UNIVERSITY OF WEST BOHEMIA

FACULTY OF ELECTRICAL ENGINEERING

DEPARTMENT OF APPLIED ELECTRONICS
AND TELECOMMUNICATIONS

BACHELOR THESIS

Matěj Kubička 2011

UNIVERSITY OF WEST BOHEMIA

FACULTY OF ELECTRICAL ENGINEERING

DEPARTMENT OF APPLIED ELECTRONICS
AND TELECOMMUNICATIONS

BACHELOR THESIS

CANopen implementation

Author: Matěj Kubička

Supervisor: Ing. Kamil Kosturik Ph.D.

Pilsen 2011

Anotace

Kubička, M. Implementace CANopen. Katedra aplikované elektroniky a telekomunikací, Západočeská univerzita v Plzni - Fakulta elektrotechnická, 2011, 63 s., vedoucí: Ing. Kamil Kosturik Ph.D.

Obsahem této bakalářské práce je návrh sady algoritmů, které implementují klíčové části protokolového stacku CANopen nad automotive sběrnicí CAN. Práce je zaměřena na low-end mikrokontroléry, které nalezneme zejména v cenově citlivých embedded systémech a na implementace běžící paralelně na koprocessoru. Součástí práce je praktické vyzkoušení navržených principů na mikropočítačové architektuře Freescale HCS12X.

Motivací k této práci je dnešní realita embedded systémů založených na technologii CANopen - CANopen je velmi otevřený a modifikovatelný, ale také těžkopádný a komplikovaný. Důsledkem jsou vyšší nároky na použitý hardware a tedy i vyšší cena výsledného produktu. Tento problém je často obcházen tak, že produkt je vybaven jen vybranými funkcemi protokolového stacku CANopen. Tato bakalářská práce se snaží navrhnout sadu "out-of-box" algoritmů, které zjednoduší implementaci protokolového stacku CANopen na levnějších mikrokontrolérech.

Klíčová slova

CANopen, CAN, Controller area network, embedded networking, embedded systémy, optimalizace kódu

Abstract

Kubička, M. CANopen implementation. Department of applied electronics and telecommunications, University of West Bohemia in Pilsen - Faculty of Electrical Engineering, 2011, 63 p., head: Ing. Kamil Kosturik Ph.D.

Aim of this bachelor thesis is to propose a set of out-of-box computer algorithms designed to implement a CANopen protocol stack. Proposed design is targeted to low-end microcontrollers typically used in price-sensitive embedded systems and to implementations running entirely on a coprocessor. Introduced algorithms and ideas were tested on computer architecture Freescale HCS12X.

Motivation for this thesis comes from CANopen embedded networking reality - CANopen is very scalable but also very heavy and complicated. This leads to additional hardware requirements for used microcontroller, which pushes the price of the product. This pitfall is commonly overcome by implementing only chosen CANopen functionality. This thesis tries to propose a set of non-standard, out-of-box solutions which may ease the function of CANopen on low-end devices.

Keywords

CANopen, CAN, Controller Area Network, embedded networking, embedded systems, code optimization

Statement

I hereby submit for review and defense the bachelor thesis, prepared at the end of study at the Faculty of Electrical Engineering University of West Bohemia.
I declare that I prepared this bachelor thesis independently, using professional literature and resources listed in the list, which is part of this thesis. I also declare that all the software used to solve this thesis is legal.

In Pilsen, on June 13, 2011 ....

Contents

List of Tables

List of Figures

List of listings

List of terms and abbreviations

3S	Three-state outputs
ACK	Acknowledgement field
BDM	Background debug module
CAN	Controller Area Network
COS	Change of state
CRC	Cyclic redundancy check
CSMA/CD	Carrier sense multiple access with collision detection
DP	Device Profiles
EDS	Electronic Data-Sheets
EEPROM	Electronically erasable programmable read-only memory
EMCY	Emergency object
EOF	End of frame
MCU	Microcontroller unit
MIPS	Million instructions per second
MPU	Microprocessor unit
NMT	Network Management
NMT ErrCtrl	Network error control services
OC	Open Collector
OD	Object Dictionary
P2P	Peer-to-peer communication
PC	Personal Computer
PDO	Process data object
PSW	Program Status Word
RPDO	Receive PDO
RTS	Request-to-send
SDO	Service data object
SOF	Start of frame
SYNC	Synchronization object
TCP/IP	Transmit control protocol with internet protocol
TIME	Timestamp object
TPDO	Transmit PDO
USB	Universal Serial Bus

Acknowledgements

First of all, I would like to thank to Department of Applied Electronics and Telecommunications (FEE, UWB) which supported me with development kits and testing tools. Writing this bachelor thesis wouldn't be possible without the support of several mentors from this department. Special thanks belong to my supervisor Ing. Kamil Kosturik Ph.D. who consulted with me on various topics practically 24 hours a day, 7 days a week. Ing. Petr Krist Ph.D. also spent significant amount of time just by discussing proposed solutions and therefore gave me many useful tips on the topic.

I would like to thank to Ing. Pavel Turjanica Ph.D., former director of Electronics Department at R&D branch of ZF Engineering company. ZF accepted me on a part-time job as an "Electronic software engineer". While working at ZF Engineering I gathered severe knowledge about embedded systems and embedded networking, which gave me perfect introduction to the topic of this thesis.

It's impossible to measure how thankful I am to my family. To my mother, who is always caring and patient and to my father, who is discussing and helping all the time. They always gave me what I needed, and now, even in my adult age they are still supporting me so I can continue in my studies. Without them none of this would be possible.

Last but not least, I would like to thank to my friend Gwen, who read the manuscript of this thesis over and over again. Thank you!

1 Introduction

"Love and work are the cornerstones of our humanness"

Sigmund Freud

As of today, networked systems are literally everywhere, starting with internet, global "network of networks", continuing through automotive industry where brand new car have 5 separate networks inside (just to make the driver more comfortable) and ending in the light bulb driven by microcontroller and controlled remotely via another network [12]. We usually don't see those networks - they are hidden, made to make our life a bit easier.

Although there are well established standards such as Ethernet [10] with TCP/IP protocol, it may not be necessarily suitable for just any use you can think of. For example, TCP/IP has quite severe overhead contained in a data packet. This overhead makes additional requirements for the network bandwidth and makes especially real-time solutions more costly and sometimes even impossible to function properly. On the other hand, some functionality of TCP/IP may be useless - for example, support for gateways and bridges to another kinds of networks is typically useless at a production plant where the network is used to control the production process. On the top of it, microcontrollers with Ethernet peripheral are usually costly, because handling fast Ethernet itself requires lot of memory and also CPU performance. And the price is a vital property of any mass-produced product, so we have to take it under consideration too. To summarize, it is probable that at least in close future, simpler buses than Ethernet will be used [1].

1.1 Embedded systems and embedded networking

Embedded systems are computer systems designed to solve specific task. This feature is in contrast to widely popular PC systems which are designed to be flexible and meet wide range of user requirements. Embedded system can be controlled by one or more controlling units (usually microcontrollers or microcomputers). When more controlling units are present, they need to exchange information and therefore need to be somehow interconnected. This is where embedded networking comes in handy. Embedded network contains no host controller [13] and the network allows a point-to-point communication between many nodes.

There are several field buses currently used in the embedded networks. Just to name some - there is Fieldbus, Profibus, Modbus, Industrial Ethernet, DeviceNET [7], CANopen [1-6] and others. Fieldbus, Modbus and Profibus are protocols based on traditional RS-485 [8], which suffers several disadvantages over CAN-based [2] networks such as a DeviceNET and CANopen. Painful problem of RS-485 is arbitration process on detected collisions via CSMA/CD [11] scheme which CAN solves in elegant way by introducing dominant/recessive bits and arbitration-free prioritization (see chapter 2.1 The underlying CAN technology).

Typical embedded network is a complex system of interconnected computer devices (or simply "nodes"). Each node provides some kind of service to the network, for example, one node can measure inputs and provide them to another node that controls the actuators. First node is acting as an input, while the second

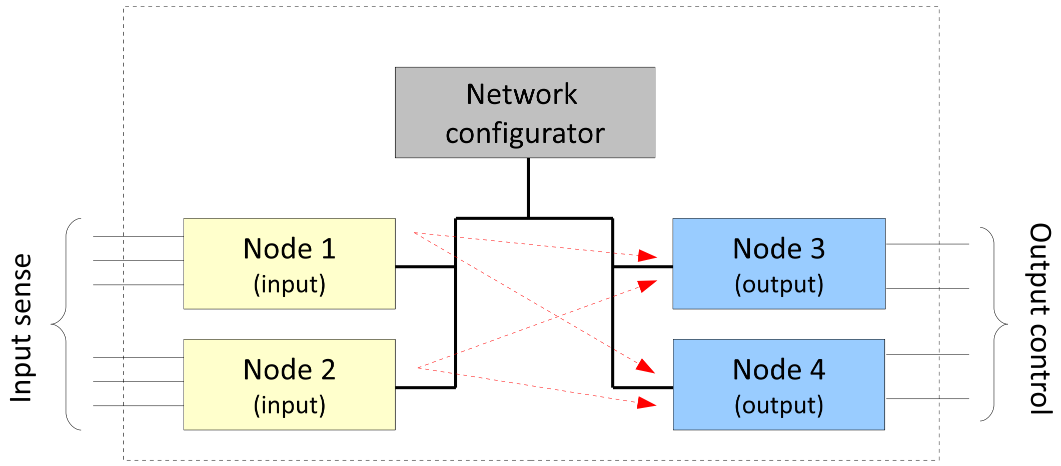


Figure 1: Embedded network example

node as an output. As there is no host controller present, there is no centralized intelligence (controlling function) and so the input/output nodes have to provide even some level of intelligent behavior to replace missing host controller - the intelligence is embedded in the network. Example of embedded network is shown on Figure 1.

Connected nodes may provide even other functionalities than just to sense inputs and control actuators. For example, we can often see network configurators and network managers who act as a network controllers and configurators.

1.2 Code requirements for embedded systems and CANopen

There are several differences between standard PC programming and programming in embedded systems - we have to consider restrictions regarded to CPU performance, program size, memory usage and to real-time processing.

1.2.1 CPU performance

CANopen is very flexible so amount of MCU time depends greatly on CANopen features compiled into the final program. Minimal CANopen implementation can be handled by simple 8-bit MCU with performance around 2 MIPS, but there are constraints coming out of using low performance MCU: device may not be able to handle high bus-loads, because processing time of a single message is longer than time between two consecutive CAN messages. Also, during the short message bursts on the bus we can observe more than 60% of MCU time on CANopen processing which limits host application. CANopen works as a service but it's also a part of the MCU program so it has to share MCU time with the host application. If CANopen takes major part of MCU time, the host application is suppressed.

1.2.2 Program size

Used program memory varies greatly with number of CANopen features compiled into the final program, with usage of different compilers and different optimization levels. Bootloader implementation with CANopen support may use about 2kB program memory while not truly implementing a CANopen compliant node. Minimum CANopen implementation can have from 4kB to 8kB and a size of full-blown implementation may vary from 20kB to 50kB or more [1].

1.2.3 Data memory usage

Volatile memory usage varies greatly for similar reasons like size of the used program memory. Typical minimum CANopen implementation uses from 100B to 200B of data memory while full-blown implementation may be using even more than 1kB [1]. Additional memory can be used for compiler specific reasons (like memory padding in structures) or for time-optimization reasons (using of 32-bit unsigned integers instead of 8-bit unsigned chars).

1.2.4 Non-volatile memory

CANopen allows save/load program configuration to/from non-volatile memory, typically using internal EEPROM [9]. Configuration size depends mainly on host application. Non-volatile storage typically contain two configurations - last saved configuration and factory configuration.

1.2.5 Real-time requirements

Real-time behavior is defined as guaranteed response time to an event. This is very important performance factor, because it allows whole system to work within specified time frames and therefore guarantees some level of performance. Certain applications may require guaranteed response time even within a fraction of millisecond. Portable CANopen implementations may have tough time guaranteeing those requirements because of different compilers used to compile the code.

1.3 Communication requirements for embedded networking

More and more complicated networks require more sophisticated solutions to provide information via the network. Many simple networks completely lack higher-level protocol, datatypes encapsulation, support for bootloader, plug&play and so on. While the network is simple and stationary, there is no need for any kind of generic communication protocol, but this is not a case of CANopen networks - they are generic in their nature. Having higher-level protocol on the bus means additional communication overhead at first, but added value may be great. Two most important features of generic higher-level protocols are: it provides a way to describe data and implements node-security mechanisms.

1.3.1 Higher-level protocol

Although low-cost microcontrollers utilize many serial buses, they usually lack buses with some higher-level protocol. Typical CAN peripheral provides some kind of data-link layer interface. It means that we can

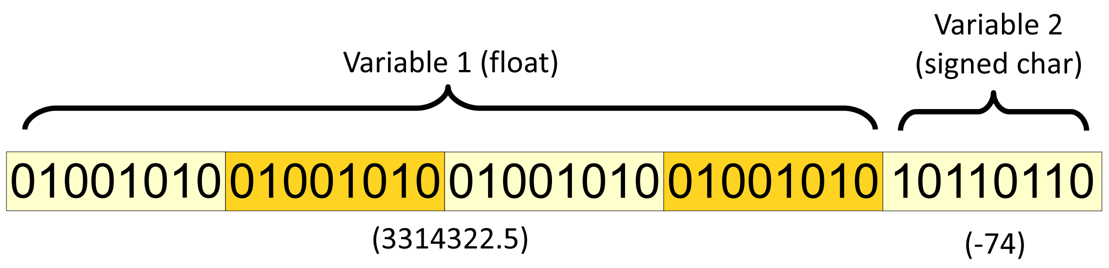


Figure 2: Datapacket variables encapsulation

send and receive databytes, but there is no description of what those data mean, there are no security checks implemented, data are not encapsulated nor described. When we start describing data, when we start adding CRC checksums, that's the point when we are implementing a higher-level protocol. *This kind of protocol don't care about how the data are sent, it cares about what they mean.*

When we start to create our own higher-level protocol, we are creating proprietary software which needs to be well documented otherwise no one with the exception of the author would be able to operate the bus. And that's only one of many pitfalls we can run into. Just to name other one - we have to design, create and test completely new kind of software, which takes time and money. Also, no third-party components can be used, because there are not any.

For these reasons it's always better to use some standardized and tested protocol - such as CANopen, or DeviceNET. They are well documented, proven to be working and third-party components are available.

1.3.2 Definition of data-types and process variables

As was already stated, devices on the bus need to recognize what the data mean. For example, when one device transmits a temperature, the receiver needs to know that the value is a temperature and the units are degrees. The situation is even more complicated when sending multiple variables in one datapacket - in order to understand to received content we need some knowledge regarding the datapacket structure - it means how the variables are organized within the message. Also, we need to understand the content of every variable - how is it encoded (whether the variable is sent in a little-endian or big-endian fashion for example).

The situation is outlined in Figure 2. The datapacket contains two variables named "Variable 1" and "Variable 2" - Variable 1 is a floating-point number, Variable 2 is a signed integer. There are several things to be seen. First of all, each variable has different size - float takes 32 bits while signed char takes only 8 bits. Furthermore, there are different encodings applied for different data-types - float is encoded as IEEE 754 [14], which is a standard for single-precision floating point arithmetic while signed char is encoded with two's complement [15].

1.3.3 Plug & play, hot-swapping

One of the most compelling and most complicated features is plug&play capability. It means that different devices understand each other and may work together [16]. Hot-swapping [17] is an extension to plug&play, it allows the devices to be connected without the need for bus restart.

