CANopen Master and Node Software

Radoslav Bortel,
Faculty of Electrical Engineering, Czech Technical University, Prague

Mentors: Mark Sibenac, Bill Kirkwood
Summer 2002

Keywords: CAN, CANopen, communication, AUV, 8051

ABSTRACT

This reports contains the description of the project, during which CANopen software was created. Also, it contains a detail description of the implemented programs. The project description part explains what CANopen standard is, and why there was a need for its implementation. In addition it talks about all the steps the project went through, about its outcomes and insufficiencies. The program description part discusses created programs - CANopen master and CANopen node software. It explains their function, providing flow charts and explanations.

INTRODUCTION

The main function of the Autonomous Underwater Vehicle (AUV) is to collect data measured by variety of scientific instruments. In order to store all the produced data, they need to be transferred to the main computer, where storage takes place. For this transfer, AUV uses its internal communication system. This communication system connects each measuring instrument and each actuator to the main computer, and allows them to talk to each other.

To be usable in the Autonomous Underwater Vehicle, the internal communication system has to meet several requirements. First, it has to be fast enough to transfer all the measured data. Then, it should allow easy adding and removing of new scientific packages without big needs for configuration. Next, it should be pressure resistant, without any requirements for additional pressure housing. Also, it should use low number of wires to allow easy cabling. In addition, it should be widely supported by industry, which allows buying off-shelf components, and using them without additional interface devices. And, finally, the communication system should be cheap to implement.

If fact, the internal communication system currently used on the AUV (based on RS-232 and Ethernet) doesn't meet all these requirements. Even though this system is fast enough and supported by industry, it has several disadvantages. First, it is not pressure

resistant - certain hardware requires one atmosphere housing, which makes the system construction difficult and bulky. Next, the system is very complex, and uses a lot of hardware and cabling, which makes the AUV construction expensive. Finally this system doesn't support quick addition of new hardware without difficult configuration. These disadvantages led to the thoughts about changing the internal communication system to the one, which is more suitable for a device as AUV.

To find a new communication system several options were considered. Systems such as Ethernet, CAN, LonTalk and RS-232/485 were compared, and their advantages and disadvantages were taken into account. Finally, as the most suitable communication system was chosen Controlled Area Network or CAN.

This system promised to have everything that AUV requires. CAN is fast enough to manage the sensor data flow. It has a high-level protocol, which allows easy ‘plug and work' functionality. All the CAN hardware parts are believed to be pressure resistant, and so they don't need any bulky housing. CAN also requires only two wires to connect each device, and used hardware is cheap. Moreover, CAN has strong industrial support. For example, National Marine Electronics Association have chosen CAN as their standard, so all their compasses, GPS system or sonars will have CAN interface on them. CiA - CAN in Automation organization strongly supports CAN systems in automation fields. Also, CAN is widely used in the car industry. This ensures that in the future many scientific instruments will have a CAN interface on them.

As each sport is connected with its tactics CAN is connected with its high-level protocols. Higher-level protocols are the sets of rules keeping which one can win an information transfer game. These protocols organizes communication between numerous number of the users connected to a network. For example, protocols set priorities to important messages, and organize slow transfers of huge data blocks. In the other words, higher-level protocols are putting order into disordered users, which request transfer of their data.

CAN has several higher-level protocols, but currently only one of them is widely used. Its name is CANopen. CANopen provides comfortable environment for data transfer between the central computer and its peripheries. It defines several sets of rules that should organize various data transfers. For example CANopen differentiates small, fast transfer and huge, slow transfer. It defines how the individual users of a network are recognized and configured. It also specifies how transferred data should be passed to data processing applications. All these features makes the data transfer effective and allows convenient 'plug and work' functionality.

Technically, the implementation of CANopen as a high-level protocol is a peace of software, which behaves according to a certain technical specification. This software 'lives' between a data processing application and circuits, which sends electronic messages on the bus. The CAN open software translates requests to send or to receive data into electronic messages running through a CAN bus. It exists in each user connected into a network. It is on the side of the main computer, and also on the side of other connected devices - e.g. scientific instruments.

The aim of this project was to write this CANopen software for both the main AUV computer and a general connected device. The software should have provided the basic CANopen transfer procedures; however, it wasn't supposed to have all the complex functionality as given in the technical specification. In fact, only the implementation of

the most important transfer functions was needed.

Therefore, the resulting code intended for the main AUV computer has ability to transfer and process data from a group of network users, but it is not able of the higher functions as 'plug and work' configuration (these features can be added).

Also, the resulting code intended for a network node (e.g. scientific instrument) has ability to respond to all the requests from the main computer - it provides and receive all the requested or sent data and passes them to the data processing application of a sensor or an actuator. Similarly, like the main computer application, the node software doesn't support network management functions.

MATERIALS AND METHODS

The task for my project was to start the implementation of the CANopen communication standard inside the AUV. The original plan was to build a small testing network, get it running, and then design appropriate hardware for the actual usage in the vehicle. The project didn't get so far, but this effort was initiated.

As the first thing in my project, I build a simple CAN network. This network was much simpler than the aimed sophisticated CANopen one, but it was a good start. The CAN network consisted of a personal computer and a microcontroler (ATMEL 8051CC01). After assembling the hardware, the network was tested. Not much surprisingly, the initial communication trials didn't work. No communication was established, and the oscilloscope was showing some strange activity on the network bus. Only studies of the CAN requirements showed, that one important element was missing. After adding this element (actually a simple termination resistor), the CAN network finally started to operate. This was a good starting point.

The next step was to move from CAN to CANopen. This required getting CANopen communication software; actually, two different kinds of software – the master and node applications. The master application should run on the main AUV computer and the node application should run on each connected network user. To obtain this software several sources were tired.

Fist, we tried one free Linux software, called CanFestival. It should have provided CANopen master functionality; however, it turned out to be problematic. CanFestival had no ability to talk to our CAN hardware, so additional drivers had to be added. Even after the driver addition CanFestival didn't work properly. It showed to have some problems when more different communication procedures were running at the same time. Finally, we decided not to use this free software, but rather buy a commercial one.

In the mean time, while I was trying to make CanFestival running, my mentors were considering to buy commercial software for a CANopen node. This software should have been a communication partner for the CanFestival master application. However, the currently available CANopen node software turned out to be rather unattractive. It was extremely expensive; moreover, buying this commercial software would also mean spending money for some redundant functionality, which our CANopen implementations didn't actually required. Therefore, I decided to implement the CANopen node software by myself.

Actually, the implementation of the basic CANopen communication functions and the interface wasn't complex. I wrote code for all the different kinds of the data transfers, and also I programmed a small data processing application, which was taking data from simple sonar based device, and translating them into CANopen messages. Once finished, my code was tested against the commercial CANopen tools running on the PC, and all the code proved to be functioning. Moreover, I happened to find some bugs in one of the beta versions of CANopen software prepared for commercial use.

