WP9 Validation on Platform

Deliverable D9pc.4
Process Control Application

WP9 Validation on Platform : Deliverable 9pc.4 Process Control Application
By Stanislav Benes, Petr Cvachoucek, and Ales Hajny

Published January 2005

Table of Contens

List of figures

List of tables

Document Presentation

Project Coordinator

Organisation: UPVLC
Responsible person: Alfons Crespo
Address: Camino Vera, 14, 46022 Valencia, Spain
Phone: +34 963877576
Fax: +34 963877576
Email: alfons@disca.upv.es

Participant List

Role	*Id.*	*Participant Name*	*Acronym*	*Country*
CO	1	Universidad Politecnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA/DRT/LIST/DTSI	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	*Date*	*Reason of change*
1_0	20/01/2005	First release

Chapter 1. Introduction

1.1. General

This document describes results of Ocera components validation on process control application platform. Process control application is modular software solution running in embedded system supervised from attached development host running on standard PC workstation. As the platform was choosen model of control system for the real high-pressure gas turbocompressor plant in Kralice (the Czech Republic).

In the first part of the document there is overview of application design description and in the second part validation test results follow. Description of the controlled application is contained in deliverable D9pc1. Validation results are divided into two groups, separate components validation and whole application validation.

1.2. Definitions, acronyms and abbreviations

Term	**Definition**
Unicon	UniControls proprietary distributed control system.
UcFramework	C++ class library used to develop platform-independent applications
CANopen	CAN-based higher layer protocol
Unicon CANopen Io subsystem	Subsystem of Unicon control system, targeted to communication with CANopen devices.

Table 1 – Definitions

Acronym	**Definition**
CobId	Communication Object Identifier
PDO	Process Data Object
SDO	Service Data Object
SYNC	Synchronization Object
NMT	Network Management
CAN	Controller Area Network

Table 2 – Acronyms

1.3. References

Num.	**Publisher**	**Document number**	**Title**
[1]	CIA	DS301 V.4.02	CANopen - Application Layer and Communication Profile
[2]	CIA	DS302 V.3.0	Framework for Programmable CANopen Devices

Table 3 – References

Chapter 2. Design overview

2.1. Overall description

The whole software system consists of base modules including control algorithm interpreter, ethernet communication subsystem and CANopen remote IO subsystem. Software modularity is based on open interfaces allowing easy connection of any new subsystem to algorithm interpreter.

The set of interfaces was developed in OCERA project and it allows connection of basic subsystems to interpreter according IEC 61131 norm.

Fig. 1 shows block diagram of process control node. Particular application can consist of number of nodes.

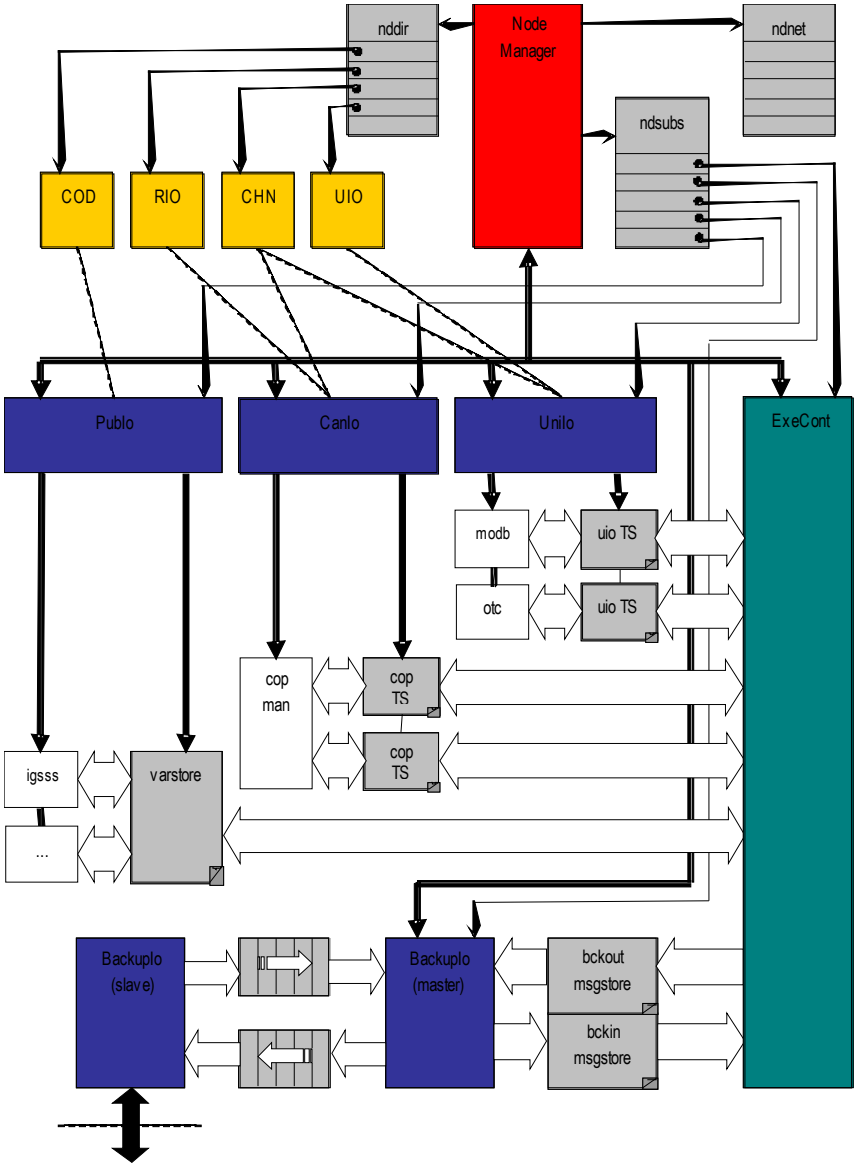

Fig 1 – Process control node bock diagram

2.2. Application porting description

2.2.1. Signals on Linux platform

Linux signals

Linux kernel works with two types of signals, standard signals and POSIX real-time signals.

Standard signals:

- only predefined signal names with defined meaning,
- no possibility for user defined signals,
- multiple instances of same signal are not queued (some signals may be lost),
- signals are not delivered in guaranteed order

POSIX real-time signals:

- signals from SIGRTMIN to SIGRTMAX (currently 32 signals),
- multiple instances of same signal are queued (no lost signals),
- signal can have an accompanying value (an integer or pointer),
- signals are delivered in a guaranteed order

Differences of framework signals and Linux signals:

Framework	**Linux**
• numbered signals using 16bit integer • unused signal numbers can be freely used by applications	• named signals • no user-defined signals
• all signals are queued	• only POSIX real-time signals are queued
• signal cannot carry additional data	• POSIX real-time signals can carry additional data

Framework signals are on Linux implemented using one POSIX real-time signal. The 16-bit signal number is stored as parameter of this signal. Signals utilized by framework are in following table.

signal name	signal value	description
SIGUCF	SIGRTMIN + 5	POSIX real-time signal used to carry the framework signal.
SIGAIO	SIGRTMIN + 6	POSIX real-time signal used for asynchronous IO operations.
SIGALRM	standard signal	signal from process interval timer
SIGINT	standard signal	signal to interrupt process

Table 4 – Signals utilized by framework

Intercept routine

The framework provides single intercept routine to handle all signals in one place. Some signals are handled specifically to provide required framework functionality not provided directly by Linux. All other signals are packed into framework messages and placed into process internal message queue for processing by application.

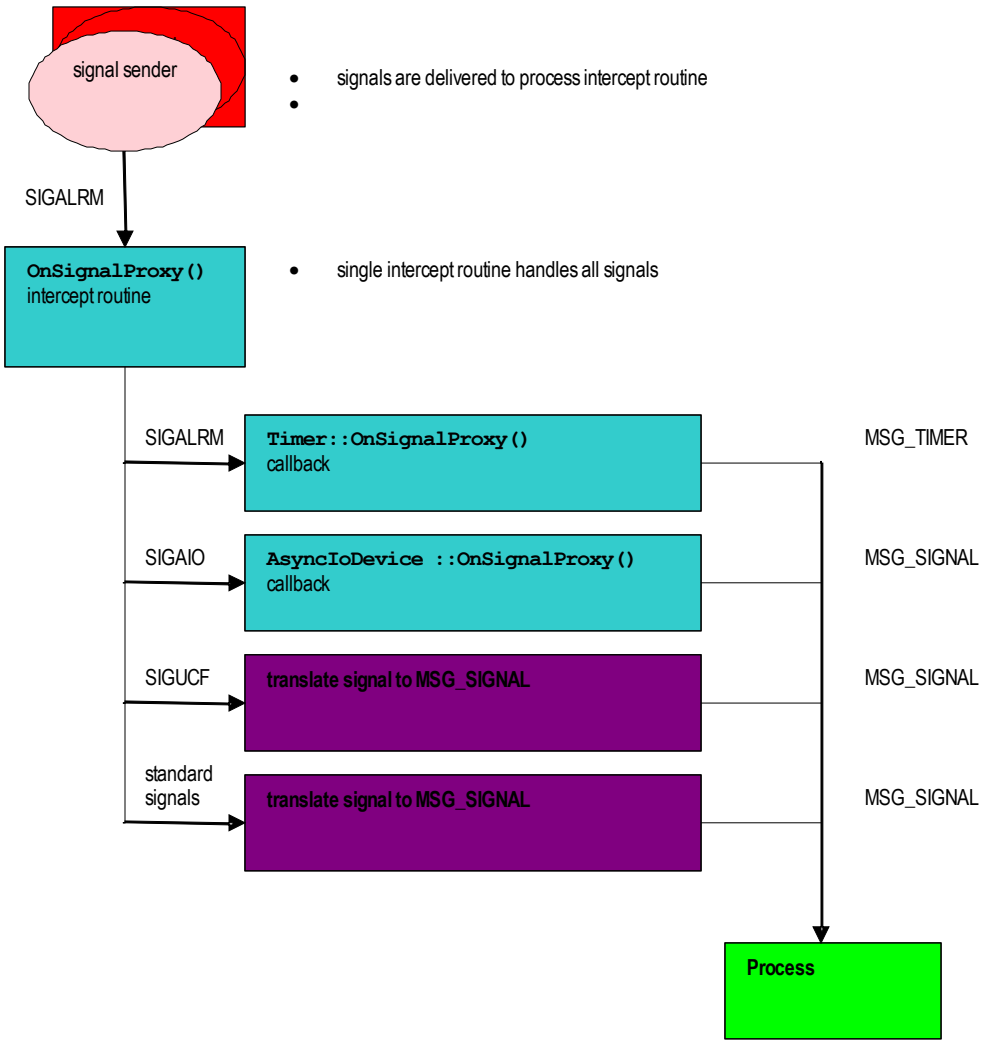

Fig 2 – Intercept routine call graph

2.2.2. Shared memory on Linux platform

SharedObject class

The Linux operating system provides memory protection. This means that process cannot directly access the memory of any other process. When the processes need to share memory, special operating system mechanisms must be explicitly used. The Linux provides several ways how to share memory between processes:

- System V IPC shared memory segments,
- virtual memory filesystem (tmpfs, formerly known as shmfs).

