WP7 – Development of Communication Components

D7.2_rep Communications components V1

Communication components

by Frantisek Vacek

by Jan Krakora

by Pavel Pisa

by Petr Smolik

by Zdenek Sebek

by Libor Waszniowski

by Zdenek Hanzalek

Published April 2003

Table of Contents

List of Figures

Document Presentation

Table 1 Project Coordinator

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

Table 2 Participant List

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

Table 3 Document version

Release	Date	Reason of change
1_0	15/04/03	First release

Communication components

Chapter 1. CAN/CANopen

1.1. Virtual CAN API (VCA)

The virtual CAN API is the interface used to connect the application threads either with the CAN hardware card or with other software layers substituting CAN bus. The application thread can live either in the Hard RT space or in the Soft RT space. In the the words we can say that VCA is a layer between the CAN driver and the application threads.

1.1.1. Summary

Name of the component

Virtual CAN API (VCA)

Author

Pavel Pisa, Frantisek Vacek

Reviewer

not validated

Layer

Low-level (not implemented yet), High-level

Version

0.1 Alfa

Status

Alfa

Dependencies

High-level version needs CAN driver module (it is connected via /dev/can).

Low-level version needs RT-CAN driver installed.

Release date

N/A

1.1.2. Description

A virtual CAN API is an interface used to connect the application threads either with a CAN hardware card or with other software layer which can substitute a CAN bus. An application thread can live either on low-level (RT-Linux) or on application-level (user space). In the other words we can say that VCA is an uniform layer between a CAN driver and the application threads on any level.

1.1.3. API / Compatibility

struct canmsg_t

`struct canmsg_t` — structure representing CAN message

Synopsis

```
struct canmsg_t {
  short flags;
  int cob;
  unsigned long id;
  unsigned long timestamp;
  unsigned int length;
  unsigned char * data;
};
```

Members

flags

extra flags for internal use

cob

communication object number (not used)

id

ID of CAN message

timestamp

not used

length

length of used data

data

data bytes buffer

Header

can.h

vca_h2log

`vca_h2log` — converts VCA handle to printable number

Synopsis

```
long vca_h2log (vca_handle_t vcah);
```

Arguments

vcah

VCA handle

Header

can_vca.h

Return Value

unique printable VCA handle number

vca_open_handle

`vca_open_handle` — opens new VCA handle from CAN driver

Synopsis

```
int vca_open_handle (vca_handle_t * vcah_p, const char * dev_name, const char * options, int flags);
```

Arguments

vcah_p

points to location filled by new VCA handle

dev_name

name of requested CAN device, if NULL, default VCA_DEV_NAME is used

options

options argument, can be NULL

flags

flags modifying style of open (VCA_O_NOBLOCK)

Header

can_vca.h

Return Value

VCA_OK in case of success

vca_close_handle

`vca_close_handle` — closes previously acquired VCA handle

Synopsis

```
int vca_close_handle (vca_handle_t vcah);
```

Arguments

vcah

VCA handle

Header

can_vca.h

Return Value

Same as libc `close` returns.

vca_send_msg_seq

`vca_send_msg_seq` — sends sequentially block of CAN messages

Synopsis

```
int vca_send_msg_seq (vca_handle_t vcah, canmsg_t * messages, int count);
```

Arguments

`vcah`

VCA handle

`messages`

points to continuous array of CAN messages to send

`count`

count of messages in array

Header

can_vca.h

Return Value

Number of sucessfully sent messages or error < 0

vca_rec_msg_seq

`vca_rec_msg_seq` — receive sequential block of CAN messages

Synopsis

```
int vca_rec_msg_seq (vca_handle_t vcah, canmsg_t * messages, int count);
```

Arguments

vcah

VCA handle

messages

points to array for received CAN messages

count

number of message slots in array

Header

can_vca.h

Return Value

number of received messages or error < 0

vca_wait

`vca_wait` — blocking wait for the new message(s)

Synopsis

```
int vca_wait (vca_handle_t vcah, int wait_msec, int what);
```

Arguments

`vcah`

VCA handle

`wait_msec`

number of miliseconds to wait, 0 => forever

`what`

0,1 => wait for Rx message, 2 => wait for Tx - free 3 => wait for both

Header

can_vca.h

Return Value

Positive value if wait condition is satisfied

vca_log

`vca_log` — generic logging facility for VCA library

Synopsis

```
void vca_log (const char * domain, int level, const char * format, ... ...);
```

Arguments

`domain`

pointer to character string representing source of logged event, it is `VCA_LDOMAIN` for library itself

`level`

severity level

`format`

printf style format followed by arguments

`...`

variable arguments

Description

This functions is used for logging of various events. If not overridden by application, logged messages goes to the stderr. Environment variable `VCA_LOG_FILENAME` can be used to redirect output to file. Environment variable `VCA_DEBUG_FLG` can be used to select different set of logged events through vca_debug_flg.

Note

only messages with `level` <= `vca_log_cutoff_level` will be logged. see `can_vca`.h

vca_log_redir

`vca_log_redir` — redirects default log output function

Synopsis

```
void vca_log_redir (vca_log_fnc_t * log_fnc, int add_flags);
```

Arguments

`log_fnc`

new log output function. Value NULL resets to default function

`add_flags`

some more flags

1.1.4. Implementation issues

Applications can be connected to CAN via VCA in two ways, either from hard real-time space or from soft real-time one. Other CAN driver is used in each case (RT-Linux or Linux resp.), but VCA remains always the same. Following the POSIX standard and VCA one can easy write applications, which can be compiled and run in both spaces.

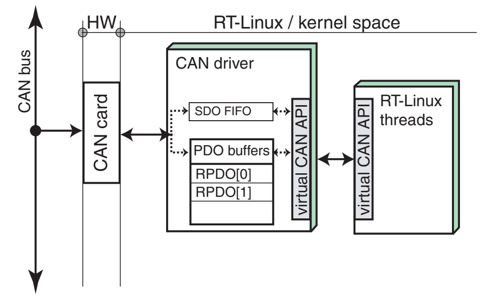


Figure 1-1. Hard real time CAN driver usage

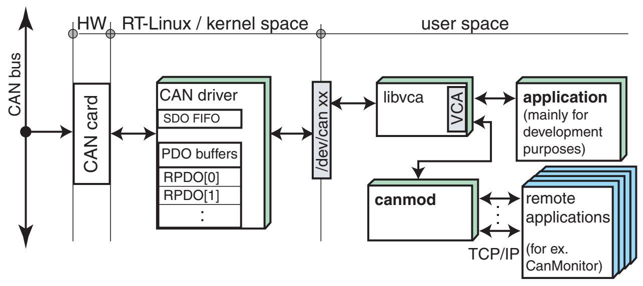


Figure 1-2. Soft real time CAN driver usage

On figure above we can also find, that user space application can be connected through TCP/IP to the *canmond* server. Canmond works like CAN/IP proxy (see canmond description).

1.1.5. Tests

Only soft real-time solution was implemented yet. No heavy tests were made. All tests were performed during CanMonitor testing (see CanMonitor tests).

All VCA sources were compiled by GNU C ver. 3.2 and linked with glibc ver. 2.2.5.

1.1.6. Examples

Directory `ocera/components/comm/can/canvca/cantest` contains two example programs - *sendcan* and *readcan*. First one shows the simplest way to send CAN message via VCA. The second one shows, how to read CAN message. You can find more information in their source codes (*sendcan.c, readcan.c*) in the same directory.

sendcan invocation: `sendcan id byte_1 ... byte_n`

Note: don't forget restart CAN device before communication (`sendcan 0 1 0`).

1.1.7. Installation instructions

All communication components can be compiled issuing `make` command in directory `ocera/components/comm/can/`. You can also make only some of them by issuing `make` in appropriate directory.

VCA components don't have special requirements on gcc or glibc version.

1.2. CAN monitor

CAN monitor is a component used to sniff on a CAN bus. It can log messages and send ones. If CAN device EDS (Electronic Data Sheet) is available CanMonitor offers alse basic SDO functionality. Package consists of three parts: *canmond, testclient* and *Can-Monitor*.

1.2.1. Summary

Name of the component

CanMonitor

Author

Frantisek Vacek

Reviewer

not validated

Layer

High-level

Version

0.1 Alfa

Status

Alfa

Dependencies

It needs interface layer providing VCA.

Release date

N/A

1.2.2. Description

Can monitor component consists of three parts. CAN proxy - *canmond*, console canmond client *testclient* and Java GUI canmond client *CanMonitor*.

1.2.2.1. canmond - CAN/CANopen proxy

Canmond is the heard of component. It works like CAN proxy , translates every CAN message to the textual, platform independent form and send it to the all connected applications. TCP connection allows clients to be placed wherever on Internet. One can also read/send CAN messages using a java applet on his HTML browser.

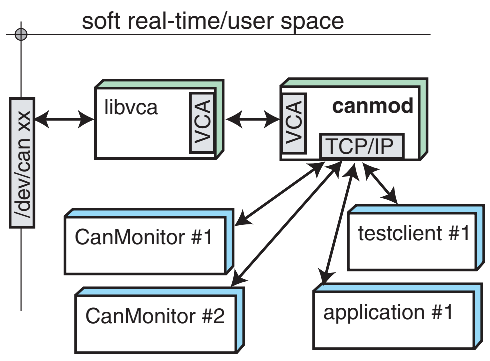


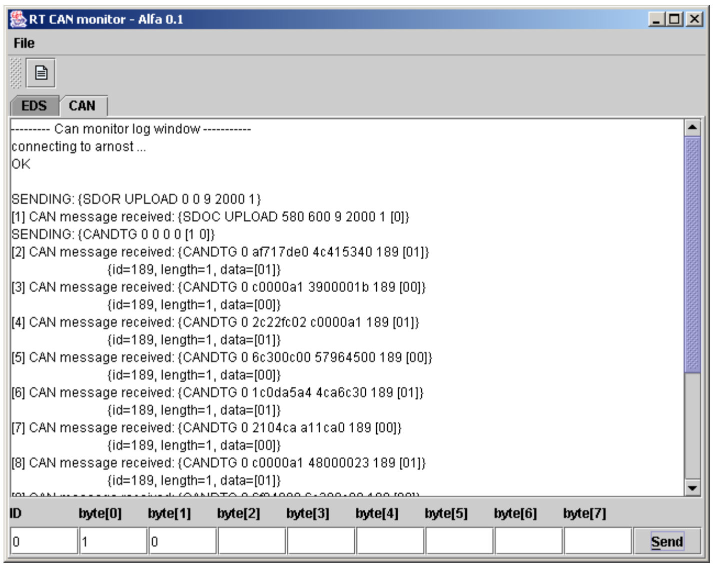
Figure 1-3. canmond server-client architecture

1.2.2.2. testclient

Testclient is an simple console based application for communication with *canmond*. It provides us basic operation on CAN/CANopen bus like sending messages and SDO communication.

1.2.2.3. CanMonitor

CanMonitor is a GUI Java based application connected to the *canmond*. Like *testclient* provides us basic CAN/CANopen communication primitives. If one has CANopen device EDS (Electronic Data Sheet), he can read/write CANopen objects just clicking on the mouse.



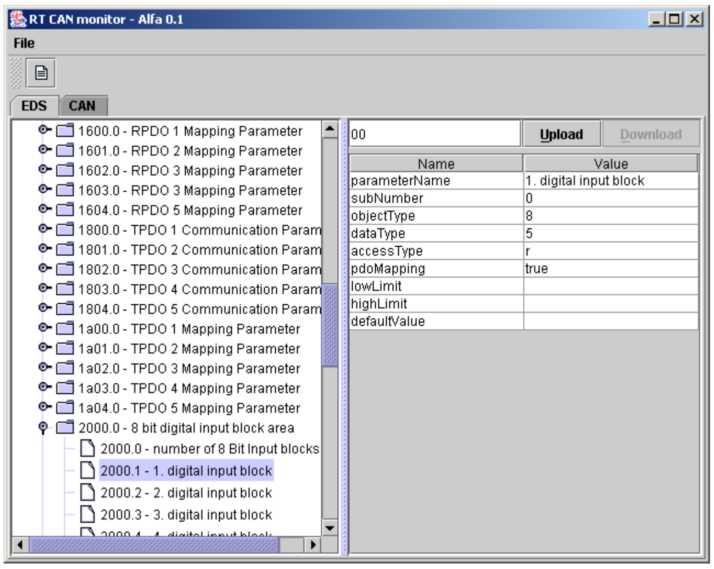
```
--------- Can monitor log window -----------
connecting to arnost ...
OK

SENDING: {SDOR UPLOAD 0 0 9 2000 1}
[1] CAN message received: {SDOC UPLOAD 580 600 9 2000 1 [0]}
SENDING: {CANDTG 0 0 0 0 [1 0]}
[2] CAN message received: {CANDTG 0 af717de0 4c415340 189 [01]}
                    {id=189, length=1, data=[01]}
[3] CAN message received: {CANDTG 0 c0000a1 3900001b 189 [00]}
                    {id=189, length=1, data=[00]}
[4] CAN message received: {CANDTG 0 2c22fc02 c0000a1 189 [01]}
                    {id=189, length=1, data=[01]}
[5] CAN message received: {CANDTG 0 6c300c00 57964500 189 [00]}
                    {id=189, length=1, data=[00]}
[6] CAN message received: {CANDTG 0 1c0da5a4 4ca6c30 189 [01]}
                    {id=189, length=1, data=[01]}
[7] CAN message received: {CANDTG 0 2104ca a11ca0 189 [00]}
                    {id=189, length=1, data=[00]}
[8] CAN message received: {CANDTG 0 c0000a1 48000023 189 [01]}
                    {id=189, length=1, data=[01]}
```

ID	byte[0]	byte[1]	byte[2]	byte[3]	byte[4]	byte[5]	byte[6]	byte[7]
0	1	0						

Figure 1-4. CAN monitor CAN messages window

CAN monitor can serve as application showing all messages on CAN bus. You can also send a raw CAN messages to the CAN bus clicking on *Send* button.



1600.0 - RPDO 1 Mapping Parameter
1601.0 - RPDO 2 Mapping Parameter
1602.0 - RPDO 3 Mapping Parameter
1603.0 - RPDO 3 Mapping Parameter
1604.0 - RPDO 5 Mapping Parameter
1800.0 - TPDO 1 Communication Param
1801.0 - TPDO 2 Communication Param
1802.0 - TPDO 3 Communication Param
1803.0 - TPDO 4 Communication Param
1804.0 - TPDO 5 Communication Param
1a00.0 - TPDO 1 Mapping Parameter
1a01.0 - TPDO 2 Mapping Parameter
1a02.0 - TPDO 3 Mapping Parameter
1a03.0 - TPDO 4 Mapping Parameter
1a04.0 - TPDO 5 Mapping Parameter
2000.0 - 8 bit digital input block area
2000.0 - number of 8 Bit Input blocks
2000.1 - 1. digital input block
2000.2 - 2. digital input block
2000.3 - 3. digital input block

00

Name	Value
parameterName	1. digital input block
subNumber	0
objectType	8
dataType	5
accessType	r
pdoMapping	true
lowLimit	
highLimit	
defaultValue	

Figure 1-5. The Object Dictionary tree view

With loaded EDS you can upload/download CANopen objects values straight to the device object dictionary (OD).