In the same time my mentors and me were also realizing that the CANopen master software, CanFestival, is not going to be useful. Again, my mentors tried to find some commercial software, but they didn't succeeded. The only CANopen master application on the market was again extremely expensive; moreover, it required some odd additional hardware for its function. Therefore, the same decision was made. I started to implement the CANopen master software by myself.

CANopen master application was more complex then the node application. It had to handle simultaneous communication with several different nodes, and it should have provided functions for simple configuration of a network. It required more than 1500 lines of the code to implement these functions. When the software was ready, it was tested with simple commercial applications, and it worked fine.

Finally I ended up with a simple test network, which is using my own software, and it's working fine. I ran a few tests, and integrated some commercial programs, which allows remotely changing software running on the network nodes. Everything was doing well.

In this point I'm passing my work to my mentors.

RESULTS

The result of this project is the CANopen master and node software.

The first peace of software, the CANopen master application, provides the basic CANopen functionality. First, it can handle all the types of data transfers, which can run simultaneously with several nodes. Then, it has implemented the standard interface for data processing applications – so called object dictionary. Moreover, the CANopen master software supports configuration of the object dictionary based on DCF files, which makes creation of the object dictionary very comfortable. On the other hand, CANopen master software doesn't support all the functions stated in the CANopen technical specification. So-called network management still needs to be implemented. However, even though not all the functionality is implemented, the essential part of the communication is already working.

The second peace of software implemented in this project is the CANopen node application. Like the CANopen master, the CANopen node software supports the basic CANopen functionality. It can communicate with all the types of data transfers, and it supports simple object dictionary - interface for a data processing application running on a node. Functionality that CANopen doesn't support is not so essential network management.

DISCUSSION

Both created applications – the CANopen master and the CANopen node – form closed complex, which can provide the remote control of the network nodes. Once put between the data processing applications on the main computer and a network node, it allows them to cooperate. This cooperation is highly organized, and provides variety of the transfer types. For small data, which needs to be transferred quickly, fast and simple formats are available. For huge data blocks, more complex and more safe formats are can be used. This way the CANopen software creates an efficient communication system.

What neither the CANopen master nor the CANopen slave provides is so called network management. These are more sophisticated functions and operations on the network - for example, continuous checking of the individual nodes for their operational state (operating or dead), or changing the identifiers of the nodes. However, even though these procedures are not implemented, they are not essential for a basic information transfer. Therefore, the CANopen communication system can run without them. Moreover, all the software is built in the way that addition of the network management functions won't require any significant changes (actually only adding of the code is needed).

CONCLUSIONS/RECOMMENDATIONS

Conclusion

The CANopen master and the CANopen node applications showed that they can provide sufficient functionality for an efficient data transfer between the main computer (as a data consumer) and individual network nodes (as data producers). Having this software, actual hardware for the CANopen network can be designed and build.

Recommendation

Even though the missing network management is not essential, its adding is recommended. It can increase network safety, and provides 'plug and work functionality'

Appendix

Software Documentation

Project Scope

This project provides a software support for the CANopen implementation. Basically, two applications were created. First is intended for the main AUV computer, and second for a general network node (e.g. scientific instrument). Both this communication programs connect data processing applications on the main computer and on network nodes. They both provide standard CANopen communication services, and standardized CANopen interface in the form of the object dictionary (OD)

CANopen Master – "Can Communicator"

The application called Can Communicator, written for industrial PC 104 with a Linux system, provides the basic CANopen master functionality. It supports all the types of data transfers, and also the standard CANopen interface – the object dictionary. However, even though the basic CANopen functionality is supported, not everything mentioned in the CANopen technical specification is implemented. The following list specifies:

Can Communicator as a CANopen master supports:

- SDO transfer:
  - Expedited
  - Domain
  - Block
- PDO processing – PDO are automatically mapped into object dictionary entries
- Object dictionary (OD)
- OD configuration based on DCF files
- Sync frame generation

Can Communicator doesn't support (even though these functions are defined in the technical specification)

- Network Management
- Any interprocess communication – basically no interface for other applications (needs to be done)

Program's Structure

The block diagram of the CANopen master application, Can Communicator, is in Fig.1. Can Communicator is situated between the data processing application (Node Control in Fig. 1) and the CAN driver. Its function is to translate the communication requests of the data processing application into the stream of CANopen messages – CANopen communication objects – and, using the driver, send these messages on a CAN bus.

To carry out this task Can Communicator uses several parts. The central part has the same name as the program itself – *Can Communicator*. This section sorts all the messages incoming from the CAN-bus, and resends them for the further processing. The messages are resent to the objects called *Nodes*. The *node* objects take care for the communication with individual network nodes. Each *Node* object in Can Communicator corresponds to one actual node in the network. They initiate and lead SDO communication, process incoming PDO's, and have ability to respond network management and error messages. Each *Node* object also owns an *Object Dictionary(OD)*, which works as an interface with data processing applications. The O*bject Dictionary* is actually used for storage of all the transferred data. To create an *Object Dictionary*, which corresponds to the object dictionary on a node, Can Communicator uses a *DCF File Reader,* which reads the OD description from a Device Configuration File (DCF). Slightly separated from all these parts, there is a *Watchdog*, which controls the time delays of the SDO transfers. If any network node doesn't respond to SDO transfer requests within a certain time, the *Watchdog* breaks the communication with such a node.

All these parts provide the basic CANopen transfer functionality. Closer description of the individual blocks is provided in the following sections

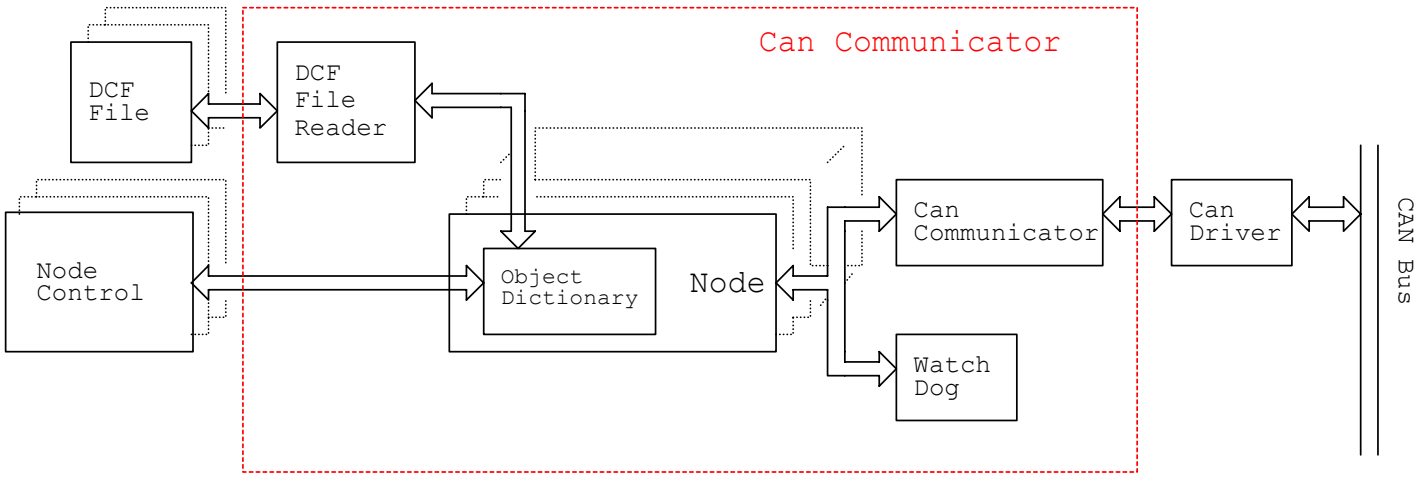

Fig. 1. Block diagram of CANopen master program – Can Communicator

Can Communicator – incoming message processor

The function of the Can Communicator block (not overall program, just the message processor part) is showed in Fig. 2.

