WP7 – Communication Components

Deliverable D7.4 Communication components V2

D7.4_rep - Communication Components

by Jan Krakora, Pavel Pisa, Frantisek Vacek, Zdenek Sebek, Petr Smolik, and Zdenek Hanzalek

Published February 2004

Table of Contents

Preface i
1. OCERA Real-Time Ethernet 1
1.1. ORTE 1
1.1.1. Sumary 1
1.1.2. Description 1
1.1.3. API / Compatibility 1
1.1.3.1. Data types 1
enum SubscriptionMode 1
enum SubscriptionType 2
enum ORTERecvStatus 2
enum ORTESendStatus 3
struct ORTEIFProp 3
struct ORTEMulticastProp 3
struct ORTECDRStream 4
struct ORTETypeRegister 5
struct ORTEDomainBaseProp 5
struct ORTEDomainWireProp 6
struct ORTEPublProp 7
struct ORTESubsProp 8
struct ORTEAppInfo 8
struct ORTEPubInfo 9
struct ORTESubInfo 10
struct ORTEPublStatus 10
struct ORTESubsStatus 11
struct ORTERecvInfo 11
struct ORTESendInfo 12
struct ORTEDomainAppEvents 12
struct ORTETasksProp 14
struct ORTEDomainProp 14
1.1.3.2. Functions 15
IPAddressToString 15
StringToIPAddress 15
NtpTimeToStringMs 16
NtpTimeToStringUs 16
ORTEDomainStart 16
ORTEDomainPropDefaultGet 17
ORTEDomainInitEvents 17
ORTEDomainAppCreate 18
ORTEDomainAppDestroy 18
ORTEDomainAppSubscriptionPatternAdd 19
ORTEDomainAppSubscriptionPatternRemove 20
ORTEDomainAppSubscriptionPatternDestroy 20
ORTEDomainMgrCreate 21
ORTEDomainMgrDestroy 21
ORTEPublicationCreate 22
ORTEPublicationDestroy 22
ORTEPublicationPropertiesGet 23
ORTEPublicationPropertiesSet 23
ORTEPublicationGetStatus 24
ORTEPublicationSend 24
ORTESubscriptionCreate 24
ORTESubscriptionDestroy 25
ORTESubscriptionPropertiesGet 26

ORTESubscriptionPropertiesSet ... 26
ORTESubscriptionWaitForPublications ... 26
ORTESubscriptionGetStatus ... 27
ORTESubscriptionPull ... 27
ORTETypeRegisterAdd ... 28
ORTETypeRegisterDestroyAll ... 28
ORTEVerbositySetOptions ... 29
ORTEVerbositySetLogFile ... 29
ORTEInit ... 30
ORTEAppSendThread ... 30
ORTESleepMs ... 30
1.1.3.3. Macros ... 31
SeqNumberCmp ... 31
SeqNumberInc ... 31
SeqNumberAdd ... 32
SeqNumberDec ... 32
SeqNumberSub ... 32
NtpTimeCmp ... 33
NtpTimeAdd ... 33
NtpTimeSub ... 34
NtpTimeAssembFromMs ... 34
NtpTimeDisAssembToMs ... 35
NtpTimeAssembFromUs ... 35
NtpTimeDisAssembToUs ... 35
Domain2Port ... 36
Domain2PortMulticastUserdata ... 36
Domain2PortMulticastMetatraffic ... 37
1.1.4. Implementation issues ... 37
1.1.5. Tests ... 39
1.1.6. Examples ... 39
1.1.7. Installation instructions ... 41
1.2. Real Time Ethernet analyzer ... 41
1.2.1. Sumary ... 41
1.2.2. Description ... 42
1.2.3. API / Compatibility ... 42
1.2.4. Implementation issues ... 42
1.2.5. Tests ... 42
1.2.6. Examples ... 42
1.2.7. Installation instructions ... 43
2. Linux/RT-Linux CAN Driver (LinCAN) ... 44
2.1. LinCAN Summary ... 44
2.1.1. Summary ... 44
2.2. LinCAN Driver Description ... 45
2.2.1. Introduction ... 45
2.3. LinCAN Driver System Level API ... 46
2.3.1. Device Files and Message Structure ... 46
2.3.2. CAN Driver File Operations ... 47
open ... 47
close ... 47
read ... 48
write ... 48
struct canfilt_t ... 49
IOCTL CANQUE_FILTER ... 49
IOCTL CANQUE_FLUSH ... 50
2.4. LinCAN Driver Architecture ... 50