1.2.3. API / Compatibility

1.2.3.1. canmond

Canmond command line arguments:

```
CANMOND - CAN monitor server
canmond [OPTION]

OPTIONS:
 -h
 --help this help screen
 -v --verbose
 -p
 --port [n] sets port where the server listens (default 1001)
 -g
 --log_level [n] sets how many log messages you will see.
  0 - fatal errors onyl
  1 - level 0 + errors
  2 - level 1 + messages
  3 - level 2 + info messages
  4 - level 3 + debug messages
  log level can be also set by environment variable CANMOND_LOG_LEVEL
```

1.2.3.2. testclient

Command line arguments

```
TESTCLIENT - canmond client
testclient [OPTION]

OPTIONS:
 -h
 --help this help screen
 -v
 --verbose tverbose
 -a
 --host [n] sets  the IP address where the server listens default is "127.0.0.1"
 -p
 --port [n] sets port on which server listens default 1001

COMMANDS:
 sendmsg id [byte1 byte2 ...]  - sends CAN message (short version)
```

```
send  {CANDTG flags cob timestamp id [byte_1 .. byte_n]}
 - sends CAN message (detailed version)
 {SDOR UPLOAD server_port client_port node index subindex}
 - uploads CANopen object from device object dictionary
 server_port, client_port can be 0 for default values (0x580, 0x600)
 {SDOR DOWNLOAD server_port client_port node index subindex [byte_1 ... byte_n]}
 - downloads CANopen object to device object dictionary
q quits
```

1.2.3.3. CanMonitor

In current version CanMonitor does not have GUI configuration dialog. It can be configured only from command line during launching.

Command line arguments

`CanMonitor -a` *host* `-n` *node* `-f` *EDS-file-name*

1.2.4. Implementation issues

1.2.4.1. canmond

Any application can attach itself to the *canmond*. It works like TCP server listening on port 1001. If an application opens socket to the server, it can send/receive text messages described in section canmond API.

canmond has simple text API to communicate with its clients. API is of following structure:

Rough CAN message format

`{CANDTG flags cob timestamp id [data_byte_1 .. data_byte_n]}`

flags, cob, timestamp, id and *data_byte_1 ... data_byte_n* are numbers in hexadecimal format. Number of bytes should be less or equal 8 in case of single CAN message.

`Example: {CANDTG 0 104500C8 6400B40 189 [0F]}`

SDO upload request

`{SDOR UPLOAD server_port client_port node index subindex}`

Requests upload of object[*index.subindex*] from device with CANopen address *node*. Uploaded data are returned in `SDOC UPLOAD` message.

server_port and *client_port* could be 0. In that case the default values, 0x580 for the server_port and 0x600 for the client_port are considered.

Example: `{SDOR UPLOAD 0 0 9 2000 1}` - request for upload of index 0x2000, subindex 0x1 of node 9.

SDO upload confirmation

`{SDOC UPLOAD server_port client_port node index subindex [data_byte_1 ... data_byte_n]}`

Confirmation message for previously requested CANopen object upload. Uploaded data are returned as a byte array *[data_byte_1 ... data_byte_n]*. Number of returned bytes can be greater than 8 in case of SDO. For description of other parameters see SDO upload request

Example: `{SDOC UPLOAD 580 600 9 2000 1 [0]}` - answer for the upload request from the paragraph above.

SDO download request

`{SDOR DOWNLOAD server_port client_port node index subindex [data_byte_1 ... data_byte_n]}`

Requests download of the byte array *[data_byte_1 ... data_byte_n]* to the CANopen device. For description of other parameters see SDO upload request

Example: `{SDOR DOWNLOAD 0 0 9 2100 1 [FF]}` - request for download one byte 0xFF to the index 0x2000, subindex 0x1 of node 9.

SDO download confirmation

`{SDOC DOWNLOAD server_port client_port node index subindex}`

Confirmation message for previously requested CANopen object download. For description of other parameters see SDO upload request

Example: `{SDOC DOWNLOAD 580 600 9 2100 1 }` - answer for the download request from the paragraph above.

Communication error and abort messages

Upon some circumstances, CANopen device aborts SDO communication. Also a communication error can occur.

In case of aborted communication *canmond* includes word `'ABORT'`, abort code (defined in CiA Standard 301) and textual representation of that code in place of returned data byte array.

Example: `{SDOC DOWNLOAD 580 600 9 2100 2 ABORT 6090011 'Sub-index does not exist.'}`

In case of communication error *canmond* includes word `'ERROR'` error code (defined in OCERA `vcasdo_fsm.h`) and textual representation of that code in place of returned data byte array.

Example: `{SDOC UPLOAD 580 600 9 2000 1 ERROR 1 'SDO transfer time out.'}`

1.2.5. Tests

Component was tested with real CANopen device WAGO 750-307.

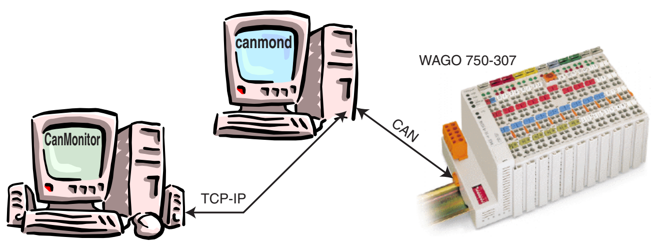


Figure 1-6. CanMonitor testing

All VCA sources were compiled by GNU C ver. 3.2 and linked with glibc ver. 2.2.5.

1.2.6. Examples

Type `make` in directory `ocera/components/comm/can/canmon/canmond`. Than type `canmond`.

You should see something like this

```
[root@arnost canmond]# ./canmond
CANMOND - the can monitor server
```

Than you can launch `testclient` program either on the same machine or on other one.

```
[fanda@mandrake canmond]$ testclient -a arnost
testclient -a arnost
finding arnost:1001 ...
found address: arnost - 147.32.84.158
connecting 147.32.84.158:1001 ...
OK
got HELLO from canmond.
```

You can also use `rdln` utility (also part of the OCERA project) in directory `ocera/components/comm/c`

```
[fanda@mandrake canmond]$ rdln testclient -a arnost
```

This utility gives the program in argument list readline facility like command history, BASH like line editing etc.

If you have a graphical environment with Java installed, you can launch CanMonitor. Type

```
canmonitor -a host_addr -n CAN_device_node_no
Example: canmonitor -a 147.32.84.158 -n 9
```

If everything works right, you should see application window like one in section Can-Monitor description. Now you can load device EDS file and upload/download CANopen objects.

1.2.7. Installation instructions

Program from this package does not need special installation. They can run from any directory. Just type `make` in `ocera/components/comm/can` directory. If you want to compile only one component, type `make` in component's directory.

Restrictions on versions of GNU C or glibc are not known in this stage of project.

JAR package of CanMonitor is not available yet, you need all *.class files in directory `class` in `CanMonitor` home to run it. CanMonitor is a pure Java application, you don't need anything more than standard JRE.

1.3. CAN device

1.3.1. Summary

Name of the component

CANopen device

Author

Pavel Pisa

Frantisek Vacek

Reviewer

not validated

Layer

Low-level, High-level

Version

N/A

Status

Design

Dependencies

High-level version needs VCA installed.

Low-level version needs RT-Linux with RT-CAN driver installed.

Release date

N/A

1.3.2. Description

CANopen device is the software solution based on CANopen FSM (Finite State Machine) threads, EDS (Electronic Data Sheet) file and HDS (Handler Definition Sheet) file. Device can be configured to work as a CANopen master or CANopen slave.

1.3.3. API / Compatibility

CANopen device should be compatible with standard industrial CANopen devices according to *CiA Draft Standard 301*.

1.3.3.1. SDO FSM API

This library should be used for SDO FSM implementation in Low-level space and in High-level one too. It is used by *CANopen device* and also by *canmond*.

struct vcasdo_fsm_t

`struct vcasdo_fsm_t` — structure representing SDO FSM

Synopsis

```
struct vcasdo_fsm_t {
  unsigned server_port, client_port;
  unsigned node;
  unsigned index, subindex;
  struct timeval last_activity;
  int bytes_to_load;
  char toggle_bit;
  int type;
  int state;
  vcasdo_fsm_state_fnc_t * statefnc;
  int err_no;
```

```
    ul_dbuff_t data;
    canmsg_t out_msg;
  };
```

Members

client_port

  number of SDO client port (default is 0x600)

node

  SDO node number

subindex

  subindex of communicated object

last_activity

  time of last FSM activity (internal use)

bytes_to_load

  number of stil not uploaded SDO data bytes (internal use)

toggle_bit

  (internal use)

type

  type of FSM (`sdofsmUploader` = 1, `sdofsmDownloader` = 2)

state

  state of SDO (`sdofsmIdle` = 0, `sdofsmRun`, `sdofsmDone`, `sdofsmError`, `sdofsmAbort`)

statefnc

  pointer to the state function (internal use)

err_no

  number of error in state sdofsmError

data

  uploaded/downloaded bytes (see `ul_dbuff`.h)

out_msg

  if `vcasdo_taste_msg` generates answer, it is stored to the `out_msg`

vcasdo_error_msg

`vcasdo_error_msg` — translates err_no to the string message

Synopsis

```
const char* vcasdo_error_msg (int err_no);
```

Arguments

`err_no`

number of error, if FSM state == `sdofsmError`

vcasdo_init_fsm

`vcasdo_init_fsm` — init SDO FSM

Synopsis

```
vcasdo_fsm_t * vcasdo_init_fsm (vcasdo_fsm_t * fsm, unsigned server_port, unsigned client_port,
unsigned node);
```

Arguments

`fsm`

fsm to init, if NULL, function creates and inits a new one

`server_port`

SDO server port number (default value 0x580 is set if parameter == 0)

`client_port`

SDO client port number (default value 0x600 is set if parameter == 0)

`node`

number of node on CAN bus to communicate with

Return Value

pointer to the (created) inited FSM, NULL in case of allocation error.

vcasdo_destroy_fsm

`vcasdo_destroy_fsm` — frees SDO FSM and also frees all its resources

Synopsis

```
void vcasdo_destroy_fsm (vcasdo_fsm_t * fsm);
```

Arguments

`fsm`

fsm to destroy

vcasdo_run

`vcasdo_run` — starts SDO communication protocol for this FSM

Synopsis

```
int vcasdo_run (vcasdo_fsm_t * fsm);
```

Arguments

fsm

SDO FSM

Return Value

not 0 if fsm->out_msg contains CAN message to send

vcasdo_fsm_taste_msg

`vcasdo_fsm_taste_msg` — try to process msg in FSM

Synopsis

```
int vcasdo_fsm_taste_msg (vcasdo_fsm_t * fsm, const canmsg_t * msg);
```

Arguments

`fsm`

fsm to process msg

`msg`

tried msg

Return Value

`sdofsmMsgRefuse`if FSM refuses `msg`, `sdofsmMsgEat` if FSM proceses `msg`, `sdofsmMsgEatAnswer` if FSM proceses `msg` and it has answer in `fsm`->out_msg prepared `sdofsmMsgEatError` `msg` was eaten, but it does not match communication protocol or abort msg was detected

vcasdo_abort_msg

`vcasdo_abort_msg` — translates abort_code to the string message

Synopsis

```
const char* vcasdo_abort_msg (__u32 abort_code);
```

Arguments

`abort_code`

abort code

Header

`vcasdo_msg.h`

1.3.4. Implementation issues

1.3.4.1. Architecture overview

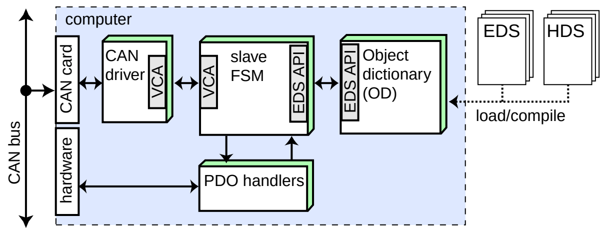


Figure 1-7. RT-CANopen device architecture

CANopen device components description

FSM

FSM (Finite State Machine) means set of RT-Linux threads providing PDO and SDO communication via VCA (see VCA). Slave FSM also calls appropriate PDO communication handler and looks into slave's object dictionary in case of SDO request.

PDO handlers

User written module containing handlers for reading/writing PDO mapped object data from/to hardware.

EDS

EDS means the *Electronic Data Sheet*, text file describing all objects in the slave object dictionary and its mapping into the PDOs. EDS is parsed in order to create slave OD representation in CANopen device.

HDS

HDS means the *Handler Definition Sheet*, a text file describing linking of PDO COB-ID with required handler in order to grant correspondence between the CANopen object value and technological process data from the hardware. For example a thermometer with the analog output connected to PC A/D converter card needs handler which reads temperature from card output port and gives it to FSM. The slave designer have to write this handler code while the FSM source code remains always the same.

1.3.4.2. RT-CANopen device in the Soft real time space

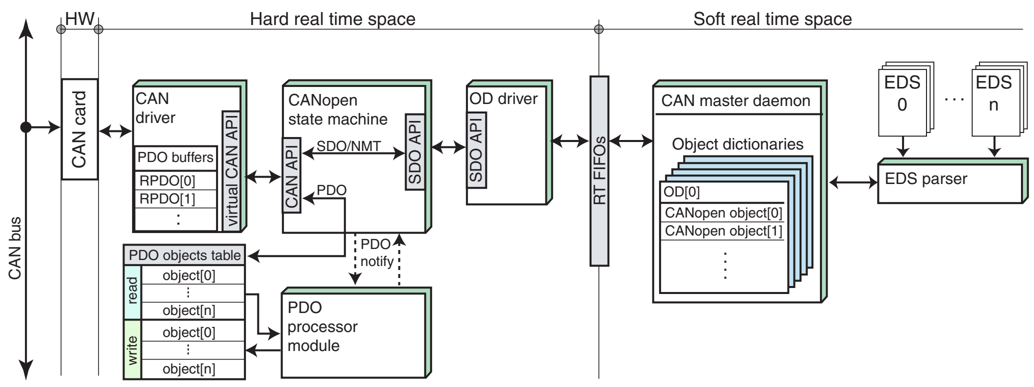


Figure 1-8. Soft real time CANopen device architecture

As can be seen on figure above CAN driver sends the CAN messages to CANopen FSM via VCA. FSM handles messages of two main categories, process data (PDO) and service data (SDO, NMT, SFO).

The process data (PDO objects) are handled separately of the SDO. Slave FSM exploits CAN driver message buffers as the buffers for the slave PDOs. This approach is necessary because some CAN chips have such buffers integrated. On the other hand this can speed up PDO object handling. Slave FSM role lies in updating this buffers after device specific event such is timer event or process object value change. The CAN driver sends objects from its buffers when needed (after SYNC object or as a response to RTR object). Consequently slave FSM has to read this buffers after WPDO object arrival.

PDO objects table is a memory mirror for transmitted and received PDOs. When a new RPDO comes, it is written into the table and PDO processor is notified by master FSM about this event. On the contrary, when PDO processor updates some object in the write part of PDO table, the FSM should be notified to allow it to transmit object change across the network.