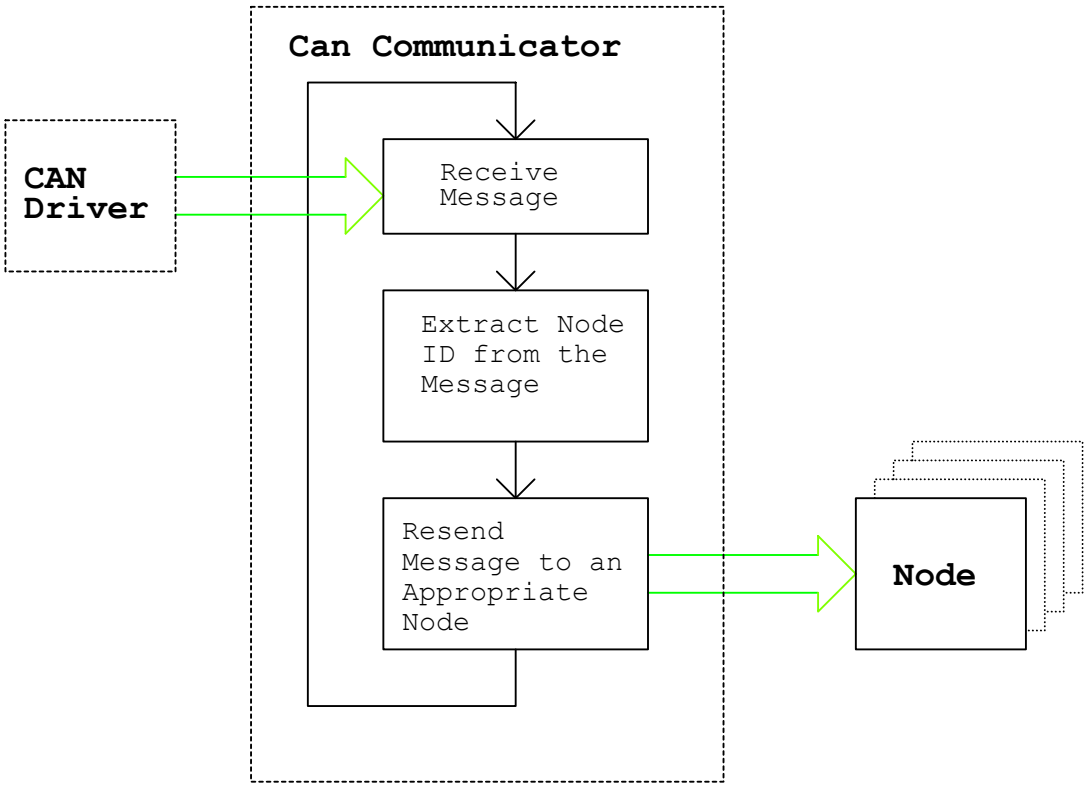

Fig. 2. Can Communicator block – incoming message processor

The Can Communicator block, running as a separate thread, receives incoming CAN messages, finds out which node the message is directed to, and then resends the message to this node.

Node object

The *Node* objects process communication with the individual nodes in the network. They provide message processing functions, which are called by the Can Communicator block after CANopen messages are received. There is exactly one *Node* object for each actual network node.

Node's message processing is showed in Fig. 3. First, the incoming message is examined for its type (SDO, PDO, NMT). Then, an appropriate processing function is called. The address of this function is obtained from the *Process Function Table.*

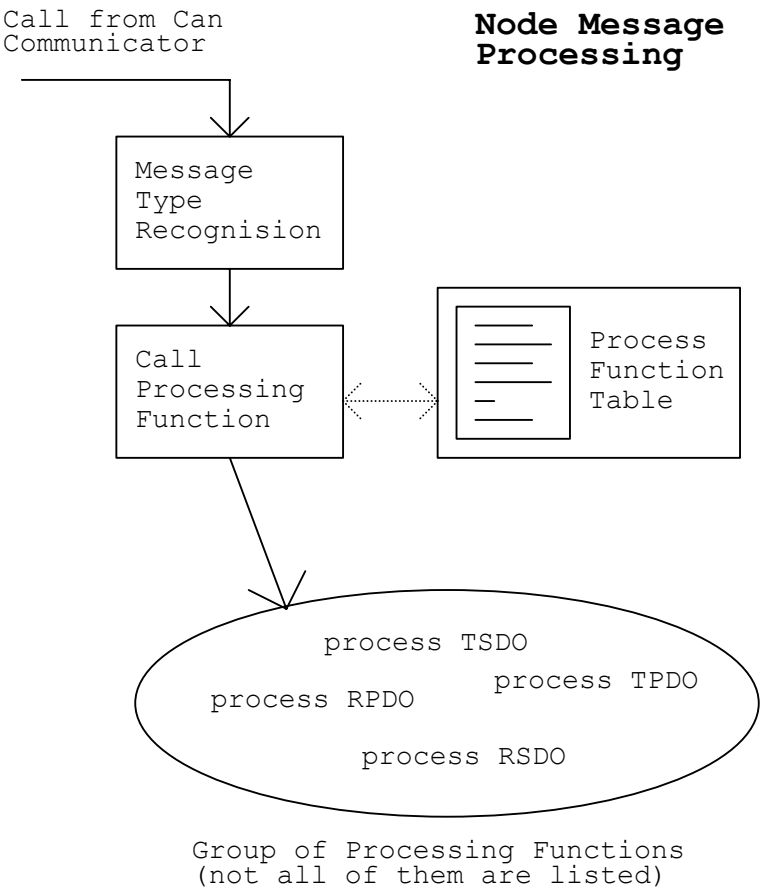

Fig. 3. Processing messages by a Node object

Functions processing individual communication objects (SDO, PDO, NMT…)

In the following text there is a description of two processing function, which treat the SDO and PDO communication objects. In general, these processing functions are defined as virtual, so redefinition and special treatment of each communication object for different nodes is possible.