System V IPC shared memory mechanism has several important disadvantages for use in framework. The most important one is that memory segments are identified only by 32 bit key (integer value). The framework needs a named memory segments. The System V IPC API provides a function for generation of segment key from an existing file name, but there still can be collisions in generated keys for different files.

Virtual memory filesystem works for applications as an obvious filesystem. The difference is that files are kept only in virtual memory, they are not stored on any other medium. Together with the memory-mapping files mechanism this can be used to share memory between different processes.

The framework uses virtual memory filesystem for shared memory implementation. The mechanism is encapsulated in SharedObject class. SharedObject class is not a public framework class, can be used only by framework itself.

SharedObject class attributes:

- implements reference counter for class instances,
- provides methods for creation and accessing shared memory,
- when the object's reference count reaches zero, the object is automatically deleted,
- objects can be sorted by their purpose

The virtual memory filesystem is usually mounted in **`/dev/shm`** mount point. Framework defines several types of shared objects and every object type have its own prefix in filename path.

object type (prefix)	description
`/module`	objects used to implement DataModule class
`/event`	objects used to implement Event class • in object is stored System V IPC semaphore key used to implement event
`/mmfile`	objects used to implement FileMapping class • in object is stored mapping object size and full pathname to mapped file
`/mutex`	objects used to implement Mutex class • in object is stored System V IPC semaphore key used to implement mutex
`/pipe`	objects used to implement Pipe class • in object is stored pipe type and other information
`/semaphore`	objects used to implement Semaphore class • in object is stored System V IPC semaphore key used to implement semaphore

Table 5 – Shared objects types

The SharedObject data layout is very simple.

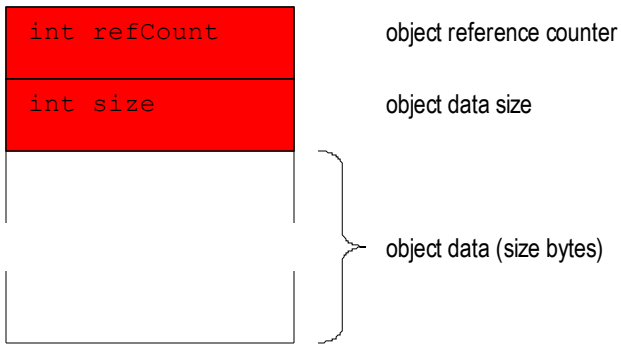

Fig 3 – SharedObject data layout

DataModule class

- DataModule class is derived from SharedObject class, so inherits all its features
- as object type is specified **`/module`** (modules can be found on **`/dev/shm/module`** path)

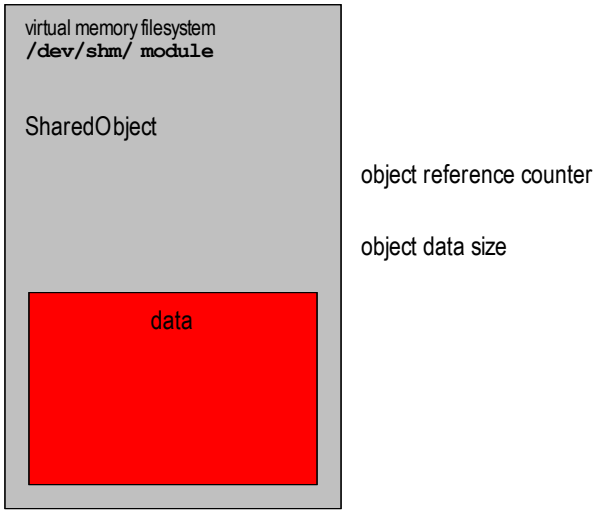

Fig 4 – SharedObject data layout for DataModule class implementation

2.2.3. Synchronization objects on Linux platform

Linux semaphores

When several processes need to use the same resources, some synchronization mechanisms are needed. The Linux provides two types of synchronization objects:

- System V IPC semaphores,
- POSIX semaphores

Unfortunately Linux doesn't support POSIX semaphores which can be accessed by different processes (although POSIX defines them). This Linux "feature" disqualifies POSIX semaphores from usage in framework.

The framework utilizes System V IPC semaphores for process synchronization.

- System V IPC semaphores are identified by 32-bit key (integer value),
- key is generated by function `ftok()` from the file representing shared object (stored in virtual memory filesystem),
- resulting key is stored in shared object, so other processes can access the right semaphore,
- `sem_get()`, `sem_ctl()` and `sem_op()` calls are used to work with semaphores,
- detailed information can be found in Linux man pages

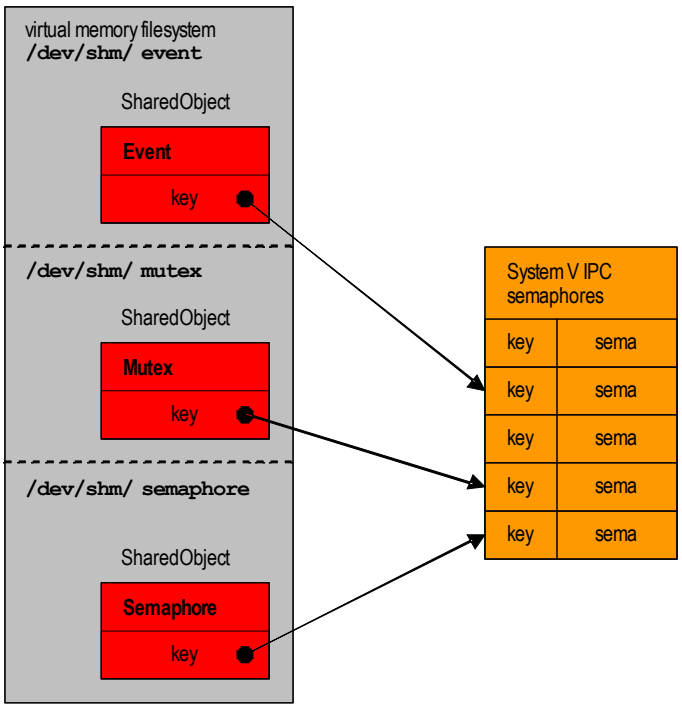

Fig 5 – Synchronization objects implementation overview

Event class

- Event class is derived from SharedObject class, object type is specified as `/event` (events can be found on `/dev/shm/event` path)
- in the shared object data is stored the System V IPC semaphore key used to access real semaphore and auto/manual event flag

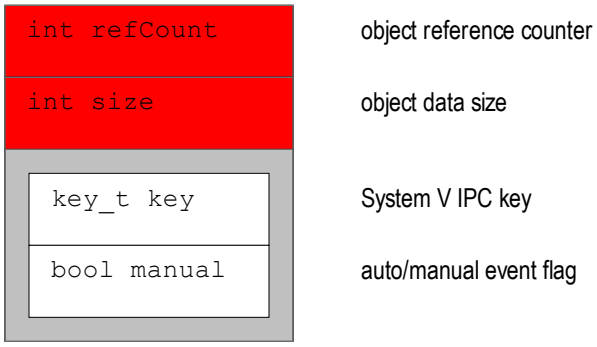

Fig 6 – SharedObject data layout for Event class implementation

Mutex class

- Mutex class is derived from SharedObject class, object type is specified as `/mutex` (mutexes can be found on `/dev/shm/mutex` path)
- in the shared object data is stored the System V IPC semaphore key used to access real semaphore

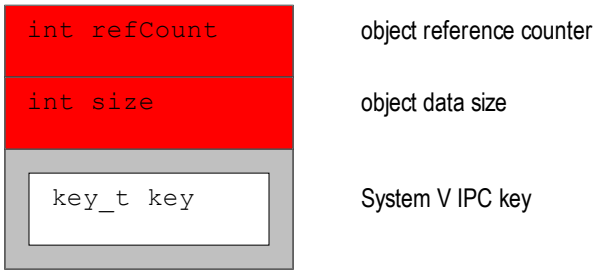

Fig 7 – SharedObject data layout for Mutex class implementation

Semaphore class

- Semaphore class is derived from SharedObject class, object type is specified as `/semaphore` (semaphores can be found on `/dev/shm/semaphore` path)
- in the shared object data is stored the System V IPC semaphore key used to access real semaphore

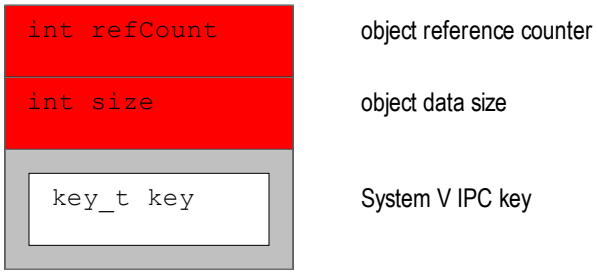

Fig 8 – SharedObject data layout for Semaphore class implementation

2.2.4. Timers on Linux platform

The Linux provides each process with three interval timers, each decrementing in a distinct time domain. When any timer expires a signal is sent to the process and the timer is (potentially) restarted. The three time domains are:

time domain	description
ITIMER_REAL	decrements in real time, delivers SIGALRM upon expiration
ITIMER_VIRTUAL	decrements only when the process is executing, delivers SIGVTALRM upon expiration
ITIMER_PROF	decrements both when the process executes and when the system is executing on behalf of the process, delivers SIGPROF upon expiration

Table 6 – Linux interval timers

The framework application is not limited to single timer, which is provided to Linux applications. The framework must emulate all application timers using single timer provided by Linux.

