add 200h (RPDO1), 300h (RPDO2), 400h (RPDO3), 500h (RPDO4). For example, the node with CAN Open node ID 7, will use the COB-IDs:

COB-ID	PDO
187h	TPDO1
207h	RPDO1
287h	TPDO2
307h	RPDO2
387h	TPDO3
407h	RPDO3
487h	TPDO4
507h	RPDO4

This default allocation is nevertheless merely a suggestion, and any allocation may be made. In fact, we can note that in the allocation of COB-IDs among the nodes, no COB-ID is shared between any two nodes. However, in view of the fact that to receive data from a specific PDO the receiving node must subscribe to that PDO's COB-ID, the subscribed COB-ID must be the same as the COD-ID used by the transmitter to send that PDO. This means that the sending and the receiving nodes must use the same COB-ID, but as the default allocation does not include any shared COB-IDs, the default allocation does not permit a transmitted PDO to be received by any other node.

In practice it is common for a node to follow the default allocation of COB-IDs for the transmit PDOs, and ignore the default allocation for the receive PDOs. This means that commonly transmit PDOs use the transmit COB-IDs allocated in the default allocation to the transmitting node, but for nodes to receive whatever COB-ID used by the transmitter for the PDO it needs to receive.

2.7 PDO Triggering

The transmission of PDOs may follow several interaction paradigms:

- publish/subscribe paradigm, time triggered
- publish/subscribe paradigm, event triggered
- client/server
- synchronised polling

In the publish/subscribe paradigm the node responsible for transmitting a PDO makes the decision when to make that transmission, which may be either event or time triggered. In the event triggered method, the PDO is transmitted as soon as an event occurs in the node. The definition of what consists an event will differ from node to the next, but is usually mapped to the change of state of the data in that. Analog values connected to analog sensors will typically be continually changing, so in these cases a minimum value of change is usually required before a 'change' event occurs.

If taken to extremes, if the data changes at a fast rate, this mechanism could result a single node taking over the bus by continually sending back-to-back messages of the same PDO. CAN Open protects against this by defining a minimum interval between two consecutive transmissions of the same PDO (the inhibit timer).

With the publish/subscribe time triggered method, node transmitting the node will periodically send the PDO, using a local timer with ms resolution. Each node uses its own timer, which are not synchronised.

In the client/server paradigm a node requests a PDO to be transmitted, and the node responsible for that transmission responds by sending that PDO out on the bus. The request is sent using a 'remote request' frame, which is identical to the requested PDO, but with 0 data bytes, and the RTR bit of the CAN header set to 1.

A variation of the client server also allows a single request from a client to be sent to all PDO transmitting nodes – the SYNC message. After receiving this SYNC message, each node will immediately send out all PDOs configured to be sent with the SYNC message. Naturally, only one PDO will be able to be sent on the bus at a time, but the medium access control of the CAN bus will guarantee that they will be transmitted one after the other, starting with the PDO with the lowest COB-ID. Any receiver listening to all these PDOs will receive them also one after the other, however it is guaranteed that the data they contain was all obtained at the same exact time at which the SYNC message was sent out.

Although the SYNC signal may be sent onto the bus at any time and at any frequency, typically it is sent periodically by a master node. The SYNC message is typically a CAN message with 0 data bytes, and with a COB-ID of 80h. However, the COB-ID to be used by the SYNC message may be configured independently on each node, and different values other than the default value of 80h may be used. One may even have some nodes respond to a SYNC message with one COB-ID, and other nodes responding to another SYNC message with another COB-ID, therefore effectively organising the nodes in groups.

2.8 PDO Configuration

Although the default allocation of COB-IDs to nodes allows for only 4 transmit PDOs and 4 receive PDOS, each node can actually be configured to transmit up to 512 PDOs, and to receive also up to 512 PDOs. The configuration of which PDOs to transmit, as well as which PDOs to receive, is all contained in the object dictionary of each node. Note however that in actual practice very few devices support receiving 512 RPDOs.

The objects in the object dictionary from index 1400h to 15FFh are used to configure the RPDOs (one index for each RPDO). Each of these objects in the above range have the following subindexes:

Subindex	Description	Data Type
0	Number of subentries	Unsigned 8
1	COB-ID	Unsigned 32

Subindex	Description	Data Type
2	Transmission type	Unsigned 8
3	Inhibit Time (not used for RPDOs)	Unsigned 16
4	Reserved (unused)	Unsigned 8
5	Event Timer	Unsigned 16

For nodes that do not support the event timer on RPDOs the last 3 subindexes (3-5) do not exist - in this case the subindex 'number of entries' will have the value 2, otherwise it will contain the value 5.

The COB-ID is the COB-ID to accept for this PDO, while transmission type will determine whether the receiver should act on this PDO immediately after receiving it, or if it will delay this action until it receives the next SYNC message.

The event timer is used to have the node generate an error message if the time between the reception of two consecutive instances of this RPDO ever exceeds this value.

Similarly, the TPDOs are configured by the objects in the object dictionary in the indexes from 1800h to 19FFh. These objects follow the same format as those used for configuring RPDOs, with the difference that the entry 'inhibit time' is now used to specify the minimum interval between the sending of two consecutive TPDOs.

The transmission type parameter may have the following values:

TransmissionType	PDO Transmission				
	Cyclic	Acyclic	Synchronous	Asynchronous	Request
0		X	X		
1-240	X		X		
241-251	Reserved				
252			X		X
253				X	X
254				X	
255				X	

2.9 Timing Analysis

Control applications typically have tight timing requirements. Consider for example a bottle that is being transported on a conveyor belt towards a filling station, where it needs to stop in such a way as to have the bottle opening aligned with the filling station. A sensor detects the arrival of the bottle at the filling station, after which a PLC (Programmable Logic Controller) sends a signal for the conveyor's speed drive to stop the conveyor. In this scenario, the signal to stop the conveyor is time critical – large delays may leave the bottle with an incorrect alignment.