2.5. Driver History and Implementation Issues 53
2.6. LinCAN Driver Internals 54
2.6.1. Basic Driver Data Structures 55
struct canhardware_t 55
struct candevice_t 55
struct chip_t 56
struct msgobj_t 58
struct canuser_t 59
struct hwspecops_t 60
struct chipspecops_t 61
2.6.2. Board Support Functions 62
template_request_io 62
template_release_io 63
template_reset 63
template_init_hw_data 64
template_init_chip_data 64
template_init_obj_data 65
template_program_irq 66
template_write_register 66
template_read_register 67
2.6.3. Chip Support Functions 67
sja1000p_enable_configuration 68
sja1000p_disable_configuration 68
sja1000p_chip_config 68
sja1000p_extended_mask 69
sja1000p_baud_rate 69
sja1000p_read 70
sja1000p_pre_read_config 70
sja1000p_pre_write_config 71
sja1000p_send_msg 71
sja1000p_check_tx_stat 72
sja1000p_set_btregs 72
sja1000p_start_chip 73
sja1000p_stop_chip 73
sja1000p_remote_request 74
sja1000p_standard_mask 74
sja1000p_clear_objects 75
sja1000p_config_irqs 75
sja1000p_irq_write_handler 76
sja1000p_irq_handler 76
sja1000p_wakeup_tx 77
2.6.4. CAN Queues Common Structures and Functions 77
struct canque_slot_t 77
struct canque_fifo_t 78
canque_fifo_get_inslot 79
canque_fifo_put_inslot 79
canque_fifo_abort_inslot 80
canque_fifo_test_outslot 80
canque_fifo_free_outslot 80
canque_fifo_again_outslot 81
struct canque_edge_t 81
struct canque_ends_t 83
canque_notify_inends 84
canque_notify_outends 84
canque_notify_bothends 84

canque_activate_edge 85
canque_filtid2internal 85
canque_fifo_flush_slots 86
canque_fifo_init_slots 86
canque_get_inslot 86
canque_get_inslot4id 87
canque_put_inslot 88
canque_abort_inslot 88
canque_filter_msg2edges 89
canque_test_outslot 89
canque_free_outslot 90
canque_again_outslot 90
canque_set_filt 91
canque_flush 91
canqueue_ends_init_gen 92
canqueue_connect_edge 92
canqueue_disconnect_edge 93
canqueue_block_inlist 93
canqueue_block_outlist 93
canqueue_ends_kill_inlist 94
canqueue_ends_kill_outlist 94
2.6.5. CAN Queues Kernel Specific Functions 95
canqueue_notify_kern 95
canqueue_ends_init_kern 95
canque_get_inslot4id_wait_kern 95
canque_get_outslot_wait_kern 96
canque_sync_wait_kern 97
canque_fifo_init_kern 97
canque_fifo_done_kern 98
canque_new_edge_kern 98
canqueue_ends_dispose_kern 98
2.6.6. CAN Queues RT-Linux Specific Functions 99
canqueue_rtl2lin_check_and_pend 99
canque_get_inslot4id_wait_rtl 99
canque_get_outslot_wait_rtl 100
canque_sync_wait_rtl 101
canque_fifo_init_rtl 101
canque_fifo_done_rtl 101
canque_new_edge_rtl 102
canqueue_notify_rtl 102
canqueue_ends_init_rtl 103
canqueue_ends_dispose_rtl 103
canqueue_rtl_initialize 103
canqueue_rtl_done 104
2.6.7. CAN Queues CAN Chips Specific Functions 104
canqueue_notify_chip 104
canqueue_ends_init_chip 104
canqueue_ends_done_chip 105
2.6.8. CAN Boards and Chip Setup specific Functions 105
can_checked_malloc 105
can_checked_free 106
can_del_mem_list 106
can_request_io_region 106
can_release_io_region 107
can_request_mem_region 107