The PDO processor module is used to synchronize PDO mapped objects values and real world data examined or set by computer hardware. Every PDO mapped object has assigned its reading and writing routine called PDO handler. These handlers are written by control system developer. Handlers placed in PDO processor share PDO table with FSM. In this table are stored/retrieved process data according to external events and occurrence of messages on the CAN bus. PDO processor is an user written set of functions designated for processing objects from read part of PDO table and generating new value of objects in write part of PDO table. New generated write objects can be sent across the CAN, if the processor notifies master FSM. This way the RPDO-TPDO mapping rules or control algorithms can be realized.

Some of SFO (Special Function Object) are handled directly by CAN driver. Such objects are the SYNC or RTR frames

Other objects (SDO, NMT) are sent through the SDO API to OD driver. OD driver is responsible for all object dictionary manipulations, that means getting and setting object values. If the PDO mapping change occurs due to SDO object processing, OD driver informs slave FSM (via SDO API) to correct PDO handlers table to reflect PDO mapping status in OD properly.

OD driver communicates with OD daemon, which resides in the user space, through RT FIFOs. OD daemon offers set of primitives to provide basic manipulations with OD like get/set object value, add object, delete object etc. OD daemon also owns slave OD in its memory space.

Slave OD can be loaded onto OD daemon memory by EDS parser. As EDS parser will probably serve CAN/CANopen monitor connected with it via Unix socket. This gives us the opportunity to control the daemon and slave OD remotely using TCP/IP. EDS parser is also responsible to read HDS and make appropriate changes mapping of handler functions in PDO processor module. This ensures that the proper handlers will be called for certain PDO objects.

The main difference between CANopen master and CANopen slave device is in OD and PDO processor module. We can say, that slave is more or less master with only one EDS in its OD and some restrictions on functionality (can't send NMT objects etc.). That means, that the slave device is a special case of the master one. Thanks to this generalization we can have also one code for the master and slave device and determinate final behavior by configuration and by loading different PDO processor module.

1.3.4.3. RT-CANopen device in the Hard real time space

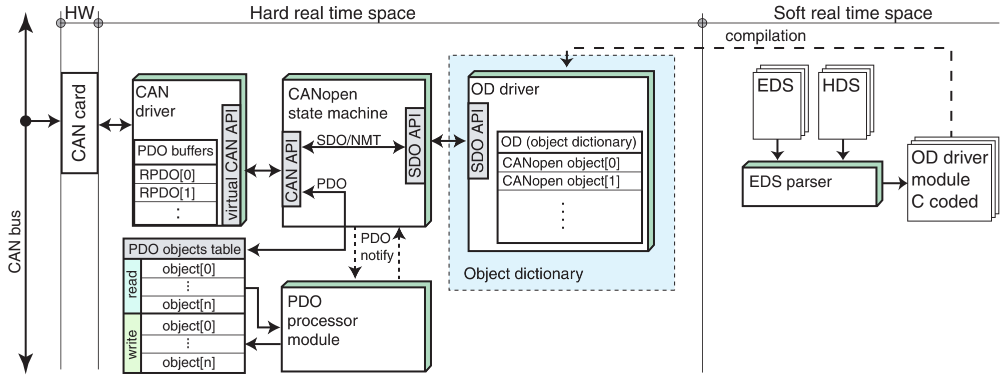


Figure 1-9. Hard real time CANopen device architecture

This architecture is very similar to the previous one. The main difference lies in OD position which is a part of OD driver module now. Every other part of slave remain the same. OD driver module is compiled from source code generated by EDS parser from the slave EDS, HDS and the empty OD driver template.

Benefits of the kernel space solution

- Faster SDO object processing.
- Slave does not need user space applications to work properly.
- Slave can be implemented to other POSIX compliant real-time OS like RTEMS.
- Suitable for CANopen slave realization in embedded systems.

Disadvantages of the kernel space solution

- OD is static, no objects can be added or removed.
- EDS parser can not explore OD any more. The diagnostic pipe has to be used for that purpose and all the information must be communicated through SDO API and slave FSM.

1.3.5. Tests

N/A

1.3.6. Examples

N/A

1.3.7. Installation instructions

N/A

Chapter 2. ORTE - OCERA Real-Time Ethernet

2.1. ORTE

The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. RTPS is new application layer protocol targeted to real-time communication area, which is build on the top of standard UDP stack. Since there are many TCP/IP stack implementations under many operating systems and RTPS protocol does not have any other special HW/SW requirements, it should be easily ported to many HW/SW target platforms. Because it uses only UDP protocol, it retains control of timing and reliability.

2.1.1. Sumary

Name of the component

OCERA Real-Time Ethernet

Author

Petr Smolik

Reviewer

not validated

Layer

High-level

Version

0.1 alfa

Status

Alfa

Dependencies

Any Ethernet adapter and standard TCP/IP stack.

Release date

N/A

2.1.2. Description

The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. This protocol is being to submit to IETF as an informational RFC and has been adopted by the IDA group. ORTE has been designed with compliance to RTPS protocol version 1.17.

2.1.3. API / Compatibility

struct ORTERcvInfo

`struct ORTERcvInfo` — description of an issue

Synopsis

```
struct ORTERcvInfo {
  PathName subsTopic;
  TypeName subsTypeName;
  NtpTime localTimeReceived;
  NtpTime remoteTimePublished;
  SequenceNumber seqNumber;
  u_char * data;
  u_short dataLength;
  u_char reverseOrder;
};
```

Members

subsTopic

  name of topic

subsTypeName

  name of data type

localTimeReceived

  time when the issue was received

remoteTimePublished

  time when the issue was published

seqNumber

  sequential number

data

  buffer containing published data

dataLength

  length of data in buffer

reverseOrder

  different endianing

Description

A message can be sent from a big endian machine to a little endian machine. Currently ORTE supports only little endianing and returns 0 in all cases.

struct ORTEPublStatus

`struct ORTEPublStatus` — status of a publication

Synopsis

```
struct ORTEPublStatus {
  u_short subsReliable;
  u_short subsUnReliable;
  u_short issues;
  u_short unacknowledgedIssues;
};
```

Members

subsReliable

count of reliable subscribers (best effort) connected on responsible publisher

subsUnReliable

count of unreliable subscribers (strict) connected on responsible publisher

issues

number of messages in sending queue

unacknowledgedIssues

number of unacknowledged issues (only for best effort)

struct ORTESubsStatus

struct ORTESubsStatus — status of a subscription

Synopsis

```
struct ORTESubsStatus {
  u_short publReliable;
  u_short publUnReliable;
  u_short issues;
};
```

Members

publReliable

count of reliable publishers (best effort) connected to responsible subscriber

publUnReliable

count of unreliable publishers (strict) connected to responsible subscriber

issues

number of messages in receiving queue

Description

Current implementation has always issues=0. It means, that all messages were sent to user application by callback function.

struct ORTESubsProp

`struct ORTESubsProp` — properties of a subscription

Synopsis

```
struct ORTESubsProp {
  NtpTime deadline;
  NtpTime minimumSeparation;
  int recvQueueSize;
  unsigned long         reliability;
};
```

Members

deadline

how long wait if publication is not available

minimumSeparation

requested minimum separation between issues

recvQueueSize

receiver's queue size

reliability

reliability requested (ORTE_RELIABILITY_BEST_EFFORTS or ORTE_RELIABILITY_STRICT)

ORTE_RELIABILITY_BEST_EFFORTS

Data are received with requested separation. Delivery in sequence order is not guaranteed. If a message has been drop, won't be resend.

ORTE_RELIABILITY_STRICT

All messages have to be acknowled by the subscribers. Every lost message will be retransmitted.

struct ORTEPublProp

`struct ORTEPublProp` — properties of a publication

Synopsis

```
struct ORTEPublProp {
  NtpTime persistence;
  int strength;
  int sendQueueSize;
  unsigned long         reliability;
};
```

Members

persistence

  porsistence of the publication

strength

  strength of the publication

sendQueueSize

  send queue size

reliability

  offered reliability of publication (see ORTESubsProp)

ORTEAppCreate

`ORTEAppCreate` — creates new ManagedApplication object

Synopsis

```
void ORTEAppCreate (ManagedApp ** papp);
```

Arguments

papp

pointer to ManagedApp

Description

Pointer to ManagedApp object (NULL if an error occured) will be returned in *papp*.

ORTEAppDestroy

`ORTEAppDestroy` — destroys ManagedApplication

Synopsis

```
void ORTEAppDestroy (ManagedApp * app);
```

Arguments

app

pointer to ManagedApp

Description

Destroys ManagedApplication object specified by *app*.

ORTEAppPublAdd

`ORTEAppPublAdd` — creates new publication

Synopsis

```
int ORTEAppPublAdd (ManagedApp * app, const char * topic, const char * typeName, NtpTime * persistence,
long strength);
```

Arguments

app

pointer to ManagedApp object which will provide this publication

topic

name of topic

typeName

data type description

persistence

persistende of publication

strength

strength of publication

Description

Returns handle to Publication object.

ORTEAppPublRemove

`ORTEAppPublRemove` — removes a publication

Synopsis

```
int ORTEAppPublRemove (ManagedApp * app, int happ);
```

Arguments

app

pointer to ManagedApp object which provides this publication

happ

handle to publication to be removed

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid publication handle.

ORTEAppPublSend

`ORTEAppPublSend` — creates new instance of data change to be publicated to its subscribers

Synopsis

```
int ORTEAppPublSend (ManagedApp * app, int happ, char * msg, u_int msg_len);
```

Arguments

app

pointer to ManagedApp object which provides this publication

happ

handle to publication

msg

pointer to buffer with data to be published

msg_len

size of data in buffer

Description

Returns ORTE_OK if successful or ORTE_QUEUE_FULL if send queue is full.

ORTEAppPublPropGet

`ORTEAppPublPropGet` — read properties of a publication

Synopsis

```
int ORTEAppPublPropGet (ManagedApp * app, int happ, ORTEPublProp * properties);
```

Arguments

app

pointer to ManagedApp object which provides this publication

happ

handle to publication

properties

pointer to ORTEPublProp structure where values of publication's properties will be stored

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid publication handle.

ORTEAppPublPropSet

`ORTEAppPublPropSet` — set properties of a publication

Synopsis

```
int ORTEAppPublPropSet (ManagedApp * app, int happ, ORTEPublProp * properties);
```

Arguments

app

pointer to ManagedApp object which provides this publication

happ

handle to publication

properties

pointer to ORTEPublProp structure containing values of publication's properties

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid publication handle.

ORTEAppPublWaitForSubs

`ORTEAppPublWaitForSubs` — waits for given number of subscriptions

Synopsis

```
int ORTEAppPublWaitForSubs (ManagedApp * app, int happ, NtpTime wait, u_short retries, u_short
no_subs);
```

Arguments

app

pointer to ManagedApp object which provides this publication

happ

handle to publication to be removed

wait

time how long to wait

retries

number of retries if specified number of subscriptions was not reached

no_subs

desired number of subscriptions

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid publication handle or ORTE_TIMEOUT if number of retries has been exhausted.

ORTEAppPublGetStatus

`ORTEAppPublGetStatus` — removes a publication

Synopsis

```
int ORTEAppPublGetStatus (ManagedApp * app, int happ, ORTEPublStatus * status);
```

Arguments

app

pointer to ManagedApp object which provides this publication

happ

handle to publication

status

pointer to ORTEPublStatus structure

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid publication handle.

ORTEAppSubsAdd

`ORTEAppSubsAdd` — adds a new subscription

Synopsis

```
int ORTEAppSubsAdd (ManagedApp * app, const char * topic, const char * typeName, NtpTime * minimumSeparation,
NtpTime * deadline, RcvMessageCallBack rcvMessageCallBack);
```

Arguments

app

pointer to ManagedApp object where this subscription will be created

topic

name of topic

typeName

name of data type

minimumSeparation

minimum time interval between two publications sent by Publisher as requested by Subscriber

deadline

-- undescribed --

rcvMessageCallBack

callback function called when new Subscription has been received

Description

Returns handle to Subscription object.

ORTEAppSubsRemove

`ORTEAppSubsRemove` — removes a subscription

Synopsis

```
int ORTEAppSubsRemove (ManagedApp * app, int happ);
```

Arguments

app

pointer to ManagedApp object

happ

handle to subscriotion to be removed

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid subscription handle.

ORTEAppSubsPropGet

`ORTEAppSubsPropGet` — get properties of a subscription

Synopsis

```
int ORTEAppSubsPropGet (ManagedApp * app, int happ, ORTESubsProp * properties);
```

Arguments

app

pointer to ManagedApp object which owns this subscription

happ

handle to publication

properties

pointer to ORTESubsProp structure where properties of subscrition will be stored

ORTEAppSubsPropSet

`ORTEAppSubsPropSet` — set properties of a subscription

Synopsis

```
int ORTEAppSubsPropSet (ManagedApp * app, int happ, ORTESubsProp * properties);
```

Arguments

app

pointer to ManagedApp object which owns this subscription

happ

handle to publication

properties

pointer to ORTESubsProp structure containing desired properties of the subscription

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid subscription handle.

ORTEAppSubsWaitForPubl

`ORTEAppSubsWaitForPubl` — waits for given number of publications

Synopsis

```
int ORTEAppSubsWaitForPubl (ManagedApp * app, int happ, NtpTime wait, u_short retries, u_short
no_publ);
```

Arguments

app

pointer to ManagedApp object which owns this subscription

happ

handle to subscription

wait

time how long to wait

retries

number of retries if specified number of publications was not reached

no_publ

-- undescribed --

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid subscription handle or ORTE_TIMEOUT if number of retries has been exhausted..

ORTEAppSubsGetStatus

`ORTEAppSubsGetStatus` — get status of a subscription

Synopsis

```
int ORTEAppSubsGetStatus (ManagedApp * app, int happ, ORTESubsStatus * status);
```

Arguments

`app`

pointer to ManagedApp object which owns this subscription

`happ`

handle to subscription

`status`

pointer to ORTESubsStatus structure

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *`happ`* is not valid subscription handle.

ORTEVerbositySet

`ORTEVerbositySet` — add a subscription

Synopsis

```
void ORTEVerbositySet (int level);
```

Arguments

`level`

level of verbosity, 0 - minimum, 5 - maximum

ORTESleepMs

`ORTESleepMs` — suspends calling thread for given time

Synopsis

```
ORTESleepMs ( x);
```

Arguments

x

time in miliseconds

NtpTimeAssembFromMs

`NtpTimeAssembFromMs` — converts seconds and miliseconds to NtpTime

Synopsis

```
NtpTimeAssembFromMs ( time,  s,  msec);
```

Arguments

time

time given in NtpTime structure

s

seconds portion of given time

msec

miliseconds portion of given time

NtpTimeDisAssembToMs

`NtpTimeDisAssembToMs` — converts NtpTime to seconds and miliseconds

Synopsis

```
NtpTimeDisAssembToMs ( s,  msec,  time);
```

Arguments

s

seconds portion of given time

msec

miliseconds portion of given time

time

time given in NtpTime structure

NtpTimeAssembFromUs

`NtpTimeAssembFromUs` — converts seconds and useconds to NtpTime

Synopsis

```
NtpTimeAssembFromUs ( time,  s,  usec);
```

Arguments

`time`

time given in NtpTime structure

`s`

seconds portion of given time

`usec`

microseconds portion of given time

NtpTimeDisAssembToUs

`NtpTimeDisAssembToUs` — converts NtpTime to seconds and useconds

Synopsis

```
NtpTimeDisAssembToUs ( s,  usec,  time);
```

Arguments

s

seconds portion of given time

usec

microseconds portion of given time

time

time given in NtpTime structure

2.1.4. Implementation issues

The RTPS protocol is implemented as a set of objects. Objects are of the following types:

Manager (M): Special object that facilitates the automatic discovery of other Managers. There is one Manager on each participating network node.
ManagedApplication (MA): An applciation that is managed by one or more Managers.
Writers (Publication, CSTWriter): provide locally available data (a composit state or stream of issues) on the network.
Readers (Subscription, CSTReader): obtain information provided by Writers.