When the signals from the sensor to the PLC, and from the PLC to the speed driver, are sent over a communication bus, they may suffer delays imposed by the bus itself. When configuring

a network it is therefore of the utmost importance to guarantee that it will be able to transmit all the required messages within the maximum delays tolerated by the application.

When dealing with a CAN Open network we can use two approaches to the timing analysis. The first is simply to verify whether that all messages will eventually be sent over the bus, in other words, if we are not placing on the bus more messages than the bandwidth of the bus itself. The second approach is to determine the maximum delay that each message may suffer during transmission, and to compare this with the application requirements.

2.10 Determining WCTT

Both of the mentioned approaches rely on the fact that we can determine how much time each message will occupy the shared bus during its transmission. This can be determined by adding up the number of bits in a message, and diving this value by the baud rate currently in use by the network.

To determine the number of bits in a message we need to know the format of a CAN frame. This was explained in chapter 5 of this module, but we repeat the frame format here for convenience:

SOF	Arbitration		Control			Data	CRC	ACK	EOF	IFS
1 bit	11 bits	1 bit	1 bit	1 bit	4 bits	0..64 bits	16 bits	2 bits	7 bits	>=3 bits
SOF=0	Identifier	RTR	IDE	r0	DLC	Data	CRC	ACK	=0	=1

Standard (Part A) CAN frame (11 bit Identifier)

By simple addition of the number of bits in the CAN frame, we can easily determine that each frame takes up a minimum of 47 bits, and a maximum of 111 bits (47+64). However, we must remember that the bit encoding algorithm uses bit stuffing of one extra bit after every 5 identical bits, and that bit stuffing is only done from the beginning of the frame up to the end of the CRC. If we take this into account, then the minimum frame size remains at 47 bits by assuming no extra bits are necessary, but the maximum frame is now increased. Bit stuffing is applied to a maximum of 99 bits, which means that a maximum of 19 extra bits may be added to each frame. This results in a maximum frame size of 130 bits. (A similar analysis to the above may be made for CAN frames with 29 bit identifiers).

At a transmission rate of 50kbaud, the maximum frame of 130 bits will occupy the bus for 2.6ms. At the highest transmission rate of 1Mbaud, this same frame will take up the bus for 130us. You may want to compare these values with those presented by the CAN timing calculator available online at http://www.esacademy.com/en/library/calculators/can-best-and-worst-case-calculator.html.

2.11 Determining Bandwidth

Once we know the bit-size of all messages we will have on our system, and the frequency with at which each of those messages will be transmitted, it becomes possible to determine if the bus has enough bandwidth to transmit them all.

The most common configurations for CAN Open is for a master to periodically transmit a synch message (size of 47 bits), which will in turn trigger the transmission of several PDOs from the nodes on the network. In this case, we merely need to add the size of all messages, including the synch message, and check whether all those bits can be sent in the period of time between two consecutive synch messages.

Let us consider, for example, a CAN Open network with 5 nodes, each transmitting 2 PDOs, with each PDO sending 8 data bytes. This will result in 10 messages of 130 bits, plus the synch message, resulting in 1347 bits being transmitted every period. If the synch message is transmitted every 100 ms, on a network running at 100kbaud, then we can transmit 100k*0.1 = 10k bits between two synch messages. Since this is clearly larger than the required 1346 bits, then the current network configuration is feasible. If for some reason we were required to reduce the transmission rate to 10kbaud, then we would no longer be able to transmit all the messages in the 100ms period (maximum of 1k bits).

The above approach can also be used even if each message has a distinct period. We merely need to multiply each message by the frequency with which that message is being sent. For example, consider the following set of messages:

Message id	**Bit size**	**Period**
PDO1	130	10ms
PDO2	50	5ms
PDO3	50	5ms
PDO4	80	2.5ms

The total bandwidth used would be:

$$130/10\text{ms} + 50/5\text{ms} + 50/5\text{ms} + 80/2.5\text{ms} = 65\text{kbaud}.$$

We would therefore require a network faster than 85kbaud, for example 100kbaud.

Note that there are several possible methods of configuring the CAN Open network in order to obtain different messages. One is to configure some PDOs to only be sent every x number of SYNCH messages. In the above example, the SYNCH message would be sent every 2.5 ms, and PDO2 and PDO3 would be configured to only be sent every 2nd SYNCH message, and PDO1 would be sent every 4th SYNCH message. Note that in this scenario the SYNCH message would take approximately 50 bits every 2.5ms, which would add to the total required bandwidth, reaching a value of 85kbaud.

Another possible scenario is for some nodes to send the PDOs periodically, using a local timer. In the above example, if we consider that each PDOx is sent by a distinct node x (i.e. POD1 sent by node 1, PDO2 sent by node 2, …), then we could have each node configured to send out the respective PDO periodically, using a local timer. In this case no SYNCH message would ever be sent out.

It is important to note that in this second scenario the local clocks of the nodes are not synchronised, and that even though PDO2 and PDO3 have the same period, the interval of time between the sending of PDO2 and PDO3 may slowly drift, and change from a minimum of 0 (sent simultaneously) up to 4.99(9)ms.

Yet another scenario is if the PDOs are sent asynchronously - upon the occurrence of some event at the sending node (e.g.: the data in the PDO changes) that triggers the sending of the PDO. Here, we do not have any guarantees of when in fact the PDOs are sent. We can however do the calculations for the worst case, which happens to be when each PDO would be sent continuously, but is in fact limited by the inhibit timer. Here, the inhibit timer becomes the value to use for the periodicity of the PDO transmission.