The implementation is based on:

- starting interval timer ITIMER_REAL provided by Linux,
- timer delivers periodically the SIGALRM signal,
- when signal is delivered, the list of application timer is checked if any of application timers has expired

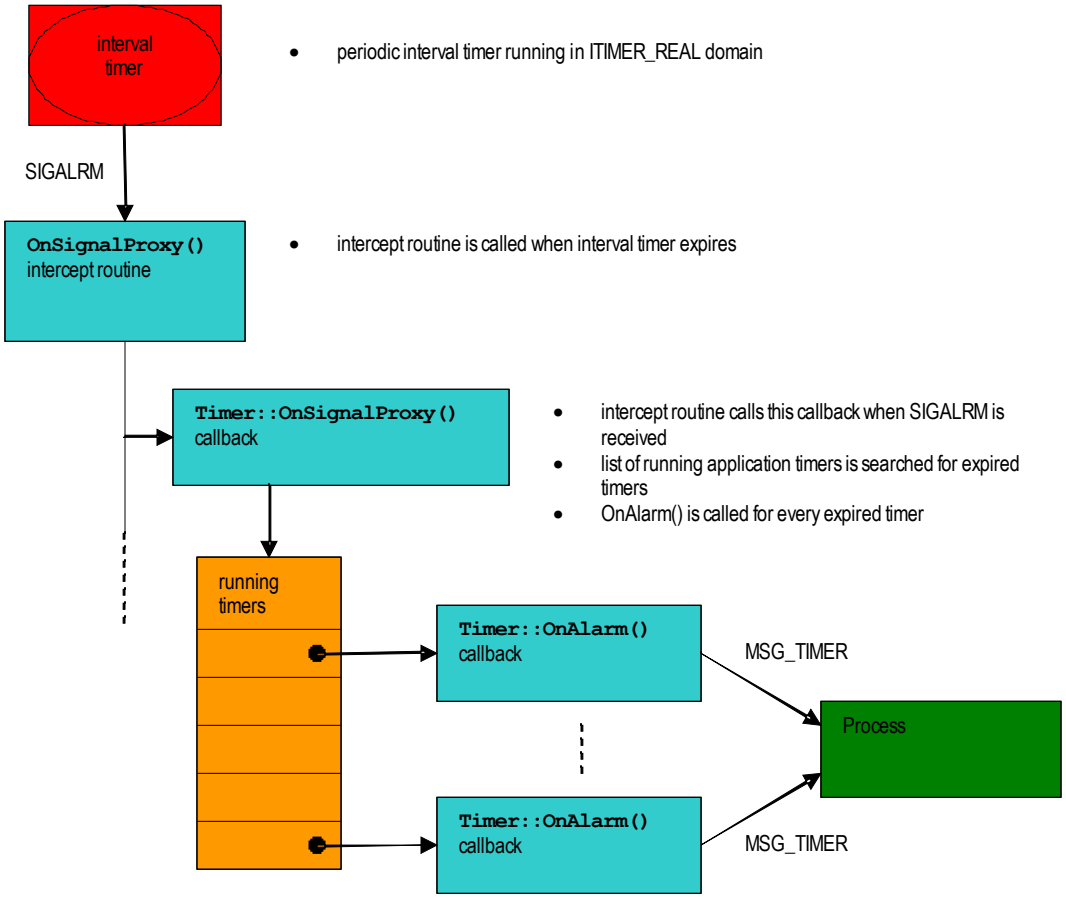

Fig 9 – Timer class implementation call graph

2.2.5. Files on Linux platform

File class - low-level file interface

The File class is on Linux implemented straightforward. There are no special differences in files-related API provided by Linux from any other operating system.

method	implemented using	description
`File::Open()`	`open()`	Open specified file.
`File::Close()`	`close()`	Close file.
`File::Write()`	`write()`	Write to file.
`File::Read()`	`read()`	Read from file.
`File::Eof()`	`fstat()`	Check for end-of-file condition.
`File::Tell()`	`lseek()`	Return current file pointer value.
`File::Seek()`	`lseek()`	Set file pointer.
`File::Size()`	`fstat()`	Return file size.

Table 7 – Low-level file interface overview

FileMapping class - memory mapped files

The Linux supports mapping files into memory using `mmap()` API.

- FileMapping class is derived from SharedObject class, object type is specified as `/mmfile`.
- in shared object data part is stored mapping object size and full path to mapped file

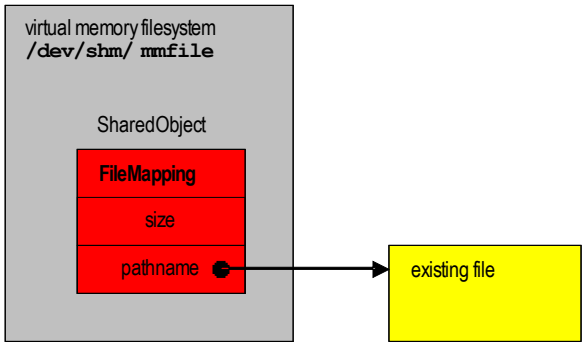

Fig 10 – File mapping implementation overview

2.2.6. Input/Output on Linux platform

I/O availability signals

The Linux supports asynchronous IO using the `fcntl()` API. The signal SIGAIO is delivered to process when IO becomes available. Because there is no possibility to define user signals, all IO operations share the same signal. The system provides only the file descriptor number of the file where IO is available. The framework must maintain the list of asynchronous IO operations and when signal is delivered the list is searched for object who serves the IO on given file.

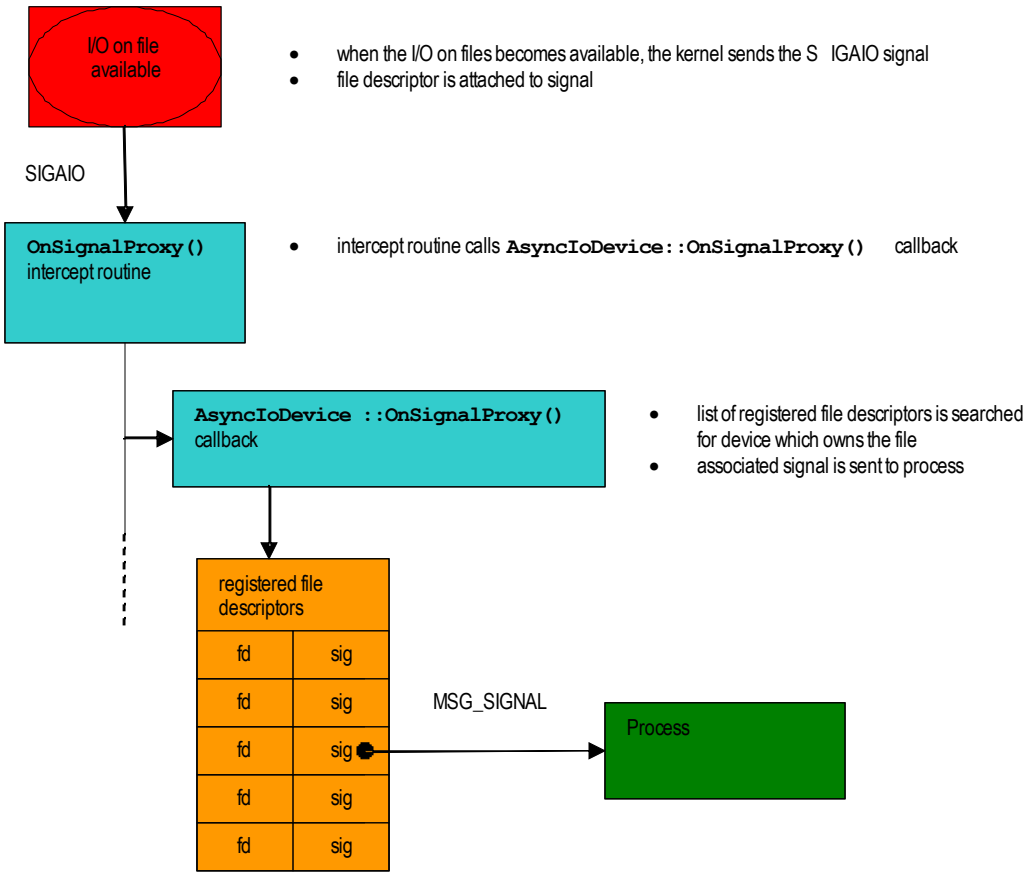

Fig 11 – Asynchronous IO notifications call graph

Serial ports

Serial ports on Linux are accessible using low-level file interface. Functions for manipulating serial ports setup are placed in termios library. Device files used to access serial ports on Linux are usually named **`/dev/ttyS0`** and so on. ComPort class is derived from AsyncIoDevice class and inherits support for I/O availability signals.

method	implemented using	description
`ComPort::Open()`	`open()`	Open specified serial port.
`ComPort::Close()`	`close()`	Close serial port.
`ComPort::Setup()`	`tcgetattr()` `tcsetattr()`	Setup serial port (baud rate, data format).
`ComPort::SetChars()`	`tcgetattr()` `tcsetattr()`	Setup transmission control characters (XON, XOFF ...).
`ComPort::Read()`	`read()`	Read from serial port.
`ComPort::Write()`	`write()`	Write to serial port.
`ComPort::SetSignal()`	`fcntl()`	Install notification signal for IO availability.
`ComPort::ReleaseSignal()`	`fcntl()`	Disable notification signal.
`ComPort::GetReadyCount()`	`ioctl()`	Return number of bytes ready for reading.

Table 8 – Serial ports interface overview

Console – terminal screen

Linux console is compatible with default ANSI/VT100 terminals. The Console class is directly derived from AnsiTerminal class and implements only low-level methods `PutString()` and `PutChar()` using the low-level file `write()` API.

Pipes – simple interprocess communication

Linux named pipes are implemented as special files on the filesystem. These files are created on virtual memory filesystem along the shared object file with additional control data. The Linux doesn't support IO availability signals for pipes, so the framework must provide its own implementation.

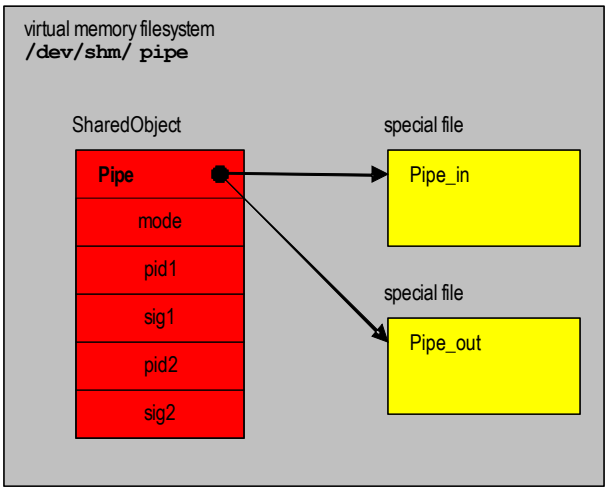

Fig 12 – Pipe implementation overview

Sockets – network communication

Linux natively supports network communication using the BSD sockets API. IO availability signals are also supported for sockets. Implementation is quite straightforward.