1.4 Price considerations in embedded systems

Devices produced in large quantities are price-sensitive and so are their internal parts, including MCU. Although the host application may not require large processor performances, full CANopen does. This means that the price of the device is partly driven by CANopen protocol implementation, not by host application. This is not desirable feature and it has to be taken into consideration while developing any CANopen devices.

There is often used backdoor - CANopen is very scalable by the definition so it's compliant with CANopen specification when manufacturers implement into the device just very basic CANopen facilities, which won't make any special requirements for used MCU, but the device capabilities get limited. In the end, as a result of this trade-off between functionality and price the two different devices may not be able to successfully exchange information between each other.

1.5 Summary & thesis cornerstones

From CANopen protocol stack specification [2] and outlined features of embedded systems we can deduce common disadvantages of any CANopen implementation. Full CANopen protocol stack is generally too heavy and complicated to run on small 8-bit microcontrollers. As a consequence, manufacturer needs to use costlier MCU/MPU which pushes the price of the product. This is commonly overcome by sacrificing some of the CANopen functionality.

This thesis tries to propose a set of non-standard, out-of-box solutions which may ease the function of CANopen on cheaper devices.

2 CANopen overview

"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools"
Douglas Adams

CANopen is a higher-level protocol created on top of the CAN bus. *CAN itself provides a way to transfer data from one point to another, while CANopen sits on the top of this service and cares about meaning of transferred data.* CAN bus was introduced in 1983 by Robert Bosch. It was originally designed for automotive industry so each car produced in Europe since 1997 has at least one CAN embedded network inside. By the time CAN proved yourself in many other industries, especially in automation and control, medicine, military and so on. The very reason why CAN was so successful for past two decades is in used safety mechanisms and arbitration-free prioritization of the messages.

Using CAN in other industries than automotive lead to need for generic higher-level protocol usable especially in automation and in real-time embedded networks. By the time we have two dominant protocol stacks used on the market - CANopen and DeviceNET. DeviceNET is quite popular in America while CANopen is popular in Europe and Asia [1].

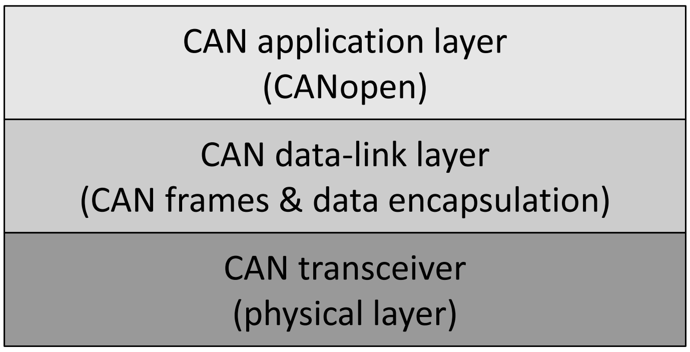


Figure 3: Layered structure of CAN protocol stack

2.1 The underlying CAN technology

CAN was designed in 1983 by Robert Bosch GmbH for automotive industry, where many small sensors needed to report small values frequently. As a result, CAN encapsulate data into short messages with maximum size of 8 bytes for the data plus overhead. Message overhead contains an 11-bit identifier, 15-bit CRC checksum, 6 bits for message properties, 7 bit for end-of-frame slot and 4 bits for delimiters. It makes 44 bits just for message encapsulation and creates 50% percent overhead on the message. This large overhead is a disadvantage sure, but the data are heavily checked for errors, which makes them very reliable. To give a flavor of its reliability, if a network with bitrate of 250kbps operates for 2000 hours per year at an average busload 25% an undetected error occurs once per 1000 years. There are several papers published on the topic of CAN reliability and performance (see [17], [18]).

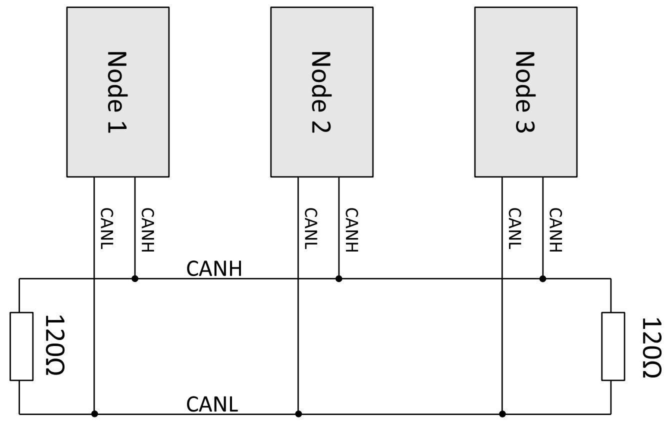


Figure 4: CAN bus interconnection example

CAN physical layer is based on ISO 11898 [19]. Typical physical medium consists of differential bus with twisted pair of wires. Open collector (OC) bus drivers are used to create recessive (log. "1") and dominant (log. "0") bus states. Open collector drivers have slow rising edge, which limits bus speed to 1Mbps, but on the other hand, OC drivers cannot be short-circuited like tri-state (3S) drivers can. Recessive state is set on the bus only if every connected node is transmitting recessive state, if a single device decides to transmit dominant state, recessive is suppressed over the dominant. This behaviour is called logical AND between the network nodes and it's the key for arbitration-free prioritization process.

If two nodes decide to transmit a message in the same time, CSMA scheme is applied - the transmitter is sensing the bus while transmitting identifier. If transmitted bit don't correspond to bus-sensed data (recessive bit of identifier was suppressed by dominant bit from other device), some other device is trying to send a message with higher priority. When this is detected, transmitter stops immediately and waits for other message to be transmitted. Beauty of this solution is in its discretion - higher-priority message is not interrupted by the collision.

Standard identifier size	11 bits
Extended identifier size (CAN2.0)	29 bits
Maximum message size	8 bytes
Maximum bitrate	1Mbps
PDO has significantly CRC field size	15 bits
Support for remote requests	Yes

Table 1: CAN-bus properties

2.2 CANopen concepts

CANopen is object-oriented protocol, objects are organized into a lookup table called Object dictionary (OD). Each entry has a 16-bit index and 8-bit sub-index (sub-indexes are used for structures and arrays) which allows addressing of 65 thousand objects. Entries from 0001h to 0FFFh are dedicated for datatype specification, entries on higher ID's are variables.

Object dictionary is used as an interface between the device and the network. Configuration of the device is done via SDO protocol (service data objects) and process data are transferred with PDO protocol (process data objects). PDO has significantly lower overhead over the SDO - PDO only provides selected data to/from the network, while SDO protocol allows accessing whole object dictionary from "outside".

Although object dictionary provides a way to address and describe all the accessible data, there is still need for the master device (or configuration tool) to recognize the structure of object dictionary. Different devices may have slightly different dictionaries (they provide different kind of data) and object dictionary is just too big for master to take "wild guesses". The solution is simple - the device has a few mandatory items in OD - mainly "device type" object and "identity" structure. Those entries allow the master device to recognize the slave.

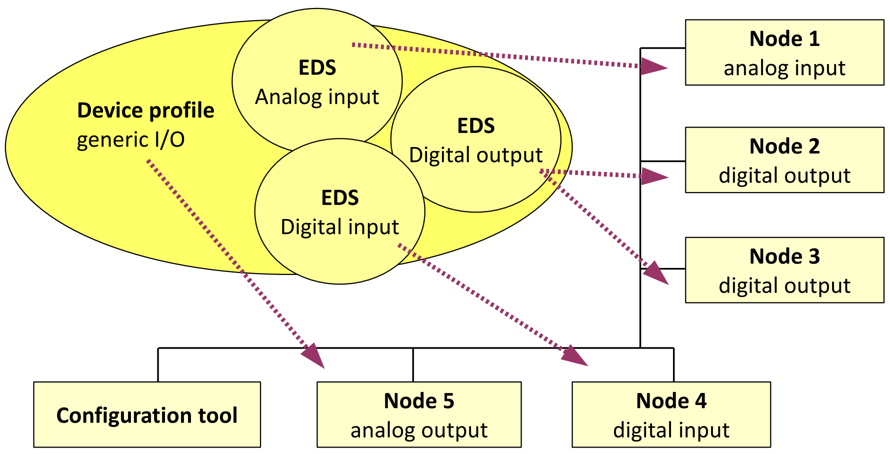


Figure 5: Usage of device profiles and electronic datasheets

2.2.1 Device profiles and Electronic datasheets

Electronic datasheets and device profiles are used by master or configuration tool to configure embedded network. Device type is variable placed on 1000h in the object dictionary and encoded as UNSIGNED32 (for datatypes specification see CiA-301 [2]). Lower 16 bits specify a device profile (DP), higher 16 bits contain manufacturer specific information regarding additional functionality. Device Profile is a specification of minimal functionality implemented in the device (and so in the object dictionary). If master recognizes given device profile, it has information about its purpose and has basic knowledge of object dictionary contents.

Identity object is a structure placed on 1018h in the object dictionary. It contains vendor ID, product code, version number and revision number. The master or a configuration tool may read this object to figure out specific information about the device, its purpose and object dictionary content. An electronic datasheet (EDS) may be applied if the device is recognized. EDS is a file similar to Microsoft Windows® .ini files with specific information about the device. There is also new XML-based version of electronic datasheets.

2.3 Detailed description of object dictionary

As was stated in the previous chapter, the object dictionary poses as an interface between the network and the device. It has 16 bit index, 8 bit sub-index and strict organization of index ranges for specific use (see Table 2).

Domain name	**Index range**
Standard types definition	0001h - 001Fh
Standard complex types definition (standard structures)	0020h - 003Fh
Manufacturer specific complex datatypes	0040h - 005Fh
Device profile specific scalar types definition	0060h - 007Fh
Device profile specific complex types definition	0080h - 009Fh
Communication specific object dictionary entries	1000h - 1FFFh
Manufacturer specific entries	2000h - 5FFFh
Device profile specific entries	6000h - 9FFFh
Reserved	A000h - FFFFh

Table 2: Object dictionary domains and ranges

Domains on indexes from 0001h to 0FFFh are used to define datatypes. Those entries are not actually present in the object dictionary, they are just used as a reference in electronic datasheets and device profiles. There are no actual data stored.

”Communication specific object dictionary entries” on range 1000h - 1FFFh are common parameters for all CANopen devices - behaviour of the device is manipulated via entries in this domain. ”Manufacturer specific entries” and ”Device profile specific entries” contain actual process and configuration data (they are application specific).

2.4 Mandatory object dictionary entries

It was already stated that CANopen is very flexible, but there are some minimal features common for all devices. Those features are: ”device type” object, ”identity” structure and ”error register” in the dictionary, support for SDO’s and PDO’s (at least partially), network management (NMT) and error control services (NMT Error Control). Mandatory dictionary entries are listed in Table 3.

Index	**Object type**	**Entry name**	**Datatype**	**Access rights**
1000h	Variable	Device type	UNSIGNED32	read-only
1001h	Variable	Error register	UNSIGNED8	read-only
1017h	Variable	Producer heartbeat time	UNSIGNED16	read-write
1018h	Structure	Identity object	IDENTITY	read-only

Table 3: Mandatory object dictionary entries

2.4.1 Device type

This entry is encoded as 32 bit unsigned integer, where upper half contains manufacturer specific information about the device capabilities and lower half contains a device profile identifier. This identifier is used to recognize device purpose, its capabilities and basic object dictionary structure.

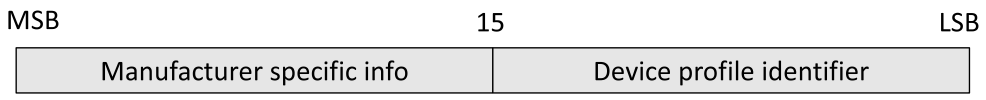


Figure 6: Structure of a device type entry

2.4.2 Error register

Error register is used to sign an error state of the device - internal errors are mapped in this entry. It contains a set of flags which describe a nature of occurred errors. See Table 4 for detailed description. Additionally, an emergency message is generated once per error event if EMCY object is supported by the protocol stack.

Bit	**M/O**	**Error type**
0	M	Generic error
1	O	Current
2	O	Voltage
3	O	Temperature
4	O	Commnucation error (overrun, bus-off)
5	O	Device profile specific
6	-	Reserved
7	O	Manufacturer specific

Table 4: Structure of error register (”M” - mandatory)

2.4.3 Producer heartbeat time

The ”producer heartbeat time” defines a cycle time to produce a heartbeat message. Time is a multiple of 1ms. For further information about heartbeat see chapter 3.5 - Heartbeat protocol. Note: There are two error control services - heartbeat and node/life guarding. One of them has to be supported by the device. Node/life guarding is not recommended for new designs.

2.4.4 Identity structure

This structure provides information about the device and its vendor. Supported entries are: vendor ID, product code, revision number and serial number. These variables can be used by master device or by configuration tool to get specific information about the device and its capabilities. See Table 5 for detailed description.

Index	**Sub-index**	**Entry name**	**Datatype**
1018h	0h	Number of supported entries (=4)	UNSIGNED8
	1h	Vendor-ID	UNSIGNED32
	2h	Product code	UNSIGNED32
	3h	Revision number	UNSIGNED32
	4h	Serial number	UNSIGNED32

Table 5: Structure of an IDENTITY datatype

Note: Identity structure is stored on index 1018h and it's of IDENTITY datatype. This datatype is specified on index 0023h (in domain Standard complex datatypes specifications) and table 5 is actually showing a structure of that datatype (for detailed description of standard CANopen datatypes see CiA-301 [2])

3 CANopen communication objects

"The best way to find yourself is to lose yourself in the service of others."
Mohandas Gandhi

Communication object is a network service realized with specific protocol. CANopen specifies 6 separate communication objects, each of them provides different service and uses different protocol. SDO object provides remote access to the object dictionary, PDO gives a fast way to share process-critical data, NMT is used to control and configure devices, TIME object provides reference time, EMCY object is used to inform devices about occurred errors and with SYNC object we can create synchronous communication.

Object	**Purpose**	**Availability**
PDO	Fast transfer of process variables	STATE_OPERATIONAL
SDO	Remote access to the object dictionary	STATE_PRE_OPERATIONAL, STATE_OPERATIONAL
SYNC	Basic network clock for synchronization	STATE_PRE_OPERATIONAL, STATE_OPERATIONAL
TIME	Provides a time reference	STATE_PRE_OPERATIONAL, STATE_OPERATIONAL
EMCY	Informs about occurred error	STATE_PRE_OPERATIONAL, STATE_OPERATIONAL
NMT	Network management, heartbeat	STATE_PRE_OPERATIONAL, STATE_OPERATIONAL, STATE_STOPPED
BOOTUP	Informs master about bootup	INITIALIZING

Table 6: Communication objects summary

3.1 Service Data Objects (SDO)

Service data object provides remote access to the object dictionary. It uses a Client/Server communication scheme where the owner of the dictionary poses as a server and the device with upload/download request poses as a client. As object dictionary entries may be of arbitrary size and maximum size of CAN message is limited, data may need to be divided into several CAN messages. SDO protocol solves this by transferring data in segments. Segmented transfer has large overhead so CANopen specification issues a special type of transfer called "block transfer" usable while transferring large blocks of data (like a program). Device which poses as a server uses a "Server SDO" (SSDO) object for communication, client uses a "Client SDO" (CSDO).