As is obvious, a mixture of the above three scenarios is obviously possible where some PDOS are sent with the SYNCH messages, others are sent periodically by the nodes, whereas others

are sent upon the occurrence of come event. In any case, the bandwidth analysis is always valid and viable.

We must also highlight that in all of the above scenarios, besides the PDOs and SYNCH messages, the CAN Open protocol will generate other messages on the bus that we have not mentioned in our explanation of the protocol. In fact, the CAN Open protocol may be configured to send heartbeat as well as node guarding messages, that will take up some extra bandwidth. We therefore advise that CAN Open networks be configured with some extra available bandwidth, or that you take these extra messages into account when determining the required bandwidth.

It should also be noted that the above analysis provides a weak scheduling guarantee; in this case it merely guarantees that each PDO will eventually find the time to be transmitted over the bus. It does not guarantee that the time at which each PDO will be effectively sent. For example, in some situations it may happen that one instance of PDO4 that was ready to be transmitted at time t_0, will finish transmitting at a time greater than t_0+2.5ms, i.e. after the next iteration of the same PDO4 is also ready to transmit.

The above situation occurs mostly because we are assuming that PDO4 is being sent with the lowest priority COB-ID out of all the PDOs in the system. One way of reducing the likelihood of the above situation occurring is to attribute priorities following the RM (Rate Monotonic) algorithm. In this algorithm the priorities are given inversely proportional to the transmission period of each PDO – the PDO sent with the lowest period would have the higher priority (the COB-ID with the lowest value), and the second highest priority to the PDO with the second lowest period, and so on monotonically for all PDOs. Even so, situations similar to that described above may still occur. This is due to the fact that once any message starts being transmitted (including low priority messages), that message cannot be interrupted for the transmission of higher priority messages – in other words, messages are scheduled with a non pre-emption algorithm.

2.12 Message Latency - 1st attempt

A stronger timing analysis is also possible, and it is based on determining the latest possible time at which a PDO will finish transmitting, in relation to the time it is released for transmission. We will call this the response time R, and as we are interested in determining its value in the worst case (wc), we are effectively trying to determine the value Rwc (the value of Rwc for each PDOi will be indicated as Rwc_i.).

In order to determine this value it is important to remember the algorithm that Is used to control the access to the CAN bus, and the fact that during simultaneous transmissions, the message with the lowest CAN ID will be transmitted before all the other messages. This CAN message ID, which in can Open is called COB-ID, will function as a fixed priority parameter. In the following discussion we will therefore call this value the PDO priority. Higher priorities correspond to COB-IDs with smaller values (i.e. closer to 0).

It is also important to recognize that the worst delay in starting the transmission of a given PDOi will occur when:

- a few moments before PDOi needs to be transmitted, the longest PDO with any priority lower than PDOi starts transmitting, therefore blocking the transmission of PDOi. This is called the blocking time B;

- during this initial interval during which PDOi is blocked, all other PDOs with a priority higher than PDOi will also become ready for transmission;
- Once the initial message (lower priority than PDOi) finishes transmission, all messages with a higher priority than PDOi will be transmitted first. During this period, the same higher priority message may be transmitted multiple times, if it happens to be released again, before all the messages with a higher priority than PDOi are finished transmitting. This interval will be called the interference time I.

The maximum delay a message will suffer before it finishes transmission will therefore be:

(6.1) $Rwc=B_i+I_i+C_i$

where C represents the time it takes to transmit the desired message.

Both C and B are easy to determine. Determining the value of I is more difficult, since we do not know *a priori* the number of times a specific PDO will be transmitted, as that will itself be dependent on the value of I. But we do know that for each PDOk (with PDOk having a higher priority than PDOi), the number of times it will be get to be transmitted before PDOi starts transmitting will be:

(6.2) $n_k = \left\lceil \frac{W_i}{T_k} \right\rceil$

(6.3) $W_i=B_i+I_i$

Here, the symbols$\lceil x \rceil$represent the ceiling of x, where any number is always rounded up to the next highest integer. So both 1.0001 and 1.999 would round up to 2.

The amount of time these nk messages take to be transmitted is

(6.4)$\left\lceil \frac{W_i}{T_k} \right\rceil C_k$

The total value of W will be the sum of B with all the Ik, where PDOk has a higher priority that PDOi:

(6.5) $W_i = B_i + \sum_{k=1}^{i-1} \left\lceil \frac{W_i}{T_k} \right\rceil C_k$

Now, unfortunately it is not possible to solve the above equation using simple algebra, because the ceiling function is not a continuous function. We can however solve that formula using an iterative method. We start with a first approximation for W, assuming each PDOk is transmitted only once:

$$W_i(0) = B_i + \sum_{k=1}^{i-1} C_k$$

We then determine a second approximation for the value of W, using the first value as a basis for determining the value of each nk:

$$W_i(1) = B_i + \sum_{k=1}^{i-1} \left\lceil \frac{W_i(0)}{T_k} \right\rceil C_k$$

We continue doing these iterations:

$$W_i(m+1) = B_i + \sum_{k=1}^{i-1} \left\lceil \frac{W_i(m)}{T_k} \right\rceil C_k$$

until we reach a a point where the approximate value of W stops increasing. Basically, we stop when

(6.6) W_i(m+1)=W$_i(m)$

which will be the true value of W.

Please note however that we have no guarantee that the iteration process will ultimately stop. Typically, in an overloaded network the value of W will continue rising. In these situations, we will typically stop the iteration process once we determine that the value of W is larger than the repetition period of the PDOi whose worst case transmission time we are trying to determine. In these cases we conclude that there are situations where PDOi will never get to transmit.