2.3. CANopen subsystem manager for Unicon

2.3.1. Overview

CANopen subsystem manager is implemented as process named CopIo (CANopen IO). The process is tightly coupled with Unicon in several ways:

- process works with configuration from Unicon Application Target Database (ATD),
- process is managed by Unicon Node Manager,
- process data from/to CANopen modules are stored in traffic store in a compatible way with older CANopen manager used in Unicon node (process copman).

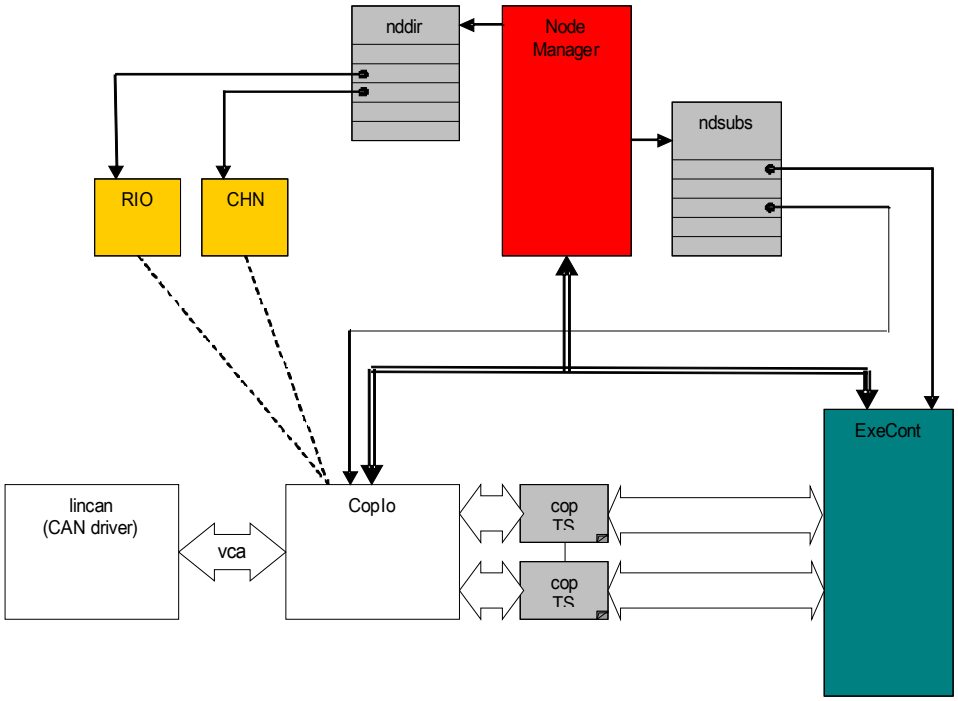

Fig 13 – CANopen subsystem manager for Unicon

2.3.2. External interfaces

Communication with Unicon Node Manager

Unicon node manager communicates with its subsystems using FIFO message queues. Every subsystem after its startup creates own input message queue and registers the queue in the node manager. The message queue is an instance of object class IpcMsgQueue. Notifications to node managers are sent to his input message queue. The list of messages and their meaning is in the following table, comprehensive description can be found in SD0577A document.

Message	Description
NDMSG_SUBSYS_REGISTER	The subsystem registers its own input message queue in the node manager.
NDMSG_SUBSYS_WATCHDOG	The subsystem registers the watchdog timer in the node manager.
NDMSG_SUBSYS_RESPAWN	The subsystem requests its own respawn (in case of fatal failure).
NDNTF_SUBSYS_ACTIVE	The subsystem notifies the node manager that its status changed to ACTIVE.
NDNTF_SUBSYS_STANDBY	The subsystem notifies the node manager that its state changed to STANDBY.
NDNTF_SUBSYS_READY	The subsystem notifies the node manager that its state changed to READY.
NDNTF_SUBSYS_FAILURE	The subsystem notifies the node manager that its state changed to FAILURE.
NDNTF_SUBSYS_RESTART	The subsystem notifies the node manager that its state changed to STARTING.
NDCMD_SUBSYS_ACTIVATE	The message is a command for subsystem to change status to ACTIVE.
NDCMD_SUBSYS_STANDBY	The message is a command for subsystem to change status to STANDBY.
NDCMD_SUBSYS_STOP	The message is a command for subsystem to change status to READY.
NDCMD_SUBSYS_SHUTDOWN	The message is a command for subsystem to shutdown itself.
NDCMD_SUBSYS_RESTART	The message is a command for subsystem to restart with new parameterization.

Table 9 – Unicon node manager messages

RIO module format specification

The RIO parameterization module contains a description of remote inputs and outputs.

module header	Module header is a structure of type sXEDA_FILE_HEAD. • creation time • project name • node name • node number • module revision
number of nets	Total number of nets.
number of modules	Total number of io modules.
array of net descriptors	Net descriptors are structures of type sRIO_NET. - every net descriptor is followed by net parameters - type and size of net parameters depends on net class (sCAN_NET_PAR for CANopen nets)
array of module descriptor offsets	Array of offsets of io module descriptors.
array of module descriptors	Io module descriptors are structures of type sRIO_MOD. Every module descriptor is followed by: • array of its input channel descriptors (array of sRIO_IOCHNL structures), • array of its output channel descriptors (arrays of sRIO_IOCHNL structures), • module internal parameters (type and size of module internal parameters depends on module class – sCAN_MOD_PAR for CANopen modules)

CHN module format specification

The CHN parameterization module contains a description of communication channels.

module header	Module header is a structure of type sXEDA_FILE_HEAD. • creation time • project name • node name • node number • module revision
number of channels	Total number of channels.
array of channel descriptor offsets	Array of offsets of channel descriptors.
array of channel descriptors	Channel descriptors are structures of type sCHNL_DESC. - every channel descriptor is followed by channel parameters - type and size of channel parameters depends on channel type (sCAN_NET_CHNL for CAN channels)

Traffic store format specification

Traffic store is a data module where process data are store stored.

traffic store header	Traffic store header is a structure of type sRIO_TS_COP. • net status • inputs enable flag • outputs enable flag • number of modules
array of module control tables	Module control tables are structures of type sRIO_COP_MCT. • number of ports in every of 4 module cages • offsets of first port of every used module cage • module status • module io channels validity bits
traffic store ports	Data are stored in ports. • port is 16 bytes of memory • cage can have max. 8 ports • every module has 4 cages

2.3.3. Decomposition to classes

The design of CANopen Io manager can be split into three basic hierarchical levels.

Io subsystem manager level

- handles communication with node manager using message queues,
- attaches to Application Target Database (ATD) modules RIO and CHN,
- opens and owns the subsystem configuration file or data module,
- creates and owns subsystem log,
- creates and owns timer providing subsystem tick (base period for timeout evaluations),
- creates and manages individual CANopen networks (can manage up to eight networks),
- evaluates overall subsystem status as composition of states of individual CANopen networks

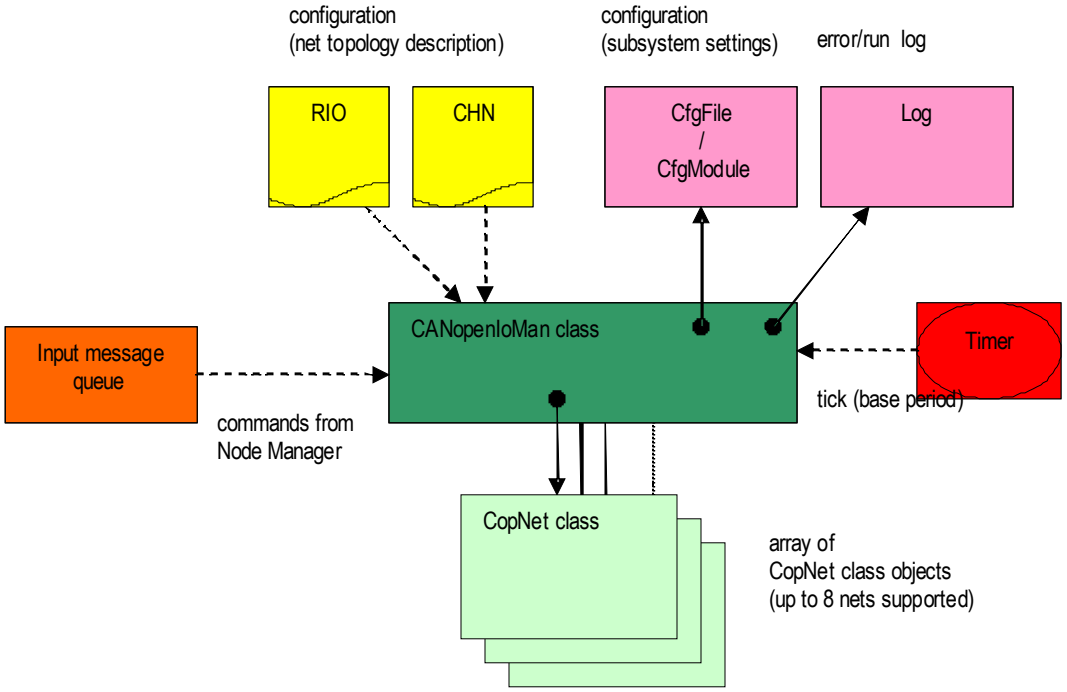

Fig 14 – Io subsystem manager level design overview

CANopenIoMan class members

data type	member		description
`IpcMsgQueue`	`msqQueue`	-	input message queue • node manager sends his commands as messages
`IpcMsgQueue`	`ndQueue`	-	node manager's input message queue • notifications and responses to node manager's commands are sent back as messages
`NdStatus`	`status`	-	Unicon subsystem status • see the [3] for detailed description
`LogExt`	`log`	-	error/run log • all significant events are recorded to log, • all errors are recorded, • log can be configured (verbosity, destination – file, data module, non-volatile memory), • log can be disabled
`Ptr<CfgBase>`	`cfg`	-	configuration database • needed to keep additional configuration not contained in ATD, • for example the communication bitrate can be specified
`RioBase`	`rio`	-	ATD RIO module • describes all remote io networks in Unicon node (include CANopen ones), • the number and types of remote io devices are described, • every CANopen network is connected to one CAN communication channel described in CHN module

`ChnBase`	`chn`	-	ATD CHN module • describes all communication channels and their parameters in Unicon node • CAN communication channel is specified by symbolic name of associated CAN controller (for example '/dev/can0')
`Timer`	`tmr`	-	timer object • used for periodic tick signal generation, • tick signal is used to feed the application timer queue tmrQueue
`TimerQueue`	`tmrQueue`	-	ordered queue of application timers • used for timeout evaluation throughout entire application, • provides services to register timer event with callback when event expires, • required for efficient implementation of large number of simultaneously running application timers
`Ptr<CopNet>`	`net[8]`		array of pointers to instances of CopNet class • every managed CANopen is represented by instance of CopNet class • when the net of given index is not defined, the pointer is NULL • array size is hard coded to 8, which defines maximum number of managed networks (can be changed if necessary)