The Manager is an independent process, which is created during application startup. It is a special Application that helps applications to automatically discover each other on the Network. Every Manager keeps track of its managees and their attributes. To provide this information on the Network, every Manager has the special CSTWriter writerApplications. The Composite State (CS) provided by the CSTWriter writerApplications are the attributes of all the ManagedApplications the Manager manages (its managees). Whenever the Manager accepts a new ManagedApplication as its managee, whenever the Manager loses a ManagedApplication as a managee or whenever an attribute of a managee changes, the CS of the writerApplications changes. Each such change creates new instance of CSChange which has to be transferred to all network objects (Managers and ManagedApplications) by means of CST protocol.

The Publication is used to publish issues to matching Subscription. The CSTWriter and CSTReader are the equivalent of the Publication and Subscription, respectively, but are used solely for the state-synchronization protocol.

The manager is composed from five kinds of objects:

WriterApplicationSelf: CSTWriter throught which the Manager provides information about its own parameters to Managers on other nodes.
ReaderManagers: CSTReader through which the Manager obtains information on the state of all other Managers on the Network.
WriterManagers: CSTWriter throught which the Manager will send the state of all Managers in the Network to all its managees.
ReaderApplications: CSTReader which is used for the registration of local and remote managedApplications.
WriterApplications: CSTWriter throught which the Manager will send information about its managees to other Managers in the Network.

A ManagedApplication is an Application that is managed by one or more Managers. Every ManagedApplication is managed by at least one Manager. TheManagedApplication has a special CSTWriter writerApplicationSelf. The Composite State of the ManagedApplication's writerApplicationSelf object contains only one NetworkObject - the application itself. The writerApplicationSelf of the ManagedApplication must be configured to announce its presence repeatedly and does not request nor expect acknowledgements. A Manager that discovers a new ManagedApplication through its readerApplications must decide whether it must manage this ManagedApplication or not. For this purpose, the attribute managerKeyList of the Application is used. If one of the ManagedApplication's keys (in the attribute managerKeyList) is equal to one of the Manager's keys, the Manager accepts the Application as a managee. If none of the keys are equal, the managed application is ignored. At the end of this process all Managers have discovered their managees and the ManagedApplications know all Managers in the Network.

The ManagedApplications now use the CST Protocol between the writerApplications of the Managers and the readerApplications of the ManagedApplications in order to discover other ManagedApplications in the Network. Every ManagedApplication has two special CSTWriters, writerPublications and writerSubscriptions, and two special CSTReaders, readerPublications and readerSubscriptions.

Once ManagedApplications have discovered each other, they use the standard CST protocol through these special CSTReaders and CSTWriter to transfer the attributes of all Publications and Subscriptions in the Network. The managedApplication is composed from seven kinds of objects.

WriterApplicationSelf: a CSTWriter throught which the ManagedApplication registers itself with the local Manager.
ReaderApplications: a CSTReader throught which the ManagedApplication receives information about another ManagedApplications in the network.
ReaderManagers: a CSTReader throught which the ManagedApplication receives information about Managers.
WriterPublications: a Writer that provides issues to one or more instances of a Subscription using the publish-subscribe protocol and semantics.
ReaderPublications: a Reader throught which the Publication receives information about Subscriptions.
WriterSubscriptions: a Writer that provides information about Subscription to Publications.
ReaderSubscriptions: a Reader that receives issues from one or more instances of Publication, using the publish-subscribe service.

Following example shows communication between two nodes (N1, N2). There are two applications running on each node - MA1.1, MA1.2 on node N1 and MA2.1, MA2.2 on node N2. Each node has it own manager (M1, M2).

1. MA1.1 introduces itself to local manager M1
2. M1 sends list of remote managers Mx and other local applications MA1.x
3. MA1.1 is introduced to all Mx by M1
4. All remote MAs are reported now to M1.1
5. Local MAs are queried for their CS (composite state)
6. All local MAs are sending their CS
7. Remote MAs are queried for their CS
8. All remote MAs are sending their CS

The corresponding publishers and subscribers with matching Topic and Type are connected and starts their data communication

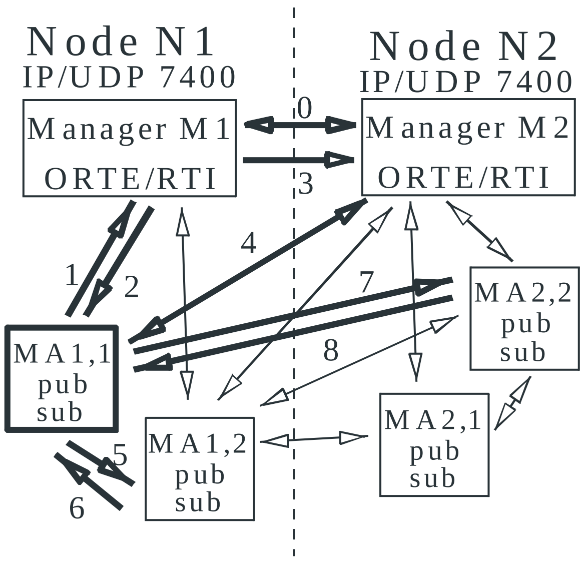


Figure 2-1. Communication among network objects.

2.1.5. Tests

There were not any serious tests performed yet. Current version has been intensively tested against reference implementation of the protocol. Results of these test indicate that ORTE is fully interoperable with implementation provided by another vendor.

2.1.6. Examples

The skeleton of an ORTE application is very simple:

```
#include <orte.h>

int main(int argc, char *argv[])
{
ManagedApp    *app1;

ORTEAppCreate(&app1);

/*
.....
here is your application dependent code
.....
*/
}
```

In order to exchange user data, the application must create the publications of its variables. Application which wants to receive an issues of published data must create a subscription. Properties of publication and subscription contain specification of Topic and TypeName, which specify an application variable within whole network. It is allowed to have more publications of same Topic and TypeName. If it subscribes to such publication, it will receive issues from all publications of the same Topic and TypeName. An publication will be created by calling function ORTEAppPublAdd. Once the publication is created, it is are ready to publish data using function ORTEAppPublSend.

```
  int h_pub;
NtpTime timePersistence;
long strength;
char msg[128];
u_long i=0;

NtpTimeAssembFromMs(timePersistence, 5, 0);     /* this issue is valid for 5 seconds */
```

```
strength=1;                                     /* strength of this publication */
h_pub=ORTEAppPublAdd(app1,
                     "HelloWorld",              /* Topic */
                     "HelloWorldData",          /* TypeName */
                     &timePersistence,
                     strength);
while (1) {
  sprintf(msg,"Hello World count:%li\n",i);
  ORTEAppPublSend(app1,h_pub,msg,strlen(msg)+1);
  ORTESleepMs(1000);                            /* sleep for 1 second */
  i++;
}
```

Subscribing application needs to create a subscription with publication's Topic and TypeName. A callback function will be then called when new issue from publisher will be received.

```
  ManagedApp    *app1;
int h_sub;
NtpTime minimumSeparation,deadline;

NtpTimeAssembFromMs(minimumSeparation, 0, 0);
NtpTimeAssembFromMs(deadline, 5, 0);
h_sub=ORTEAppSubsAdd(app1,
                     "HelloWorld",             /* Topic */
                     "HelloWorldData",         /* TypeName */
                     &minimumSeparation,
                     &deadline,
                     rcvCallBack);             /* callback function */
while (1) {
  ORTESleepMs(1000);
}
```

The callback function is shown in the following example:

```
void rcvCallBack(ORTERcvInfo *rcvInfo,u_char status)
{
switch (status) {
  case 0:                       /* Issue */
    printf("%s",rcvInfo->data);
    break;
  case 1:                       /* Deadline */
    printf("\ndeadline\n");
    break;
}
}
```

There must be the Manager process running on each network node. This manager must be started manualy before any other ORTE-enabled application. Manager process will be created by program **ORTEManager** with following options:

```
-P, --peer IPAddress1:IPAddress2:...:IPAddressn
-p, --port port
-v, --verbosity level
-V, --version
-h, --help
```

Each manager has to know where are other managers in the network. Their IP addresses are therefore specified as IPAddressX parameters. All managers must use the same port, the default port is 7400.

Example:

ORTEManager -P 147.32.86.167:147.32.86.186 -v 3

Now you are ready to run your ORTE enabled application.

There are following examples available:

HelloWorld: Very simple program demonstrating how to create an application which will publish some data and another application, which will subscribe to this publication.
Ping: Similar to HelloWorld example, publication and subscription is in one source code.
Teletype: More complicated example demonstrating functionality of various settings such as persistence, minimum separation etc.

2.1.7. Installation instructions

There are no any special steps in order to install ORTE package. Simply untar instalation package into desired directory, enter this directory and issue following commands:

./configure

make

make install

2.2. Real Time Ethernet analyzer

Real Time Ethernet analyzer is a module which adds support for RTPS protocol into Ethereal (http://www.ethereal.com) network analyzer.

2.2.1. Sumary

Name of the component	Real Time Ethernet analyzer
Author	Zdenek Sebek
Reviewer	not validated
Layer	High-level available
Version	0.1 alfa
Status	Alfa
Dependencies	Ethereal source code.
Release date	N/A

2.2.2. Description

Real Time Ethernet analyzer is not standalone tool. It is the module which is compiled into Ethereal network analyzer and adds support for RTPS protocol.

2.2.3. API / Compatibility

not applicable

2.2.4. Implementation issues

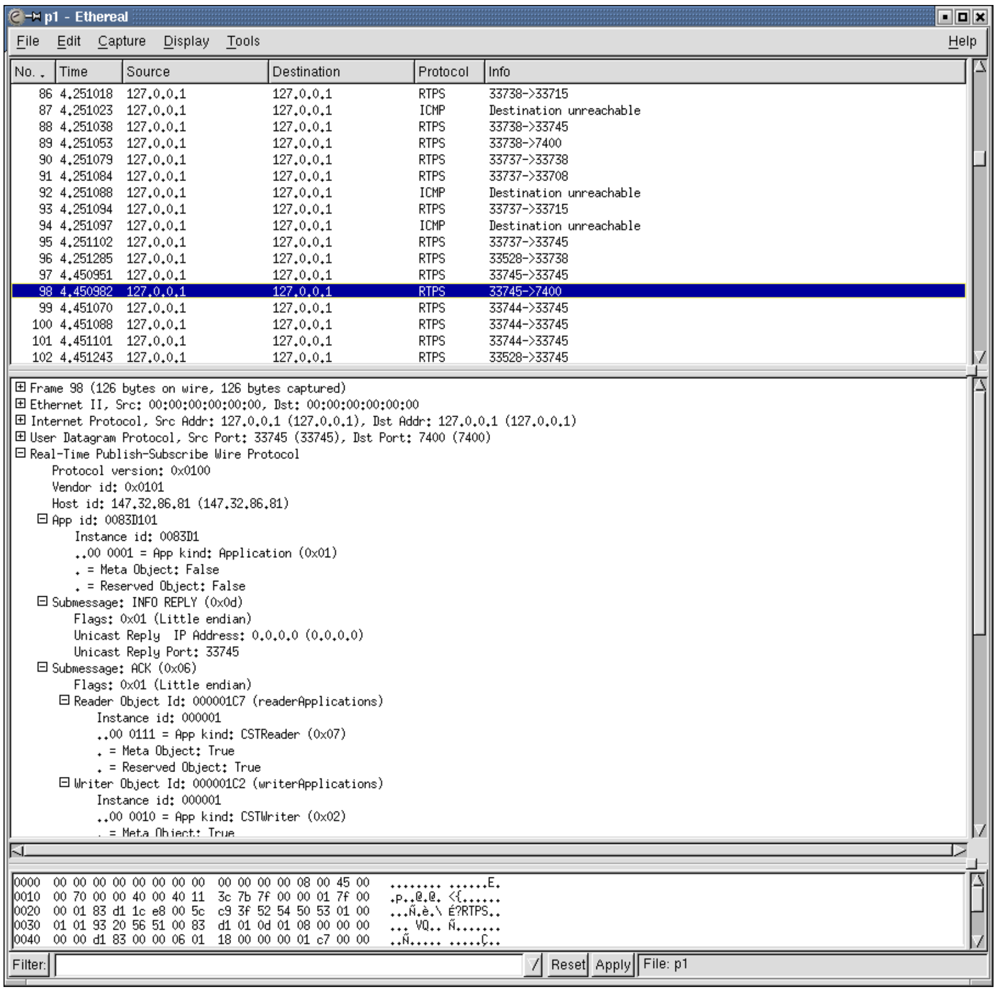
Internal structure is completly driven by requirements for Ethereal's modules. It consists of single function, which receives data as they were received from network, analyzes them according to RTPS data format description and vizualizes them by standard Ethereal's means.

2.2.5. Tests

The tests performed were focusing on evaluation of abilities to correctly parse whole set of RTPS commands. There are no other real-time parameters to be tested, because analyzis of received network frames is performed off-line and there are not any time constraints.