SDO processing

Processing of the Transmit SDO (SDO sent by a network node), carried out by process_TSDO function, is showed in Fig. 4. The processing starts when Node's message processor the calls process_TSDO function. After the call, the process_TSDO examines the message's command specifier. Than, appropriate SDO transfer processing function is searched, based on status the variables of the communication process. If the SDO transfer processing function is found, communication can continue by calling this function. If the search is not successful, a communication error has occurred, and the transfer is aborted. In case of a successful search the process_TSDO also updates the watchdog, which takes care for the communication delays.

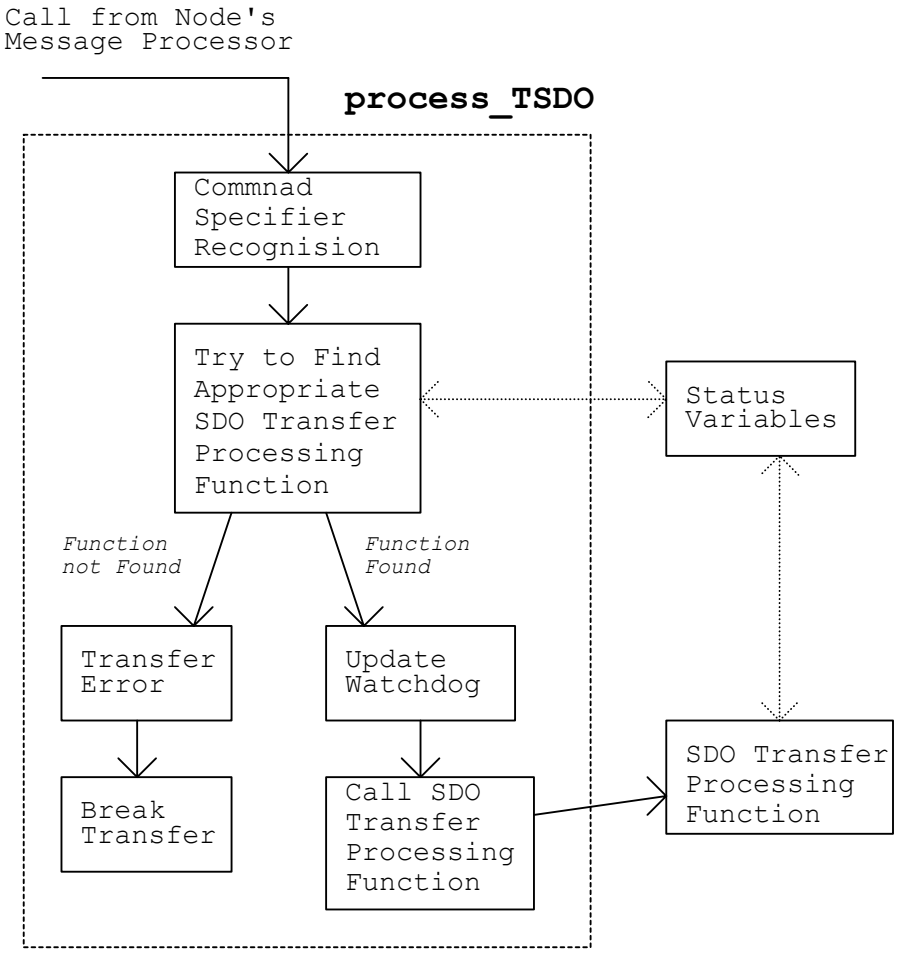

Fig. 4. Processing of Transmit SDO's

PDO processing

After receiving a PDO, the data contained in the message should be stored into the object dictionary. Specifically, PDO's data should be stored into the place where the received PDO is mapped. Storage of these data is accomplished by a PDO processing function, called process_PDO.

The flow chart of the PDO processing is showed in Fig. 5.

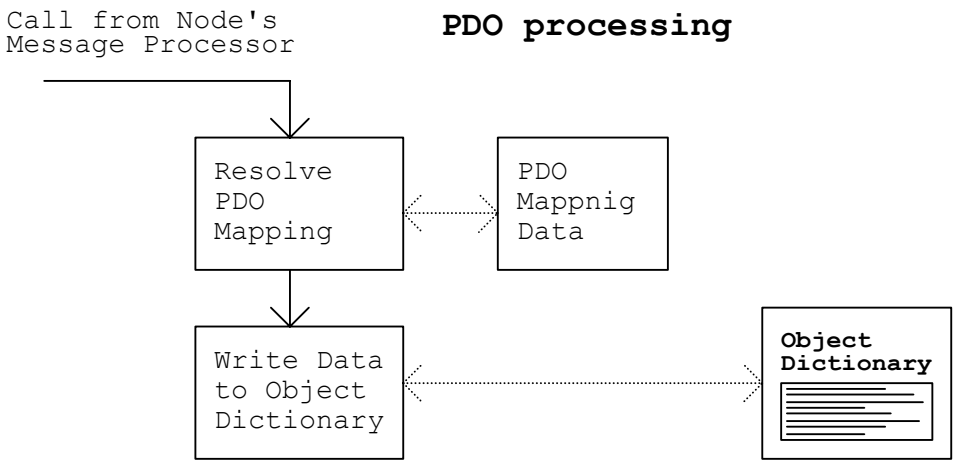

Fig. 5. Processing of PDO's

After the call from Node's message processor, the PDO processing function finds out which OD entry the incoming PDO is mapped to. Then, PDO's data are stored there.

Object Dictionary

Object dictionary (OD) is an interface between the CANopen communication and a data processing application. It is the place where all the transferred data are stored. Structure of the implemented OD is shown in Fig. 6. The OD consists from a group of index objects, each of which can have several subindex objects. The subindex objects contain actual data stored in the OD. The structure of the OD is based on a chain list; individual objects are inherited from LIST and ITEM objects, by which this chain list is implemented (Fig. 7).

Basic access to the OD is provided by the group of functions – *open, close, read,* and *write*. The o*pen* and *close* functions find position of the requested data according to the passed index and subindex values. They also lock the object dictionary, and so

prevent multiple accesses into one OD entry. *Read* and *write* functions read or store data into the OD entry.