”SDO download” service starts when client is requesting a server to download data from the server’s object dictionary by sending a ”Initiate SDO download request” message where is specified which data are requested to be downloaded from the dictionary (by providing the index and the sub-index). Server responds with confirmation in form of ”Initiate SDO download response” message and then client periodically requests next segment by sending ”Download SDO segment” message. Transfer may be aborted at any time by sending ”Abort SDO transfer request”. SDO upload service is analogous to described SDO download service.

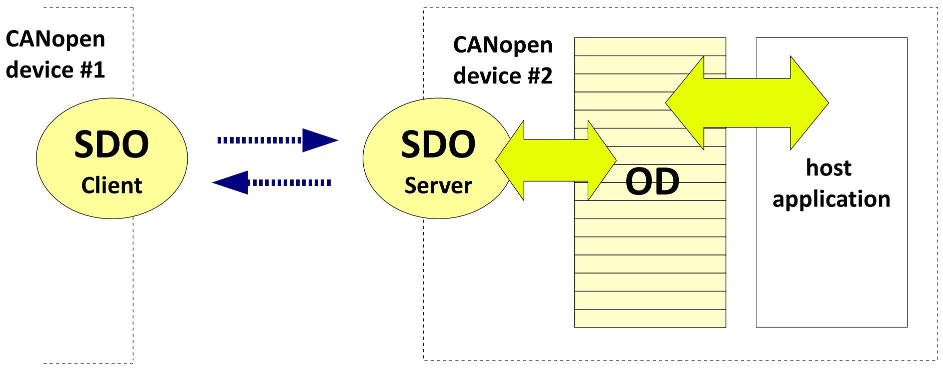


Figure 7: Client accesses server’s object dictionary

3.2 Process Data Objects (PDO)

Each device on the bus has some function. Function itself may be very general term, but from the network point of view a function is always expressed as input/output data (usually referred as a process data). Process data are critical in their nature so we need to transfer them as soon as possible with ideally no overhead. SDO services may provide this transfer, but considering SDO protocol overhead and that data itself have to be remotely requested by client, it may not be ideal choice.

Take the following example - when controlling production plant we may require to respond to some critical condition in fraction of millisecond, otherwise production plant may get damaged. If we use an SDO transfer, client have to request current process data many times in single millisecond to fulfill given condition. Due to maximum speed of 1Mbps on CAN it may not even be possible. If we use different method or triggering technique, we may actually reach those conditions. And that’s where PDO comes in handy.

PDO services are basically a shortcut for process data, they are designed to realize a real-time transfer with no overhead. PDO services have higher priority than SDO transfer and they support a number of different triggering techniques.

Two kinds of PDO’s are supported - a Transmit PDO (TPDO) and a Receive PDO (RPDO). A Transmit PDO is transmitting process data when triggered (using push model of Producer/Consumer relationship, see chapter 4), while Receive PDO is consuming data appeared on the bus (or remotely requesting them by using a pull model).

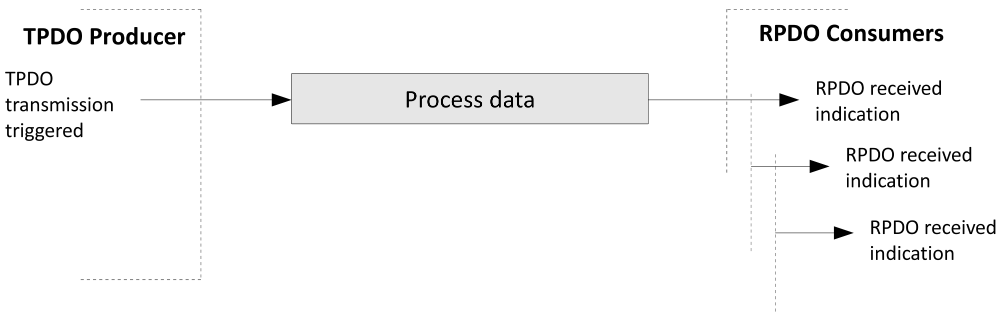


Figure 8: PDO transmission and reception - push model is used

The master device or configuration tool is mapping process variables into the TPDOs and RPDOs as a part of configuration process. Mapped data are transferred in a single CAN message, encoded variable-by-variable, bit-by-bit. Only limitation for mapped variables is in total length of the coded bitstream - size cannot exceed maximum CAN message size (8 bytes). As CANopen supports variables with arbitrary length, this may prove to be a bit problematic. For example, it's possible to map 64 variables of length 1 bit into a PDO (64 bits = 8 bytes).

Triggering techniques

- **Event driven** TPDO transmission is triggered by the occurrence of internal event, or for synchronous PDOs by the expiration of SYNC transmission period (PDO may be configured to transmit every n occurrences of SYNC message, for further information see [2] or section 3.3 - Synchronization object).
- **Timer driven** TPDO transmission is triggered by the expiration of presetted time period. For example a PDO may be transmitted each n milliseconds.
- **Remotely requested** A transmission of TPDO was remotely requested by other device using RTR message and pull model of Producer/Consumer scheme (see Chapter 4 - CANopen communication model).

Synchronous and asynchronous TPDOs are distinguished. Synchronous TPDOs are triggered by reception of a SYNC message while asynchronous TPDOs are triggered internally as a response to occurred event, by timer expiration or by remote request. Although asynchronous event-driven TPDOs are still the fastest way to inform about occurred event, synchronous TPDOs deliver advantages in form of bus-load predictivity. Created latencies may not be critical for some types of process data.

3.3 Synchronization object (SYNC)

Synchronization object provides a basic network clock. The SYNC message is transmitted periodically by SYNC producer and received by devices with support for synchronous TPDOs. TPDO may be configured to trigger transmission each n occurrences of SYNC message.

Synchronization message has zero length (it doesn't carry any data), message is recognized due to message identifier. SYNC producer uses push model to transmit message on the bus periodically. Length of the transmission period is set in the object dictionary with "Communication cycle period" entry (index 1006h). Value is a multiple of one microsecond.

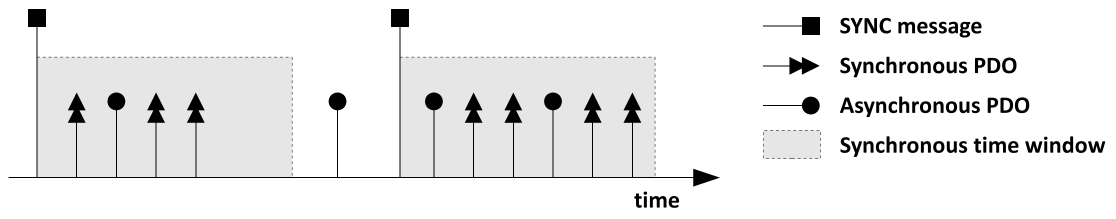


Figure 9: Synchronous and asynchronous PDOs

When synchronization message occurs on the bus a time window with defined length is created. All synchronous PDOs are transmitted inside this window - when this happens, devices are fighting for the bus and many messages may be transmitted one after another. This generates a short bursts of 100% bus-load.

3.4 Emergency object (EMCY)

Emergency message is generated when internal error on the device occurs. There may be several reasons - host application error, communication error (message overrun), voltage, current, temperature and so on (CANopen specifies about twenty types of errors and exceptions). EMCY message is produced once per error and contains an error code and the error register (object dictionary entry 1001h). When error condition is gone, another EMCY message is produced with error code 0000h (error reset). Error register is a part of the EMCY message so EMCY consumer has a way to recognize whether there are still pending errors.

3.5 Timestamp object (TIME)

Timestamp object is providing a time reference to the network. Push model is used for transmission, the message itself has high priority and contains only a TIME_OF_DAY structure (specified in Listing 1).

Listing 1: TIME_OF_DAY structure

```
/* TIME_OF_DAY structure (source CiA-301) */
typedef struct
{
    UNSIGNED28     ms; /* time in miliseconds after midnight */
    VOID4          padding;
    UNSIGNED16     days; /* number of days since 1/1/1984 */
} TIME_OF_DAY;
```

3.6 Network management (NMT)

Every CANopen device has to implement a state machine. Network management services allow master device (or configuration tool) to remotely change the state. The device is set into the ”initializing” state after a hardware reset and when initialization process is done the state is automatically changed to ”pre-operational”. Special BOOTUP message is generated on this transition so master is informed.

The device may be reconfigured during the ”pre-operational” state, SDO services are available. Typically, master is going to reconfigure the device and then change state to ”operational” or ”stopped”. State transitions may be committed only by the master device or by a host application itself. State transition is also a by-product of a hardware reset.

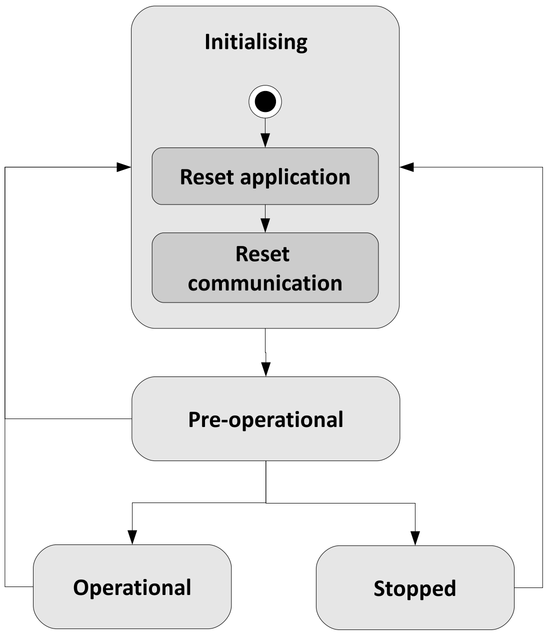


Figure 10: CANopen state machine overview

Availability of communication objects depend on actual state of the device. All communication objects are available in ”operational” state, PDO is disabled while in ”pre-operational” state and only NMT services are available when ”stopped”.

3.7 Heartbeat protocol

CANopen supports two types of error control services - the Heartbeat and Node/life guarding. Heartbeat is a preferred way, node/life guarding is not recommended for new designs. Heartbeat protocol is a part of the NMT communication object. The protocol is using a push model of producer/consumer scheme. Each device in the network produces a heartbeat message within specified time frame. Consuming device (typically a master) is controlling the whole network for proper heartbeat function. This concept is similar to watchdog timer used as a diagnostic tool on microcontrollers - if heartbeat producer won’t produce a message within specified time an exception will be raised.

4 CANopen communication model

"Beware of bugs in the above code; I have only proved it correct, not tried it."
Donald Knuth

Communication model specifies approaches for communication in the network. CANopen supports synchronous and asynchronous communication (see SYNC and TIME communication objects). Synchronous messages are transmitted as a response for received SYNC message, asynchronous messages may be transmitted at any time. Due to event character of the synchronous communication, it is possible to set an inhibit time for transmitted objects, so no starvation occurs on the network (situation when a low priority messages are suppressed by a stream of higher priority messages). Different communication objects use different communication relationships with respect to their functionality.

Model type	**Usage**	**Objects**
Master/Slave	Only unconfirmed services	SDO
Client/Server	Mostly confirmed, complicated protocols	NMT
Producer/Consumer	"Give or take" approach, simple protocols	PDO, TIME, EMCY, BOOTUP, Heartbeat

Table 7: Communication model usage

4.1 Client/Server relationship

This relationship specifies peer-to-peer (P2P) communication between two nodes. Client is requesting server with some task (usually to upload or download data) and server performs requested task and responds accordingly. Please note that single device may act as a server and client simultaneously - he can request data from other devices (pose as a client) and also provide data on request (pose as a server).

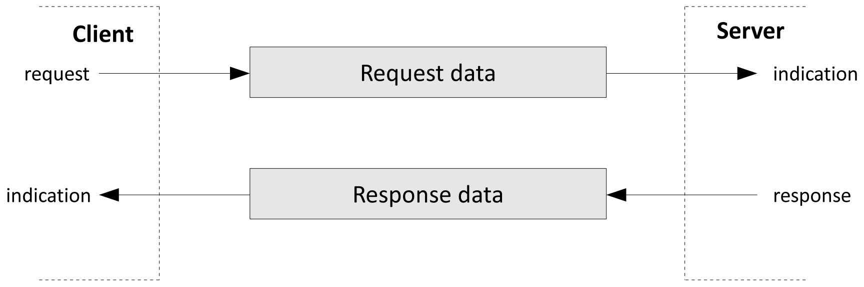


Figure 11: Client/Server communication relationship

This relationship scheme is used by SDO protocol - client is requesting to upload/download data to/from the object dictionary on the server. At least one SDO server is mandatory for every CANopen device (called "Default SDO").

4.2 Master/Slave relationship

There can be only one device within the network posing as a master for specific functionality, other devices are always slaves. The master start communication with the slave devices - he can give a direct command, or request data with CAN remote request message (RTR frame). Relationship may be unicast (peer-to-peer, master-to-slave) or broadcast (master-to-slaves).

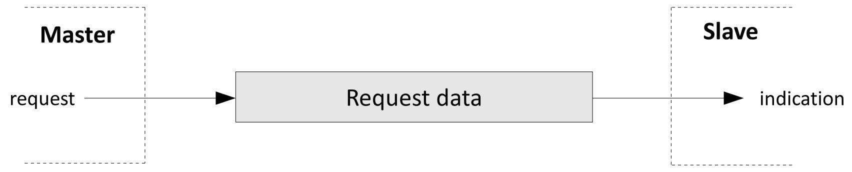


Figure 12: Unconfirmed Master/Slave relationship

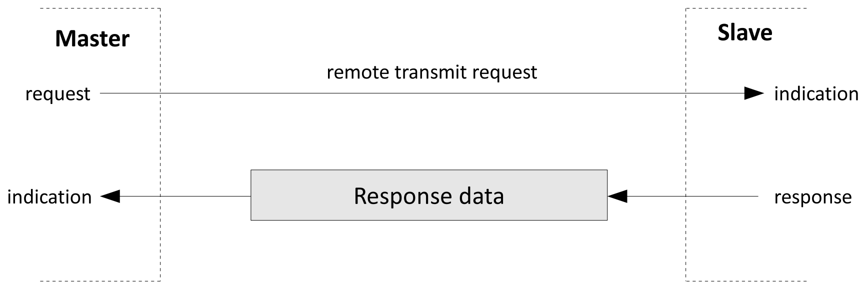


Figure 13: Confirmed Master/Slave relationship

Master/Slave relationship is used by NMT protocol for network management. It allows the master to setup the slave devices, start/stop selected nodes, request restart and so on.

4.3 Producer/Consumer relationship

This relationship implements push/pull communication model. Push model is issued by a producer who push the data onto the network and zero or more consumers who internally indicate reception. Pull model is using remote requests (RTR messages) to pull the data from the consumers.