CANopenIoMan class operations

method		**description**
`Init()`	-	initialization method of framework application, provides: • command line arguments parsing, • opens configuration file/module, • calls InitResources() method, • calls StartupSubsystem() method, • registers application to Unicon node manager
`InitResources()`	-	initializes general subsystem resources as: • opens and configures log, • creates and configures input message queue, • installs periodic tick timer
`StartupSubsystem()`	-	performs steps necessary to startup entire subsystem: • connect to ATD modules directory, • find activated RIO and CHN ATD modules, • build resources for all CANopen networks defined in RIO
`ShutdownSubsystem()`	-	performs steps necessary to shutdown subsystem: • free resources allocated for all CANopen networks
`ActivateSubsystem()`	-	performs steps necessary to activate subsystem: • activate all managed CANopen networks
`StandbySubsystem()`	-	performs steps necessary to standby subsystem: • standby all managed CANopen networks
`StopSubsystem()`	-	performs steps necessary to stop subsystem: • stop all managed CANopen networks, • network resources remain allocated
`RestartSubsystem()`	-	performs subsystem restart as sequence of: • subsystem shutdown, • subsystem startup with new RIO and CHN modules

`BuildAllNets()` - builds resources for all networks:
- iterates through RIO module, search CANopen networks,
- checks consistency of information in RIO module,
- for every found CANopen network calls its BuildNet() method

`BuildNet()` - build resources for single network:
- iterate through RIO module,
- search for modules connected to given CANopen network,
- allocate instance of class associated with found module type
- attach module to network

CANopenIoMan class handlers

method	**description**
`OnMsgQueue()`	- input message queue handler 1. signal is sent when there is a message in input message queue 2. handler reads message, 3. calls operation associated with received command
`OnTick()`	- periodic timer signal handler • application timer queue is evaluated for expired timers
`OnCanMsg()`	- CAN controller signal handler • network's OnCanMsg() handler is called
`OnNetStatusChange()`	- network status change notification handler • method is called by managed CANopen networks when their status changes • overall subsystem status is updated depending on states of all managed networks
`OnException()`	- uncatched exception handler • method is called when exception is thrown and not catched by any other catch block • exception is logged into subsystem log

Network level

- opens and owns CAN controller device used for given CANopen network,
- creates and owns traffic store for process data storage,
- creates and manages set of CANopen module objects (they represent individual devices on the network),
- provides SYNC message generation,
- evaluates overall CANopen network status as composition of states of individual CANopen modules

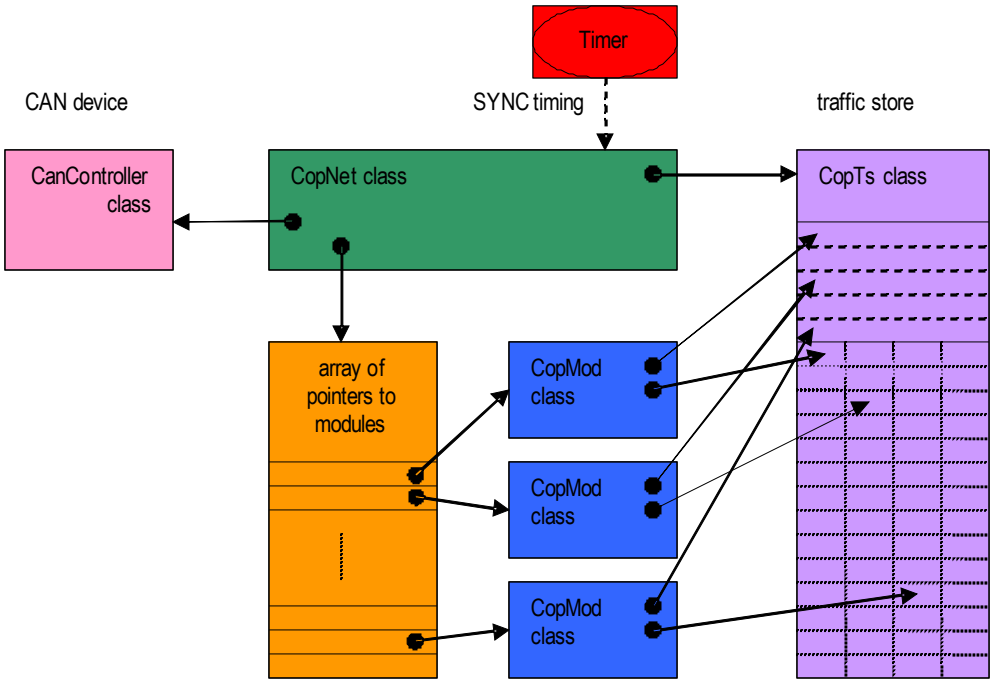

Fig 15 – CANopen network level overview

CopNet class members

data type	member		description
`CopStatus`	`status`	-	CANopen network status
`CANopenIoMan *`	`man`	-	pointer to CANopenIoMan which owns this network • pointer is used to call OnNetStatusChange() callback method

`CanController`	`can`	-	CAN controller class
			• provides an abstract layer between the application and real hardware
			• this allows to easily port the application to use various CAN controller hardware
			• current implementation supports:
			  ○ lincan driver on Linux platform,
			  ○ usb-can driver on Windows platform,
			  ○ vcan driver on OS9 platform

`CopNetDesc`	`nd`	-	CANopen network descriptor in RIO module • descriptor provides network configuration parameters, ie. SYNC message period
`CopTs`	`ts`	-	traffic store object • traffic store is used for process data stotage
`Array<Ptr<CopMod>>`	`rio`	-	array of pointers to CANopen modules • size of array is equal to number of network modules
`CopMod *`	`idMap[128]`	-	CANopen module ID cache • cache for fast resolving of “module ID to C++ object” • in cache is address of appropriate C++ object representing CANopen module with given CANopen module ID

CopNet class operations

method		description
`RegisterModule()`	-	registers new module into network internal structures • called during building of network resources, • module is queried for required number of ports in traffic store • address of module object is stored in module ID cache
`BuildNetwork()`	-	creates network resources for all registered modules • called as final step of building of network resources, • traffic store is created, • all registered modules are attached to their MCTs and ports in traffic store, • CAN controller is opened
`ActivateNetwork()`	-	performs steps necessary to activate network • all modules are switched to pre-operational state • node guarding is enabled for all modules • module configuration is started
`StandbyNetwork()`	-	performs steps necessary to standby network • no messages are sent to the network in standby mode, • all PDOs are intercepted and process data are transferred to traffic store (traffic stores in active and standby nodes are synchronized)
`StopNetwork()`	-	performs steps necessary to stop network • SYNC message producer is disabled, • node guarding for all modules is disabled
`SendMsg()`	-	sends raw CAN message to network
`SendNmt()`	-	prepares and sends NMT message to network • message can be addressed to specified module, • or can be sent as broadcast to all modules
`SendNdg()`	-	prepares and sends NDG message to network • message is sent to specified module only
`SendSync()`	-	prepares and sends SYNC message to network

CopNet class handlers

method		description
`OnCanMsg()`	-	received CAN message handler • message type and module ID is extracted, • appropriate module object handler is called
`OnSyncTimer()`	-	SYNC timer callback handler • SYNC message is sent to the network • OnSync callback is called for all modules
`OnModStatusChange()`	-	module status change handler • method is called by modules when their status changes • overall network status is updated depending on states of all modules • and appropriate operation is called when needed

Module level

- keeps pointer to module control table in traffic store (in module control table are stored process data validity bits and module status bits),
- keeps pointers to module process data in traffic store ports,
- owns SDO state machine for service data objects upload/download,
- provides NDG message generation for given module,
- evaluates timeouts for responses to NDG messages and SDO communications,
- provides mechanism for module configuration (module can be configured using a sequence of SDO downloads before its switched into an operational state)

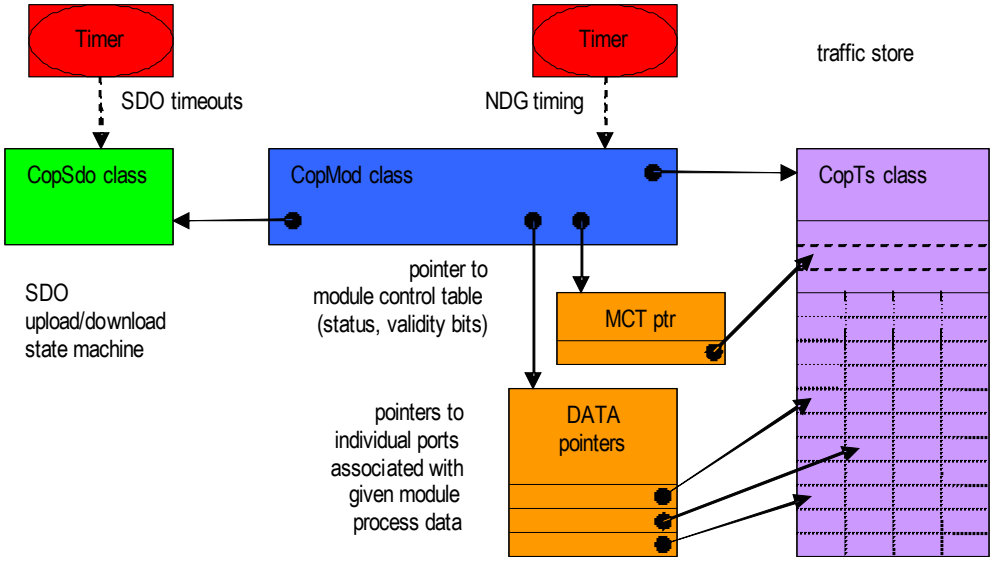

Fig 16 – CANopen module level overview

CopMod class members

data type	member		description
`CopStatus`	`status`	-	CANopen module status
`CopNmtState`	`nmt`	-	module NMT state
`CopModDesc`	`md`	-	CANopen module descriptor in RIO module • descriptor provides module configuration parameters, ie. NDG message period, timeouts ...
`CopNet *`	`net`	-	pointer to network object to which module belongs • pointer is used to call network's OnModStatusChange() callback
`CopTs *`	`ts`	-	pointer to network's traffic store object
`CopTsMct *`	`mct`	-	address of module's control table in traffic store • MCT contains module status • and validity bits of its input and output channels
`CopSdo`	`sdo`	-	SDO state machine object 1. controls SDO uploads and downloads