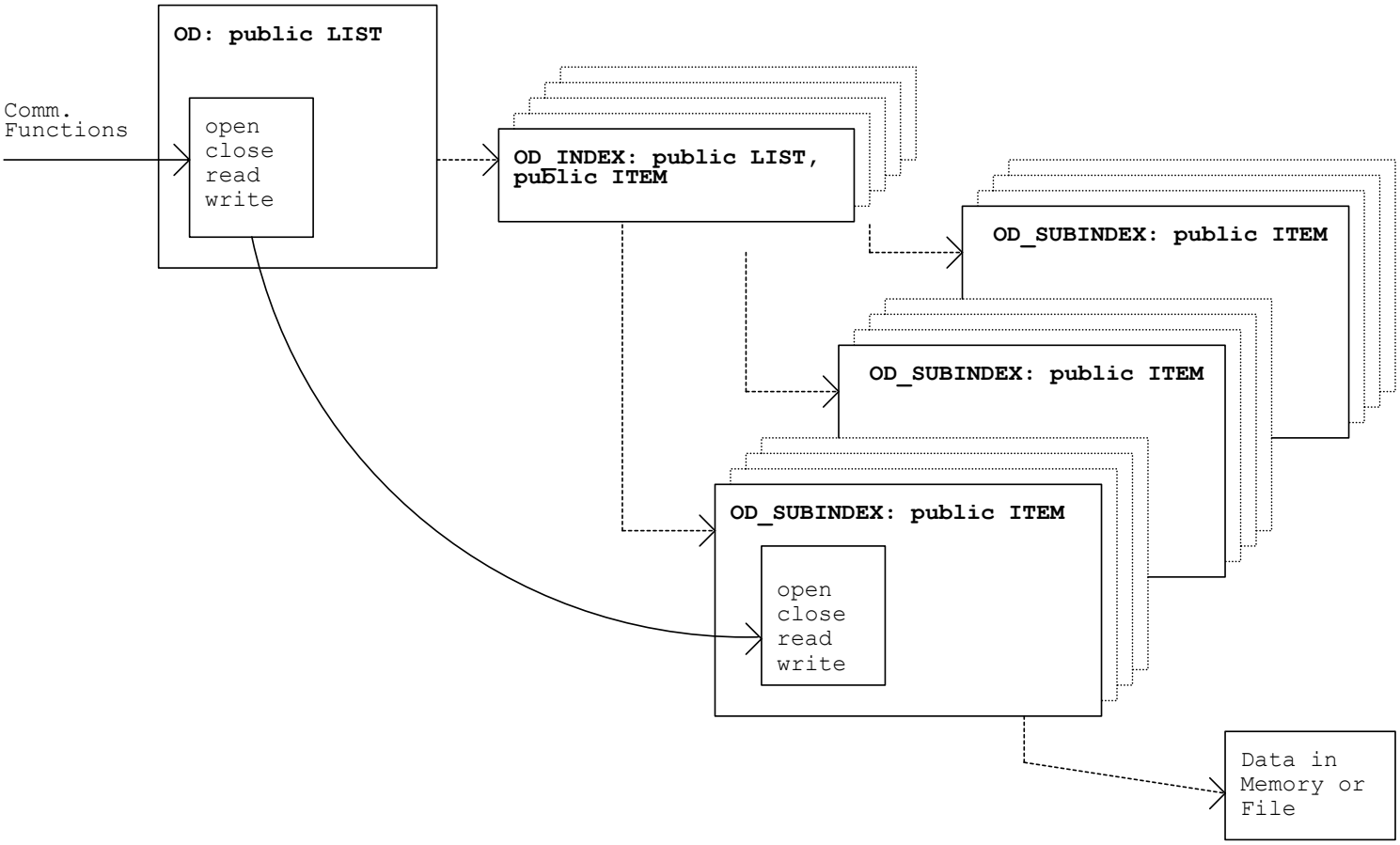

Fig. 6. Structure of Object Dictionary

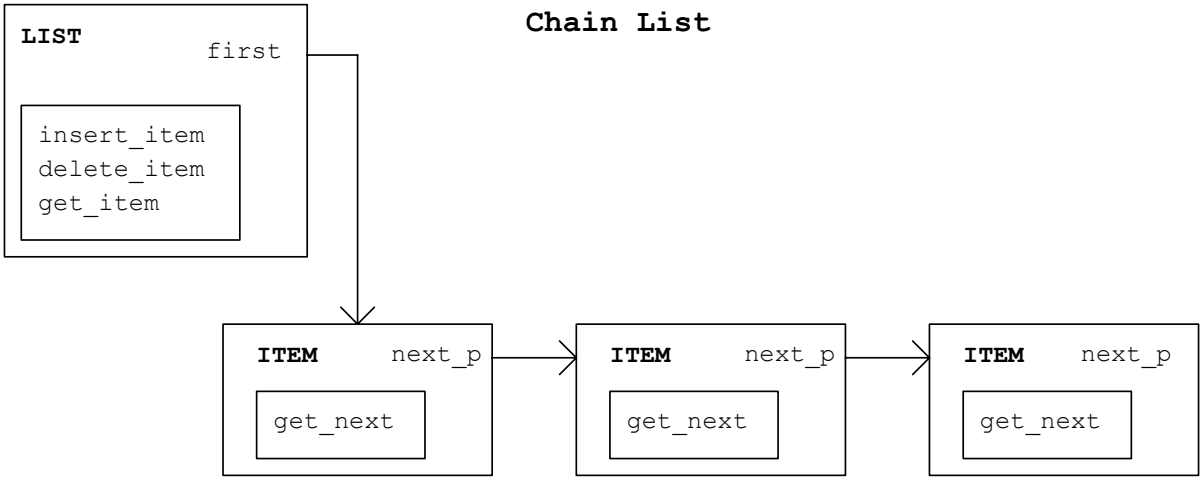

Fig. 7. Structure of chain list that OD is based on

DCF reader

The Device Configuration File (DCF) reader creates an object dictionary based on a DCF file. This allows creating complex OD structures by simply editing a text DCF file, or using a DCF file provided by a hardware vendor. The structure of the DCF reader is shown in Fig. 8.