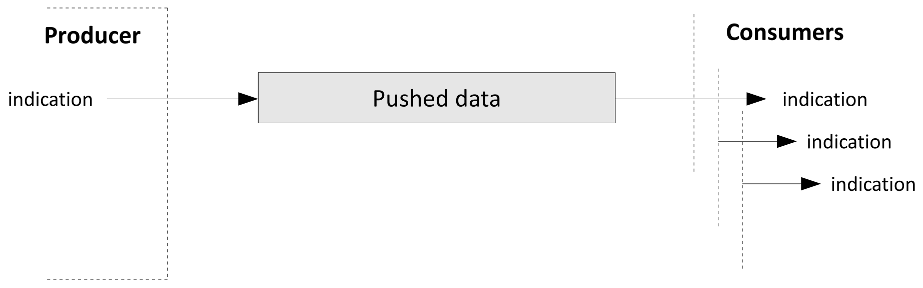


Figure 14: Producer/Consumer relationship - Push model

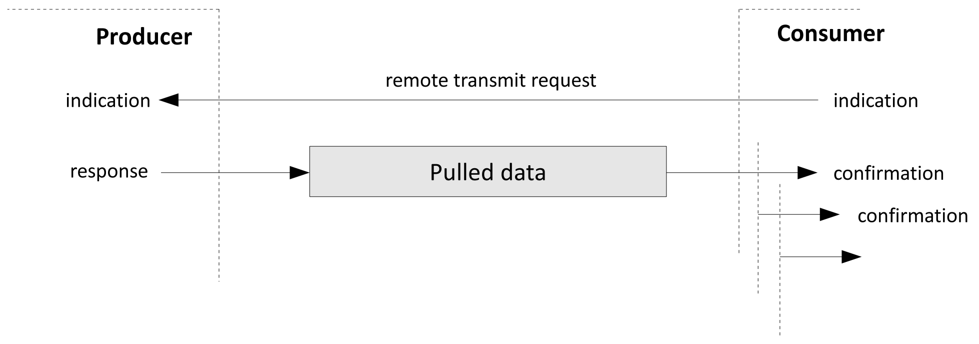


Figure 15: Producer/Consumer relationship - Pull model

Push model is used at least partially by all CANopen protocols with the exception of SDO and NMT. Bootup protocol and protocols attached to PDO, EMCY, SYNC, TIME and heartbeat services are using this model. Pull model is used for remotely requested PDO objects, although it is not recommended to use this triggering technique in new designs. The difference between pull model and confirmed master/slave relationship is in response - pull model may have more than one response consumers. Request is generated by a consumer and addressed to a producer (in other words one of the consumers is requesting production).

5 CANopen software architecture

"There is nothing worse than a brilliant image of a fuzzy concept."
Ansel Adams

There are several constraints limiting actual CANopen implementation. They always create a trade-off between required device capabilities and the price. Proposed concept contains implementation design specialized for low-performance microcontrollers with respect to their common properties. Two architectures will be outlined in this chapter - architecture designed to run with the host application on the same processor and architecture designed to run on separate coprocessor. Although both architectures are built from the same blocks, there are important differences and we have to take them under consideration.

This software design has generally well defined interfaces between layers and functional blocks. It allows simple replaceability of selected parts of code, which gets useful especially when implementing device-specific optimizations. Programmer only needs to replace selected part and keep the same interface. Then he has secured correct function of the program.

5.1 Shared CPU architecture

Shared CPU architecture covers functional design on a device where the host application and the protocol stack run on the same CPU and therefore the CPU time is multiplexed between them. The protocol stack is periodically invoked by the host application to ensure proper function of the software while all the data are exchanged via the object dictionary. The situation is outlined on Figure 16.

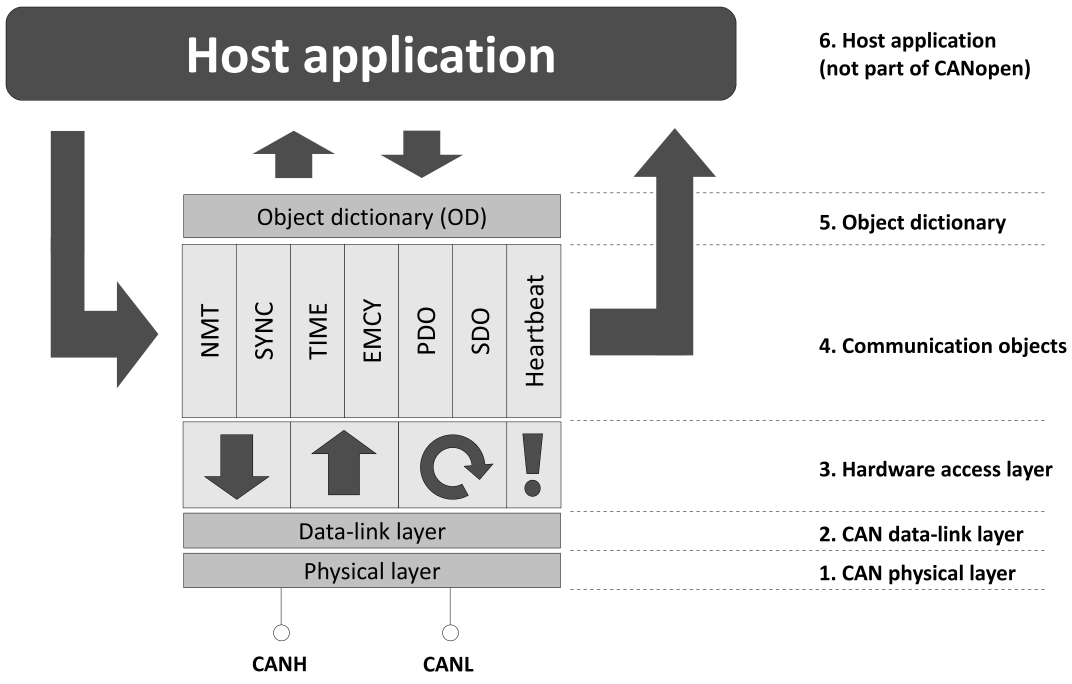


Figure 16: The protocol stack design

This design uses layered architecture with five distinct layers. Each layer uses services of the lower layer to provide an extended service for a higher layer. It means that when we want to send some data on fourth layer (communication objects), we have to call specific function on the third layer (hardware access layer). Third layer then uses services of second layer to request a data to be sent while giving more specific information about the message. Then the second layer (CAN data-link layer) encapsulates data into the message, adds a CRC check (among other things) and requests the first layer (CAN physical layer) to actually transmit the message. As you can see while the information is "bubbling" through the layers it's gathering more and more specific shape and so finally in the end the information is completely transformed into a CAN message.

Lowest two layers (CAN data-link layer and CAN physical layer) are actually implemented on hardware level. As a consequence the third layer needs to be device-dependent, so it has to use a device specific code to manipulate registers of a CAN peripheral on the microcontroller. It is desirable for the third layer to be well specified and very thin in the meaning of the source code size because it has to be rewritten for each new device it works on. Using of a device-specific code is a thing we cannot avoid by any means.

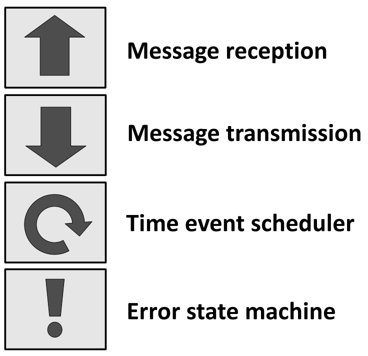


Figure 17: Hardware access layer description

Hardware access layer doesn't implement any actual part of CANopen protocol stack, it just provides a number of services needed by all communication objects (PDO, SDO, NMT and so on). Although it's not directly related to CANopen it has to be widely inspired by CANopen communication model and by common requirements of CANopen communication objects. Only in this way it can provide services optimized for use by CANopen protocol stack.

Hardware access layer services

- **Message transmission** is a service implemented to provide a standard way to transmit a message. When one of the communication objects wants to send a message it has to allocate an empty transmit buffer first (which may prove to be problematic if the buffers are full). If the buffer was successfully allocated. the content may be stored into the buffer and then requested to be transmitted

- **Message reception** service filters and recognizes received message so it can invoke according communication object with according event handler.

- **Time event scheduler** is scheduling a time events. Communication object has to describe when the event should be raised and which action should be taken.
- **Error state machine** is providing a standard way to propagate internal errors and errors detected on the CAN bus. CANopen protocol stack requires some level of error checking so this service provides that feature. It is not necessarily connected to EMCY communication object (See chapter 3.6 Emergency object).

Fourth layer (communication objects) is implementing actual CANopen protocol stack with all its blows and whistles. It is periodically invoked by host application so each communication object can perform required tasks.

Fifth layer is the object dictionary itself. It works as an interface between the device and the network - both communication objects and host application may manipulate with the data. Object dictionary may be stored in non-volatile memory (EEPROM/Program memory) with only read-access. Object dictionary content is referencing variables to the RAM memory or contain constants directly.

5.2 Using a coprocessor

A coprocessor is a peripheral used to offload main processing core (CPU). Coprocessor design differs from manufacturer to manufacturer, it may not generally be fully functioning CPU, it may lack some of the basic instructions or have instruction set optimized for one specific purpose [20]. History of coprocessors is very long, starting with popular Intel 80x86 processor series and ending at CUDA technology and PhysX physical engine. In embedded systems, coprocessor usually works as a substitute processor with limited instruction set designed to off-load the main processor [20]. Main processor may fetch instructions to the coprocessor directly, or assign a program to run, or process only interrupts.

5.2.1 Advantages & disadvantages

It allows us to create a quick CANopen implementation on relatively cheap devices (there are 16-bit microcontrollers with a coprocessor available). The coprocessor application may act as a virtual CANopen peripheral - CPU overhead over CANopen is none (or at least minimal). Device response time is becoming limited only by used coprocessor frequency, the host application is not limiting CANopen anymore.

On the contrary, using a two processing units bring issues with parallel processing. Sharing system resources and exchanging information (playing "ping-pong") requires special hardware support in form of software semaphores and hardware access locking. Debugging a program who's running partially in the CPU and partially in the coprocessor may not be simple, or even possible.

5.2.2 Coprocessor running a parallel program

Microcontroller with this type of coprocessor has practically two processors inside. If both processors are equal (have the same capabilities, instruction set and boot techniques) the device can be considered as a dual-core. The other common case is that a coprocessor is optimized for some kind of special application (math calculations, video processing), or two devices are present in a single package (like DSP and MPU).

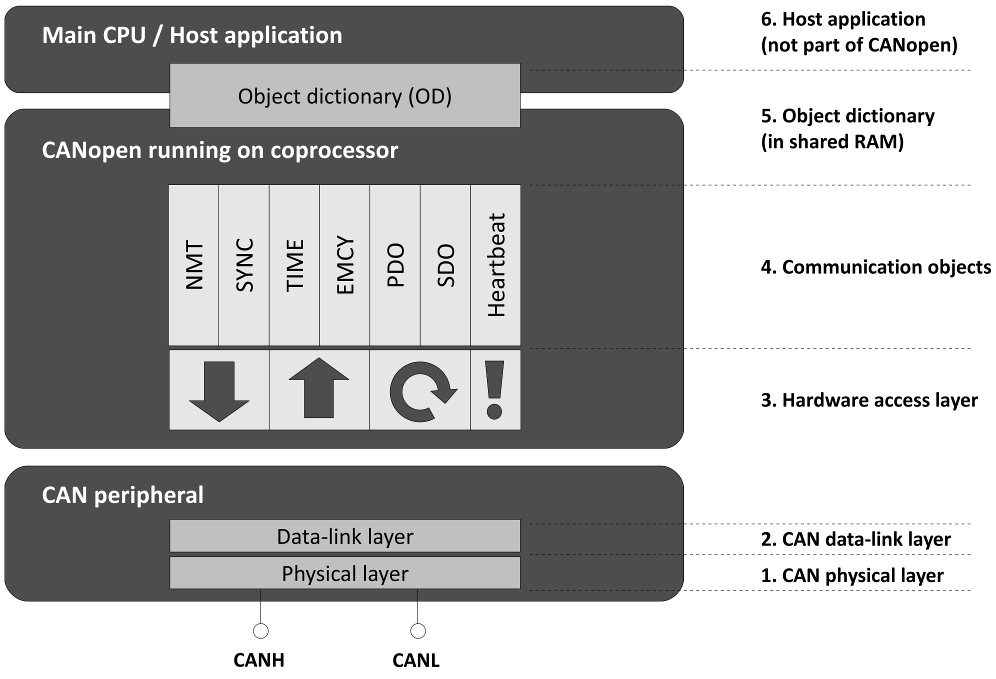


Figure 18: CANopen running on a coprocessor

A CANopen architecture for this kind of coprocessor doesn't differ from the case described in chapter 5.1 Shared CPU architecture. The only difference is, that there is no host application on the second processor, only CANopen protocol stack.

5.2.3 Coprocessor for interrupt handling

Simpler microcontrollers utilize coprocessor in simpler form - coprocessor is just a bus peripheral designed to handle interrupt service routines. The main CPU has to configure and manually enable coprocessor operation. There is no actual program present, only a number of interrupt vectors mapped to a program memory. When the interrupt occurs, the coprocessor will start processing instructions on given interrupt vector. When it gets to "return from interrupt service routine" instruction the coprocessor turns idle.

Software architecture for this kind of coprocessor is different from previous. The protocol stack is no longer needed. Normally, we want the interrupt service routine (ISR) to be finished as soon as possible and thus we move processing of any time consuming algorithm into a protocol stack (which is invoked periodically by the host application). In this case, the processing is done entirely inside the interrupt service routine and thus there is generally no need for protocol stack - the responses are generated immediately.

Note: Some CANopen solutions may require a multi-level maskable interrupts. This topic is not in the scope of this thesis. For further information see [21].

5.3 Communication object model

There are six distinct communication objects (NMT, TIME, SYNC, EMCY, PDO and SDO). Each provides different service and can be configured as master or slave (producer or consumer respectively). While there are differences between any two communication objects, we can still find many similarities. This feature calls for some kind of standardized behaviour which will simplify the code and allow applying optimization techniques.

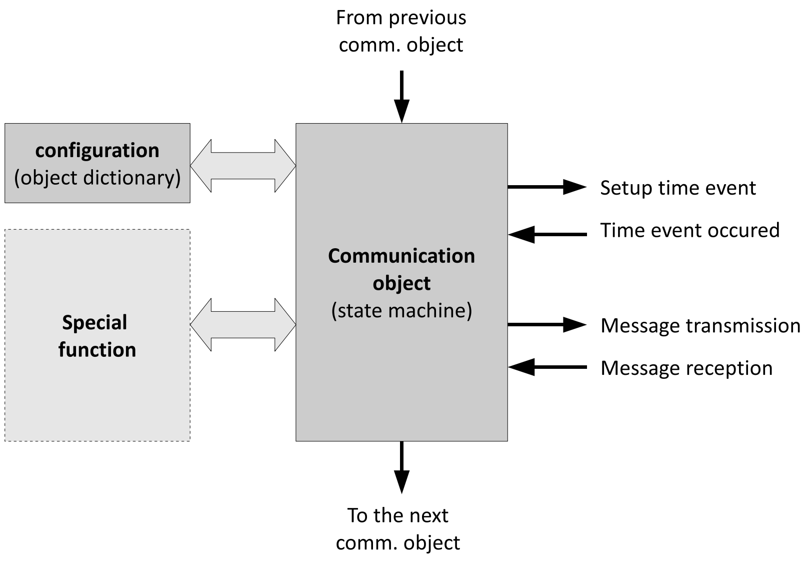


Figure 19: Communication object model

Model of a communication object is shown on Figure 19. Core of any communication object is in the state machine which implements required behaviour. Configuration of the state machine is stored usually in the object dictionary so configuration can be modified remotely with the SDO services. As was stated above, there is always some kind of special functionality attached to the communication object (NMT communication object changes device state, RPDO stores received process data). Every communication object needs the ability to transmit/receive data and many objects also require timing facilities.

Communication object may be invoked in two ways. First way is by a host application while "bubbling" through the protocol stack and second by the interrupt request generated when a new message was received (with one small exception of CAN error interrupt which may invoke EMCY communication object).