2.2.6. Examples

The structure of a sample RTPS message is shown on the Ethereal's window screenshot.



```
No.  Time      Source         Destination    Protocol  Info
 86 4.251018  127.0.0.1      127.0.0.1      RTPS      33738->33715
 87 4.251023  127.0.0.1      127.0.0.1      ICMP      Destination unreachable
 88 4.251038  127.0.0.1      127.0.0.1      RTPS      33738->33745
 89 4.251053  127.0.0.1      127.0.0.1      RTPS      33738->7400
 90 4.251079  127.0.0.1      127.0.0.1      RTPS      33737->33738
 91 4.251084  127.0.0.1      127.0.0.1      RTPS      33737->33708
 92 4.251088  127.0.0.1      127.0.0.1      ICMP      Destination unreachable
 93 4.251094  127.0.0.1      127.0.0.1      RTPS      33737->33715
 94 4.251097  127.0.0.1      127.0.0.1      ICMP      Destination unreachable
 95 4.251102  127.0.0.1      127.0.0.1      RTPS      33737->33745
 96 4.251285  127.0.0.1      127.0.0.1      RTPS      33528->33738
 97 4.450951  127.0.0.1      127.0.0.1      RTPS      33745->33745
 98 4.450982  127.0.0.1      127.0.0.1      RTPS      33745->7400
 99 4.451070  127.0.0.1      127.0.0.1      RTPS      33744->33745
100 4.451088  127.0.0.1      127.0.0.1      RTPS      33744->33745
101 4.451101  127.0.0.1      127.0.0.1      RTPS      33744->33745
102 4.451243  127.0.0.1      127.0.0.1      RTPS      33528->33745
```

```
⊞ Frame 98 (126 bytes on wire, 126 bytes captured)
⊞ Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:00:00:00:00:00
⊞ Internet Protocol, Src Addr: 127.0.0.1 (127.0.0.1), Dst Addr: 127.0.0.1 (127.0.0.1)
⊞ User Datagram Protocol, Src Port: 33745 (33745), Dst Port: 7400 (7400)
⊟ Real-Time Publish-Subscribe Wire Protocol
    Protocol version: 0x0100
    Vendor id: 0x0101
    Host id: 147.32.86.81 (147.32.86.81)
  ⊟ App id: 0083D101
        Instance id: 0083D1
        ..00 0001 = App kind: Application (0x01)
        . = Meta Object: False
        . = Reserved Object: False
  ⊟ Submessage: INFO REPLY (0x0d)
        Flags: 0x01 (Little endian)
        Unicast Reply  IP Address: 0.0.0.0 (0.0.0.0)
        Unicast Reply Port: 33745
  ⊟ Submessage: ACK (0x06)
        Flags: 0x01 (Little endian)
      ⊟ Reader Object Id: 000001C7 (readerApplications)
            Instance id: 000001
            ..00 0111 = App kind: CSTReader (0x07)
            . = Meta Object: True
            . = Reserved Object: True
      ⊟ Writer Object Id: 000001C2 (writerApplications)
            Instance id: 000001
            ..00 0010 = App kind: CSTWriter (0x02)
            . = Meta Object: True
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 70 00 00 40 00 40 11  3c 7b 7f 00 00 01 7f 00   .p..@.@. <{......
0020  00 01 83 d1 1c e8 00 5c  c9 3f 52 54 50 53 01 00   ...Ñ.è.\ É?RTPS..
0030  01 01 93 20 56 51 00 83  d1 01 0d 01 08 00 00 00   ... VQ.. Ñ.......
0040  00 00 d1 83 00 00 06 01  18 00 00 00 01 c7 00 00   ..Ñ..... .....Ç..
```

Figure 2-2. Screenshot

2.2.7. Installation instructions

First you need download source code distibution of Ethereal network analyze from http://www.ethereal.co and unpack it. Current implementation has been succesfully tested with Ethereal version 0.9.6 and 0.9.7. Untar instalation package into directory containg Ethereal's source. Edit file `Makefile.in`. Find all occurences of string `packet-rtsp` (yes, rtsp, it is not a typo) and add similar entries with string `packet-rtps`. Now you can compile Ethereal analyzer by following commands:

./configure

make

```
make install
```

Chapter 3. Verification

3.1. CAN model by timed automata /Petri Nets

3.1.1. Summary

Name of the component

CAN model by timed automata /Petri Nets

Description

This component is theoretical study offering methodology **tool support** for analysis of distributed system consisting of n independent processors and deterministic communication bus (CAN). In order to verify distributed RT system, application designer needs to create a model of application tasks and to interconnect this model with the communication bus model provided by this component. Finally he/she needs to define system properties to be verified (deadlock, missed deadline etc.). This component can be used either in a design phase or it can be used to verify existing implementation.

Author

Jan Krakora, Zdenek Hanzalek

Reviewer

not validated

Layer

High-level available

Version

0.1 Alfa

Status

Analysis

Dependencies

Not validated

Release date

2003-04-01

3.1.2. Description

3.1.2.1. Problem statement

This section deals with a design conception of theoretical study offering methodology tool supporting analysis of distributed Real Time (RT) systems. Figure 3-1 illustrates mayor topic of verification of distributed systems . The figure shows a control system consisting of n independent processors and CAN communication bus. Let us consider the parallel running applications in the real-time operating system (RTOS) environment and further let us consider the communication protocol behaving in Real-time manner.

The crucial problem is whether the general real-time control system (RTCS) [Buttazoo97] behaves in RT manner. This problem can be split into three subproblems that can be further composed together:

- application SW (modeled by application developer)

- RTOS (study of preemptive and cooperative schedulers) - see "Verification of cooperative scheduling and interrupt handlers" component
- RT communication - CAN (Medium Access Control modeling) - addressed in this component

Corresponding three sub models can be further combined to create RTCS model and it's possible behavior can be defined. Desired behavior of the RTCS has to be specified in the form of properties (e.g. deadlock, missed deadline, ...).

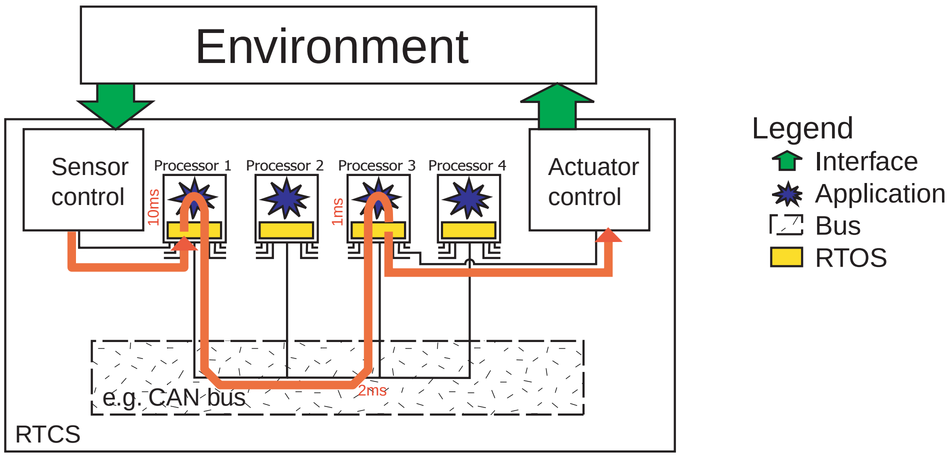


Figure 3-1. Real time control system structure with denotation of computation/communication times

Goal of "Verification of cooperative scheduling and interrupt handlers" and this component is to provide:

- model of RTOS and CAN
- develop examples of typical applications
- provide methodology for model checking of RTCS

To resolve the above mentioned problem we use a mathematical formalisms based on:

- system specification by means of communicating automata
- design system behavior formulated by means of CTL
- verification algorithm

While using this component the application developer can verify his RT applications that are communicating via CAN by checking of properties like for example whether all task deadlines are satisfied or whether the message is received before another one. This approach is an alternative to the one known as VOLCANO [Tindell94], and it offers more general framework for verification. Specifically it can be combined with RTOS and application SW.

3.1.2.2. CAN bus description

This section introduces a basic terminology further used in CAN model. It can be skipped by the reader familiar with this technology.

Controller Area Network (CAN) [CAN] is a serial bus system especially suited to interconnect smart devices to build smart systems or sub-systems.

3.1.2.2.1. Real-time data transmission

In real-time processing the urgency of messages to be exchanged over the network can differ greatly: a rapidly changing dimension, e.g. engine load, has to be transmitted more frequently and therefore with less delays than other dimensions, e.g. engine temperature.

The priority at which a message is transmitted compared to another less urgent message is specified by the identifier of each message. The priorities are laid down during system design in the form of corresponding binary values and cannot be changed dynamically. The identifier with the lowest binary number has the highest priority.

Bus access conflicts are resolved by bit-wise arbitration on the identifiers involved by each station observing the bus level bit for bit. This happens in accordance with the "wired and" mechanism, by which the dominant state overwrites the recessive state. The competition for bus allocation is lost by all those stations (nodes) with recessive transmission and dominant observation. All those "losers" automatically become receivers of the message with the highest priority and do not re-attempt transmission until the bus is available again.

Transmission requests are handled in the order of the importance of the messages for the system as a whole. This proves especially advantageous in overload situations. Since bus access is prioritized on the basis of the messages, it is possible to guarantee low individual latency times in real-time systems.

3.1.2.2.2. Message frame formats

The CAN protocol supports two message frame formats, the only essential difference being in the length of the identifier. The so-called CAN standard frame, also known as CAN 2.0 A, supports a length of 11 bits for the identifier, and the so-called CAN extended frame, also known as CAN 2.0 B, supports a length of 29 bits for the identifier.

- CAN standard frame

  A message in the CAN standard frame format begins with the start bit called "Start Of Frame (SOF)", this is followed by the "Arbitration field" which consist of the identifier and the "Remote Transmission Request (RTR)" bit used to distinguish between the data frame and the data request frame called remote frame. The following "Control field" contains the "IDentifier Extension (IDE)" bit to distinguish between the CAN standard frame and the CAN extended frame, as well as the "Data Length Code (DLC)" used to indicate the number of following data bytes in the "Data field". If the message is used as a remote frame, the DLC contains the number of requested data byte. The "Data field" that follows is able to hold up to 8 data byte. The integrity of the frame is guaranteed by the following "Cyclic Redundant Check (CRC)" sum. The "ACKnowledge (ACK) field" compromises the ACK slot and the ACK delimiter. The bit in the ACK slot is sent as a recessive bit and is overwritten as a dominant bit by those receivers which have at this time received the data correctly. Correct messages are acknowledged by the receivers regardless of the result of the acceptance test. The end of the message is indicated by "End Of Frame (EOF)". The "Intermission Frame Space (IFS)" is the minimum number of bits separating consecutive messages. If there is no following bus access by any station the bus remains idle.

- CAN extended frame

  A message in the CAN extended frame format is likely the same as a message in CAN standard frame format. The difference is the length of the identifier used. The identifier is made up of the existing 11-bit identifier (so-called base identifier) and an 18-bit extension (so-called identifier extension). The distinction between CAN standard frame format and CAN extended frame format is made by using the IDE bit which is transmitted as dominant in case of a frame in CAN standard frame format, and transmitted as recessive in case of a frame in CAN extended frame format. As the two formats have to co-exist on one bus, it is laid down which message has higher priority on the bus in the case of bus access collision with different formats and the same identifier / base identifier: The message in CAN standard frame format always has priority over the message in extended format.

CAN controllers which support the messages in CAN extended frame format are also able to send and receive messages in CAN standard frame format. When CAN controllers which only cover the CAN standard frame format are used in one network, then only messages in CAN standard frame can be transmitted in the entire network. Messages in CAN extended frame format would be misunderstood. However there are CAN controllers which only support CAN standard frame format but recognize messages in CAN extended frame format and ignore them (version 2.0 B passive).

3.1.2.2.3. Detecting and signaling errors

Unlike other bus systems, the CAN protocol does not use acknowledgment messages but instead signals any errors immediately as they occur. For error detection the CAN protocol implements three mechanisms at the message level:

- Cyclic Redundancy Check (CRC).

  The CRC safeguards the information in the frame by adding redundant check bits at the transmission end. At the receiver these bits are re-computed and tested against the received bits. If they do not agree there has been a CRC error.

- Frame check.

  This mechanism verifies the structure of the transmitted frame by checking the bit fields against the fixed format and the frame size. Errors detected by frame checks are designated "format errors".

- ACK errors.

  As already mentioned frames received are acknowledged by all receivers through positive acknowledgment. If no acknowledgment is received by the transmitter of the message an ACK error is indicated.

The CAN protocol also implements two mechanisms for error detection at the bit level:

- Monitoring.

  The ability of the transmitter to detect errors is based on the monitoring of bus signals. Each station which transmits also observes the bus level and thus detects differences between the bit sent and the bit received. This permits reliable detection of global errors and errors local to the transmitter.

- Bit stuffing.

  The coding of the individual bits is tested at bit level. The bit representation used by CAN is "Non Return to Zero (NRZ)" coding, which guarantees maximum efficiency in bit coding. The synchronization edges are generated by means of bit stuffing. That means after five consecutive equal bits the transmitter inserts into the bit stream a stuff bit with the complementary value, which is removed by the receivers.

If one or more errors are discovered by at least one station using the above mechanisms, the current transmission is aborted by sending an "error flag". This prevents other stations accepting the message and thus ensures the consistency of data throughout the network. After transmission of an erroneous message that has been aborted, the sender automatically re-attempts transmission (automatic re-transmission). There may again competition for bus allocation.

However effective and efficient the method described may be, in the event of a defective station it might lead to all messages (including correct ones) being aborted. If no measures fr self-monitoring were taken, the bus system would be blocked by this. The CAN protocol therefore provides a mechanism to distinguishing sporadic errors from permanent errors and local failures at the station. This is done by statistical assessment of station error situations with the aim of recognizing a stations own defects and possibly entering an operation mode where the rest of the CAN network is not negatively affected. This may go as far as the station switching itself off to prevent messages erroneously from being recognized as incorrect .

3.1.3. API/Compatibility

Not applicable.

3.1.4. Implementation issues

3.1.4.1. Model of bit-wise arbitration

The model of CAN arbitration is shown in Figure 3-2. The model describes the control MAC mechanism of the CAN bus for one message accessing the bus. The location *no_trans_needed* represents a situation when the arbitration model is waiting for trans_request from application process. The locations *send_bit_to_bus*, *listen_bus*, *check_next_bit* represent the arbitration process. The locations *request_denied* and *request_success* describe arbitration result in arbitration process.

First of all let us assume there is only one transmission request from one application process at each node. The message is in CAN data frame format and without loose of generality the first bit (SOF) was correctly sent. In first step the first bit from the Arbitration filed is sent to the bus (transition *send_bit_to_bus* -> *listen_bus*). At the same time the bus is sensed by the transmitting node and both transmitted (*id* local variable) and sensed (*signal* global variable) bits are compared. If they are identical and end of the Arbitration field (*i==(nsigi-1)*) was not reached the next bit is proceeded (*check_next_bit* location). If the arbitration is finished the node wins the arbitration (*request_success* location).After than the next message frame bits (DLC, Data field etc.) are send to the bus.

The CAN Arbitration field model designed in UPPAAL [UPPAAL] includes the information about the duration of each bit-time given by invariant *t<=1* in listen_bus location and guards *t>=1*, *t>=0* on outgoing transitions.

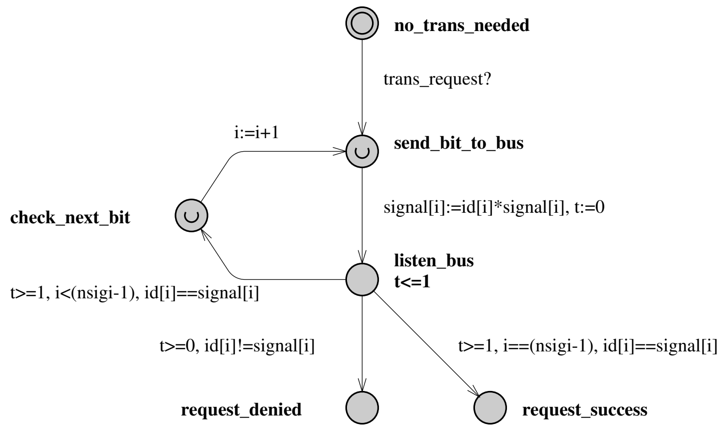


Figure 3-2. Arbitration model

3.1.4.2. MAC arbiter implementation

The implementation of the MAC arbitration model is depicted in Figure 3-3. There are three sections:

- arbitration section described already in Figure 3-2
- synchronization section (*waiting_for_free_bus*-> *send_bit_to_bus* transition) that is used to synchronize all transmitting nodes prior to arbitration (this part realizes broadcast communication [UPPAAL]) and
- data transmission section characterized by locations *trans_section*, *trans_section_finished* and *trans_finished*

The principle is as follows. If the trans_request comes then the node is in waiting state (*waiting_for_free_bus*) till the bus isn't free. If the bus is idle the arbitration starts (synchronization by *broadcast_synch* channel variable). If the transmission was denied (*trans_denied*) the transmission request is immediately repeated (by *bs_i1* local variable) and the node is again waiting for the bus (*waiting_for_free_bus* location). Otherwise the node message is sent (*trans_section*). When all message's bits were sent (*trans_section_finished*) the bus gets idle and the application process is informed about the end of transmission (*trans_finished* -> *no_trans_needed* transition).