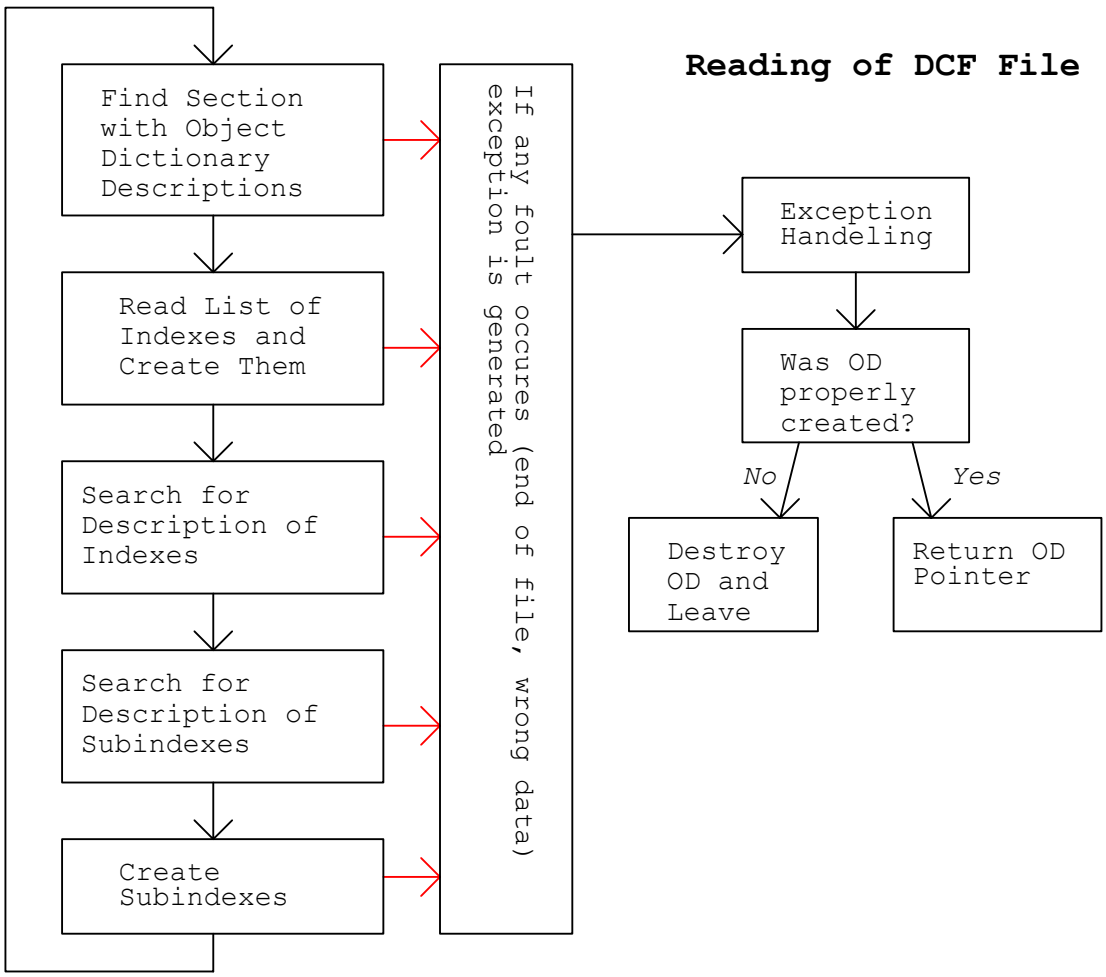

Fig. 8. Reading of DCF file

The DCF reader successively searches through a DCF file, looking for the sections which describes OD entries. When such a section is found, the description of OD indexes and subindexes is read. Also, the data types of the entries are recognized, and the appropriate part of memory is allocated.

If, in the course of the reading, any problems occur – anything from the end of the

file to an unreadable value – an exception is generated. Then, in the *exception handling* section, the DCF reader decides whether the OD was already crated (which most probably has happened when exception was generated at the end of file) or the OD wasn't created (which most likely happened if there were any syntax errors in the DCF file). If the OD was created its pointer is returned. If the OD wasn't created, any of its formed structures are destroyed, and the DCF reader returns a NULL pointer.

Watchdog

The watchdog is a part of the Can Communicator application, which tracks communication delays. Its task is to break any transfer with a node, which stopped responding. Watchdog's structure is shown in Fig. 9.

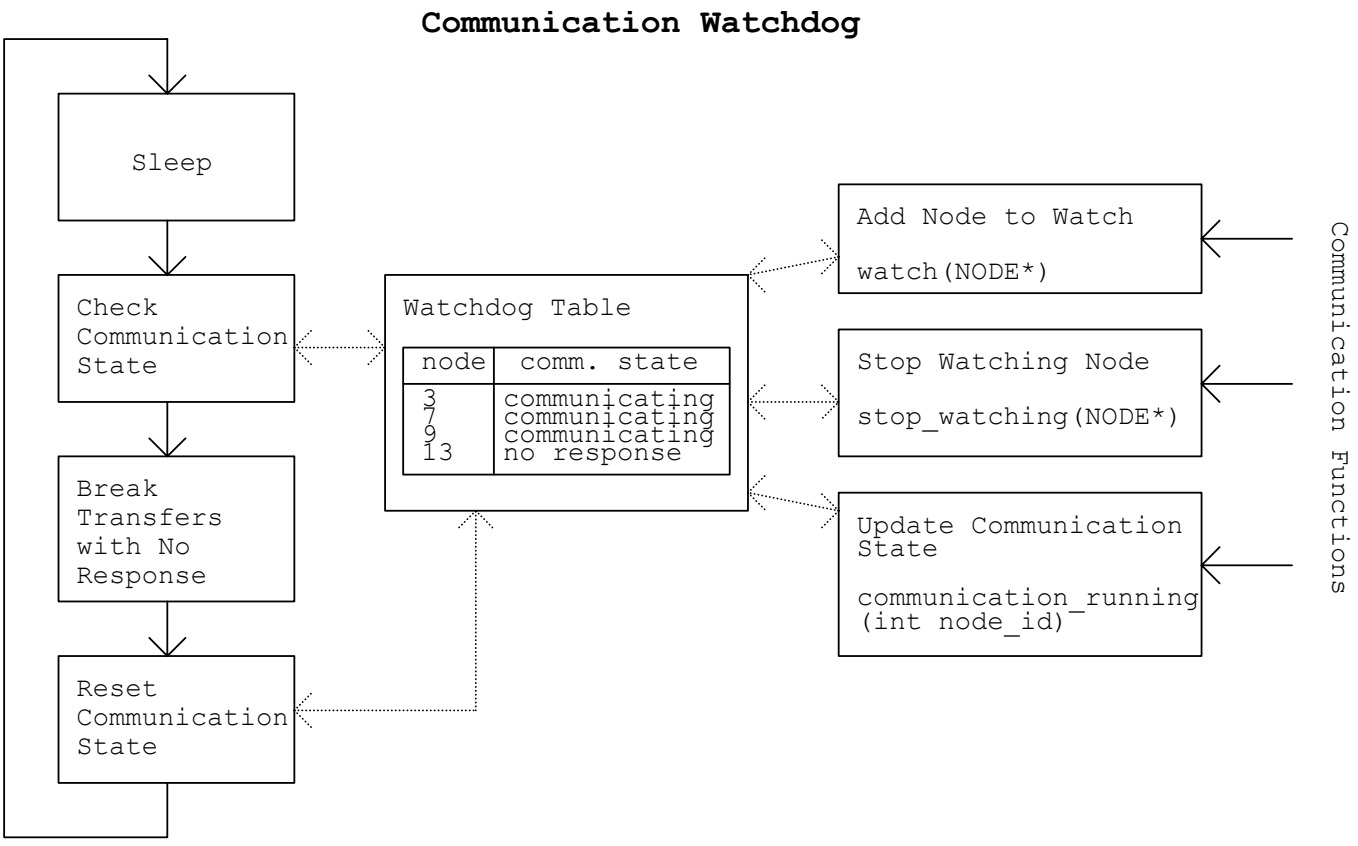

Fig. 9. Structure of communication Watchdog

The watchdog runs as a separate thread, and its function consists from the following stages: First, the watchdog sleeps for some time, giving a chance to the

network nodes to respond to Can Communicator requests. After the watchdog wakes up, it checks the internal table for any not communicating nodes. If such a node is found, the communication with it is aborted. After this, records of the communication in the internal table are cleared, and watchdog sleeps again.

To ask the watchdog to track communication, to say that the communication is running, or to say that the communication is over (no further tracking is needed), watchdog provides interfacing functions – *watch*, *stop_watching* and *communication_running*. These function updates the internal watchdog table, where watched nodes are indicated as communicating or not.