When invoked by an interrupt source we usually need to process the message as quickly as possible, so we cannot afford to schedule a time event or to send a response message. We usually just process received data and change state machine if needed. All the time consuming work is done when invoked by the host application - messages are transmitted and time events scheduled. As a consequence, CANopen is a bit slower with responses. Please note that it isn't necessarily a disadvantage, too short response times may create short bursts of load on the bus which may overwhelm connected devices.

Besides, if CANopen would work entirely inside the interrupt service it may disrupt the host application. For example when controlling a BLDC motor we need a precise timing of output pulses, but when CANopen takes let's say 50% of CPU time, the timer would not be able to inform the host application about occurred events - CANopen interrupt service routines have taken away half of the CPU time. Multi-level interrupts would solve this, but their using is tricky and it is not recommended to use unless you really know what you are doing [21].

Note: When using a coprocessor it is desirable to do all the time-consuming work inside the interrupt. Described restrictions don't apply in this situation. Coprocessor is processing the interrupt, not CPU, so host application is not disrupted by any means. Very fast CANopen implementation can be created in this way. See chapter 5.2 for further information.

6 Hardware access layer design

"If at first, the idea is not absurd, then there is no hope for it."
Albert Einstein

Hardware access layer provides a number of services to the protocol stack. The protocol stack performance strongly depends on implementation of this layer. Several out-of-box solutions are introduced in this chapter, they came from deep analysis of communication objects and communication model. Used approaches are result of multi-objective optimization which is vital for this software to work on cheaper microcontrollers.

6.1 Message transmission

A series of steps have to be done when we want to send a message. At first, we need to allocate an empty transmit buffer, or empty message box (sometimes called also "mailbox"). The device typically have only three transmit buffers, but extension in the form of many message boxes can be created in RAM and connected to the CAN peripheral. When we want to send a message in the message box, data from RAM are copied to the empty transmit buffer using a special DMA channel. This implementation is targeted specifically to simple devices that may not have a DMA controller inside, so the design is created in the way in which it can cope with at least three transmit buffers

Note: At least three transmit buffers are needed in order to be able to send uninterrupted sequence of messages, see CAN2.0 specification for further details [19].

6.1.1 Problems to solve

When we want to allocate a buffer, there may not be any available at the moment. This has to be somehow resolved, we cannot let messages disappear due to full hardware resources and we still have to keep the real-time requirements of the system. The most simple solution is to interrupt the allocation request and don't care. This is not really a solution, it just delegates the problem to the upper layer. If we really want a solution, we need to handle the problem directly, we need an algorithm which will allocate the buffer as soon as possible, but not sooner. The messages in the transmit buffers are all waiting for the bus (they were already requested for transmission), so sooner or later the empty buffer will appear.

Other issue is with the error detection mechanism - when we request a message to be sent the message may not be transferred for several reasons (bus-overload, bus-off). We need a way to figure this out somehow. Traditional solution is to setup a timeout for the message to be transmitted (it is recommended to use timeouts even by official CANopen specification). This concept of timeouts is proven to be working, but it creates additional CPU overhead and additional requirements for the data memory if there is no hardware support in form of some additional timers (which may not be our case). This design introduces a concept of tryouts, instead of timeouts. When allocation request is received and all hardware and software resources being used at the moment, the allocation algorithm starts to repeat itself until try-counter overflows. In this way, when the bus has issues (is heavily disturbed, or disconnected) all the transmit buffers are getting full after a while and therefore at some point, the problem is detected. In the nature of the problem, the tryouts detect the same type of issues as timeouts, but they don't require any additional CPU overhead or additional memory or special hardware.

6.1.2 The transmission balancer

As was stated above, in some cases we have only three transmit buffers and no message boxes available. Common solution to this problem is to create a software buffer with own message boxes. This may sound simple (we just need to replace a DMA channel with CPU and according software) but it's forgotten to keep the real-time behaviour - message boxes may contain several messages requested for transmission. It raises a question - which message has the priority? Which message should be transmitted first? Message boxes implemented on hardware usually solve this automatically with internal prioritization system, which is exactly a thing we need in a software solution to keep that real-time behaviour.

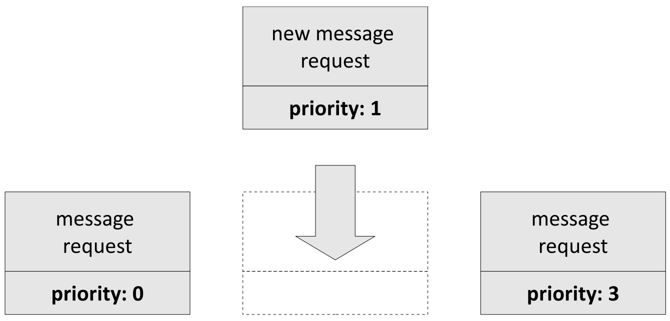


Figure 20: The basic principle of transmission balancer

Prioritization system usually requires some kind of algorithm with linear asymptotic complexity (search algorithm), which may be just too time consuming and therefore not affordable while feeding a transmit buffer with new data (this is usually done in the interrupt service routine). Listing 2 illustrates the problem of searching for the right message box and copying it from the RAM to the CAN peripheral. Both used algorithms have linear asymptotic complexities. Please note that this is a software solution - hardware implemented message boxes are doing this automatically.

Listing 2: Sending a message box

```
/** Sent event handler. This function should be called by the
 * CAN driver when a message is sent. */
__interrupt void msgsent(void)
{
    /* msgbox_t is a structure containing the messages in RAM */
    msgbox_t * msgbox = pick_next_msgbox(); /* Search alogrithm (O(n)) */
    copy_msgbox_to_txbuffer(msgbox); /* Copy algorithm (O(n)) */
    request_to_send(msgbox);
}
```

To keep the real-time behaviour we need a solution which sorts the messages by the priority when allocating a buffer - we need a schedule. Buffers are allocated usually while ”bubbling” through the protocol stack and therefore this operation is not time critical. In other words, we just move the prioritization problem from interrupt service routine to a protocol stack, where the time is not the essence. On the contrary, creating a schedule of data structures with size of a message box (about 14 bytes each message box) may be just too time consuming even when the time is not critical.

Introduced solution of transmission balancer is leaving the concept of message boxes entirely. It uses a feature of the communication objects outlined above - they work as state machines. When allocating a new message, it is not required to do the allocation immediately. Message is required to be allocated when there is an empty transmit buffer and no other higher-priority message pending. In other words, we can register the request and get back to it when hardware resources are available.

When allocating we just need to know the message identifier (to determine a priority) and a pointer to a callback function. When callback function is called, it may directly store data into the transmit buffer - message is created and stored immediately so no additional memory is required for the message boxes.

Listing 3: Message request structure

```
/* Define the type of standard callback */
typedef void (* __near fcn)(key8_t key);
/* Define the standard key */
typedef unsigned int key_y;
/* Define the standard identifier */
typedef unsigned int cob_id_t;

/* tx balancer message request structure */
typedef struct {
    void * next;      /* Pointer to the next entry */
    cob_id_t value;   /* Mesage priority (same as message id) */
    evhandler_t fcn;  /* Callback (function pointer) */
    key8_t key;       /* state machine key (used by comm. objects) */
} msgreq_t;
```

Implementation of transmission balancer uses sorted linear linked-list and internal callbacks in form of function pointers. When we add a message into the transmission balancer the message is stored in a sorted manner. Linear linked list is used to simplify the sorting algorithm, but it requires additional memory to keep the links between the entries.

When a content of the transmit buffer is transmitted an empty space is made and we can setup next message there. As was outlined on Listing 2, there are two steps needed. At first, highest priority message needs to be chosen - as transmission balancer is sorting the list of requested messages, highest priority message would be the very first entry in the list. To fill the buffer with content we have to call the callback function which will do the task. A big difference between a message boxes and the transmission balancer is in the approach to fulfill the content. Message boxes are using two-step approach - the message is created somewhere else and then moved to the transmit buffer. Transmit balancer on the hand stores the message directly to the buffer (if possible, otherwise it postpones the allocation). Memory for the message boxes is not needed, because the data are stored to the buffer *a priori*, not *a posteriori* as it does the solution with message boxes.

Please note that message boxes may be implemented on hardware level, or manually as a part of CAN driver and therefore transmission balancer may be suppressed if needed.

6.1.3 Request-to-send approaches

Message can be requested to be transmitted in three distinct ways. The first and most simple way is to directly request the message to be sent with no addendum. The second way is designed for confirmed messages. When we transfer this kind of message we expect to receive a response. Some action should be taken if the response is not received, so this approach automatically sets a timeout and calls a callback function if message is not received in given time. This behaviour is achieved by creating a time event with the time scheduler (described below). On the other hand, if the message is received within given timeout, the time event is disabled (time scheduler is designed to provide a quick way to disable a time event, see chapter 6.3).

Third approach allows us to create a consecutive stream of messages which is useful for transferring large blocks of data (SDO block transfer). Communication object can register a sent notification on a message in a transmit buffer. When the message in the buffer is sent a callback function is called and it may directly store new message and request it for the transmission.

Note: Sent notifications are handled in the "message sent" interrupt together with the transmission balancer. The balancer has priority over sent notifications - if the balancer is not empty a message request (taken from the balancer) is stored into the buffer and the sent notification is pushed into the transmission balancer (kind of similar to a "castling" move in a chess game). This solution is not perfect, pushing a message into the transmission balancer is adding/sorting algorithm with linear complexity, but it is still better than preferring a message with no regard to its priority.

6.1.4 Implementation of message transmission service

Although transmission service may look like complex monstrosity, it is actually quite small, simple and elegant. From software point of view, the message needs to be allocated, filled with the data and requested for transmission. This defines three separate parts of the service. Message allocation is done on the third layer, between communication object and CAN data-link layer, content is created by the communication object itself and so is the request for transmission when the message is prepared (both is done in the same callback function).

Allocation is outlined on the Figure 21. The algorithm tries to put the message directly to the transmit buffer and if it isn't successful, the transmission balancer is used. If even transmission balancer is full on first try, the algorithm tries again and again until it is successful or try-counter overflows. The program basically runs in a loop until an empty place is made in the balancer - until at least one of allocated messages is sent. Tryouts are last measure possibility, the algorithm will not run into them unless there is a problem on the bus. This algorithm from its nature won't work inside the interrupt service routine.

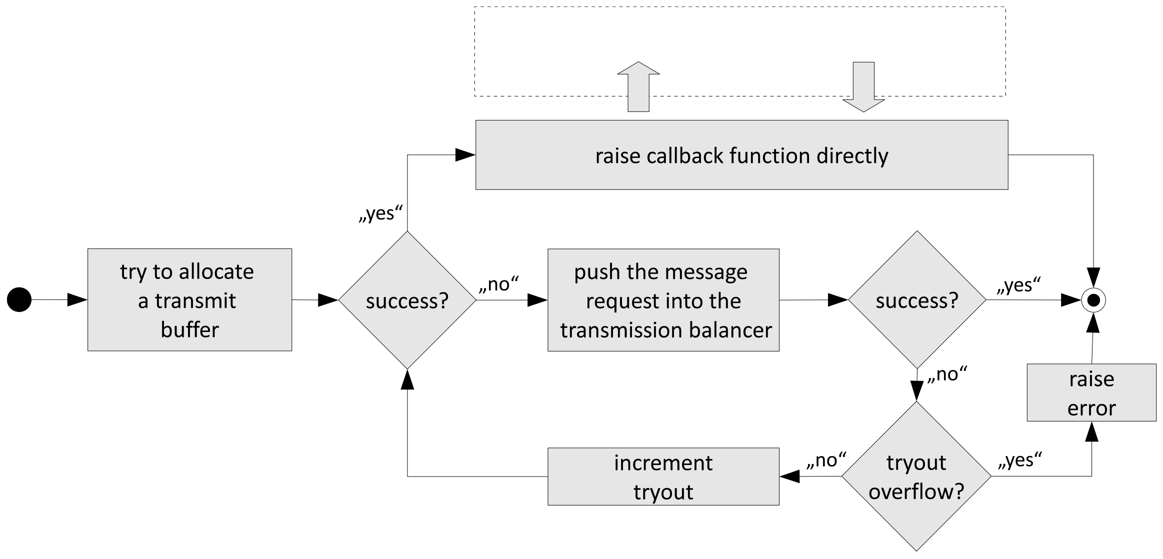

raise callback function directly
„yes“
try to allocate a transmit buffer
success?
„no“
push the message request into the transmission balancer
success?
„yes“
„no“
raise error
increment tryout
„no“
tryout overflow?
„yes“

Figure 21: Message allocation with transmission balancer and tryouts

When a place in the transmit buffers is made (i.e. message which was occupying one of the buffers was sent) the transmission interrupt is called. In that moment, we can fill empty buffer with a new message from the transmission balancer - highest priority message request will be pulled from the balancer, filled with contents and requested for the transmission.

Feeding the message with the content is done from the callback function. Several functions like save-byte(..) and rts(..) are defined for this purpose.

6.2 Message reception

When a new message is received we run some kind of conditional algorithm to figure out the purpose of the message, then we can invoke appropriate handler. This algorithm is usually created from switch-case structure with linear asymptotic complexity (O(n)). Because this is quite straight forward, we can define the interfaces and come up with another solution, possibly quicker, centralized and with fewer requirements for the program memory.

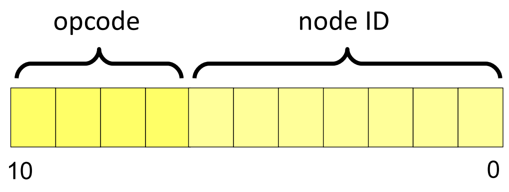


Figure 22: Predefined connection set - identifier content

Switch-case takes large piece of program memory and the algorithm is relatively slow due to its linear complexity (as was stated above). Two distinct solutions are proposed as a replacement. First solution is based on CANopen's predefined connection set (PCS). It has the best possible asymptotic complexity

(O(1)) and uses very little memory. This solution works only for slave devices and doesn't allow making changes in the connection set, which is not entirely CANopen compliant. Second algorithm is based on sorted translation table stored in the program memory. The algorithm is quicker (O(log(n))) and allows changes in the connection set.

Note: Both algorithms can cooperate with filtering techniques applied on CAN data-link layer (CAN filters and masks) as recommended by Robert Bosch's CAN2.0 specification [19]. Their configuration is hardware-specific and doesn't fall into the scope of this thesis.

6.2.1 Predefined Connection Set

CANopen specification (CiA-301) defines a "Predefined connection set", it is a lookup table which connects identifiers to communication objects. This connection set is used at least after bootup, specific identifiers can be altered by the master device. It uses standard CAN identifier with 11 bits and although connection set can be changed to extended identifiers with 29 bits, it is not recommended. Using a long identifier creates additional message overhead (about 65% minimum) and weakens CRC security check.

Each identifier has two parts - opcode and node ID, opcode (operation code) specifies requested service, node ID addresses specific device. As you can see on the Figure 22, the opcode has four highest significant bits of the identifier, lower seven bits are used to determine node ID. It means that 16 services on 127 devices are supported.

6.2.2 Solution for CANopen slave devices

NMT slave device provides a number of services recognized by the opcode in the identifier. This device doesn't expect messages from other devices with the information about their state (BOOTUP message / Heartbeat). From predefined connection set analysis came up that only messages with node ID which matches this device and broadcast messages without node ID should be accepted. Filtering of messages with according node ID is done on the CAN data-link layer and choosing specific handler for received message depends on hardware access layer (in other words, on this solution).