Just to summarise, remember that in all of the above formulas:

- C_i represents the transmission time of PDO_i.
- T_i represents the transmission period of PDO_i.
- hp(i) represents all the PDOs with higher priority than PDO_i.
- C_k represents the transmission time of PDO_k.
- $\lceil x \rceil$represents the ceiling function (round up to nearest integer).

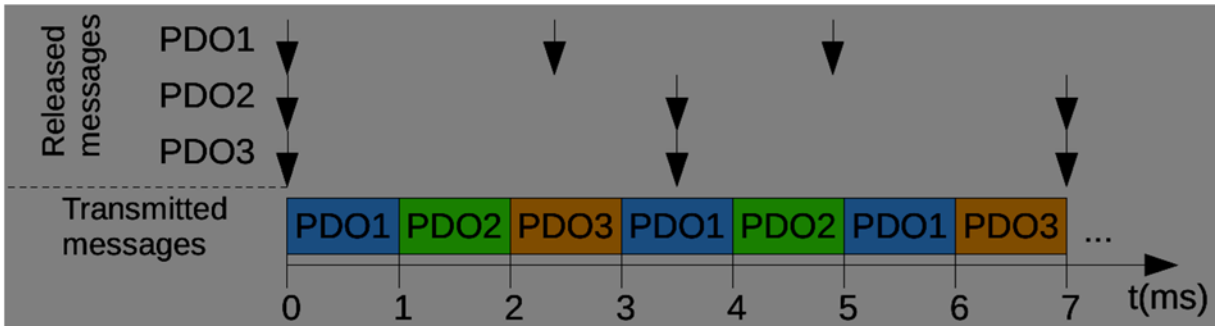
2.13 Message Latency – 2nd attempt

Unfortunately, the above analysis is not completely correct. Consider, for example the following scenario (taken directly from Davis et al.) with 3 PDO messages :

Message	Priority	Period	Deadline	Tx time
PDO1	1	2.5ms	2.5ms	1ms
PDO2	2	3.5ms	3.25ms	1ms
PDO3	3	3.5ms	3.25ms	1ms

Using the analysis presented in the previous sub-chapter to determine the worst case response time (R) of message PDO3 (i.e. the lowest priority message, which corresponds to the message with the highest COB-ID) will result in Rwc3 = 3ms.

This is not the true value, because the true worst case does not occur during this first attempt at transmitting message PDO3. In the following figure is a representation of the true transmission sequence.



Here you can see that for first transmission of PDO3, it is released at t=0ms, and ends its transmission at t=3ms, resulting in a response time R=3-0=3ms. For the second transmission of

PDO3, released at t=3,5ms, the transmission ends at t=ms, which corresponds to a response time R=7-3,5=3,5ms. The real response time is therefore larger than that calculated with the presented algorithm, because that algorithm assumes that the worst case response time R will always occur in the first transmission. Now that assumption is not valid since the transmission of the PDO3 can delay the beginning of the transmission of higher priority PDOs in the next iteration.

To correctly determine the response time of a PDOi, we must therefore start off by determining at what time, after the simultaneous release of PDOi and all other higher priority PDOs, will all the messages that were released up to that point finish their transmission. In other words, we want to determine at what time instant will the bus become free, after the first release of PDOi. Notice that this is exactly the same time that we called W(where W = B + I) in the previous algorithm, but instead of considering only the PDOs with higher priority than PDOi, we will include the PDOi messages too. Here we will call this time the busy interval for priority level of PDOi, or in short, the level i busy period L_i.

$$(6.7)\ L_i = B_i + \sum_{k=1}^{i} \left\lceil \frac{L_i}{T_k} \right\rceil C_k$$

For the above example, when we determine this value L_3 for PDO3, we will obtain:

$$L_3(0) = B_i + \sum_{k=1}^{3} C_k = 0 + 1 + 1 + 1 = 3$$

$$L_3(1) = B_i + \sum_{k=1}^{3} \left\lceil \frac{L_3(0)}{T_k} \right\rceil C_k = 0 + \left\lceil \frac{3}{2.5} \right\rceil 1 + \left\lceil \frac{3}{3.25} \right\rceil 1 + \left\lceil \frac{3}{3.25} \right\rceil 1 = 0 + 2 + 1 + 1 = 4$$

$$L_3(2) = B_i + \sum_{k=1}^{3} \left\lceil \frac{L_3(1)}{T_k} \right\rceil C_k = 0 + \left\lceil \frac{4}{2.5} \right\rceil 1 + \left\lceil \frac{4}{3.25} \right\rceil 1 + \left\lceil \frac{4}{3.25} \right\rceil 1 = 0 + 2 + 2 + 2 = 6$$

$$L_3(3) = B_i + \sum_{k=1}^{3} \left\lceil \frac{L_3(2)}{T_k} \right\rceil C_k = 0 + \left\lceil \frac{6}{2.5} \right\rceil 1 + \left\lceil \frac{6}{3.25} \right\rceil 1 + \left\lceil \frac{6}{3.25} \right\rceil 1 = 0 + 3 + 2 + 2 = 7$$

$$L_3(4) = B_i + \sum_{k=1}^{3} \left\lceil \frac{L_3(3)}{T_k} \right\rceil C_k = 0 + \left\lceil \frac{7}{2.5} \right\rceil 1 + \left\lceil \frac{7}{3.25} \right\rceil 1 + \left\lceil \frac{7}{3.25} \right\rceil 1 = 0 + 3 + 2 + 2 = 7$$

Once we know the busy level period L, we can determine how many times the PDOi message will be transmitted in that period.

$$(6.8)\ n_i = \left\lceil \frac{L_i}{T_i} \right\rceil$$

In the example above, this value would be

$$n_3 = \left\lceil \frac{L_3}{T_3} \right\rceil = \left\lceil \frac{7}{3.5} \right\rceil = 2$$

which corresponds to the 2 times PDO3 is sent in the 7ms drawn in the figure.