Initiating SDO transfer

The sequence of the steps that starts an SDO transfer is showed in Fig. 10.

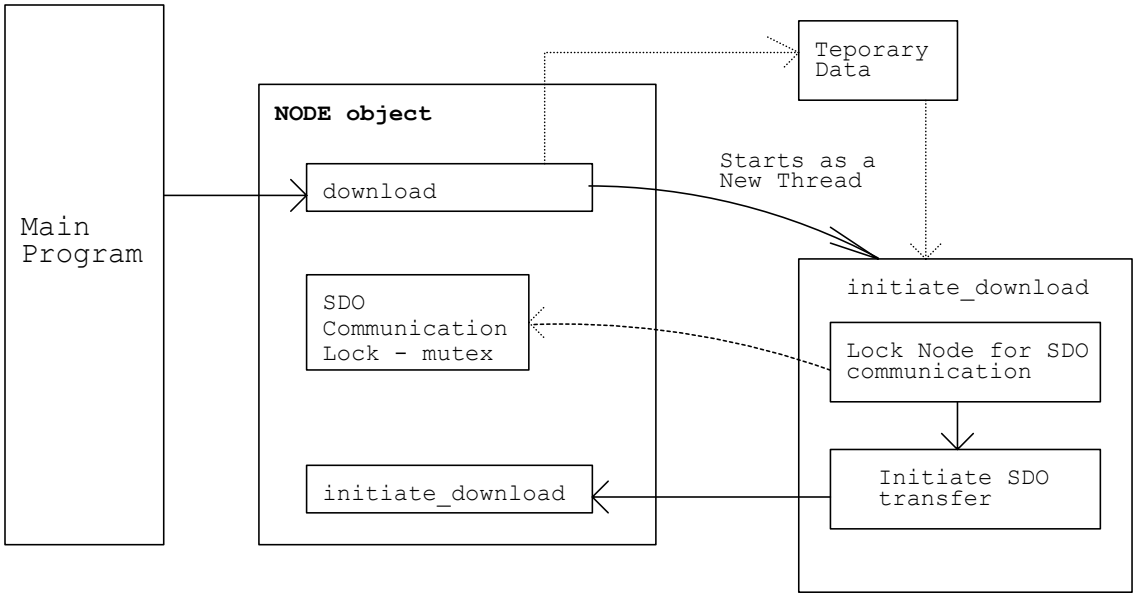

Fig. 10. Sequence of steps initiating SDO transfer

When an SDO transfer is required by the main program, it is initiated thorough calling one of the *download* or *upload* functions provided by the NODE objects. What follows than is that this *download* or *upload* function creates a new thread, which is a special *initiate_download* or *initiate_upload* function outside the NODE object. After this thread is started, it tries to lock the NODE object for the SDO communication (Two different SDO transfer with the same network node cannot run simultaneously. This situation is prevented here). If this locking is successful (the competent mutex is not already locked) the thread goes on, and calls *initiate_download* or *initiate_upload* function of the NODE object, which finally starts sending SDO messages. If the competent mutex is already locked, it means that some SDO communication is running, and so the thread is blocked until this transfer is over.

This slightly complicated structure is handy for the main program. The main loop

can request an SDO transfer, and then turn to another activity. The main program doesn't have to wait until the already running SDO transfers are over; moreover, with this system, several SDO transfers can be required at one point.

CANopen node application

The CANopen node application works as a provider of the CANopen services to the program, which controls node's hardware. It processes incoming communication objects, replies to them, and sends or stores transferred data. Also, these data are provided to the hardware controlling program thorough an object dictionary.

The structre of the CANopen node application is shown in Fig. 11

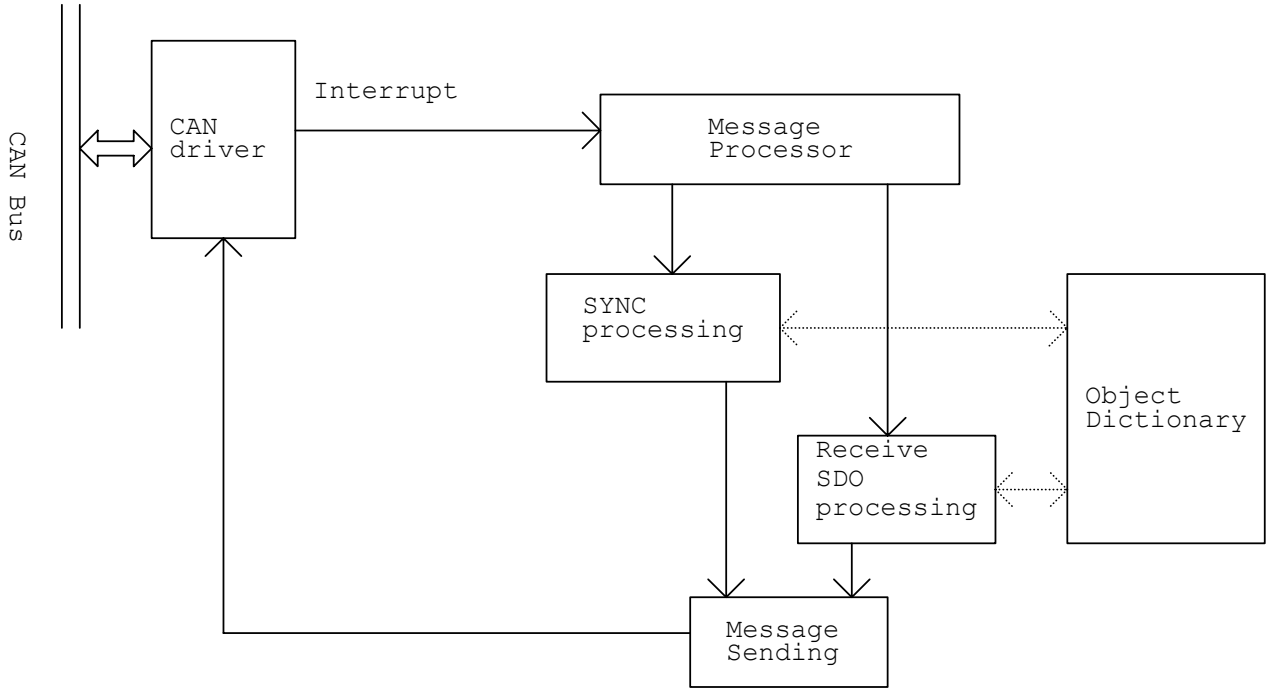

Fig. 11. Block diagram of CANopen node application

Each function of the program is started by an interrupt generated by an incoming message. The interrupt is caught by the *CAN driver*, which then calls the *Message Processor* function. This function recognizes the type of the message, and calls the next processing function, which is specialized only on one messages type (SDO, SYNC…). This specialized function, storing or reading data from the object dictionary, creates and sends an appropriate response. The detail description of the mentioned parts follows.

Message Processor

The structure of the Message Processor is shown in Fig. 12.