In order to achieve complexity of O(1) we have to directly deduce associated event handler from given identifier. Let's consider a function with a message identifier on the input and a function pointer on the output. As was stated above, opcode describes the requested service. Considering discussed properties, this function may be just a translation table with 16 entries for 16 different opcodes. If we have function pointers on appropriate positions in the table we can pick line with index equal to opcode (see Table 8 for example).

With this table (which can be stored in the program memory) we can always simply decide which action should be taken. Reception interrupt has to read message identifier, then dig out the opcode and use it to decide which row in the translation table should be used. It's simple, fast and straightforward. Disadvantage of this solution is in its static behaviour - CANopen specification requires ability to dynamically change the connection set (for example if more than four PDO's are used).

#	Opcode	Function pointer	Key	#	Opcode	Function pointer	Key
0	0000	NMT handler	-	8	1000	RPDO handler	2
1	0001	SYNC handler	-	9	1001	TPDO handler	3
2	0010	TIME handler	-	10	1010	RPDO handler	3
3	0011	TPDO handler	0	11	1011	SSDO handler	0
4	0100	RPDO handler	0	12	1100	CSDO handler	0
5	0101	TPDO handler	1	13	1101	dummy handler	-
6	0110	RPDO handler	1	14	1110	dummy handler	-
7	0111	TPDO handler	2	15	1111	dummy handler	-

Table 8: Translation table for received messages

Note: Column ”key” in Table 8 is used for communication objects and their state machines (to decide which machine is used if more of them is inside a communication object - case of multiple PDO and SDO objects).

6.2.3 Solution for CANopen master devices

If we need a device with the ability to dynamically change the connection set, we cannot use previous solution - the translation table would have to contain complete range of identifiers (2048), not only 16 opcodes. In order to reduce size of the translation table we can use a lookup table with some kind of search algorithm which will optimize memory usage, but make the algorithm slower in general (O(log(n)) at best). To make this work we need a table like in previous case with one more column - the message identifier. This table is called a lookup table because we have to search for the identifier inside (the identifier is no longer an entry number in the table, there are only chosen identifiers connected to specific handlers). If the translation table is sorted we can use binary search algorithm with logarithmic complexity to locate received identifier in the memory. Connection set can be changed simply by changing the identifier in the translation table.

Advantage of this solution is in its complete scalability - we can change the connection set, use extended identifiers and analyze complete identifier, not just the opcode. Analyzing a complete identifier comes handy when the device works as a NMT master and a heartbeat consumer. The translation table then contains entries with node ID’s from all the heartbeat producers and so it can directly say which device has sent the heartbeat message.

6.3 Time scheduling algorithm

CANopen requires many timers in order to work properly - for example SDO is using response timeouts, PDO inhibit times, SYNC and TIME producers use timer to create time delay between two consecutive messages and so on. If purely hardware resources are used for timing, only some kind of hardware driver is present. It’s usually only about configuration and all the work is done on a hardware level with no

CPU overhead. The problem is that most of the target devices won't have needed hardware facilities (like array of modulus down counters for example), so we still need a way around it. The question is how to design software which will replace missing array of timers with only one timer and some additional CPU overhead.

Whatever is actual time scheduling algorithm the interface works in a request/response way. We request to setup the time event by providing a pointer to function and a time-delay. As a response the function is called when time runs out. This is very general concept which allows us to use different implementations of time scheduler algorithm.

6.3.1 Scheduling principles

The goal is to create an algorithm which can simply call a function with given time offset. When we look at the problem from the target device point of view, this solution is especially useful on small and cheap devices (typically 8bit MCU's) with limited peripherals. Those devices usually cannot provide some kind of absolute time reference (CPU time register). As a consequence, the scheduling process needs to be done in relative time completely, which offers several design challenges. If we want a reasonable precision, we have to implement quite complex algorithm (maybe even with corrections for additive offsets).

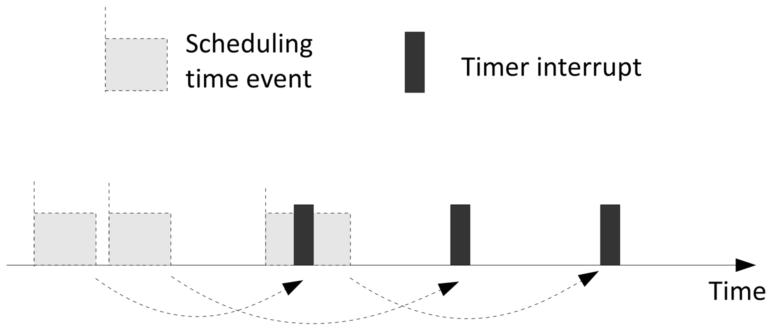


Figure 23: Event scheduling

We need to deal with a situation when many time events are set and only one timer with one interrupt is present. The very basic idea is to somehow schedule the interrupt to be raised after a time-span between two consecutive scheduled time-events. For this we need configurable timer (modulus down counter or timer with compare register) and some kind of time base. Time base is a minimum recognizable time between two events - it discretizes occurred events in time.

Time is always relative in this system - it just counts elapsed time between request and respond. Some CPU architectures have support for absolute time measures, it's usually a 64 bit counter incremented each CPU clock cycle, but the problem is that this kind of clock is mainly present on 32 bit architectures. Work with 64 bit register on an 8-bit device wouldn't be very effective.

A new request is always processed by some kind of adding/sorting algorithm with linear asymptotic complexity (O(n)) at best. That in itself predisposes the scheduling algorithm for time non-critical situations - it shouldn't run in the interrupt service routine. The scheduling algorithm itself alters the plan of interrupts (the schedule) in a way that a new time event is raised in proper time while other scheduled events aren't destroyed nor damaged.

When time runs up a timer interrupt is raised and response generated. It means that interrupt service routine will take registered time-event from the schedule and call function pointer with given key (key is an "unsigned char" value used to choose the state machine inside a communication object).

Unscheduling algorithm does exactly opposite to scheduling algorithm, so its asymptotic complexity is linear at best too and it shouldn't be used in the interrupt service routine. That may prove problematic because in some situations it is required to unschedule a time event inside the interrupt service routine. Solution is in a scheduler with schedule reconstruction ability - unscheduling can be done just by disabling the event and when scheduling another event, scheduler will detect the disabled event and remove it entirely. Disabling itself is not just a type of signalization for the scheduler - disabled event can actually occur before it is completely removed so it also has to redirect function pointed to some "dummy" function. Then no action is taken if disabled event occurs.

This ability to "disable" event instead of unscheduling allows us to work effectively with the memory - it can quickly remove event and also when scheduling a new event, new entry in the schedule can replace the disabled event (i.e. memory is saved). This concept proves to be especially useful when waiting for response messages (response timeouts). Timeouts are typical for long waiting times and without disabling and reconstruction the timeout would be taking a memory space for unnecessary long time.

6.3.2 Scheduler implementation

As was stated above, the algorithm is built around a schedule of time events. It is realized as a forward linear linked list generally sorted in a timely manner. Each schedule entry is a structure which contains pointer to the time-event handler (a function), a pointer to the next entry and relative time between two consecutive entries. There are three entry points to the algorithm: First and most important is a timer IRQ, which is time-critical and so it was designed with constant algorithm complexity (O(1)). The second entry point is for removing already scheduled entries, this function is designed as time-critical. With its constant alg. complexity it can be called within IRQ, which has proven to be useful in some applications. The last entry point is a scheduler itself, this function shouldn't be called within any IRQ (multi-level IRQ's are not considered) and therefore, it isn't timely critical. Scheduling algorithm complexity is linear (O(N)). Due to optimization scheduler does three tasks in one cycle:

- Memory allocation
- Sorting
- Schedule reconstruction

The first problem is about memory space allocation, second one deals with sorting (from lowest time-delay to highest) and schedule reconstruction deals with removed events it reconstructs interconnections in linear linked list after removal of one or more entries.

6.3.3 Timing issues

Timer is using a user-defined time base to move from a continous time to discrete domain. Minimum time base is generally not limited, the scheduler can use the timer to measure own scheduling time and automatically correct created additive error. Lower the time base is, higher is the precision we've got. On the other hand, there is always a trade-off between scheduling precision and lower time base, because lower the time base means more CPU time consumed by the timer IRQ. This solution overcomes the trade-off problem by introducing a concept of floating time base. The timer IRQ is not called periodically, but only when really needed. It is achieved with programmable time-steps between two IRQ's. In the end, user defined time base is truly just a measure of precision. Of course, the trade-off still exists, but its consequences are suppressed so scheduler precision can get higher.

7 Results

"We must reinforce argument with results."

Booker T. Washington

As was already stated, this solution targets to low-end devices with just common peripherals. The solution was designed from the sketch with this disadvantage in mind and so just common peripherals are required for full functionality.

Software replacements were designed to replace missing/unused peripherals. Those replacements create CPU overhead which cannot be overcome (although its size strongly depends on used software design). Introduced design with its architecture allows us to replace selected parts of the hardware access layer with different implementation - so we can use peripherals in case that they aren't actually missing.

Required hardware resources are:

- CAN controller and CAN transceiver
- One 16-bit timer with comparation

CAN controller may not actually be a part of the MCU, it may be an external device (like MCP2515 [22]) controlled via simple point-to-point serial bus.

With the time scheduler only one timer is required for all timing facilities (there are many in CANopen). The timer has to have a compare register with comparation interrupt, or it has to work as modulus down-counter with pre-load register and underflow interrupt (those are precise timing facilities common even on low-end devices).

7.1 Description of measurement suite

Source code was compiled under CodeWarrior IDE version 5.9.0 [build 5294] and debugged in Hi-wave/True-time debugger for HCS12X. It was tested on educational kit CSM-12D (AXIOM Manufacturing, [24]) with MC9S12XDT512 on board (16-bit MCU with HCS12X CPU and XGATE coprocessor, [25]). Hardware debugger USBDM_JM V1.6 [23] was used for debugging.

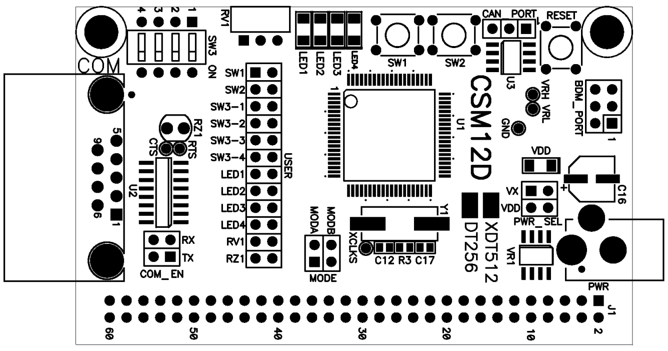


Figure 24: CSM-12D silk screen

CAN-based network was running on speed of 50kbps. The network consisted from CSM-12D educational board and a USB-CAN adapter. USB-CAN adapter [26] provided USB/CAN gate and therefore access to the network to a PC.

7.2 Methodology

Time-related variables were measured in bus cycles and expressed as time difference related to 10Mhz bus clock. Measurement was done under simulation when possible, although in tasks where CAN communication was involved the measurement was done with a debugging tool and by analysis of disassembled program.

To measure average overhead of more complex algorithms a special free-running timer/counter was used (tested hardware architecture doesn't provide any kind of absolute time reference). The value of timer/counter register was sampled on the start and on the end of tested piece of code. The difference between start and stop equals to number of bus cycles in between. Average values were created from weighted average of at least 20 samples.

Listing 4: An algorithm used for collecting data

```
/* number of samples */
#define NO_SAMPLES      30
/* corrections (unit: bus-cycles) */
#define START_OFFSET    +2
#define STOP_OFFSET     -3

ui16_t meas_start[NO_SAMPLES];
ui16_t meas_stop[NO_SAMPLES];
ui8_t p = 0;

void regstart(void) {
  meas_start[p] = TCNT + START_OFFSET;
}

void regstop(void) {
  meas_stop[p++] = TCNT + STOP_OFFSET;
  p%=NO_SAMPLES;
}
```

7.3 Build description

Tested software represents minimum CANopen functionality as described in CiA-301 v4.01 [2]. Many optional features aren't implemented, including dynamic PDO mapping (ability to change PDO identifiers and content), EMCY object and SYNC object. See Table 9 for the list of supported CANopen features.

The build was designed especially for testing purposes - it uses all 8 PDO objects (8 bytes of process data each) in order to ensure worst-case performance conditions. TPDO objects support two transmission triggers - SYNC is used to simulate situation when many messages are required to be sent on the bus (all

	Tested protocol stack
NMT	slave
Heartbeat	producer only
Node/Life guarding	
Default SDO server	✓
Expedited SDO transfer (datatypes up to 4 bytes)	✓
Segmented SDO transfer	
Supported TPDO/RPDO	4/4
TPDO COS trigger	
TPDO RTR trigger	
TPDO SYNC trigger	✓
TPDO Event timer trigger	✓
Dynamic PDO configuration	
Timer resolution	8μs

Table 9: Tested protocol stack features

four TPDO) and event timer is used to test time-scheduler performance in real application. SDO, NMT and Heartbeat are required by CANopen specification.

Time scheduler with ability to schedule up to 5 time events was used (one time slot is for heartbeat and four for TPDO objects). Transmission balancer with 3 slots for message requests was compiled into the program. Six messages may wait for the transmission at one moment - three in the transmit buffers and three in the transmission balancer. TPDO objects take 4 message slots, heartbeat and SDO take the other two. Sent notifications were also compiled to the program, although they are not actually used by any communication object (they are necessary for SDO block transfer, which isn't supported in this build). Sent notifications affect size of the "sent interrupt overhead" (that's why this feature was added to the build).

7.4 General parameters

Several parameters were measured:

- **Program size** - Size of compiled CANopen implementation. Unrelated parts of the program (like startup routines or host application) are not considered. Unit is 1 byte.
- **Used memory** - CANopen memory usage. Memory is initialized statically and predefined during the compilation time. Unit is 1 byte.
- **Receive interrupt overhead** - It's defined as a propagation time between first interrupt occurrence and start of specific handler related to content of the message.

- **Sent interrupt overhead** - Sent interrupt overhead is defined as a time needed to process interrupt itself. This value describes minimal overhead created by each sent message.
- **Timer interrupt overhead** - Timer interrupt overhead describes how much CPU time is lost each timer interrupt occurrence. In other words, this value describes minimal overhead created by each occurred time event.
- **Timer rescheduling offset** - A time required by timer interrupt to pull the next time event from the schedule and to commit rescheduling. This time creates additional time-offsets and distorts scheduling precision. This value is equal to time interrupt overhead.

Note: A time needed to process interrupt varies with used compiler and architecture. For example, some compilers may need to manually save a PSW (Program Status Word) register when interrupt occurs while other architectures may implement this on hardware level.

Measured variable	**Value**
Program size	2458B
Used memory	135B
Receive interrupt overhead	4.3μs (43 bus cycles)
Sent interrupt overhead	1.4μs (14 bus cycles)
Timer interrupt overhead	2.8μs (worst-case)

Table 10: General parameters

7.5 Hardware access layer

Hardware access layer consists mainly from the message transmission and reception routines and from the time scheduler. Message transmission and time scheduler were extensively tested (results are shown below). Message reception consists from relatively straightforward algorithm with constant complexity (O(1)). The overhead of message reception is completely described by value of ”Receive interrupt overhead” and therefore no additional measurements were conducted.

7.5.1 Message allocation time

”Message allocation time” is time needed to allocate a transmit buffer, or if needed, to register a message request in the transmission balancer. Three values were measured - minimum, average and maximum. Results of this measurement are deeply related to tested application.