What remains is to determine the worst case response time of each of these n_i messages. We will call these $R_{i,q}$, where q will take the the value from 0 to n_i-1. The worst case response time of the PDOi, will be the maximum of these values

(6.9) $R_i = \max_{q=0..n_i-1} R_{i,q}$

For the above example, we will have

$R_3 = \max_{q=0..n_3-1} R_{3,q} = max(R_{3,0}, R_{3,1})$

To determine the value of each $R_{i,q}$, we first determine the time at which it will start its transmission, which is given by the equation:

(6.10) $W_{i,q}(m+1) = B_i + qC_i + \sum_{k=1}^{i-1} \left\lceil \frac{W_{i,q}(m)+t_{bit}}{T_k} \right\rceil C_k$

where t_{bit} represents the time it takes to transmit one bit on the CAN bus.

Once again this equation needs to be solved iteratively. Possible starting values for the iterations are

(6.11) $W_{i,q}(0) = W_{i,q-1} + C_i$

(6.12) $W_{i,0}(0) = B_i + \sum_{k=1}^{i-1} C_k$

The response time of each message can then easily be determined by

(6.13) $R_{i,q} = W_{i,q} + C_i - q \cdot T_i$

Once again, for the running example, we will need to calculate $R_{3,0}$, $R_{3,1}$, and $R_{3,2}$, each of which can easily be determined from $W_{3,0}$, $W_{3,1}$, and $W_{3,2}$.

Using equation 6.12 to determine a starting point, and 6.10 for the iterations, we obtain:

$$W_{3,0}(0) = B_3 + \sum_{k=1}^{3-1} C_k = 0 + 1 + 1 = 2$$

$$W_{3,0}(1) = B_3 + 0 \cdot C_3 + \sum_{k=1}^{3-1} \left\lceil \frac{W_{3,0}(0) + t_{bit}}{T_k} \right\rceil C_k = 0 + 0 + \left\lceil \frac{2 + t_{bit}}{2.5} \right\rceil 1 + \left\lceil \frac{2 + t_{bit}}{3.5} \right\rceil 1 = 2$$

Using equation 6.13:

$$R_{3,0} = W_{3,0} + C_3 - 0 \cdot T_3 = 2 + 1 - 0 \cdot 3.5 = 3$$

Now, using equation 6.11 as the starting point, and 6.10 for the iterations, we obtain:

$$W_{3,1}(0) = W_{3,1-1} + C_3 = 2 + 1 = 3$$

$$W_{3,1}(1) = B_3 + 1 \cdot C_3 + \sum_{k=1}^{3-1} \left\lceil \frac{W_{3,1}(0) + t_{bit}}{T_k} \right\rceil C_k = 0 + 1 + \left\lceil \frac{3 + t_{bit}}{2.5} \right\rceil 1 + \left\lceil \frac{3 + t_{bit}}{3.5} \right\rceil 1$$
$$= 0 + 1 + 2 + 1 = 4$$

$$W_{3,1}(2) = B_3 + 1 \cdot C_3 + \sum_{k=1}^{3-1} \left\lceil \frac{W_{3,2}(1) + t_{bit}}{T_k} \right\rceil C_k = 0 + 1 + \left\lceil \frac{4 + t_{bit}}{2.5} \right\rceil 1 + \left\lceil \frac{4 + t_{bit}}{3.5} \right\rceil 1$$
$$= 0 + 1 + 2 + 2 = 5$$

$$W_{3,1}(3) = B_3 + 1 \cdot C_3 + \sum_{k=1}^{3-1} \left\lceil \frac{W_{3,2}(2) + t_{bit}}{T_k} \right\rceil C_k = 0 + 1 + \left\lceil \frac{5 + t_{bit}}{2.5} \right\rceil 1 + \left\lceil \frac{5 + t_{bit}}{3.5} \right\rceil 1$$
$$= 0 + 1 + 3 + 2 = 6$$

$$W_{3,1}(4) = B_3 + 1 \cdot C_3 + \sum_{k=1}^{3-1} \left\lceil \frac{W_{3,2}(3) + t_{bit}}{T_k} \right\rceil C_k = 0 + 1 + \left\lceil \frac{6 + t_{bit}}{2.5} \right\rceil 1 + \left\lceil \frac{6 + t_{bit}}{3.5} \right\rceil 1$$
$$= 0 + 1 + 3 + 2 = 6$$

$$R_{3,1} = W_{3,1} + C_3 - 1 \cdot T_3 = 6 + 1 - 1 \cdot 3.5 = 3.5$$

We can now correctly determine the worst case response time for PDO3 as

$$R_3 = \max_{q=0..n_3-1} R_{3,q} = max(R_{3,0}, R_{3,1}) = max(3,3.5) = 3.5\text{ms}$$

2.14 Bibliography

- "Embedded Networking with CAN and CANopen", Olaf Pfeiffer, Andrew Ayre, Christian Keydel, Revised First Edition, Published by Copperhill Technologies Corporation, 2008
- "CANopen: high-level protocol for CAN-bus", H. Boterenbrood, NIKHEF, Amsterdam, March 20, 2000
- "CANOpen Implementation Guidelines", G.Gruhler(Ed.), Bernd Dreier, STA Reutlingen, Germany, ESPRIT Project 22171, http://www.fh-reutlingen.de/~www-sta
- "CANOpen Memento", Francis Dupin, August 2009, Version 1.5, LIVIC – 14, route de la Minière. 78000 Versailles Satory. France, http://www.inrets.fr/ur/livic