CopMod class operations

method		description
`ReservePorts()`	-	module returns required number of ports in traffic store 1. called during building of network resources
`Attach()`	-	attach module to its resources in traffic store 1. called as final step of building of network resources 2. address of module's control table in traffic store is stored, 3. addresses of module's process data in traffic store ports are stored 4. addresses of module's channels validity bits are stored
`EnableNodeGuarding()`	-	enables or disables node guarding for module • module installs or deletes timer to generate NDG messages
`Configure()`	-	module configuration is started
`ConfigureNextSdo()`	-	start download of next configuration SDO object • module is configured when all SDOs are successfully downloaded

CopMod class handlers

method		description
`OnSync()`	-	network SYNC handler • called when SYNC event occurs, • module's receive PDOs are prepared and sent
`OnNodeGuardMsg()`	-	NDG message handler • called when response to NDG message is received, • module NMT state is stored,
`OnNodeGuardTimer()`	-	node guarding timer handler • NDG message is sent to module
`OnNodeGuardTimeout()`	-	node guarding timeout handler • called when module didn't respond to NDG message,
`OnTransmitPdoMsg()`	-	transmit PDO message handler • called when TPDO is received • PDO is processed and data are placed to traffic store • appropriate validity bits are set
`OnReceivePdoMsg()`	-	receive PDO message handler • called when RPDO is received in standby mode • PDO is processed and data are placed to traffic store
`OnTransmitSdoMsg()`	-	transmit SDO message handler • called when transmit SDO message is received, • SDO state machine object callback is called
`OnReceiveSdoMsg()`	-	receive SDO message handler • called when receive SDO message is received • no actions executed (message is ignored)
`OnEmergencyMsg()`	-	emergency message handler • called when emergency message is received • message data are copied into MCT in traffic store • module status in MCT is updated
`OnSdoStatusChange()`	-	SDO status change handler • called from SDO state machine when SDO download or upload is finished

CanController class

CanController is a class which encapsulates CAN controller device. The purpose of this class is to define an interface between the application and various CAN controllers.

- open/close CAN controller device
- services to write sequence of raw CAN messages to controller output queue
- services to read received raw CAN messages from controller input queue
- services to subscribe notification of CAN message reception

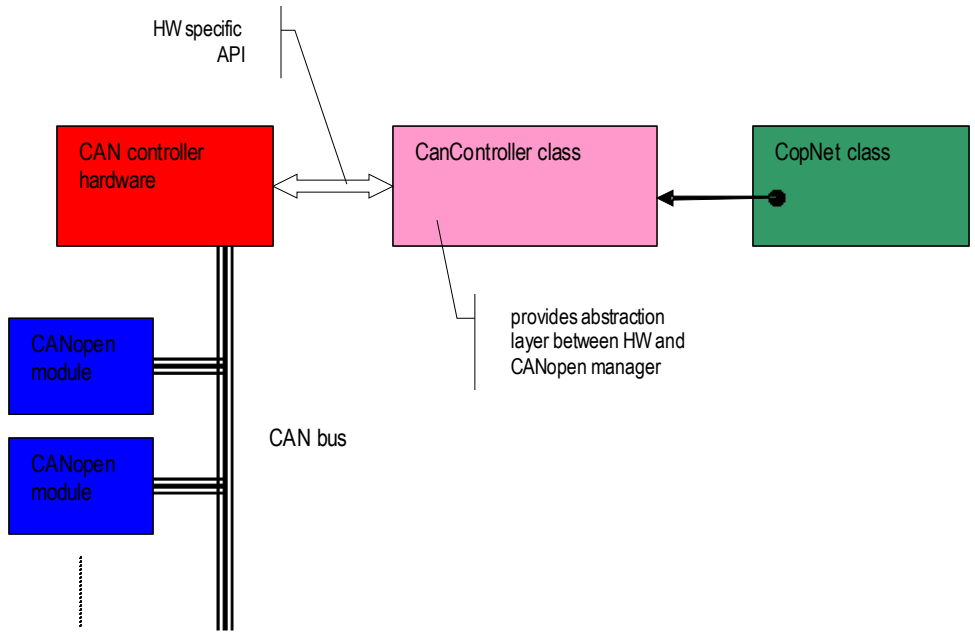

Fig 17 – CanController class

CanController class methods

method		**description**
`Open()`	-	open specified CAN controller device • device is specified by its name (ie. '/dev/can0', 'vcan0' ...) • the CAN bus bitrate can be also specified, or left on default value (default value is HW/driver dependant, usually configured in driver setup)
`Close()`	-	close open CAN controller device
`SendSeq()`	-	send a sequence of CAN messages • more than one message can be sent in one call (this allows to lower the overhead of system call - switch to driver kernel space)
`RecvSeq()`	-	read a sequence of CAN messages from driver input queue • method returns number of CAN messages read, • method never blocks the caller if there are no messages available to read
`SetNotifySignal()`	-	set notification signal number • notification signal is sent by driver when first CAN message arrives into drivers input queue

CanMsg struct

CANopen manager works with CAN messages represented by structure CanMsg. Using own structure is needed to fulfill requirement to work with various drivers. Every driver usually defines its own structure representing raw CAN message. CanController class interface works only with CanMsg structures, thus isolates the rest of the application from driver-specific issues. The disadvantage is some performance loss, because CanController class must always repack CAN messages to and from driver-specific format.

CanMsg struct members

data type	member		description
`unsigned int`	`flags`	-	message flags • CANMSG_RTR ... remote request message • CANMSG_EXT ... message with extended COB-ID
`unsigned int`	`id`	-	message COB-ID
`unsigned int`	`length`	-	message length (number of used data bytes)
`unsigned char`	`data[8]`	-	message data

CopTs class

CopTs is a class which encapsulates the traffic store. Its purpose is to simplify access to traffic store for CopMod objects.

- creates traffic store and initializes its contents
- provides methods for accessing the module control tables and extracting information from them
- provides methods for accessing data ports
- traffic store format is binary compatible with older ‘copman’ application, only for compatibility reasons (applications accessing the process data stored in the traffic store don’t need to be updated)

CopTs class members

data type	member		description
`DataModule`	`mod`	-	traffic store data module
`CopTsHdr *`	`ts`	-	address of traffic store header
`CopTsMct *`	`mct`	-	address of MCT array in traffic store
`char *`	`port`	-	address of first port in traffic store

CopTs class methods

method		description
`Open()`	-	opens existing traffic store • traffic stores are numbered, the data module name is constructed as ‘dm_tscopX’ where X is traffic store number
`Create()`	-	create new traffic store • traffic stores are numbered, the data module name is constructed as ‘dm_tscopX’ where X is traffic store number
`Close()`	-	close traffic store
`GetNetStatus()`	-	return CANopen network status • network status is stored in traffic store header

`SetNetStatus()` - set CANopen network status

`GetNumMods()` - return number of module control tables in traffic store

`GetModuleControlTable()` - return address of specified module control table

`GetPortAddress()` - return address of specified port of given module

`GetNumPortsInCage()` - return number of ports in specified cage of given module

`GetPortOffset()` - return offset of port in specified cage of given module

`GetModStatus()` - return specified module status

- module status is store in its MCT

`SetModStatus()` - set specified module status

`GetInpValBitsAddr()` - return address of given module inputs validity bits

- input channels validity bits are stored in MCT

`GetOutValBitsAddr()` - return address of given module outputs validity bits

- output channels validity bits are stored in MCT

`GetEmergencyMsgAddr()` - return address of module's CANopen emergency message

- emergency message data are stored in MCT

Chapter 3. Validation tests

3.1. CAN monitor/analyzer tests

There is no need to develop custom software to test functions of OCERA developed CAN monitor. The OCERA software suite contains a set of basic utilities and examples which can be used for specified test.

Components tested

Component	Description
lincan	CAN device driver for Linux/RTLinux/OCERA kernel
canslave	CANopen slave component.
canmaster/canmond	CANopen master component with monitor daemon.
canmond-simple	CAN monitor daemon component.
canmonitor	CAN monitor user interface component.
readcan	Command line utility for reading raw CAN messages on the bus.
sendcan	Command line utility for sending raw CAN messages to the bus.
EDS file	Electronic Data Sheet with description of emulated/analyzed CANopen slave.

Table 10 – Components used in CAN monitor/analyzer test

Test setup

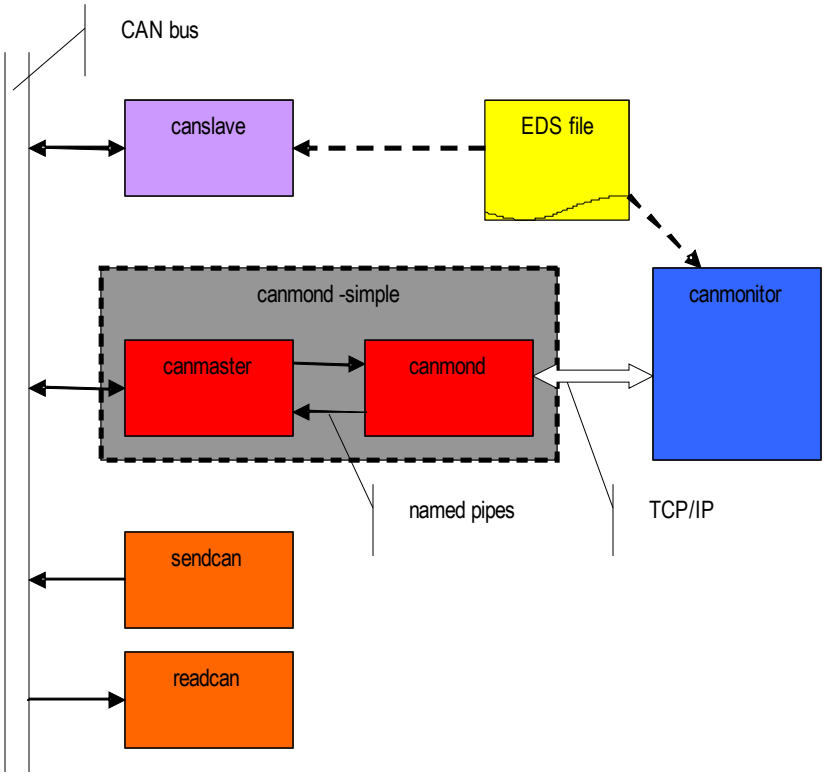

Fig 18 - CAN monitor/analyzer test setup

- components can run on the same or on different computers,
- when all components run on single computer, the lincan device driver can run in virtual mode (no real CAN device needed)