can_release_mem_region ... 108
can_base_addr_fixup ... 108
register_obj_struct ... 109
register_chip_struct ... 109
init_hw_struct ... 110
init_device_struct ... 110
init_chip_struct ... 111
init_obj_struct ... 111
init_hwspecops ... 112
init_chipspecops ... 112
can_chip_setup_irq ... 113
can_chip_free_irq ... 113
2.6.9. CAN Boards and Chip Finalization Functions ... 114
msgobj_done ... 114
canchip_done ... 114
candevice_done ... 114
canhardware_done ... 115
2.7. LinCAN Usage Information ... 115
2.7.1. Installation Prerequisites ... 115
2.7.2. Quick Installation Instructions ... 116
2.7.3. Installation instructions ... 116
2.7.4. Simple Utilities ... 118
readburst ... 119
sendburst ... 120
3. CAN/CANopen ... 122
3.1. Virtual CAN API (VCA) ... 122
3.1.1. Summary ... 122
3.1.2. Description ... 122
3.1.3. API / Compatibility ... 122
3.1.3.1. VCA API ... 122
struct canmsg_t ... 122
vca_h2log ... 123
vca_open_handle ... 123
vca_close_handle ... 124
vca_send_msg_seq ... 124
vca_rec_msg_seq ... 125
vca_wait ... 125
vca_log ... 126
vca_log_redir ... 126
3.1.4. Implementation issues ... 127
3.1.5. Tests ... 128
3.1.6. Examples ... 128
3.1.7. Installation instructions ... 128
3.2. CAN device ... 128
3.2.1. Summary ... 128
3.2.2. Description ... 128
3.2.3. API / Compatibility ... 129
3.2.3.1. PDO processor API ... 129
struct vcapdo_mapping_t ... 129
struct vcapdolst_object_t ... 129
struct vcapdolst_root_t ... 130
struct vcaPDOProcessor_t ... 131
vcaPDOProcessor_init ... 131
vcaPDOProcessor_destroy ... 132
vcaPDOProcessor_setOD ... 132

vcaPDOProcessor_createPDOList 132
_vcaPDOProcessor_disconnectDinfoLinks 133
vcaPDOProcessor_makeDinfoLinks 133
vcaPDOProcessor_processMsg 134
3.2.3.2. SDO FSM API 134
struct vcasdo_fsm_t 134
vcasdo_fsm_upload1 135
vcasdo_fsm_download1 135
vcasdo_read_multiplexor 136
vcasdo_error_msg 136
vcasdo_init_fsm 137
vcasdo_destroy_fsm 137
vcasdo_fsm_idle 137
vcasdo_fsm_run 138
vcasdo_fsm_abort 138
vcasdo_fsm_upload 138
vcasdo_fsm_download 139
vcasdo_fsm_taste_msg 140
vcasdo_abort_msg 140
3.2.3.3. Object Dictionary API 140
struct vcaod_root_t 140
struct vcaod_object_t 141
_vcaod_find_object 142
vcaod_get_value 143
vcaod_set_value 143
vcaod_od_free 144
vcaod_load_eds 144
vcaod_dump_od 145
vcaod_get_dinfo_ref 145
3.2.3.4. canslave command line parameters 146
3.2.3.5. CANmaster command line parameters 146
3.2.4. Implementation issues 146
3.2.4.1. Architecture overview 147
3.2.4.2. CANopen slave 148
3.2.4.3. CANopen master 148
3.2.5. Tests 149
3.2.6. Examples 149
3.2.7. Installation instructions 149
3.3. CAN monitor 149
3.3.1. Summary 149
3.3.2. Description 149
3.3.2.1. canmond - CAN/CANopen proxy 149
3.3.2.2. testclient 150
3.3.2.3. CanMonitor 150
3.3.3. API / Compatibility 151
3.3.3.1. canmond 151
3.3.3.2. testclient 152
3.3.3.3. CanMonitor 152
3.3.4. Implementation issues 152
3.3.4.1. canmond 152
3.3.5. Tests 154
3.3.6. Examples 154
3.3.6.1. Example 1 - connecting to real CANopen device 154
3.3.6.2. Example 2 155
3.3.7. Installation instructions 155