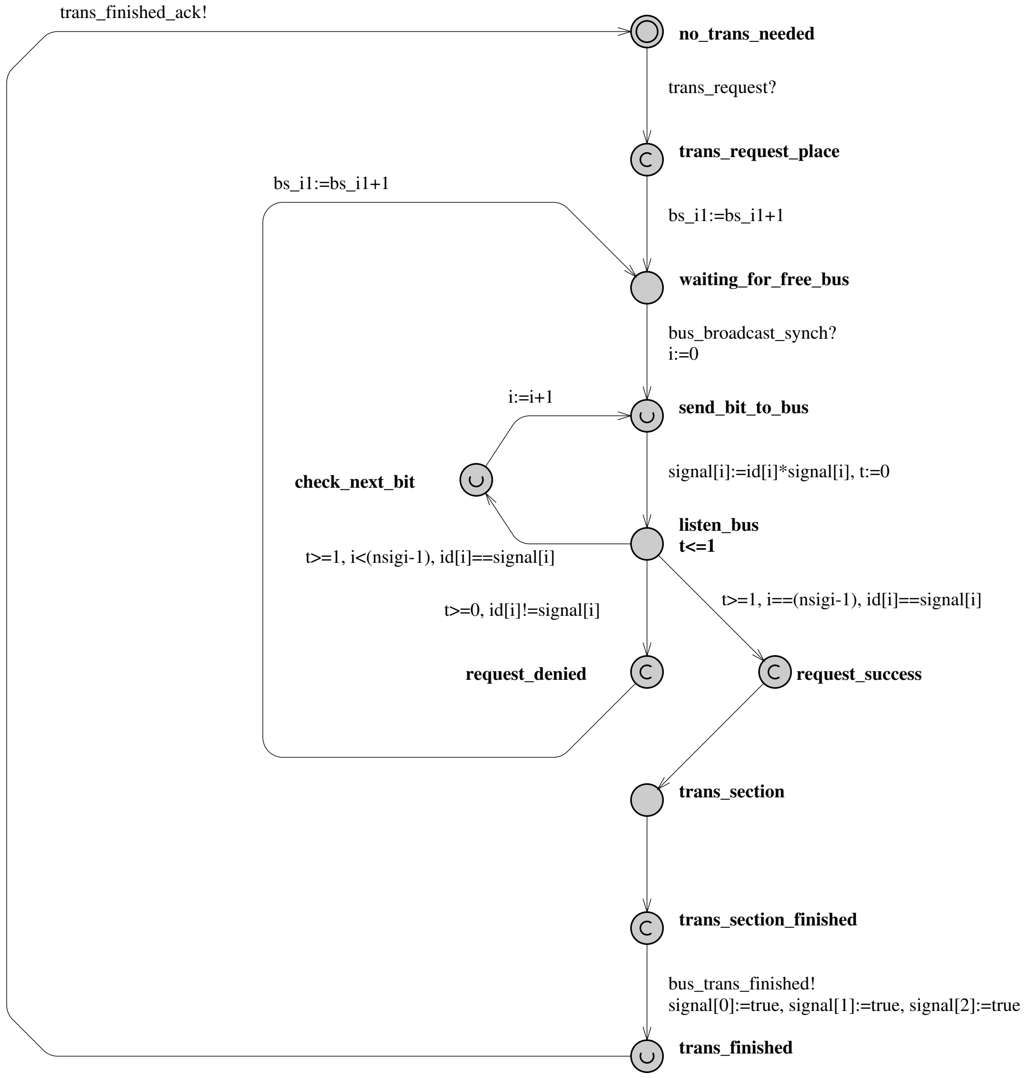


Figure 3-3. MAC model

The Figure 3-4 depicts the physical bus behavior. The bus can be either idle or busy. The "idle" means there is no activity on the bus and the "busy" corresponds to active bus. The *bc_place1* and *bc_place2* locations are part of the broadcast communication model.

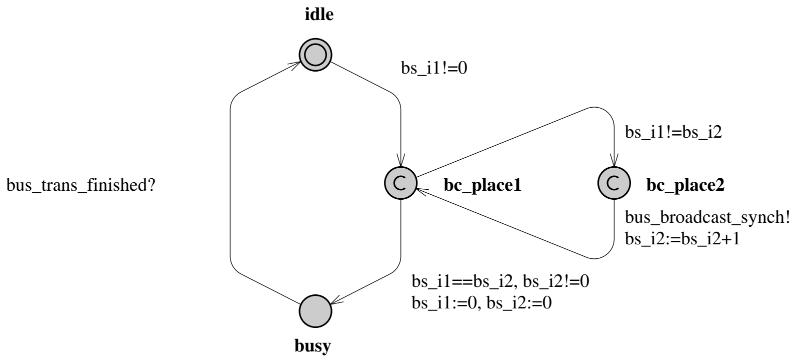


Figure 3-4. Bus model

3.1.5. Tests

Not applicable.

3.1.6. Examples

3.1.6.1. Implementation of process

In Figure 3-5 there are 3 nodes (one application process on each node) using the MAC arbitration model. The application process 1 repeatedly sends the high priority message, the application process 2 repeatedly sends the medium priority message and the application process 3 repeatedly sends the low priority message. The question is how to design the application processes and how to verify the properties of the whole model.

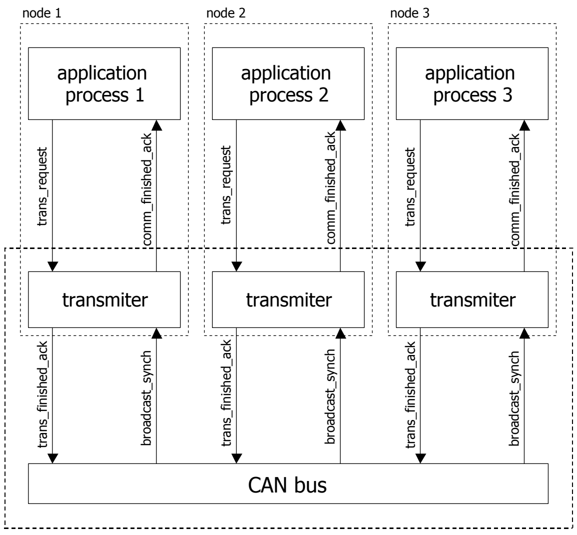


Figure 3-5. Example model

The application process is depicted in Figure 3-6. The first state (*no_communication_activity* location) represents a situation when the process does not perform any communication activity (e.g. it performs some computations). The second one describes situations the process has some data to be transmitted(*communication* location).

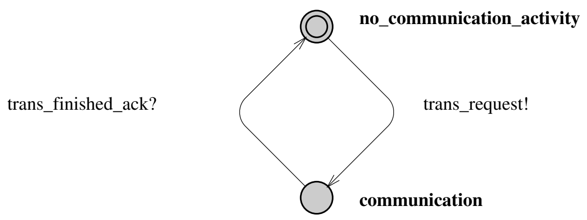


Figure 3-6. Application process model

Verification of the model:

The properties to be verified are the following:

1. Does the system include deadlock?
2. Is there any state in which two messages are sent on the bus at the same time?
3. Does higher priority message win the arbitration?

These properties have to be transformed to the verification tool formalism. The formula (formula form is described in [UPPAAL]) are:

1. A[](not deadlock)
2. E<> not (MAC_1.trans_section and MAC_2.trans_section and MAC_3.trans_section)
3. All the following formulae have to be satisfied:
   - E<> (MAC_1.request_denied and MAC_2.trans_section and MAC_3.request_denied);
   - E<> (MAC_1.request_denied and MAC_2.trans_section and MAC_3.no_trans_needed);
   - E<> (MAC_1.request_denied and MAC_2.no_trans_needed and MAC_3.trans_section);
   - E<> (MAC_1.no_trans_needed and MAC_2.trans_section and MAC_3.request_denied);
   - E<> (MAC_1.trans_section and MAC_2.no_trans_needed and MAC_3.no_trans_needed);
   - E<> (MAC_1.no_trans_needed and MAC_2.trans_section and MAC_3.no_trans_needed);
   - E<> (MAC_1.no_trans_needed and MAC_2.no_trans_needed and MAC_3.trans_section);

The results of UPPAAL analyze are:

1. Property satisfied
2. Property satisfied
3. All properties are satisfied

Timed automata in UPPAAL can also contain for example time information which can influence behavior of application in global point of view. For example the computing time or ping-pong response times. Application developer can also check if the evolution of states is running in right way. For example:

- Does exist the sequence:

  1. send high priority message
  2. communication via bus for high priority message was permitted
  3. send low priority message
  4. communication via bus for low priority message was permitted too?

3.1.7. Installation instructions

Not applicable.

Bibliography

[Holzmann91] Gerard J. Holzmann, 1991, Prentice Hall, *Design and validation of computer protocols*.

[Buttazoo97] C. Buttazzo, 1997, Kluwer Academic Publisher, *Hard Real-time computing systems: Predictable Scheduling Algorithms and Applications*.

[Best98] Eike Best and Bernd Grahlmann, 1998, *Programming Environment based on Petri nets: Docummentation and User Guide Version 1.8*.

[Best93] Eike Best and R. P. Hopkins, 1993, *B(PN)^2 - A Basic Petri Nets Programming Notation*.

[UPPAAL] *UPPAAL tool: http://www.docs.uu.se/docs/rtmv/uppaal/*.

[PEPTOOL] *PEP tool: http://theoretica.informatik.uni-oldenburg.de/~pep/*.

[Tindell94] Ken Tindell and A. Burns, 1994, *Guaranteeing Message Latencies on Controller Area Network (CAN)* .

[CAN] *CAN description: http://www.can-cia.de*.

3.2. Verification of cooperative scheduling and interrupt handlers

3.2.1. Summary

Name of the component

Verification of cooperative scheduling and interrupt handlers

Description

This component is a theoretical study offering methodology and tool support for model checking of real-time applications running under multitasking operating system. Theoretical background is based on timed automata by Allur and Dill. As this approach does not allow to model pre-emption we focus on cooperative scheduling. The cooperative scheduler under assumption performs rescheduling in specific points given by "yield" instruction in the application processes. In the addition, interrupt service routines are considered, and their enabling/disabling is controlled by interrupt server considering specified server capacity. The server capacity has influence on the margins of the computation times in the application processes. Such systems, used in practical real-time applications, can be modelled by timed automata and further verified by existing model checking tools. The approach is illustrated in the form of examples in the real-time verification tool UPPAAL.

Author

Libor Waszniowski, Zdenek Hanzalek

Reviewer

not validated

Layer

High-level available

Version

0.1 Alfa

Status

Analysis

Dependencies

Not validated

Release date

N/A

3.2.2. Description

The aim of this chapter is to show, how timed automata [Alur94] can be applied to modelling of real time software applications running under operating system with cooperative scheduling. Model checking theory based on timed automata and implemented in model checking tools (e.g. UPPAAL[David]) can be used for verifying time parameters or safety and liveness properties of proposed models. The application under consideration runs under multitasking operating system, it consists of several process, it includes mechanisms for interrupt handling, and it uses inter-process communication primitives like semaphores, queues etc. Since the processes are not truly concurrent, they share the processor, it is needed to model the scheduler.

Timing analysis of software (especially with concurrency and synchronisation) is not trivial problem and it requires sophisticated methods and analysis tools. Several special purpose methods have been developed in the area of real time scheduling [Buttazzo97],[Liu2000]. These methods e.g. rate monotonic analysis (RMA) [Sha91] are very successful for analysis of time-driven systems with periodic processes. To deal with non-periodic processes in event-driven systems, the standard method is to consider the non-periodic process as the periodic one using the minimal inter-arrival time as process period. The analysis based on such model is too pessimistic in some cases since inter-arrival times can vary over time [Fersman02]. Incorporation of inter-process communication primitives leads to pessimistic results as well.

To achieve more precise analysis, process models allowing more precise and complex timing constraints are needed. In [Fersman02] the timed automata are extended by asynchronous processes i.e. processes triggered by events to provide model for event-driven systems, which is further used for schedulability analysis. Processes (in [Fersman02] called tasks) associated to locations of timed automaton are executable programs characterised by its worst-case execution time, deadline and other parameters for scheduling (e.g. priority). Transition leading to a location in such automaton denotes an event triggering the process and the guard on transition specifies the possible arrival times of the event. Released processes are stored in a process queue and they are assumed to be executed according to a given scheduling strategy. Both non-preemptive and preemptive scheduling strategies are allowed. In the case of non-preemptive processes, the schedulability checking problem can be transformed to the reachability problem for ordinary timed automata. In the case of preemptive processes, the schedulability checking problem can be transformed to a reachability problem for bounded time automata with subtraction. Both of these problems are decidable [Fersman02].

The model based on the above mentioned extended timed automata can deal with non-periodic processes in more accurate manner than for example RMA, which does not contain any representation of internal process structure and inter-process communication. Therefore any worst-case blocking time in RMA(e.g. inter-process communication) must be involved in the worst-case execution time.

Approaches based on the worst case computation time of the whole process (e.g. RMA [Sha91] or timed automata with asynchronous processes [Fersman02]) lead to pessimistic conclusion in schedulability analysis since the worst case blocking time is considered for the resource sharing.

This disadvantage is overcome by more detailed process model proposed in [Corbett96] providing a method for constructing models of real time Ada tasking programs. Time, safety or liveness properties of produced model based on constant slope linear hybrid automata can be automatically analysed by HyTech verifier. The state of the hybrid automaton consists of various state variables representing an abstraction of program's state and also of continuous variables used to measure the amount of CPU time allocated to each process. A transition of the hybrid automaton represents execution of the sequential code segment. The timing constraints of the transition are derived from the time bounds of the corresponding code. Even thought author reports that the analysing algorithm does usually terminate in practice, the reachability problem for hybrid automata is undecidable in general.

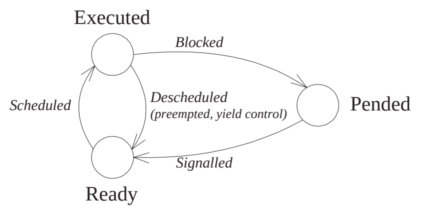
Hybrid automaton (or some its subclass e.g. stopwatch automaton [Cassez2000]) is needed to model preemption since it is necessary to accumulate computing time of each process separately. The continuous variable used to measure the amount of CPU time allocated to each process must be stopped when the corresponding process is preempted and must progress when the corresponding process is executed. Such behaviour cannot be modelled by timed automaton that does not allow stopping of the clock variable when the process was preempted.

Preemptive schedulers are known to provide higher utilisation of processor than cooperative ones [Buttazzo97]. On the other hand the processor utilisation is less important criterion when the schedulability can be proven for a given set of processes under cooperative policy. Moreover the cooperative scheduling has some advantages important especially for hard real time applications. In cooperative scheduling, process specifies when it is willing to release CPU to another process. Then it is easy to make sure all data structures are in a defined state. Applications using cooperative scheduling are therefore easier to program and to debug.