3.2. Communication latency test

Components tested

Component	Description
lincan	CAN device driver for Linux/RTLinux/OCERA kernel

Table 11 – Components used in communication latency test

Test setup

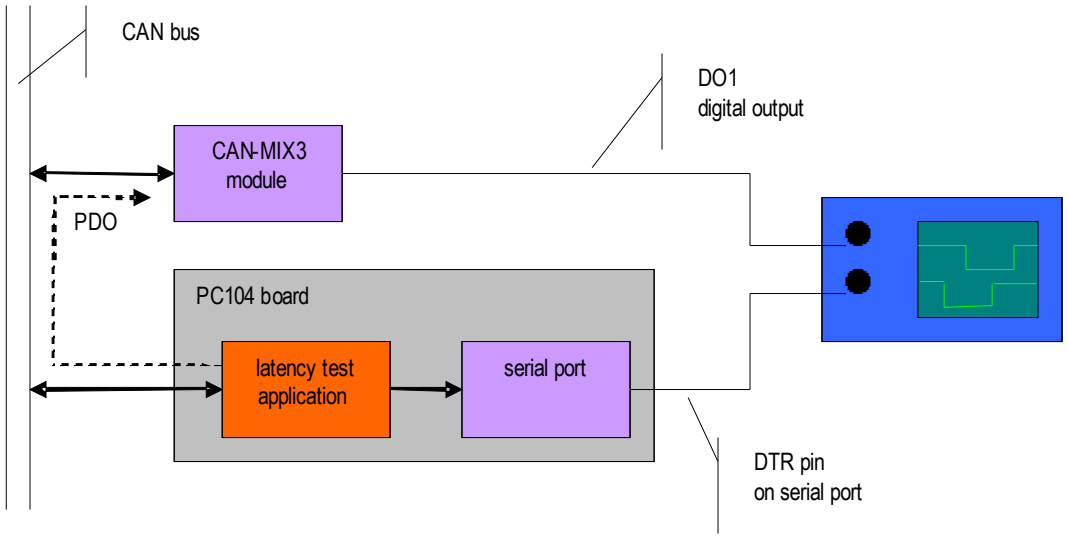

Fig 19 – Communication latency test setup

Required hardware

Hardware	Description
PC104 computer	computer board running Linux/OCERA components (Dig.Logic MSM586SEN/SEV)
CAN controller	CAN controller board supported by lincan driver (Advantech PCM3680)
CAN-MIX3 module	intelligent CANopen io module produced by Unicontrols
oscilloscope	two-channel oscilloscope for taking measurements

Table 12 – Required hardware for communication latency test

Test application flowchart

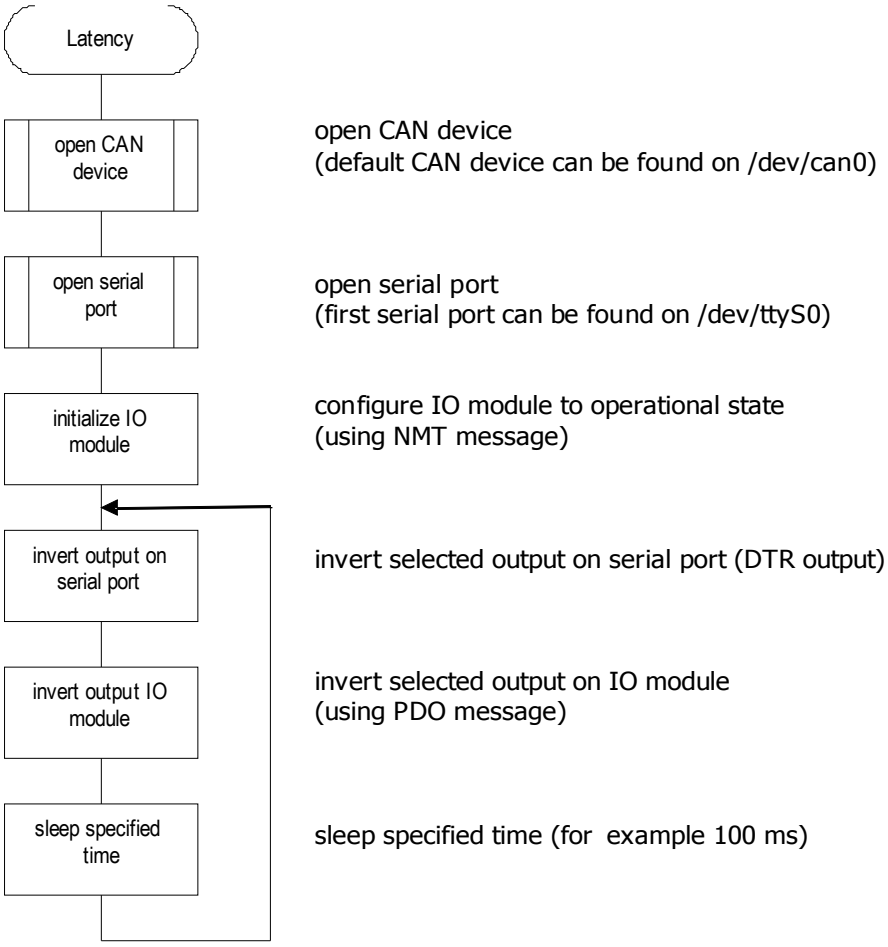

Fig 20 – Communication latency test flowchart

Results

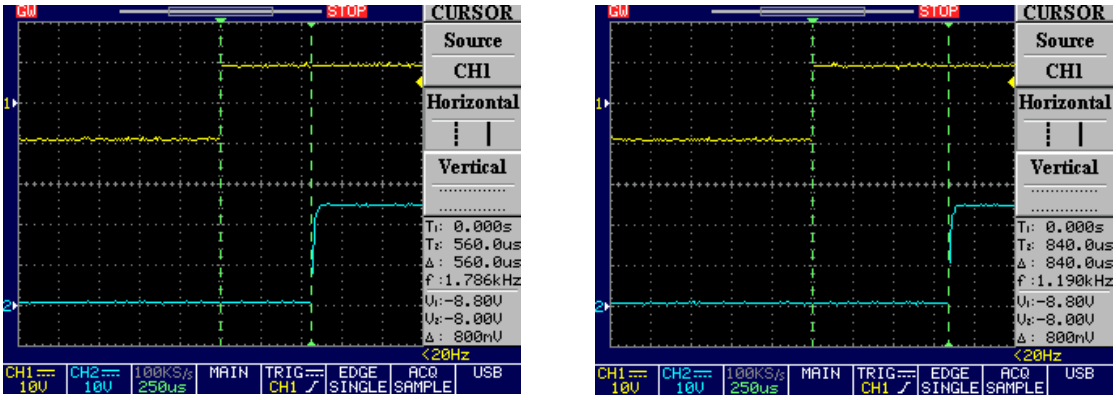

Fig 21 – Measured latency on digital output

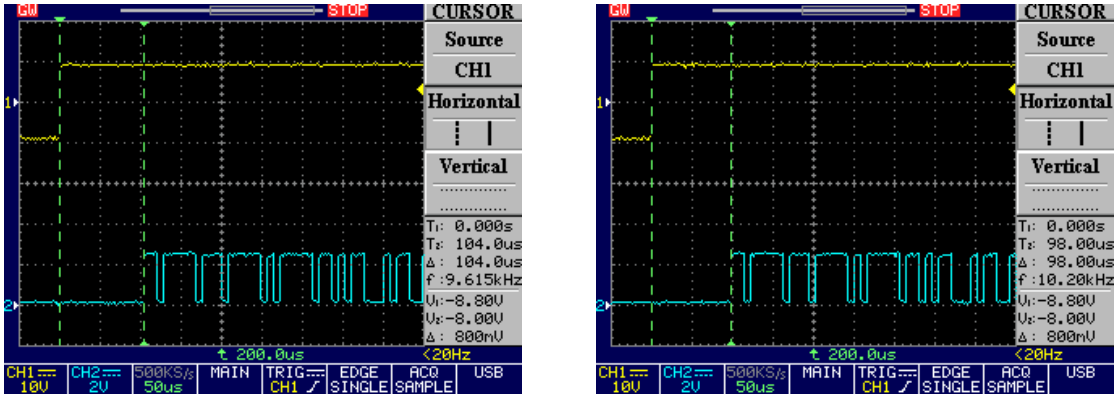

Fig 22 – Measured latency of CAN message

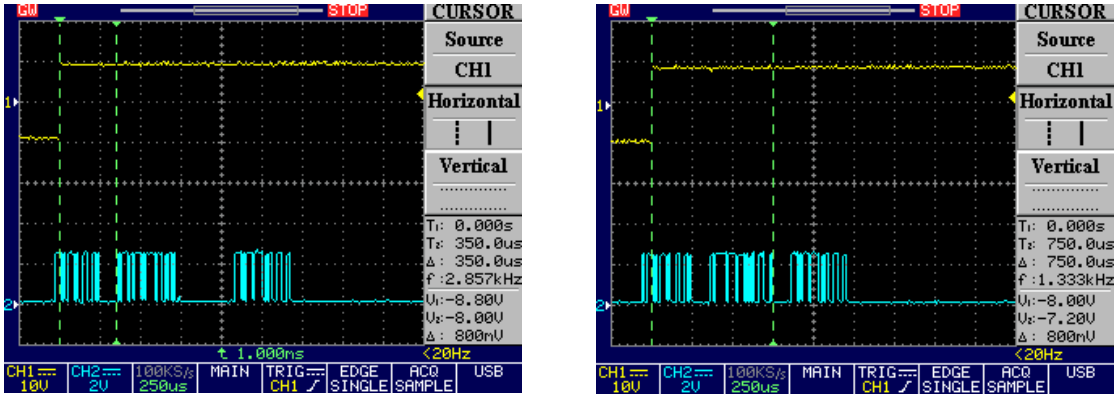

Fig 23 – Measured latency with NDG messages on line

3.3. Communication throughput test

Components tested

Component	Description
lincan	CAN device driver for Linux/RTLinux/OCERA kernel

Table 13 – Components used in communication throughput test

Test setup

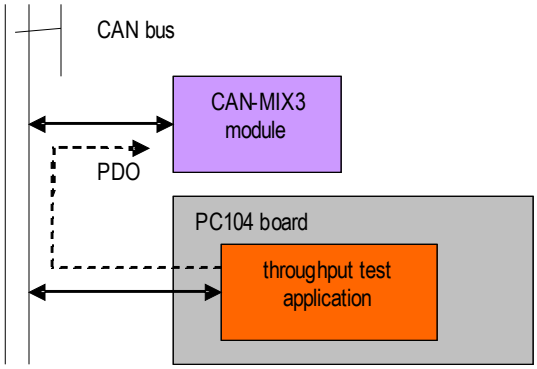

Fig 24 – Communication throughput test setup

Required hardware

Hardware	Description
PC104 computer	computer board running Linux/OCERA components (Dig.Logic MSM586SEN/SEV)
CAN controller	CAN controller board supported by lincan driver (Advantech PCM3680)
CAN-MIX3 module	intelligent CANopen io module produced by Unicontrols