4. Verifications 157
4.1. CAN model by timed automata 157
4.1.1. Sumary 157
4.1.2. Description 157
4.1.2.1. Problem statement 157
4.1.2.2. CAN bus description 158
4.1.2.2.1. Real-time data transmission 158
4.1.2.2.2. Message frame formats 159
4.1.2.2.3. Detecting and signalling errors 159
4.1.3. API/Compatibility 160
4.1.4. Implementation issues 160
4.1.4.1. Bit-wise arbitration model 161
4.1.4.2. Transceiver model 161
4.1.4.3. Bus model 162
4.1.5. Tests 162
4.1.6. Examples 162
4.1.6.1. Case study 1 - Application process model 163
4.1.6.1.1. Comparison with traditional approach 164
4.1.6.2. Case study 2 - Anti-lock Braking System 165
4.1.6.2.1. Verification 170
4.1.7. Installation instructions 170
Bibliography 171
4.2. Verification of cooperative scheduling and interrupt handlers 171
4.2.1. Sumary 171
4.2.2. Description 172
4.2.2.1. Abstract 172
4.2.2.2. Introduction 172
4.2.3. API/Compatibility 175
4.2.4. Implementation issuses 175
4.2.4.1. Cooperative scheduling 175
4.2.4.2. Interrupts 177
4.2.4.3. Inter process communication primitives 179
4.2.4.3.1. Semaphore 179
4.2.4.4. Conclusion and future work 180
4.2.5. Implementation issues 181
4.2.6. Tests 181
4.2.7. Examples 181
4.2.7.1. Example of system with interrupt 181
Bibliography 182

List of Tables

4-1. Process parameters table 164
4-2. Results of the experiment related to property 4 and 5 165
4-3. Results of the experiment related to property 4 and 5 with the operating system delay 165
4-4. Processor 1 RTOS system parameters 170
4-5. Processor 2 RTOS system parameters 170
4-6. Message Parameters 170

List of Figures

1-1. Communication among network objects .... ??
1-2. Screenshot 42
2-1. LinCAN architecture 51
2-2. LinCAN message FIFO implementation 51
2-3. LinCAN driver message flow graph edges 52
2-4. CAN hardware model in the LinCAN driver 52
3-1. Hard real time CAN driver usage example 127
3-2. Soft real time CAN driver usage example 127
3-3. RT-CANopen device architecture 147
3-4. Connecting canmond 150
3-5. CAN monitor CAN messages window 150
3-6. The Object Dictionary tree view 150
3-7. The CanMonitor configuration dialog 151
3-8. CanMonitor testing 154
4-1. Real time control system structure with denotation of computation/communication times 158
4-2. CAN message frame format 161
4-3. Arbitration model (in UPPAAL like notation) 161
4-4. Transceiver model 162
4-5. Bus model 162
4-6. Case study system configuration 163
4-7. Periodic application process model 163
4-8. Sporadic application process model 163
4-9. Worst-case response time equation 165
4-10. Worst-case queueing delay equation 165
4-11. Structure of the distributed system - ABS control 166
4-12. Two state brake pedal 166
4-13. Brake pedal model 166
4-14. Processor 1 structure 166
4-15. Timed automaton model of Task1 167
4-16. Timed automaton model of Task2 167
4-17. Processor 2 structure 168
4-18. ABS controller state diagram 168
4-19. Timed automaton model of ABSTask algorithm 169
4-20. Timed automaton model of brakes 169
4-21. Timed automaton model of acceleration sensor 169
4-22. State transition diagram of the process in the multitasking operating system 174
4-23. Example of monoprocessor scheduling anomaly 175
4-24. Model of the application process executed under cooperative scheduling policy 175
4-25. Synchronisation of cooperative scheduler with application processes 176
4-26. One automaton (*Schi*) of the cooperative scheduler in Figure 4-25 176
4-27. System architecture with interrupt server 178
4-28. *Interrupt server* routines 178

4-29. *Computation* location considering interrupts 178
4-30. Time diagram of ISR execution within *interrupt server* 178
4-31. Model of process containing Take and Give one semaphore 180
4-32. Scheduler model containing Take and Give of one semaphore (extension of Figure 4-26) 180
4-33. Automaton wFifoQueue providing writing to the FIFO queue........................... 180
4-34. Interconnection of sample automata 181
4-35. Model of Environment generating IRQ 182
4-36. Model of hardware interrupt controller 182
4-37. *ISR_Server* model 182
4-38. Model of high-priority process *Proc_Int* 182
4-39. Model of low-priority periodic process *Proc_Period* 182

Preface

This document presents elaborated version of communication component design report. It is composed of three chapters. The first and second chapters deal with design of communication stacks - RT Ethernet or CAN/CANopen respectively including analysis tools. The third chapter presents Basic features of verification methodology of distributed systems based on standard verification tools.

Chapter 1. OCERA Real-Time Ethernet

1.1. 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.