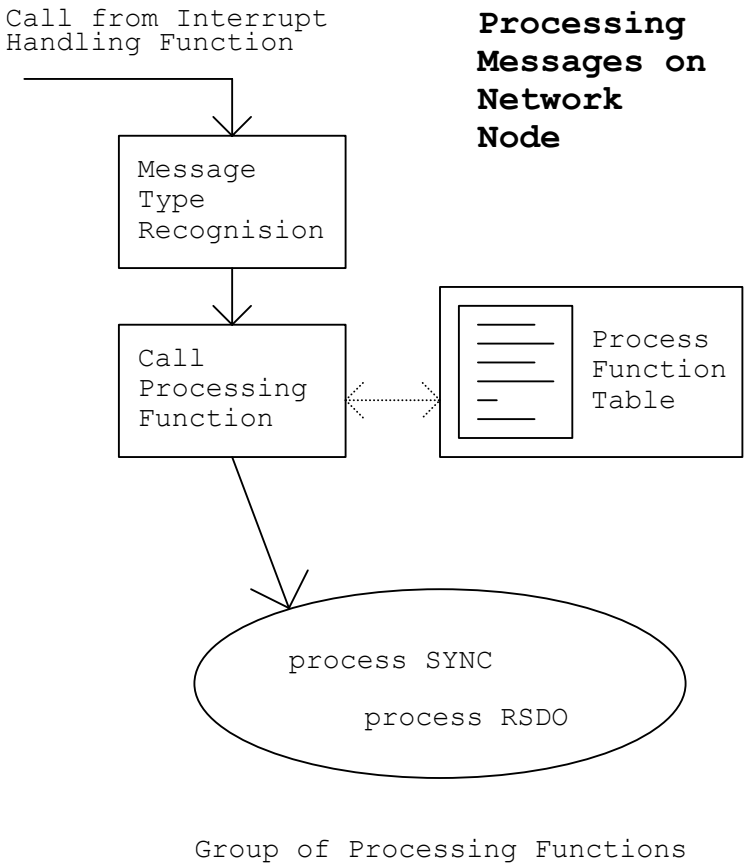

Fig. 12. Block diagram of Message Processor

The message processing algorithm consists of following steps: First the *message processor* recognizes the message type, and then, based on this type, searches for the address of the next processing function in the *Process Function Table*. If this address is found, the processing function is called. If not nothing happens.

Receive SDO processing

The behavior of the Receive SDO processing function, process_RSDO, is

indicated in Fig. 13.

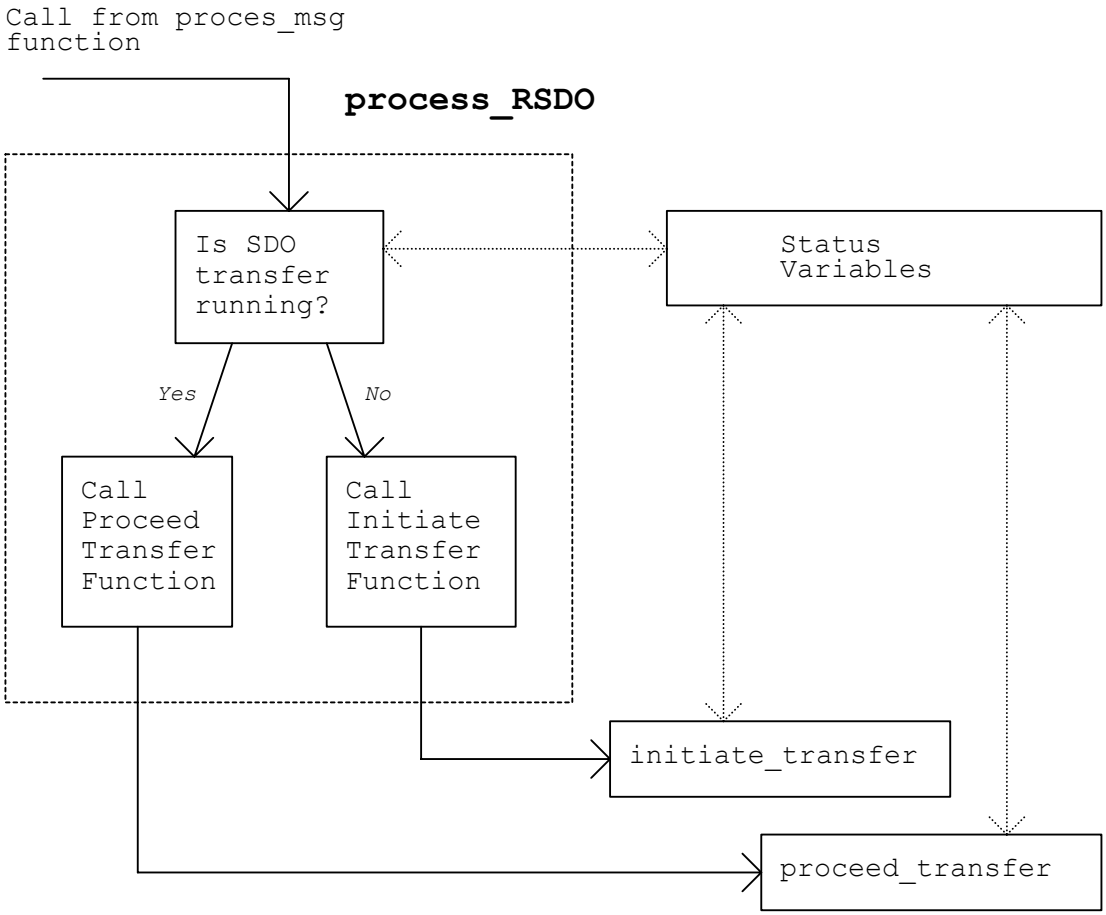

Fig. 13. Block diagram of processing of Receive SDO messages

The process_RSDO function has a simple task. It figures out whether any SDO transfer is already running, and based on this information, it calls one of two functions, which response to the incoming message. If the SDO transfer is not running the *initiate_transfer* function is called. If the SDO transfer is in progress, the *proceed_transfer* function is called. Both these functions respond based on the *Status Variables* of the transfer and they can also access the *object dictionary*.

Processing of SYNC message

In the tested CANopen node application, the response to the SYNC message was sending of the PDO. However, this reaction can differ for each node, and can by coded into the body of the *process_sync* function. Sending of the PDO, implemented in the tested case, was used to provide the data collected from a distance measuring sensor.

ACKNOWLEDGEMENTS

I want to thank my mentor Mark Sibenac for all his time and very useful advices. Also, I thank Hans Thomas for all his help with Linux programming. I feel thankful to all the AUV team members, who let me work between them. Especially, I want to thank George Matsumoto for his care about all of us – the summer interns.

References:

Boterendrood, H.(2000): CANopen high-level protocol for CAN-bus, NIKHEF, Amsterdam.

Farsi, M., Barosa, M. (2000): CANopen Implementation, applications to industrial networks, Research Studies Press LTD, Hertfordshire, England.