	Min.	**Avg.**	**Max.**
Message allocation time	8.1μs	11.2μs	25.5μs

Table 11: Message allocation time

Please note that theoretical maximum is plus inifinity, although this situation occurs only as a result of a faulty condition (problems on the bus). Allocation algorithm is fastest when at least one transmit buffer is available - there is no need to push a message request into the balancer. Worst-case condition occurs when all transmit buffers and all slots for message requests are taken.

Average value of "message allocation time" strongly depends on the bus load and on triggering method used for the transmitted messages. In practice, average value tends to converge to minimum, unless the bus is under heavy load.

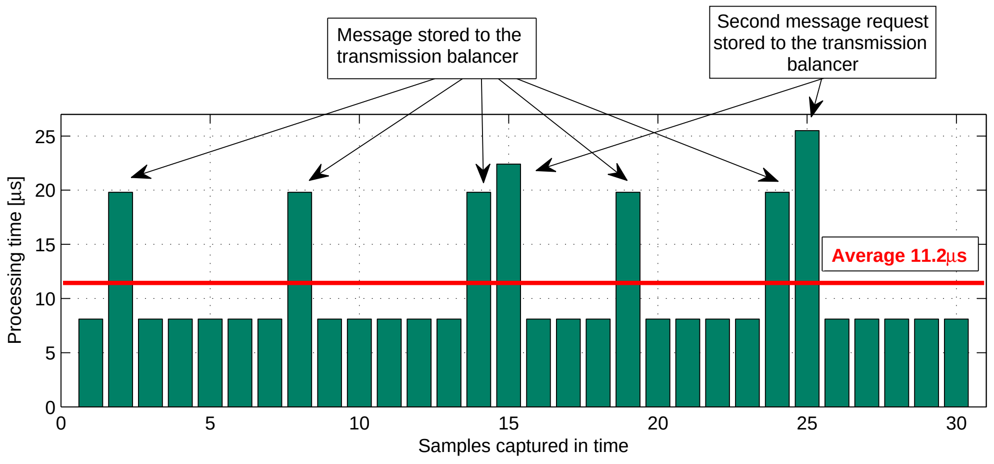


Figure 25: A series of sampled message allocation times

7.5.2 The transmission balancer

Pushing a new message into the transmission balancer is an adding/sorting algorithm with a linear complexity (O(n)). Time needed to push a new message into the balancer is strongly related to a way how internal linked list is currently organized in the memory. This test creates a worst-case condition for nth pushed entry into the balancer by adding a message request to the end of the schedule and to the farthest place in the memory, see Figure 26.

Note: Due to specific requirements for creating worst-case condition, this test is not related to tested application. Nevertheless, the results are still related to used architecture (Freescale HCS12X).

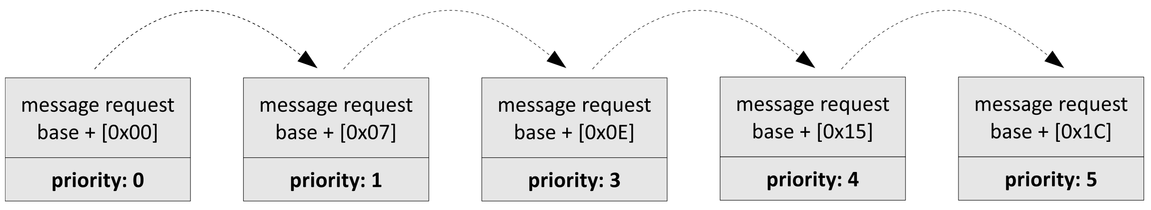

message request base + [0x00]
priority: 0
message request base + [0x07]
priority: 1
message request base + [0x0E]
priority: 3
message request base + [0x15]
priority: 4
message request base + [0x1C]
priority: 5

Figure 26: Pushing a message into the balancer under worst-case condition

Figure 26 shows how the worst-case condition is created both in the memory and in the linked list (a message request takes 7 bytes of the memory). Used structure is actually simple - entries are always scheduled to the end of the list. For test reasons, no message request is ever pulled from the schedule.

Figure 27 *de facto* shows worst-case processing time needed to push a message into the transmission balancer with chosen size (from 1 to 30). For detailed results see Appendix A.

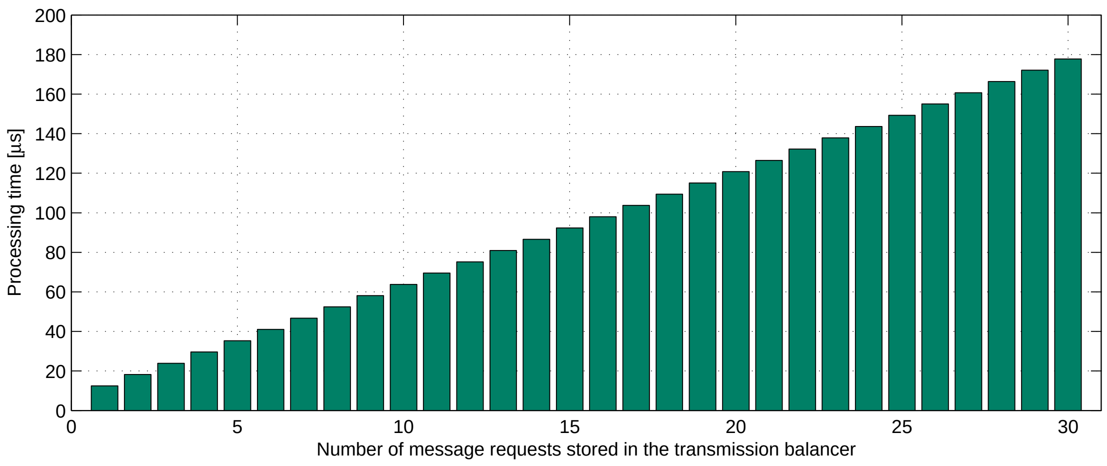


Figure 27: The transmission balancer - processing time versus load

7.5.3 The time scheduler - rescheduling time

The time scheduler works in a similar way to transmission balancer (it uses sorted linear linked list). Although it's based on same ideas, the realization is different, the time scheduler is way more complicated and also time scheduler supports a schedule reconstruction while rescheduling (see chapter 6.3 for further information).

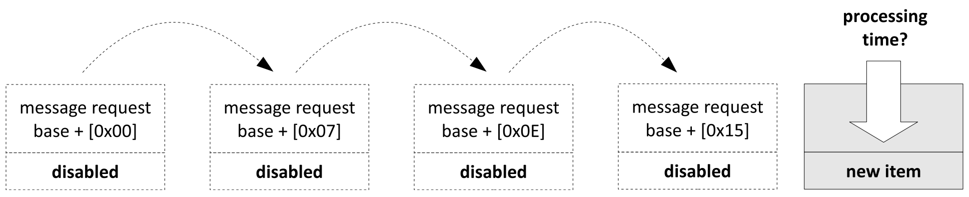

message request base + [0x00]
disabled
message request base + [0x07]
disabled
message request base + [0x0E]
disabled
message request base + [0x15]
disabled
processing time?
new item

Figure 28: Time scheduler with reconstruction - test outline

"Rescheduling without reconstruction" was measured like pushing a message into the transmission balancer (see chapter 7.5.2). "Rescheduling with reconstruction" was also measured in the same way, but all the prepared events in the linked list had to be marked as "unscheduled" first, so the scheduler will commit reconstruction of all entries in the schedule. Situation is outlined on Figure 28. Data structures had to be especially prepared for each measurement - special "feeding" algorithm was used, for further information see Listing 5.

Listing 5: Preparation of schedule with disabled entries only

```
ui8_t x,d;

/* Time scheduler */
schevent_t tsched[30];
schevent_t * start = tsched;
ui16_t roof = 0;

for(x=0;x<30;x++) {
  /* Reset schedule */
  schreset(tsched,30);
  start=tsched; roof=0;

  /* Schedule x items, higher d - higher time offset */
  for(d=0;d<x;d++) schedule(d, test_callback);
  /* Unschedule x items */
  for(d=0;d<x;d++) *(tsched+d)->fcn = dummy_callback;

  /* commit the measurement on prepared data set */
  regstart();
  schedule(100, test_callback);
  regstop();
}
```

Figure 29 shows how scheduling time depends on scheduler load while rescheduling (green bars) and while rescheduling and reconstructing at once (yellow bars). The measurement clearly shows that algorithm with reconstruction is even faster than just rescheduling. **It means that each unscheduled entry makes rescheduling process faster.** Consequences of this feature are discussed in chapter 8.

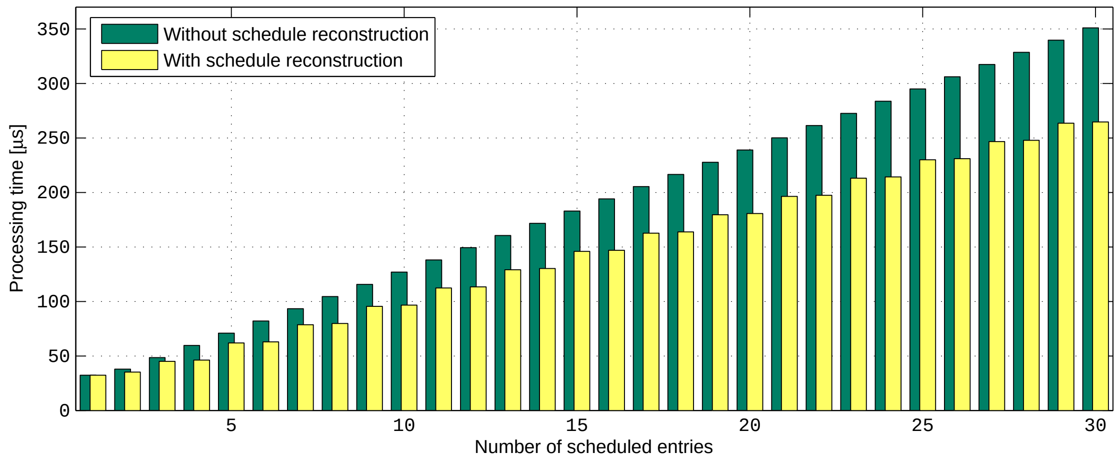
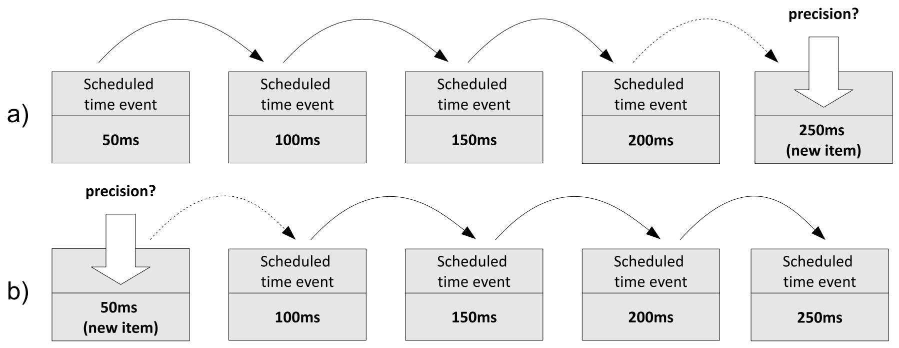


Figure 29: Time scheduler - rescheduling time

7.5.4 The time scheduler - precision

Precision of the time scheduler depends on several factors (they are described in chapter 6.3). For purposes of this test, tested time scheduler hasn't utilized any correction mechanism.



a)
Scheduled time event 50ms
Scheduled time event 100ms
Scheduled time event 150ms
Scheduled time event 200ms
precision?
250ms (new item)
b)
precision?
50ms (new item)
Scheduled time event 100ms
Scheduled time event 150ms
Scheduled time event 200ms
Scheduled time event 250ms

Figure 30: Time scheduler precision - test outline

Two tests were made - scheduling a 10 consequent time events with offset of 50ms (starting with sooner time events) and scheduling the same time events, but in opposite order (starting with farthest time events). As a result, in the first case we schedule always to the end of the schedule, in second case we schedule always before the first entry of the schedule. Situation is outlined on Figure 30.

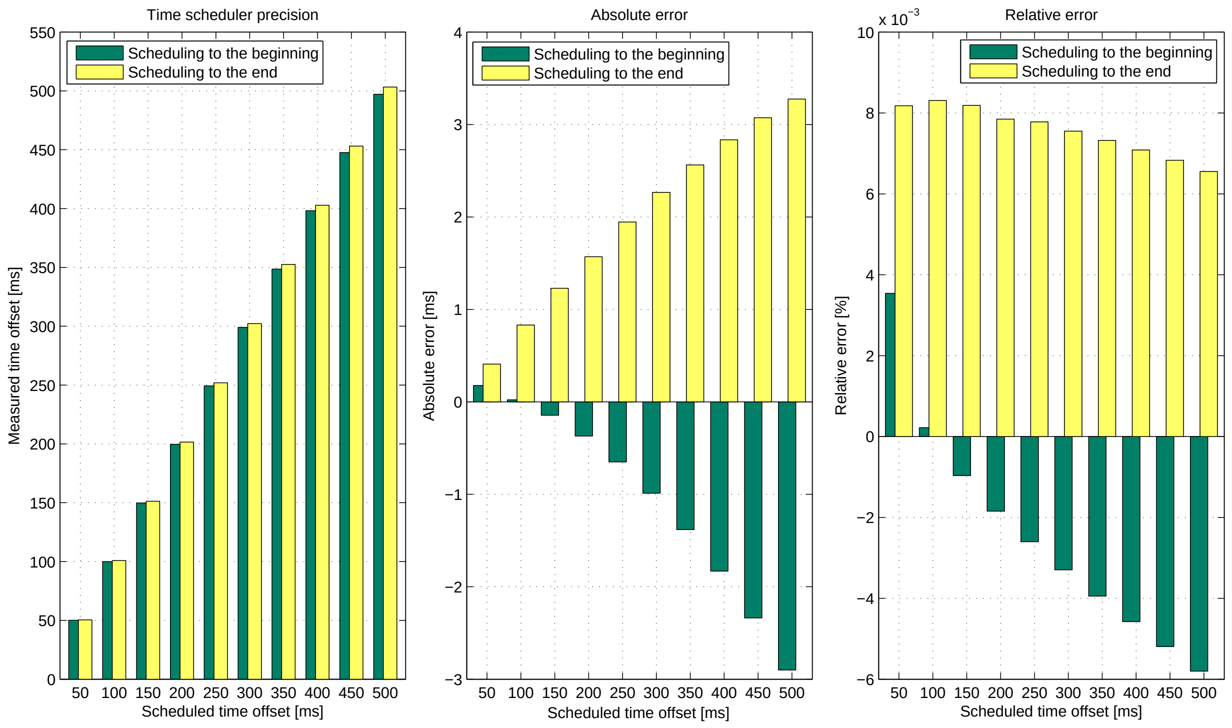


Figure 31: Time scheduler precision - test results

As you can see on Figure 31, when scheduling to the beginning we get generally smaller error rates, but the size of the absolute error rises exponentially with the load (with the number of scheduled time events). On the other hand, when scheduling to the end, given error rates are generally larger, but absolute error rises logarithmically (relative error is slowly falling). It seems that for small schedules it's better to schedule to the beginning, although the results also suppose that there may exist a threshold, after which it's better to schedule to the end.