1.1.1. 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

1.1.2. 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.

1.1.3. API / Compatibility

1.1.3.1. Data types

enum SubscriptionMode

Name

`enum SubscriptionMode` — mode of subscription

Synopsis

```
enum SubscriptionMode {
  PULLED,
  IMMEDIATE
};
```

Constants

PULLED
: polled

IMMEDIATE
: using callback function

Description

Specifies whether user application will poll for data or whether a callback function will be called by ORTE middleware when new data will be available.

enum SubscriptionType

Name

`enum SubscriptionType` — type of subcsription

Synopsis

```
enum SubscriptionType {
  BEST_EFFORTS,
  STRICT_RELIABLE
};
```

Constants

BEST_EFFORTS
: best effort subscription

STRICT_RELIABLE
: strict reliable subscription.

Description

Specifies which mode will be used for this subscription.

enum ORTERecvStatus

Name

`enum ORTERecvStatus` — status of a subscription

Synopsis

```
enum ORTERecvStatus {
  NEW_DATA,
  DEADLINE
};
```

Constants

NEW_DATA
: new data has arrived

DEADLINE
: deadline has occurred

Description

Specifies which event has occured in the subscription object.

enum ORTESendStatus

Name

`enum ORTESendStatus` — status of a publication

Synopsis

```
enum ORTESendStatus {
  NEED_DATA,
  CQL
};
```

Constants

NEED_DATA
: need new data (set when callback function specified for publciation is beeing called)

CQL
: transmit queue has been filled up to critical level.

Description

Specifies which event has occured in the publication object. Critical level of transmit queue is specified as one of publication properties (ORTEPublProp.criticalQueueLevel).

struct ORTEIFProp

Name

`struct ORTEIFProp` — interface flags

Synopsis

```
struct ORTEIFProp {
  int32_t ifFlags;
  IPAddress ipAddress;
};
```

Members

ifFlags
: flags

ipAddress
: IP address

Description

Flags for network interface.

struct ORTEMulticastProp

Name

`struct ORTEMulticastProp` — properties for ORTE multicast (not supported yet)

Synopsis

```
struct ORTEMulticastProp {
  Boolean enabled;
  unsigned char  ttl;
  Boolean loopBackEnabled;
  IPAddress ipAddress;
};
```

Members

enabled
: ORTE_TRUE if multicast enabled otherwise ORTE_FALSE

ttl
: time-to-live (TTL) for sent datagrams

loopBackEnabled
: ORTE_TRUE if data should be received by sender itself otherwise ORTE_FALSE

ipAddress
: desired multicast IP address

Description

Properties for ORTE multicast subsystem which is not fully supported yet. Multicast IP address is assigned by the ORTE middleware itself.

struct ORTECDRStream

Name

`struct ORTECDRStream` — used for serialization

Synopsis

```
struct ORTECDRStream {
  char * buffer;
  char * bufferPtr;
  Boolean needByteSwap;
  int length;
};
```

Members

buffer
: buffer for data

bufferPtr
: current position within buffer

needByteSwap
: ORTE_TRUE if it is necessary to swap byte ordering otherwise ORTE_FALSE

length
: buffer length

Description

Struct *ORTECDRStream* is used by serialization and deserialization functions.

struct ORTETypeRegister

Name

`struct ORTETypeRegister` — registered data type

Synopsis

```
struct ORTETypeRegister {
  const char            * typeName;
  ORTETypeSerialize serialize;
  ORTETypeDeserialize deserialize;
  unsigned int          getMaxSize;
};
```

Members

typeName
: name of data type

serialize
: pointer to serialization function

deserialize
: pointer to deserialization function

getMaxSize
: max data type length in bytes

Description

Contains description of registered data type. See *ORTETypeRegisterAdd* function for details.

struct ORTEDomainBaseProp

Name

`struct ORTEDomainBaseProp` — base properties of a domain

Synopsis

```
struct ORTEDomainBaseProp {
  NtpTime expirationTime;
  NtpTime refreshPeriod;
  NtpTime purgeTime;
  NtpTime repeatAnnounceTime;
  NtpTime repeatActiveQueryTime;
  NtpTime delayResponceTimeACKMin;
  NtpTime delayResponceTimeACKMax;
  unsigned int          HBMaxRetries;
  unsigned int          ACKMaxRetries;
  NtpTime maxBlockTime;
};
```