2.15 Further Reading

- "Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revisited", R. Davis, A. Burns, R. Bril, J. Lukkien, Springer, Real-Time Systems Journal, April 2007, Volume 35, Issue 3, pp 239-272, (Open Access)
- "CANopen Communication Profile Features", CAN in Automation (CiA) International Users and Manufacturers Group, Am Weichselgarten 26 x D-91058 Erlangen
- "CiA 301 CANopen application layer specification", free downloadable from CAN in Automation
- "CiA 306 CANopen Electronic Data Sheet (EDS) specification"
- "CiA 311 CANopen XML-EDS specification"
- "CiA 401 CANopen device profile specification for generic I/O modules", free downloadable from CAN in Automation
- "CiA 402 CANopen device profile for motion controllers and drives" (same as IEC 61800-7-201/301)

Software

- CANopenNode: CANopen based stack for communication in embeded control systems, by jani22, http://sourceforge.net/projects/canopennode/

– “MicroCANOpen”, 2003, as described in the book "Embedded Networking with CAN and CANopen", Source code downloaded from files.esacademy.com/index.php?dir=CANopen%20Book%20Files/code/&file=MCOv2.00.zip, http://www.canopenbook.com/microcanopen.htm

3 LAB – Week 11

In the following 3 lab sessions, you will be implementing a CAN Open device on the Arduino. This first lab session will be focused on implementing the basic data structures for the Object Dictionary.

3.1 Create the Data Structures

Notice that the object dictionary may contain a large number of entries, and that each entry may require very different number of bytes to store the required data. Implementing the Object Dictionary as a simple array of entries, with one position in the array for each possible entry, will require an array with up to 65536 positions. If each position takes up only one byte (which is clearly insufficient), just this array will use up 64kbytes of memory.

However, a very great number of the positions in the Object Dictionary will be unused, and do not need to store any data. We can use this property to reduce the size of the array - we suggest that you use a map instead.

A map can be implemented as an array of a structured data type. The size of the array will correspond to the maximum number of entries the map (in this case, the Object dictionary) will ever contain. The structure itself will contain an element indicating the index in the object dictionary to which this entry in the map corresponds to, another element indicating the datatype of this entry, as well as a pointer to the the location where the element is in fact stored.

```
typedef struct {
  u16 index;
  u16 data_location; /* references position in eo[] array */
  u8 datatype;
} od_entry_t;


#define  OD_max_objects 100;
od_entry_t  od[OD_max_objects];


/* array for object entry data */
#define  OE_max_size 1000;
u8 oe[ OE_max_size 1000];
```

Notice that with the way the data structure is built, the data associated to each object is in fact stored in the oe array. Notice too that each object can use up as many bytes as necessary in this oe array, with some objects using up only one byte, and others using 32 bytes or more.

3.2 Populate the Data Structures

Write some code that will guarantee that the OD will contain some entries when the program starts. You should start with the entries that define the datatypes themselves – 0001h to 025Fh. Remember that not all of these will be used by your application. However, at the very least, the entries for the pre-defined datatypes (0001h – 0023h) should be populated.

You should now also create an additional two or three entries for testing purposes. Make sure that each entry referencs in the oe array does not overlap.

To indicate that an entry in the od array is not currently being used to store an entry in the object dictionary, we can set the index to 0000h (a reserved index). Write an initialisation procedure to do this at startup.

3.3 Determining an Object Size

Write two functions that will return the number of bytes used by an object or sub-entry.

```
int od_getsize_object(u16 object_index, u8 *size);
int od_getsize_subentry(u16 object_index, u8 subentry_index, u8 *size);
```

Each of these functions must look up the object in the object dictionary to determine its datatype, and then look up that datatype in the object dictionary to determine the size of the object,

Remember that for the entries in the OD that define each datatype (0001h – 025Fh), the stored values will indicate the size, in bits, of the respective datatype.

Each function should return 0 on success, and a negative number when a error occurs (example: object not found in the OD).

3.4 Reading from the Object Dictionary

Write two functions for reading from the object dictionary.

```
od_read_object(u16 object_index, u8 *data, u8 *size);
od_read_subentry(u16 object_index, u8 subentry_index, u8 *data, u8 *size);
```

Each function should return 0 on success, and a negative number when a error occurs (example: object not found in the OD).

Notice that it is up to the functions themselves to determine the size (number of bytes) of each object, and to tell the caller the size.

The function od_read_object() must:

1) find the object in the object dictionary
2) determine the number sub-entries this object contains
3) determine the total number of bytes of the object
4) proceed with copying the data to the 'data' parameter from the oe array

The function od_write_subindex() must:

5) find the object in the object dictionary
6) determine the number sub-entries this object contains
7) determine the total number of bytes of the specified sub-entry
8) determine the position in the oe for this subentry
9) proceed with copying the data to the 'data' parameter from the oe array

In each of these algorithms, (1) will involve reading the object's datatype, and looking up this datatype in the object dictionary itself. It is probably best to simply call the functions that you have just finished implementing.

3.5 Writing to the Object Dictionary

Write two functions for writing to the object dictionary. One function should be able to store one sub-entry at a time. The other function will store a complete object.

```
od_write_object(u16 object_index, u8 *data);
```

```
od_write_subentry(u16 object_index, u8 subentry_index, u8 *data);
```

Each function should return 0 on success, and a negative number when a error occurs (example: object not found in the OD).