	Absolute error [ms]			**Relative error [%]**		
	Min.	Avg.	Max.	Min.	Avg.	Max.
Scheduling to the beginning	0.409	2	3.276	0.655	0.756	0.831
Scheduling to the end	0.022	1.08	2.9	0.022	0.319	0.580

Table 12: Time scheduler precision - measured error rates

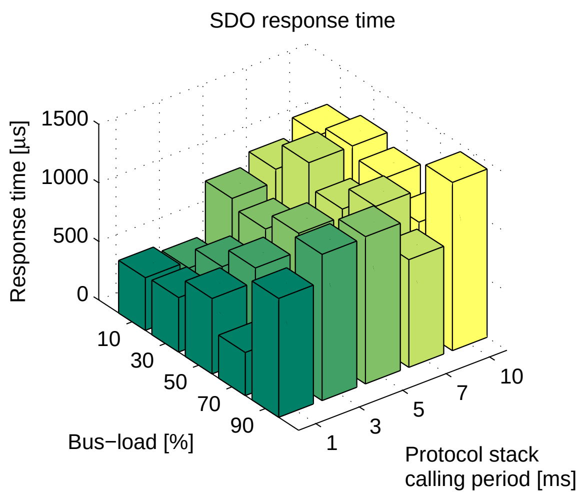
7.6 System response

As was stated in chapter 5, this CANopen implementation consists of two parts: part accessed by received interrupt and part accessed by the protocol stack. The first part is running inside an interrupt service routine and therefore is time-critical (because running interrupt suppresses a host application). Protocol stack is called by the host application on periodical basis. The protocol stack generates responses to various requests which occurred since last call. In this way it's ensured that the CANopen implementation itself cannot suppress the host application.

	Received IRQ [μs]			**Protocol stack** [μs]		
	Min.	Avg.	Max.	Min.	Avg.	Max.
Network management (NMT)	-	12.7	-	42.7	51.2	66.7
Process data object (PDO)	-	2.6	-	119.8	131.4	137.4
Service data object (SDO)	-	44.4	-	55.5	70.5	73.2
Synchronisation object (SYNC)	-	17.2	-	175.9	177.5	177.5
Heartbeat service	-	-	-	85.9	99.7	102
Processing time	7.9	33.1	54.2	6.7	- [1]	571.6

Table 13: Processing times

Response time to requested action is vital property of any real-time embedded system. Figure 32 shows measured response times related to a bus load and to a protocol stack update period. They were measured as a time between registering a request and time when a response was requested for transmission. Please note that this is not a time when the message was actually transmitted - there is additional time required to get the transmission slot on the bus. Bus-load condition is ment as a total load of the bus with no regard to the source of the transmissions - siginificant part of the bus-load was generated by measured device itself.



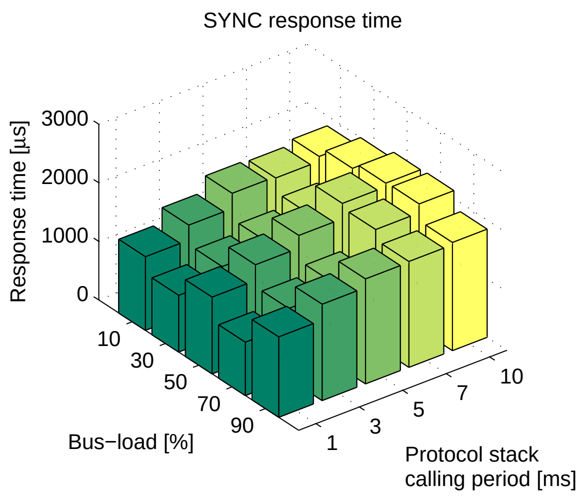


Figure 32: Response times

[1]This variable cannot be measured accurately. The protocol stack only responds to requests, so the amount of work to be done doesn't depend on how often the protocol stack is called.

8 Conclusions

"A man who carries a cat by the tail learns something he can learn in no other way."
Mark Twain

Aim of this thesis was to develop a light-weight CANopen implementation suitable for low-end microcontrollers with limited CPU performance and general absence of advanced peripherals (like timer-arrays and DMA). Proposed software design counts with those deficiencies and introduces the transmission balancer as a replacement for CAN message boxes and also the time scheduler as a replacement for timer-arrays (required for creating many simultaneous time-events). The design uses powerful architecture which allows replacing of the transmission balancer or the time scheduler with hardware driver if needed.

Proposed algorithms were designed especially to suite real-time requirements of embedded networks based on CANopen. Both transmission balancer and time scheduler were analyzed and tested in real application. The results clearly show that both transmission balancer and the time scheduler have potential to effectively replace missing hardware resources. Although they will always create additional CPU overhead, it may not necessarily be a problem, if the processing is done when the time is not critical (outside the interrupt service routine).

The transmission balancer is algorithm which allows scheduling messages for transmission. Main idea is to replace message boxes (CAN message containers in RAM) with unique message requests. Message requests are sorted by priority in the transmission balancer. When the message request acquires a transmission buffer, the callback function is called to "feed" the buffer with data.

The advantage over message boxes is that no additional hardware is required (DMA channel) and also there is no need to transfer data from RAM to CAN peripheral, which theoretically *makes possible for the transmission balancer to be faster than hardware-implemented message boxes.* This feature hasn't been tested yet and it is one the things planned for further research.

The time scheduler is complicated scheduling algorithm which has to deal with generally unlimited number of time events. There are several problems - for starters we cannot literally stop time, so it is complicated to get reasonable scheduling precision. On the other hand, it's programmer-friendly, because programmer just have to say which function should be called and when.

Key feature of the time scheduler is the ability to do the unscheduling even in time-critical parts of code, which is essential for concept of timeouts. Traditionally, unscheduling algorithm has opposite function to rescheduling and so it has linear complexity at best (O(n)). Proposed solution of unschedulling not only allows to literally unschedule within one or two bus cycles, but *it also makes rescheduling process faster* and automatically creates free space for another time-events.

Performance results of time scheduler may speak for yourself - in worst-case situation, the scheduling would be done within 65μs with precision larger than 99.7% (relates to tested case - 10Mhz bus clock, uses schedule with size of 5 and schedules at least 50ms to future).

Specially prepared CANopen protocol stack with the transmission balancer and the time scheduler was created and putted under series of tests to prove proposed concepts and real-time behaviour. The solution is CANopen compliant - network management, process/service data objects and heartbeat are supported. The protocol stack is utilizing 4 RPDO and 4TPDO objects, each of them contains 8 bytes of data (maximum). This configuration is pretty standard in practice so the results may speak about usability of this solution in real-world applications.

Compiled program takes about 2.5kB of program memory and uses only 135B of data memory (with all schedules). Worst-case processing time of received message is slightly larger than lowest time needed to transmit another message frame on maximum speed (1Mb/s). Time needed to process whole protocol stack is between 6.7μs and 571.6μs.

To sum up, this thesis aimed to propose a set of out-of-box algorithms which may replace more traditional approaches. This goal was achieved and the algorithms were successfully tested. Nevertheless, proposed solutions still need to prove yourself in real-world applications. Concept of tryouts (instead of timeouts), time scheduler, transmission balancer - they all work on paper and in laboratory, but the question is how reliable will they be in practise. This is the task for further research.

A Measured data

#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]
1	81	8.1	11	81	8.1	21	81	8.1
2	198	19.8	12	81	8.1	22	81	8.1
3	81	8.1	13	81	8.1	23	81	8.1
4	81	8.1	14	198	19.8	24	198	19.8
5	81	8.1	15	224	22.4	25	255	25.5
6	81	8.1	16	81	8.1	26	81	8.1
7	81	8.1	17	81	8.1	27	81	8.1
8	198	19.8	18	81	8.1	28	81	8.1
9	81	8.1	19	198	19.8	29	81	8.1
10	81	8.1	20	81	8.1	30	81	8.1

Table 14: Message allocation time

#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]
1	125	12.5	11	695	69.5	21	1265	126.5
2	182	18.2	12	752	75.2	22	1322	132.2
3	239	23.9	13	809	80.9	23	1379	137.9
4	296	29.6	14	866	86.6	24	1436	143.6
5	353	35.3	15	923	92.3	25	1493	149.3
6	410	41.0	16	980	98.0	26	1550	155.0
7	467	46.7	17	1037	103.7	27	1607	160.7
8	524	52.4	18	1094	109.4	28	1664	166.4
9	581	58.1	19	1151	115.1	29	1721	172.1
10	638	63.8	20	1208	120.8	30	1778	177.8

Table 15: Transmission balancer load

#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]
1	324	32.4	11	1123	112.3	21	1963	196.3
2	352	35.2	12	1134	113.4	22	1974	197.4
3	451	45.1	13	1291	129.1	23	2131	213.1
4	462	46.2	14	1302	130.2	24	2142	214.2
5	619	61.9	15	1459	145.9	25	2299	229.9
6	630	63.0	16	1470	147.0	26	2310	231.0
7	787	78.7	17	1627	162.7	27	2467	246.7
8	798	79.8	18	1638	163.8	28	2478	247.8
9	955	95.5	19	1795	179.5	29	2635	263.5
10	966	96.6	20	1806	180.6	30	2646	264.6

Table 16: Rescheduling with reconstruction

#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]	#	Bus-cycles	Time [μs]
1	324	32.4	11	1381	138.1	21	2501	250.1
2	380	38.0	12	1493	149.3	22	2613	261.3
3	485	48.5	13	1605	160.5	23	2725	272.5
4	597	59.7	14	1717	171.7	24	2837	283.7
5	709	70.9	15	1829	182.9	25	2949	294.9
6	821	82.1	16	1941	194.1	26	3061	306.1
7	933	93.3	17	2053	205.3	27	3173	317.3
8	1045	104.5	18	2165	216.5	28	3285	328.5
9	1157	115.7	19	2277	227.7	29	3397	339.7
10	1269	126.9	20	2389	238.9	30	3509	350.9

Table 17: Rescheduling without reconstruction

#	Wanted offset [ms]	Measured offset [ms]	Absolute error [ms]	Relative error [%]
1	50	50.409	0.4090	0.81800
2	100	100.831	0.8310	0.83100
3	150	151.228	1.2280	0.81867
4	200	201.570	1.5695	0.78475
5	250	251.945	1.9450	0.77800
6	300	302.266	2.2655	0.75517
7	350	352.563	2.5625	0.73214
8	400	402.835	2.8345	0.70862
9	450	453.074	3.0735	0.68300
10	500	503.276	3.2760	0.65520

Table 18: Scheduling to the beginning (precision measurement)

#	Wanted offset [ms]	Measured offset [ms]	Absolute error [ms]	Relative error [%]
1	50	50.177	0.1770	0.35400
2	100	100.022	0.0220	0.02200
3	150	149.855	-0.1450	-0.09667
4	200	199.631	-0.3690	-0.18450
5	250	249.350	-0.6505	-0.26020
6	300	299.012	-0.9885	-0.32950
7	350	348.619	-1.3815	-0.39471
8	400	398.169	-1.8310	-0.45775
9	450	447.663	-2.3370	-0.51933
10	500	497.100	-2.9000	-0.58000

Table 19: Scheduling to the end (precision measurement)

Response time [μs]						
		Protocol stack update period				
		1ms	3ms	5ms	7ms	10ms
Bus-load	10%	449.3	689.0	841.9	949.2	1098.1
	30%	465.0	587.4	766.8	1189.8	1190.0
	50%	643.0	761.4	914.5	979.4	1103.5
	70%	367.1	615.7	686.4	1186.9	909.8
	90%	1013.1	1247.5	1257.0	916.8	1435.7

Table 20: SDO response time

Response time [μs]						
		Protocol stack update period				
		1ms	3ms	5ms	7ms	10ms
Bus-load	10%	1257.5	1514.4	1771.4	1747.3	1830.1
	30%	970.7	1177.4	1328.1	1505.7	1998.7
	50%	1308.6	1577.3	1792.6	2056.5	2117.7
	70%	909.2	1202.4	1365.7	1979.5	2133.2
	90%	1369.7	1644.0	1791.1	1804.1	1884.7

Table 21: SYNC response time

References

[1] Pfeiffer Olaf, Andrew Ayre, Christian Keydel (2003). *Embedded Networking with CAN and CANopen.* Revised First Edition. ed. San Clemente, CA: Copperhill Technologies Corporation.

[2] CAN in Automation e. V. (2002). *CiA Draft Standard 301 v4.02.* Nuremberg, Germany. [Accessed 2010]. Available from: <http://www.can-cia.org/index.php?id=specifications>.

[3] Wikipedia (2011). *Controller area network* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Controller_area_network>.

[4] CAN in Automation e. V. (2011). *CANopen* [online]. [Accessed 3 June 2011]. Available from: <http://www.can-cia.org/index.php?id=canopen>.

[5] Wikipedia (2011). *CANopen* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/CANopen>.

[6] IXXAT Automation GmbH (2011). *CANopen Solutions* [online]. [Accessed 3 June 2011]. Available from: <http://www.canopensolutions.com/>.

[7] Wikipedia (2011). *DeviceNET* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/DeviceNet>.

[8] Wikipedia (2011). *EIA-485* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Rs-485>.

[9] Wikipedia (2011). *EEPROM* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/EEPROM>.

[10] Wikipedia (2011). *Ethernet* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Ethernet>.

[11] Wikipedia (2011). *Carrier sense multiple access with collision detection* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Carrier_sense_multiple_access_with_collision_detection>.

[12] Renesas Electronics Europe (2010). *Switch it Smart - Intelligent power devices* [online]. [Accessed 3 June 2011], p.4. Available from: <http://www2.renesas.eu/_pdf/U18602EE2V0PF00.PDF>.

[13] Wikipedia (2011). *Ethernet* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Ethernet>.

[14] Wikipedia (2011). *IEEE 754-1985* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/IEEE_754-1985>.

[15] Wikipedia (2011). *Two's complement* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Two%27s_complement>.

[16] Wikipedia (2011). *Plug and play* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Plug_and_play>.

[17] Manoj Kumar, Ajit Kumar Verma and A. Srividya (2008). *Response-Time Modeling of Controller Area Network (CAN).* In: Vijay K. Garg, Roger Wattenhofer, Kishore Kothapalli, (ed). Distributed Computing and Networking, Berlin, Germany: Springer-Verlag Berlin and Heidelberg GmbH , pp494.

[18] Thomas Nolte, Hans Hansson, and Christer Norstrom (2003). *Probabilistic Worst-Case Response-Time Analysis for the Controller Area Network* [online]. [Accessed 3 June 2011]. Available from: <http://www.mrtc.mdh.se/publications/0521.pdf>.

[19] Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling, ISO Standard 11898:2003

[20] Wikipedia (2011). *Coprocessor* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Coprocessor>.

[21] Wikipedia (2011). *Interrupt* [online]. [Accessed 3 June 2011]. Available from: <http://en.wikipedia.org/wiki/Interrupt>.

[22] Microchip Technology Inc. (2005). *MCP2515* [online]. [Accessed 13 June 2011]. Available from: <http://ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf>.

[23] Peter O'Donoghue (2007). *USBDM (JMxx Version) V4.5* [online]. [Accessed 13 June 2011]. Available from: <http://usbdm.sourceforge.net/USBDM_V4.5/USBDM_JMxx/html/index.html>.

[24] AXIOM Manufacturing (2006). *CSM-12D* [online]. [Accessed 13 June 2011]. Available from: <http://www.axman.com/files/CSM12D_UG.pdf>.

[25] Freescale Semiconductor (2006). *MC9S12XDP512 Datasheet* [online]. [Accessed 13 June 2011]. Available from: <http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf>.

[26] IMFsoft, Ltd. (2006). *USB-CAN adapter TRIPLE drivers* [online]. [Accessed 13 June 2011]. Available from: <http://imfsoft.com/hardware/products/usb-can-adapter-triple-drivers.asp>.