Members

expirationTime
: specifies how long is this application taken as alive in other applications/managers (default 180s)

refreshPeriod
: how often an application refresh itself to its manager or manager to other managers (default 60s)

purgeTime
: how often the local database should be cleaned from invalid (expired) objects (default 60s)

repeatAnnounceTime
: This is the period with which the CSTWriter will announce its existence and/or the availability of new CSChanges to the CSTReader. This period determines how quickly the protocol recovers when an announcement of data is lost.

repeatActiveQueryTime
: ???

delayResponceTimeACKMin
: minimum time the CSTWriter waits before responding to an incoming message.

delayResponceTimeACKMax
: maximum time the CSTWriter waits before responding to an incoming message.

HBMaxRetries
: how many times a HB message is retransmitted if no response has been received until timeout

ACKMaxRetries
: how many times an ACK message is retransmitted if no response has been received until timeout

maxBlockTime
: timeout for send functions if sending queue is full (default 30s)

struct ORTEDomainWireProp

Name

`struct ORTEDomainWireProp` — wire properties of a message

Synopsis

```
struct ORTEDomainWireProp {
  unsigned int          metaBytesPerPacket;
  unsigned int          metaBytesPerFastPacket;
  unsigned int          metabitsPerACKBitmap;
  unsigned int          userMaxSerDeserSize;
};
```

Members

metaBytesPerPacket
: maximum number of bytes in single frame (default 1500B)

metaBytesPerFastPacket
: maximum number of bytes in single frame if transmitting queue has reached *`criticalQueueLevel`* level (see *`ORTEPublProp`* struct)

metabitsPerACKBitmap
: not supported yet

userMaxSerDeserSize
: maximum number of user data in frame (default 1500B)

struct ORTEPublProp

Name

`struct ORTEPublProp` — properties of a publication

Synopsis

```
struct ORTEPublProp {
  PathName topic;
  TypeName typeName;
  TypeChecksum typeChecksum;
  Boolean expectsAck;
  NtpTime persistence;
  u_int32_t reliabilityOffered;
  u_int32_t sendQueueSize;
  int32_t strength;
  u_int32_t criticalQueueLevel;
  NtpTime HBNornalRate;
  NtpTime HBCQLRate;
  unsigned int          HBMaxRetries;
  NtpTime maxBlockTime;
};
```

Members

topic
: the name of the information in the Network that is published or subscribed to

typeName
: the name of the type of this data

typeChecksum
: a checksum that identifies the CDR-representation of the data

expectsAck
: indicates wherther publication expects to receive ACKs to its messages

persistence
: indicates how long the issue is valid

reliabilityOffered
: reliability policy as offered by the publication

sendQueueSize
: size of transmitting queue

strength
: precedence of the issue sent by the publication

criticalQueueLevel
: treshold for transmitting queue content length indicating the queue can became full immediately

HBNornalRate
: how often send HBs to subscription objects

HBCQLRate
: how often send HBs to subscription objects if transmittiong queue has reached *`criticalQueueLevel`*

HBMaxRetries
: how many times retransmit HBs if no replay from target object has not been received

maxBlockTime
: unsupported

struct ORTESubsProp

Name

`struct ORTESubsProp` — properties of a subscription

Synopsis

```
struct ORTESubsProp {
  PathName topic;
  TypeName typeName;
  TypeChecksum typeChecksum;
  NtpTime minimumSeparation;
  u_int32_t recvQueueSize;
  u_int32_t reliabilityRequested;
  //additional parametersNtpTime                deadline;
  u_int32_t mode;
};
```

Members

topic
: the name of the information in the Network that is published or subscribed to

typeName
: the name of the type of this data

typeChecksum
: a checksum that identifies the CDR-representation of the data

minimumSeparation
: minimum time between two consecutive issues received by the subscription

recvQueueSize
: size of receiving queue

reliabilityRequested
: reliability policy requested by the subscription

deadline
: deadline for subscription, a callback function (see *`ORTESubscriptionCreate`*) will be called if no data were received within this period of time

mode
: mode of subscription (strict reliable/best effort), see *`SubscriptionType`* enum for values

struct ORTEAppInfo

Name

`struct ORTEAppInfo` —

Synopsis

```
struct ORTEAppInfo {
  HostId hostId;
  AppId appId;
  IPAddress * unicastIPAddressList;
  unsigned char          unicastIPAddressCount;
  IPAddress * metatrafficMulticastIPAddressList;
  unsigned char          metatrafficMulticastIPAddressCount;
  Port metatrafficUnicastPort;
  Port userdataUnicastPort;
  VendorId vendorId;
  ProtocolVersion protocolVersion;
};
```