Notice that it is up to the functions themselves to determine the size (number of bytes) of each object, so the functions do not need a parameter to indicate the number of valid bytes in the data array.

The algorithms these functions implement are very similar to the previous functions, just the directions of copying is different.

3.6 Mapping Physical I/O to the Object Dictionary

Add an entry in the Object dictionary that will correspond to the current value in the joystick of the Arduino CAN shield. Add yet another entry that will correspond to the values of the two leds in the CAN shield.

Write a small cyclic program that will keep these values updated (copy from joystick to OD, and from OD to leds). If you prefer, you can write this as a periodic interrupt.

This functionality will be used in the next lab, but you can test it by calling the function to read the 'joystick' object from the Object Dictionary, and by calling the function to write to the 'leds' object in the Object Dictionary.

3.7 Optimizations

If you have time, it is suggested that you add more optimisations to the OD data structure.

The first optimization is to use the memory that stores the data_location element to instead store the data of objects that take up 2 or less bytes. For example, when an object is of a simple datatype, and the datatype is smaller or equal to 2 bytes, then instead of storing the data in the oe array, and having data_location reference this position, the data can be stored directly where data_location would be stored. We suggest the use of a 'union' for this.

Another optimization is to keep the od array sorted by the object index. This way looking up in the array can use optimised algorithms (for example, binary search).

4 LAB – Week 12

In this lab you will start implementing the CAN Open communication protocol itself. We will start with the transmission and reception of PDOs. This should be relatively simple considering the Object Dictionary data structures have been implemented in the previous lab.

4.1 Configuring the TPDO OD Entry

As was explained in the lecture, the format of the PDOs that are sent and received are completely defined by an entry in the Object Dictionary. You must therefore start off by adding

an appropriate entry in the Object Dictionary for a Transmit PDO, for example in the index 1800h.

Start by defining a cyclic TPDO, although we can at first completely ignore this parameter.

You can also choose to send the data from the 'joystick' object you created in the previous lab session. If you have a periodic interrupt copying the status of the joystick to the 'joystick' object in the OD, sending this PDO will effectively send the status of the joystick over the CAN network.

4.2 Transmit a PDO

Write a function to transmit a PDO. The function should take as parameter the object entry that specifies the data to place in the PDO.

```
pdo_send(u16 od_index);
```

This function must

1) read the object in the OD that specifies the data to place in the PDO

for each sub-entry:

   1. get the data from the object dictionary
   2. place it in the next available slot of the PDO frame

2) send the PDO, using the correct message ID (basically the COB-ID stored in the OD)

To send the PDO, you can use the functions you already used when sending data over the CAN bus.

You can test this function by reading the messages sent over the CAN bus using a second Arduino.

4.3 Transmit and Receive a Synch Message

On one Arduino, write a small program that simply sends a synch message, and then reads whatever comes up on the network.

On the second Arduino, write a small program that continuosly waits for a synch message. When one is received, then go through all the configured TPDOs in the OD, and send any TPDO that is configured to be sent as a response to a synch message.

4.4 Receive a PDO

Write a function to receive a PDO. Remember that you should first configure a RPDO in the object dictionary, and how to map the received data. For example, you can configure to map a RPDO (with the same COB-ID used by the previous TPDO) to the 'leds' object in the OD (this 'leds' object was created in the previous lab session).

```
pdo_recv(void);
```

Change the program that periodically transmits the synch message, to call the pdo_recv() function after sending the synch message. If everything is correctly configured, you will now be able to change the status of the leds of one Arduino, by activating the joystick of the second Arduino.

Don't forget that you will need a background interrupt task in both Arduinos. In one Arduino it will copy the state of the joystick to the 'joystick' object in the OD. In the other Arduino it will copy the 'leds' object in the OD to the leds in the CAN shield.

4.5 Send and Receive SDO

If you have time, or as an extra-curricular activity, we suggest you implement a function to send an SDO. If possible write another function to receive an SDO.

5 LAB – Week 13

In this lab session we will determine the response time of the CAN network.

During the following session, when asked to measure the time delay, you can either do so by one of three methods. The first is to set and reset a physical output on the Arduino, and use an oscilloscope to measure the time between the set and reset operations. The second is to use an internal hardware timer. The third is to use the micros() or millis() functions of the Arduino libraries, which actually use a hardware timer.

5.1 Measure Processing Time

Setup a network where one Arduino periodically sends a synch message, and the second Arduino responds with a single PDO (this is basically what you did in the previous lab session).

Measure the time it takes to process the request by the Arduino receiving the synch message. In other words, measure the time your code takes to determine that you received a synch message, and the time it takes to build (but not send) the response PDO frame.

5.2 Measure Round-Trip Time

In the Arduino sending the synch message, measure the time it takes to send the synch message, and finish receiving the synch message. More precisely, start counting the time immediately before starting to send the synch message, and as soon as you have finished receiving the response PDO. This time should be the time it takes to transmit both the synch and the PDO messages over the CAN bus, added by the time the receiving Arduino took to process these messages (i.e. the time measure in the previous sub-section).

Taking into account the transmission speed of your network, calculate how long it should take to transmit both the synch and the response PDO. Compare this time to the measured time.

Repeat this exercise for various speeds of the network.

5.3 Measure Response Time

Configure both Arduinos to have a background task (i.e. an interrupt handler) periodically copy the physical I/O (joystick and leds) to/from the corresponding objects in the OD.

Configure the Arduino sending the synch message to send these periodically, at a different rate.

Using an oscilloscope, measure the time it takes between activating the joystick of the Arduino sending the PDOs, and the time the led on the Arduino sending the synch messages reflects this change.