In this deliverable we present another important advantage of cooperative scheduling that is possibility to create mathematical model of the application based on timed automata and to verify its time, safety and liveness properties. Opposite to the model of the system with preemption based on hybrid automata, this approach has guarantied termination of verification algorithm due to decidability of reachability problem and model checking of timed computation tree logic (TCTL) problem. Moreover timed automata are one of the most studied models for real time systems and several model checkers are available (e.g. Kronos and UPPAAL[David])

Multitasking operating system and scheduling anomaly

Several processes share one processor in the systems with multitasking. The processor sharing is managed by the scheduler according to the scheduling policy. Process changes its state (state from the point of view of operating system) according to the state transition diagram in Figure 3-7 representing both, cooperative scheduling (*"yield control"* on *Deschedule* transition) or preemptive scheduling (*"preempted"* on *Deschedule* transition).



Executed	-	the process is executed on the processor
Pended	-	execution of the process cannot continue due to waiting for some event (communication, timer, etc.)
Ready	-	process is ready for execution and it is waiting in ready queue for the processor.

Figure 3-7. State transition diagram of the process in the multitasking operating system

Several multiprocessor time anomalies are known in the scheduling theory [Buttazzo97], [Graham69], [Liu2000]. Similar non-linear behaviour (a shortening of the computation time leading to the prolongation of the completion time) can be found on one processor regardless the scheduling policy (preemptive or cooperative), when the processes contain computations, resource sharing and idle waiting (notice that idle waiting is processed in parallel with computation of another process).

Example depicted in Figure 3-8 shows a high priority processes *P-high* and a low priority process *P-low* sharing one resource represented by a semaphore *Sem* . The processes consist of computations with specified deterministic computation time, of idle waiting with specified deterministic delay and of inter process communication through semaphore, which can be hold by only one process. The computation times and delays given behind slash are assumed to be constant. The computation time of *CompC/C* is *C* =2 in the instance a) or *C* =1 in the instance b).

In the instance a) regardless the scheduling policy (priority based preemptive or priority based cooperative) the semaphore is taken by *P-high* first. Consequently the process *P-high* is completed in 7 time units and the process *P-low* is completed in 9 time units, see Figure 3-8 a). In the instance b), the semaphore is taken by process *P-low* first and consequently the process *P-high* is completed in 9 time units and the process *P-low* is completed in 10 time units, see Figure 3-8 b).

The shortening of the computation time in the process P-low (*C* shorted from 2 to 1) leads to the prolongation of the completion time of both processes. As a consequence this example illustrates some important phenomena:

even for preemptive scheduling policy the low priority process influences completion time of the high priority process (due to the shared resource)

when one wants to make use of the internal process structure, then it is needed to specify lower margins of computation times even for schedulability analysis (studying the upper margin of the process completion time).

Based on these observations we provide the models including upper and lower margins of the computation time, inter process communication primitives and delays. In addition to that we provide a simple solution for verification of models including interrupts.

```
Process P-high              Process P-low
{                           {
  Delay/2                     CompC/C
  Take Sem                    Take Sem
  CompA/1                     Delay/3
  Give Sem                    Give Sem
  Delay/1                     CompD/1
  CompB/3                     CompE/1
}                           }
```

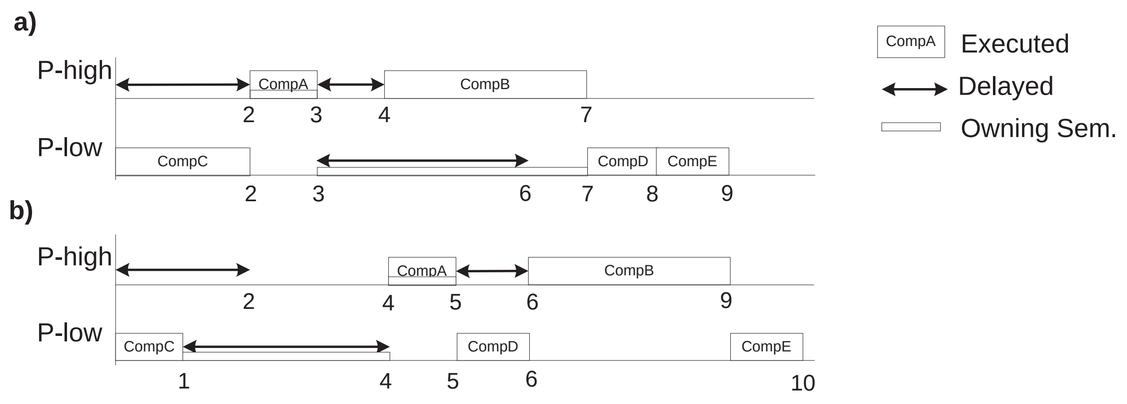


Figure 3-8. Example of monoprocessor scheduling anomaly

3.2.3. API/Compatibility

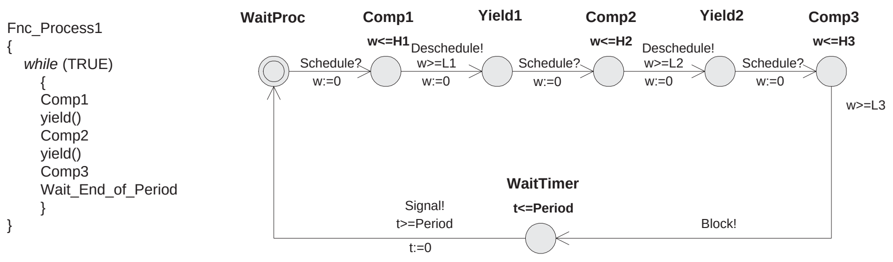
Not applicable.

3.2.4. Implementation issues

3.2.4.1. Cooperative scheduling

Cooperative scheduling enables to deschedule currently executed process only in explicitly specified points, where the system call *yield()* is called or where the process is waiting.

The example of the application process model is depicted in Figure 3-9. We can recognise four types of locations there. Except one location *WaitTimer* , where the process does not require processor, there are several *Computation* locations corresponding to sequential blocks of code (*Comp*) requiring non-preemptible execution on the processor. *Computations* do not contain any blocking operation. Each two successive *Computation* locations are separated by one *Yield* location corresponding to yield instruction where the process can be descheduled and then it waits there until it is scheduled again. *WaitTimer* location is followed by *WaitProc* location where the process waits until it is signalled and consequently scheduled.



```
Fnc_Process1
{
   while (TRUE)
      {
      Comp1
      yield()
      Comp2
      yield()
      Comp3
      Wait_End_of_Period
      }
}
```

Figure 3-9. Model of the application process executed under cooperative scheduling policy

As each part of the program modelled by *Computation* location cannot be affected by the preemption, its finishing time is known a priory and equal to computation time bounded by interval L,H (lower and upper margins allowing to involve uncertainty of execution time due to non-modelled code branching inside the computations, bus errors, cache faults, page faults, cycle stealing by DMA device, etc.). *Computation* locations are therefore guarded by standard time conditions supported by timed automata.

The following behaviour of the cooperative scheduler is assumed: if the processor is free, the process with the highest priority among all processes in the ready queue is scheduled. The currently executed process will run until it voluntarily relinquishes processor by calling system call *yield()* or until it is blocked. The model of the cooperative scheduler is created as the network of automata synchronised with application processes through synchronisation channels as depicted in Figure 3-10. *Deschedule* channel is used to signal that the process relinquishes the processor (by *yield()*). The scheduler chooses the highest priority ready process and enables its execution through *Schedule* channel.

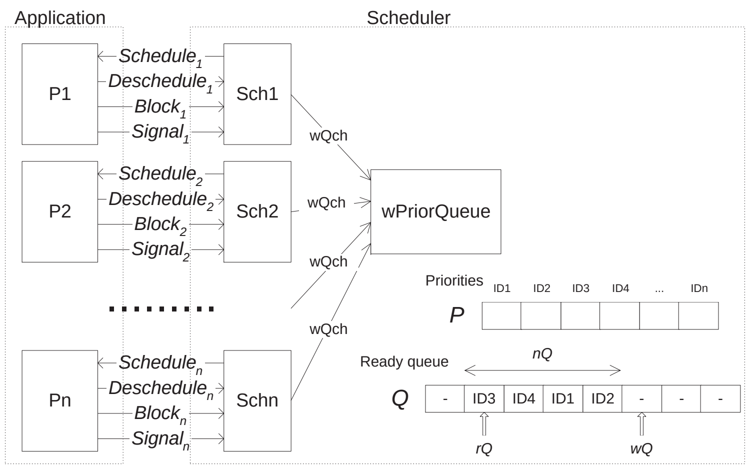


Figure 3-10. Synchronisation of cooperative scheduler with application processes

One automaton of the cooperative scheduler model (*Sch* $_i$) is depicted in Figure 3-11.

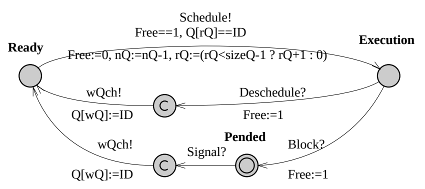


Figure 3-11. One automaton (*Schi*) of the cooperative scheduler in Figure 3-10

Each process is identified by unique integer *ID* (0,1,2,...). Priority of the process is stored in global array *P* , indexed by *ID* . *ID* s of all processes, which are in *Ready* state, are stored in queue modelled as global array *Q* of the size *sizeQ* representing circular buffer. The integer *nQ* is the number of elements in the queue. The integer *rQ* is the position for reading of the first element in *Q* and the integer *wQ* is position of the first empty element in *Q* as is depicted in Figure 3-10. Processes are ordered in descending order according to their priorities in *Q* (*rQ* points to the ready process with highest priority). Therefore *Q* must be reordered after writing new *ID* to the *Q* on the position *wQ* . Ordering according priorities is provided by automaton *wPriorQueue*. Reordering mechanism is started by synchronisation channel *wQch*.

Note on modelling of context switch time:

Please notice that the model of the scheduler proposed in Figure 3-11 is simplified by assumption that the context switch does not take any time. But for proper exploration of time properties of real-time system the context switch time should be considered. Because the context switch in cooperative scheduling occurs once per *Computation* location, context switch time can be involved in the computation time of each *Computation*.

3.2.4.2. Interrupts

Interrupts are usually used for fast handling of asynchronous external events. Interrupt is particularly important in cooperative scheduling since a low priority process cannot be preempted and therefore a high priority process cannot be used to handle asynchronous event when short requesting time is required. When the interrupt request (IRQ) arrives from the environment and corresponding interrupt is enabled, currently executed process is interrupted and interrupt service routine (ISR) is executed. The *relative finishing* *time F* of currently executed *Computation* is therefore prolonged by computation time of ISR (C_{ISR}) and it is no more equal to known *computation time* . In the timed automata process model it is needed to change upper margin *H* of each computation location. Each *H* is prolonged by *MaxSC* , the value corresponding to the processor time reserved for all interrupt service routines. Since the number of interrupt requests depends on the environment, the total computation time of all ISR (C_{ISR}) is not known a priory and moreover the existence of its upper bound is not guaranteed.

The *interrupt server* limiting amount of CPU time spent for interrupts (similar to deferrable server [Buttazzo97][Larsen95]) is used to guarantee that C_{ISR}does not exceed *MaxSC* value . The lower margin *L* of computation location is not affected by interrupts (situation when computation time reaches the lower bound and no interrupt occurs). The architecture of the system with *interrupt server* is depicted in Figure 3-12. Interrupt service routines are not called directly when some interrupt is requested, but they are wrapped by the code of *ISR_Server()* function (see Figure 3-13). The *interrupt server* has specified *server capacity SC* , which is filled by the value *MaxSC* at the beginning of each computation. The function *Fill_Server(MaxSC)* listed in Figure 3-13 is used for it. When an interrupt occurs the *server capacity SC* is decreased by the value of corresponding C_{ISR} and *interrupt server* checks if the remaining capacity *SC* is sufficient for handling next *ISR* . If not the corresponding *IRQ* is disabled. This check is provided when *SC* changes, once by *Fill_Server()* and repeatedly on each interrupt by *ISR_Server()* (both listed in Figure 3-13). Notice that C_S , the computation time of *ISR_Server()* , is considered. Further *H* has to be prolonged by C_{FS} , the computation time of the function *Fill_Server()* (see Figure 3-14).

Figure 3-15 shows the time diagram when *IRQ1* occurred twice within computation *Comp1* . Suppose system containing two sources of interrupts (*IRQ1* and *IRQ2*) with the following computation times: $C_{Comp1} = 21$, $C_{FS} = 4$, $C_S = 4$, $C_{ISR1} = 4$, $C_{ISR2} = 7$ and $MaxSC1 = 17$. The routine *Fill server* is executed at the beginning of *Comp1* at time *0* . This routine sets the server capacity *SC* to the value *MaxSC1* and it checks if this value is sufficient for handling all interrupt service routines. Interrupt request *IRQ1* occurs at time 9, execution of *Comp1* is interrupted and execution of *ISR_Server()* routine is started. This routine decreases server capacity *SC* by computation time of interrupt server C_S and by computation time of interrupt service routine C_{ISR1} . Then it starts interrupt service routine *ISR1* and then it checks if the remaining server capacity *SC* is sufficient for next interrupt request handling. Since this is not the case of *IRQ2* ($SC=9 < C_S + C_{ISR2} = 11$), the *IRQ2* is disabled. Then the execution of *Comp1* continues until it is again interrupted by the second occurrence of *IRQ1* at time *25* . After this interrupt handling, the remaining server capacity *SC* is only *1* that is not sufficient for handling any interrupt. Therefore both interrupt requests are disabled. The server capacity *SC* is replenished with the new value *MaxSC2* by routine *Fill server* at the beginning of next computation *Comp2* at time *41* . Notice that the function *ISR_Server()* supposes that the hardware does not support nested interrupts (*ISR_Server()* cannot be interrupted by another interrupt).