Table 14 – Required hardware for communication throughput test

Test application flowchart

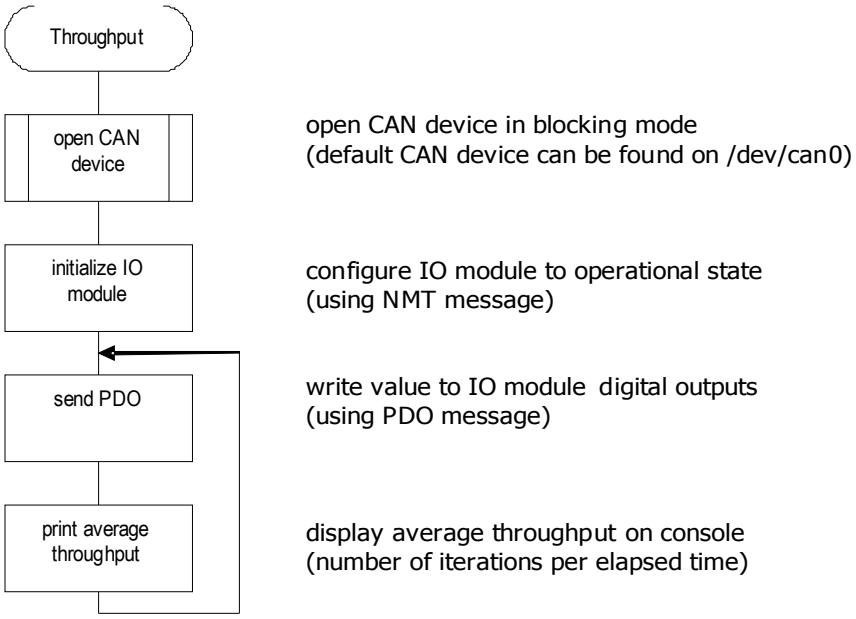

Fig 25 – Communication throughput test flowchart

Results

Results were measured with CAN-MIX module at CAN bus speed 125 kb/s.

Bytes per message	Messages per second	Bytes per second
0	2349.1	N/A
1	2063.1	2063.1
2	1738.0	3476.0
3	1580.4	4741.2
4	1451.8	5807.2
5	1326.9	6634.3
6	1223.5	7340.8
7	1113.9	7797.3
8	1039.8	8318.0

Table 15 – Measured communication throughput

3.4. Scheduler delay test

Results

Comparison of scheduling accuracy of soft real-time versus hard real-time tasks

HW: Pentium M 1.5 GHz
soft-rt: 2.4.18
hard-rt: 2.4.18-ocera-1.0.0

	soft-rt	hard-rt
Average:	0.500249034	0.500000002
Min:	0.498200178	0.499997920
Max:	0.519970179	0.500002592
Delta:	0.021770000	0.000004672
Stddev:	0.001652106	0.000000719

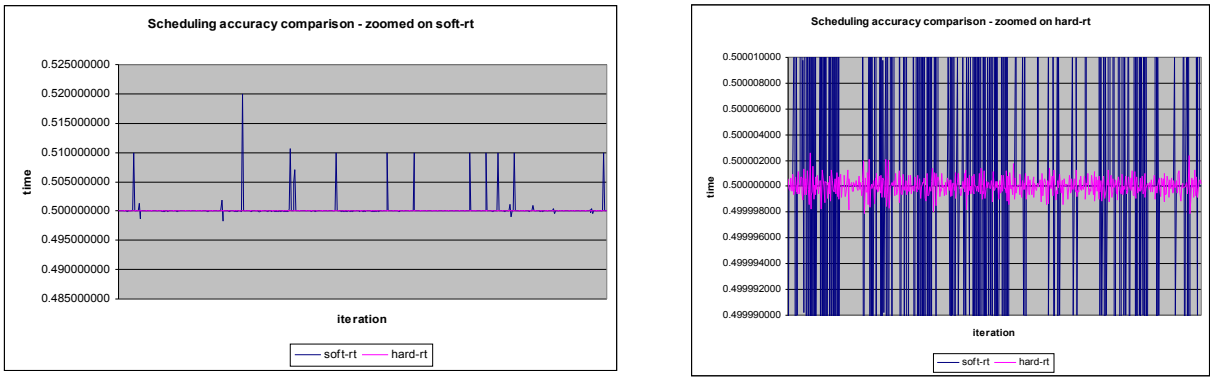

Fig 26 – Measured scheduler latency

Chapter 4. Validation of Process Control Application

4.1. Test results

The process control application was tested on system with PowerPC processor board VMP1 (MPC8240/250 MHz), operating system Linux, kernel 2.4.18-ocera-1.1.
The interpreted control algorithms consits of 9 tasks, tasks 2, 3 and 10 were interpreted with the period 25 ms, tasks 4, 5, 6, 7, 8 and 9 with period 50 ms (fig. 1).
The interpreted database contains about 50000 variables which are evaluated in each period. There is a statistics of interpreted tasks in the figure 1, item StartCnt contains number of interpreted periods, item Over1Cnt contains number of periods when interpretation wasn't finished in time. We can see that interpretation was performed practicaly without a delay.

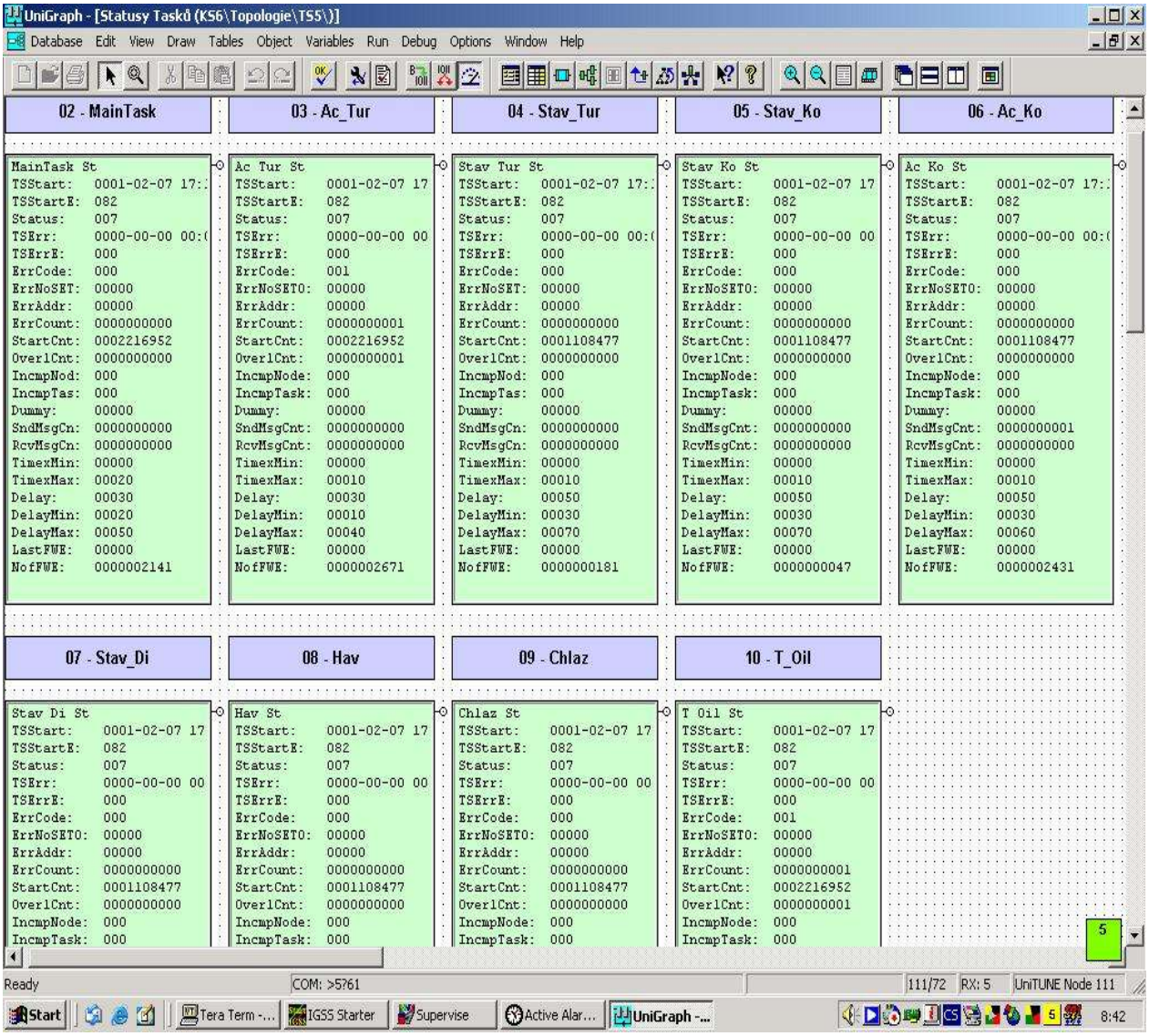

Fig 27 – Statistics of interpreted tasks

The main functions of the real time part of UniCAP were tested both in OS9 and Linux+RTLinux environment by using the Process Control Application.
There is a chart showing regulation of rotates of the turbine in fig. 2, UniCAP applications on the control system and on the technology simulator were executed in Linux+RTLinux environment.

Fig 28 – Regulation of rotates of the turbine

Chapter 5. Conclusion

All parameters requested by process control application are satisfied. The following table shows the results for defined criteria of the porting of the real time part of UniCAP from OS9 to Linux+RTLinux environment.

Parameter name	**Parameter description**	**Qualitative or quantitative criterion**	**Test result**
Process Control Application (PCA) - operational state	Operational state of PCA must be available the same way as in OS9	Operational state of PCA was reached the same way as in OS9	Passed
PCA – blocking conditions	PCA should react to blocking conditions the same way as in OS9	PCA reacted to blocking conditions the same way as in OS9	Passed
PCA – warnings	PCA should give the same warnings on the same conditions as in OS9	PCA gave the same warnings on the same conditions as in OS9	Passed
PCA – cancellation of operational state	PCA should cancel operational state on the same conditions as in OS9	PCA canceled operational state on the same conditions	Passed
PCA – regulation of rotates	The time of change of real rotates according to requested rotates	Not longer time in Linux+RTLinux than in OS9 environment	Passed