This value should vary significantly from a maximum and a minimum values. Determine the expected values analytically, and compare to the values you obtained in your measurements.

6 Seminar – Week 11

Taking into account the typical properties of the Object Dictionary, discuss and propose data structures to store this information in an embedded device programmed using the C programming language.

Take into account that the OD has a potential 65536 indexes, but that very few of these possible entries are actually used in a real-world application. Also note that some of these entries are read-only, whereas others may be written to. Remember too that each entry may be of a different data type, requiring a different number of bytes to store the necessary information. This information can vary from 1 bit to 2kBytes for a single object (or even more if strings and domain data are taken into account).

7 Seminar – Week 12

The transmission of PDO messages can be triggered by several mechanisms, including time-triggered (periodically) and event triggered (data changes). Assuming that functions already exist for the transmission and receipt of PDOs, study and propose an architecture for the software that will implement a CAN Open slave capable of responding to all of these triggers simultanesouly.

8 Seminar – Week 13

The CAN Open network can work in both an event-triggered and a time-triggered approach, or possibly even a mixture of these two, all depending on how it is configured.

Analyse and discuss the possible advantages and drawbacks of each approach. In what situations is each approach more appropriate.

Consider not only the time it takes to respond to events, but also the possibility of tolerating and detecting faults in the system.

9 Mini-project – Week 11

In the previous two weeks of your mini-project work, you have integrated a CAN bus communication to the Arduino of your industrial process control setup. This setup should currently consist of a plant floor simulator controlled by a process control software; this control software connects via Modbus/RTU to a slave running on the Arduino; and the Arduino connects via CAN bus to other Arduinos.

In the next three weeks your mini-project work will consist in introducing the CAN-Open protocol to the CAN-bus network. In this first week you will start by integrating into the code of your project the data structure used to store a CAN-Open Object Dictionary on the Arduino. This data structure was developed during the current week's lab session, along with functions to access the data structure.

The students should now design a mapping between the modbus data model, and the data model used by CAN-Open. Implement this mapping in your project's code. One possible method of doing this is to have the modbus slave write and read data directly from the CAN-Open OD data structures. Another method is to add a background task that will periodically do this mapping. Discuss the advantages and drawbacks of each alternative that you consider, and implement the one you consider best.

10 Mini-project – Week 12

In this week's lab session you have implemented the CAN-Open protocol for transmission and receipt of PDO frames. You should now integrate this protocol into your project and use it to fulfil the requirements defined in weeks 9 and 10.

During those weeks you exchanged data between Arduinos directly reading and writing to the CAN bus. This same data should now be exchanged by using the CAN-Open protocol. Once

you have integrated the software that implements the PDO transmission protocol, to exchange the desired data should only need to correctly configure the transmit and receive PDOs in the Object Dictionary of each Arduino.

To recap, you should now have three or four Arduinos connected by a CAN network. In each of these Arduinos you should now have a led that shows the RUN status of the control processes connected to this same Arduino, as well as the two Arduinos immediately before and after. You should also have an extra STOP2 button that will not only stop the control process of the current Arduino, but also the control process on the next Arduino too.

11 Mini-project – Week 13

In this week's session of the mini-project the students will not advance the project any further. They will instead use the current setup of the project to do some timing analysis.

We will continue using the same setup as before: Each PC is connected to one Arduino board through a serial connection (emulated over USB). Several Arduinos are connected on the same CAN-Open network. A process control software runs on the PC.

We will assume that the process control software is reading some information on an Arduino, and that it must react to this information. To simplify development effort, we will simply have:

- the control process reads the state of a button
- the control process writes this state to a led

Basically we want a led to light up when we press a button, and to switch off when we release a button. However, the code that controls the reading of the button, and writing to the led, will run on the PC.

To start with, implement the above mention control process to read a button and control a led on the Arduino to which the PC is connected. The data will therefore 'only' need to travel over the Modbus/RTU connection.

Press the button several times, and measure the maximum and minimum delay between the pressing of the button, and the led lighting up. You can measure this delay either using an oscilloscope directly connected to the button and led pins, or by using a program in the Arduino.

Configure the CAN-Open network to send the state of the button and the led on one Arduino to/from another Arduino. Now use the same control process on the PC to read a button on an Arduino over the CAN-Open network. Nte that the information needs to go over the Modbus/RTU connection, as well as the CAN-Open network. Repeat the button pressing, and measure the maximum and minimum delay between pressing the button and the led lighting up. Repeat this measurement with the CAN-Open network running at different communication speeds (bitrate).

Simpler alternative

If the time available in this mini-project session is not enough to complete all the previous experiments, a quicker alternative is to:

- run the 'process control' software that reads the button and writes to the led running on one Arduino
- have the button and the led on another Arduino

You will now only need to configure the CAN-Open network to transmit the data between two Arduinos. Measure the maximum and minimum time delay between pressing the button and the leds lighting up. Repeat this experiment using different speeds on the CAN-Open bus.

12 References

- "Embedded Networking with CAN and CANopen", Olaf Pfeiffer, Andrew Ayre, Christian Keydel, Revised First Edition, Published by Copperhill Technologies Corporation, 2008
- "CANopen: high-level protocol for CAN-bus", H. Boterenbrood, NIKHEF, Amsterdam, March 20, 2000
- "CANOpen Implementation Guidelines", G.Gruhler(Ed.), Bernd Dreier, STA Reutlingen, Germany, ESPRIT Project 22171, http://www.fh-reutlingen.de/~www-sta
- "CANOpen Memento", Francis Dupin, August 2009, Version 1.5, LIVIC – 14, route de la Minière. 78000 Versailles Satory. France, http://www.inrets.fr/ur/livic