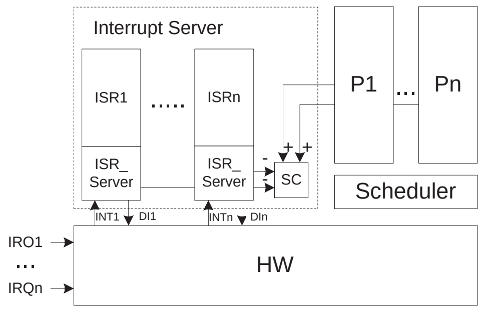


Figure 3-12. System architecture with interrupt server

```
Fill_Server (MaxSC)
{
    Disable_INT
    SC :=MaxSC
    Check for all IRQ
        if $(SC – C_{ISR} - C_S) < 0$
            Disable IRQ
        else
            Enable IRQ
    Enable_INT
}

ISR_Server ()
{
    SC := $SC – C_{ISR} - C_S$
    call ISR
    Check for all IRQ
        if $(SC – C_{ISR} - C_S) < 0$
            Disable IRQ
        else
            Enable IRQ
}
```

Figure 3-13. ***Interrupt server*** **routines**

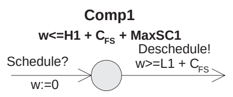


Figure 3-14. ***Computation*** **location considering interrupts**

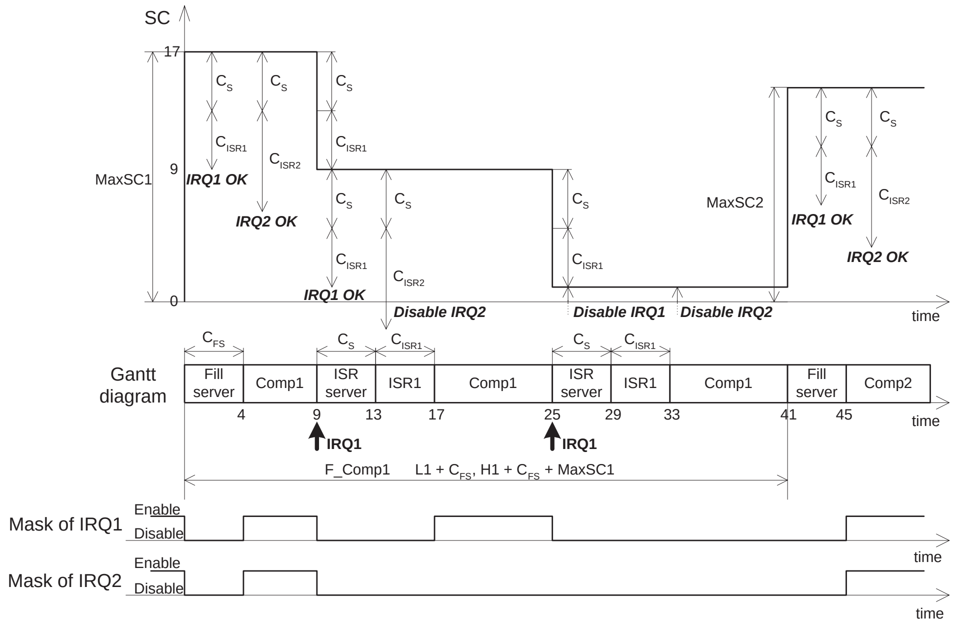


Figure 3-15. Time diagram of ISR execution within ***interrupt server***

Choice of MaxCS value for different locations depends on application requirements and it is specified at the design stage. Section 3.2.6, *Examples* section shows an example application with one IRQ, two processes of different priority and one semaphore (semaphore is discussed in Section 3.2.4.3.1, *Semaphore*).

3.2.4.3. Inter process communication primitives

Very important part of each multitasking application (and source of many possible errors) is communication between processes and their synchronisation. Operating system usually provides many facilities to manage inter process communication. It is not intention of this paper to introduce models of all possible kinds of inter process communication. We only show on example of *semaphore* how to extend the proposed model of scheduler and application. The context switch time is not considered for simplification in this section.

3.2.4.3.1. Semaphore

The semaphore is the primitive used mostly for synchronisation and mutual access to resources. It can be taken or given by process using the system calls *Take()* or *Give()* . When the semaphore is given, its value is increased. When the semaphore is taken, its value is decreased. When the value of the semaphore is zero, it cannot be taken and the process attempting to take it is blocked until the semaphore is given by other process. This blocking time can be bounded by timeout. When more than one processes are blocked on one semaphore, they are waiting in priority queue or FIFO (First In First Out) queue. This basic behaviour of semaphore can be modified according to the purpose it is dedicated to. We suppose the semaphore being of counting type with value ranging from zero to *MaxCount* .

In this section we introduce model of the process using semaphore. In addition it is needed to extend the scheduler model. Example of application process model is depicted in Figure 3-16. The process attempts to take the semaphore by synchronisation *Take!* . Then it waits in location *WaitSem* until the semaphore is taken (synchronisation *Taken?*) or until timeout expires (synchronisation *TOut!*). The synchronisation *Give!* is used to give the semaphore. Notice that giving the semaphore is not blocking operation and therefore the semaphore is given on the transition entering the *Computation* location. On the other hand taking semaphore is blocking operation and therefore transitions with *Taken?* and *TOut!* lead to the location *WaitProc* where the process waits for the processor. Notice also that all synchronisations *Take!* , *Taken?* , *TOut!* and *Give!* correspond to only one semaphore. (Another name of the synchronisations should be used for the next semaphore in the application.).

```
Fnc_Process1
{
    while (TRUE)
    {
        Comp1
        Result := Take (Sem, TimeOut)
        if (Result == TOut)
        {
          Comp4
          Yield()
        }
        else
        {
          Comp2
          Give (Sem)
        }
        Comp3
        Wait_End_of_Period
    }
}
```

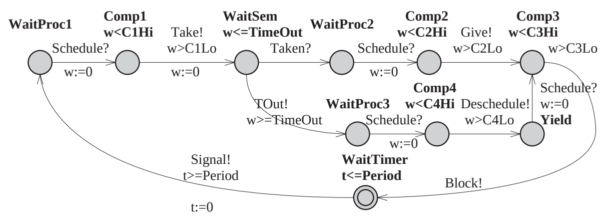


Figure 3-16. Model of process containing Take and Give one semaphore

Scheduler model for application with one semaphore is depicted in Figure 3-17. The scheduler of executed process is asked for taking the semaphore by synchronisation *Take?* . If the semaphore is empty (*Sem==0*), the processor is relinquished (*Free:=1*), *ID* of the process is written to the queue of the semaphore (*SemQ*) and the queue (FIFO or priority) is reordered by synchronisation *wSemQch!* . The scheduler and the process then wait the in location *WaitSem* until the semaphore is given by another process or until its time-out expires.

If the semaphore is not empty (*Sem>0*) its value is decreased and the synchronisation *Taken!* is immediately followed by synchronisation *Schedule!* to move the process to the next computation location. The processor is not relinquished in this case.

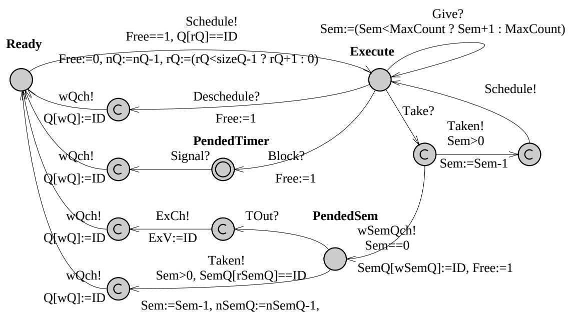


Figure 3-17. Scheduler model containing Take and Give of one semaphore (extension of Figure 3-11)

The queue of the processes waiting for the semaphore (*SemQ*) can be FIFO or priority queue. When the queue is priority queue, its elements (*ID* s of processes in this case) must be reordered according to priorities when the next process issues *Take* on empty semaphore. The only difference is the name of the queue (*SemQ* , *wSemQch* , *nSemQ* , *rSemQ* , *wSemQ*). Reordering is not necessary when FIFO is used. For compatibility with scheduler automaton in Figure 3-17 the automaton *wFifoQueue* depicted in Figure 3-18 is used.

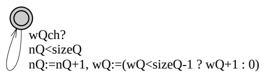


Figure 3-18. Automaton wFifoQueue providing writing to the FIFO queue

3.2.4.4. Conclusion and future work

The cooperative scheduling approach given in this chapter avoids preemption modelling by hybrid automata. Model of the application processes and cooperative scheduler is based on timed automata, for which model checking of TCTL property problem is decidable (opposite to hybrid automata). Interrupts and inter-process comunication - the most important aspect of real time embedded applications - are taken into consideration in proposed model. With respect to the processor utilisation and reaction time the system conceived in this chapter is not the most efficient one, but due to simplicity reasons many embedded applications are often based on similar cooperative scheduling mechanisms handling interrupts separately, so this approach is not just an academic idea.

Existing approaches for design and analysis of real-time applications, like Rate Monotonic Analysis (using preemptive scheduling based on priority assignment respecting the rate of periodic processes), use very elegant way of deciding whether the application is schedulable or not. Another approach based on timed automata with asynchronous processes [Fersman02] is suited for schedulability analysis of aperiodic processes. But both of these approaches do not consider internal process structure. As a consequence they provide too pessimistic results, especially when the application uses inter-process communication. Beside of that with respect to RMA it is needed to mention, that the model checking approach provides a room for verifying more complex properties (e.g. detection of deadlocks in communication, specification of buffer size,...). Model checking provides also room for modelling of more complex time behaviour of the controlled system, running truly in parallel with the control system (modelled as separate automaton).

Moreover this approach offers a frame work to combine verification of RTOS and CAN communication network (see CAN model by timed automata /Petri Nets component) with verification of fault-tolerant applications (see work package 6 - Fault Tolerant component). In order to reach full compatibility with RT Linux it is needed to study the Kernel intervals and to use different tools (e.g. Hytech) so that the preemptive can be modelled.

3.2.5. Tests

Not applicable.

3.2.6. Examples

3.2.6.1. Example of system with interrupt

Consider application depicted in Figure 3-19. It consists of two processes scheduled by cooperative scheduling (model of scheduler automaton is not depicted here because it is identical to automaton in Figure 3-17). First process *Proc_Period* is periodically executed with low priority (Figure 3-24). The second process *Proc_Int* with high priority is intended for handling external aperiodic events (Figure 3-23). It is waiting for semaphore that is given within interrupt service routine. Interrupt requests (*IRQ*) are generated by model of *Environment* (Figure 3-20). If the interrupt request is enabled (*EN>0*), hardware interrupt controller *InterruptCtrl* (Figure 3-21) generates interrupt (*INT*). Than it waits until interrupt service routine is finished (signaled by channel *iRet*). All other *IRQ* are ignored before *iRet* . Interrupt (*INT*) invokes *ISR_Server* (Figure 3-22). The integer variable *SC* represents capacity of the interrupt server. After each interrupt, *SC* is decreased by constant *C_ISR* representing computation time of interrupt service routine plus *ISR_Server* routine. If remaining *SC* is not sufficient for next interrupt (*SC-C_ISR<0*), the interrupt is disabled (*EN:=0*).

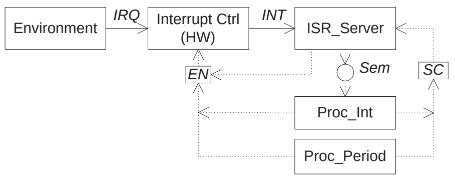


Figure 3-19. Interconnection of sample automata

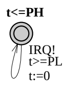


Figure 3-20. Model of Environment generating IRQ

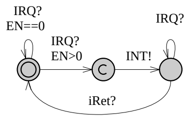


Figure 3-21. Model of hardware interrupt controller

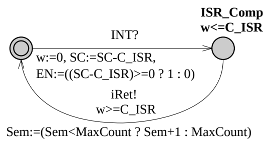

ISR_Comp
w<=C_ISR
INT?
w:=0, SC:=SC-C_ISR,
EN:=((SC-C_ISR)>=0 ? 1 : 0)
iRet!
w>=C_ISR
Sem:=(Sem<MaxCount ? Sem+1 : MaxCount)

Figure 3-22. *ISR_Server* model

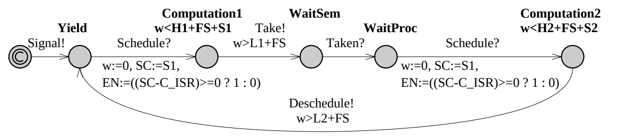


Figure 3-23. Model of high-priority process *Proc_Int*

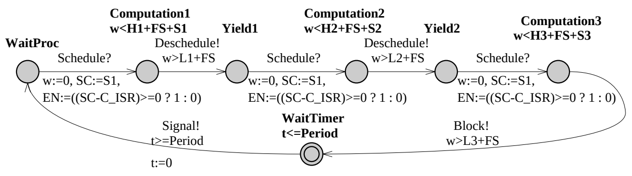


Figure 3-24. Model of low-priority periodic process *Proc_Period*

Bibliography

[Katoen99] Joost-Pieter Katoen, , and , 1998/1999, *Concepts, Algorithms, and Tools for Model Checking.: Lecture Notes of the Course "Mechanised Validation of Parallel Systems" (course number 10359)*.

[Clarke96] Edmund M. Clarke, Jeannette M. Wing, and , 1996, *Formal methods: state of the art and future directions.: Vol. 28, no 4, pp 623-643*.

[Alur93] R. Alur, C. Courcoubetis, and D. Dill, 1993, *Model-checking in dense real-time. Information and Computation: 104(1): 2-34*.

[Alur94] R. Alur, D. Dill, and , 1994, *A theory of timed automata: Theoretical Computer Science 126:183-235*.

[Alur91] R. Alur, T. Henzinger, and , 1991, *Logics and Models of Real Time: A Survey. In Real-Time: Theory in Practice: REX Workshop, LNCS 600, pp. 74-106*.

[David] A. David, , and , , *Uppaal2k: Small Tutorial. Documentation to the verification tool Uppaal2k: http://www.docs.uu.se/docs/rtmv/uppaal/*.

[Buttazzo97] Giorgio Buttazzo, , and , 1997, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications:* .

[Sha91] Lui Sha, M. Klein, and J. Goodenough, 1991, *Rate Monotonic Analysis for Real-Time Systems.: 129-155. Foundations of Real-Time Computing: Scheduling and Resource Management. Boston, MA*.

[Graham69] R. Graham, , and , 1969, *Bounds on multiprocessing timing anomalies: SIAM J. Appl. Math., 17 (1969), pp. 416-429*.

[Larsen95] Kim G. Larsen, Paul Pettersson, and Wang Yi, 1995, *Model-Checking for Real-Time Systems: In Proceedings of the 10th International Conference on Fundamentals of Computation Theory, Dresden, Germany, 22-25 August, 1995. LNCS 965, pages 62-88, Horst Reichel (Ed.)*.

[Liu2000] Liu, W.S. Jane, and , 2000, *Real-time systems: ISBN 0-13-099651-3*.

[Shaw89] A. Shaw, , and , 1989, *Reasoning about time in higher-level language software: IEEE Transactions on Software Engineering, vol. 15*.

[Corbett96] J. C. Corbett, , and , 1996, *Timing analysis of Ada tasking programs: IEEE Transactions on Software Engineering., 22(7), pp. 461-483.*

[Cassez2000] F. Cassez , K. Larsen, and , 2000, *The Impressive Power of Stopwatches: In Proceedings of CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 2000 CONCUR\'2000. LNCS 1877, p. 138 ff., .*

[Bouyer2000] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit, 2000, *Are Timed Automata Updatable ?: In Proc. 12th Int. Conf. Computer Aided Verification (CAV\'00), LNCS, Vol.1855, pp. 464-479.*

[Amnell01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas S. Hune, Bertrand Jeannet, Kim Larsen, M. Olivier Möller, Paul Pettersson, Carsten Weise, and Wang Yi, 2001, *UPPAAL - Now, Next, and Future: MOVEP'2k, LNCS Tutorial 2067.*

[Fersman02] Elena Fersman, Paul Pettersson, and Wang Yi, 2002, *Timed Automata with Asynchronous Processes: Schedulability and Decidability: In Proceedings of 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2002, Grenoble, France, April 8-12, 2002, pp.67-82, Springer-Verlag, 2002. Lecture Notes in Computer Science, Vol.2280.*

[Holzmann91] Gerard J. Holzmann, 1991, *Design and Validation of Computer Protocols: 512 pgs. ISBN 0-13-539925-4 hardcover (USA), ISBN 0-13-539834-7 paperback (international edition).*