Members

hostId

hostId of application

appId

appId of application

unicastIPAddressList

unicast IP addresses of the host on which the application runs (there can be multiple addresses on a multi-NIC host)

unicastIPAddressCount

number of IPaddresses in *unicastIPAddressList*

metatrafficMulticastIPAddressList

for the purposes of meta-traffic, an application can also accept Messages on this set of multicast addresses

metatrafficMulticastIPAddressCount

number of IPaddresses in *metatrafficMulticastIPAddressList*

metatrafficUnicastPort

UDP port used for metatraffic communication

userdataUnicastPort

UDP port used for metatraffic communication

vendorId

identifies the vendor of the middleware implementing the RTPS protocol and allows this vendor to add specific extensions to the protocol

protocolVersion

describes the protocol version

struct ORTEPubInfo

Name

`struct ORTEPubInfo` — information about publication

Synopsis

```
struct ORTEPubInfo {
  const char           * topic;
  const char           * type;
  ObjectId objectId;
};
```

Members

topic
: the name of the information in the Network that is published or subscribed to

type
: the name of the type of this data

objectId
: object providing this publication

struct ORTESubInfo

Name

`struct ORTESubInfo` — information about subscription

Synopsis

```
struct ORTESubInfo {
  const char           * topic;
  const char           * type;
  ObjectId objectId;
};
```

Members

topic
: the name of the information in the Network that is published or subscribed to

type
: the name of the type of this data

objectId
: object with this subscription

struct ORTEPublStatus

Name

`struct ORTEPublStatus` — status of a publication

Synopsis

```
struct ORTEPublStatus {
  unsigned int          strict;
  unsigned int          bestEffort;
  unsigned int          issues;
};
```

Members

strict
: count of unreliable subscription (strict) connected on responsible subscription

bestEffort
: count of reliable subscription (best effort) connected on responsible subscription

issues
: number of messages in transmitting queue

struct ORTESubsStatus

Name

`struct ORTESubsStatus` — status of a subscription

Synopsis

```
struct ORTESubsStatus {
  unsigned int          strict;
  unsigned int          bestEffort;
  unsigned int          issues;
};
```

Members

strict
: count of unreliable publications (strict) connected to responsible subscription

bestEffort
: count of reliable publications (best effort) connected to responsible subscription

issues
: number of messages in receiving queue

struct ORTERecvInfo

Name

`struct ORTERecvInfo` — description of received data

Synopsis

```
struct ORTERecvInfo {
  ORTERecvStatus status;
  const char           * topic;
  const char           * type;
  GUID_RTPS senderGUID;
  NtpTime localTimeReceived;
  NtpTime remoteTimePublished;
  SequenceNumber sn;
};
```

Members

status
: status of this event

topic
: the name of the information

type
: the name of the type of this data

senderGUID
: GUID of object who sent this information

localTimeReceived
: local timestamp when data were received

remoteTimePublished
: remote timestam when data were published

sn
: sequencial number of data

struct ORTESendInfo

Name

`struct ORTESendInfo` — description of sending data

Synopsis

```
struct ORTESendInfo {
  ORTESendStatus status;
  const char           * topic;
  const char           * type;
  GUID_RTPS senderGUID;
  SequenceNumber sn;
};
```

Members

status
: status of this event

topic
: the name of the information

type
: the name of the type of this information

senderGUID
: GUID of object who sent this information

sn
: sequencial number of information

struct ORTEDomainAppEvents

Name

`struct ORTEDomainAppEvents` — Domain event handlers of an application

Synopsis

```
struct ORTEDomainAppEvents {
  ORTEOnMgrNew onMgrNew;
  void * onMgrNewParam;
  ORTEOnMgrDelete onMgrDelete;
  void * onMgrDeleteParam;
  ORTEOnAppRemoteNew onAppRemoteNew;
  void * onAppRemoteNewParam;
  ORTEOnAppDelete onAppDelete;
  void * onAppDeleteParam;
  ORTEOnPubRemote onPubRemoteNew;
  void * onPubRemoteNewParam;
  ORTEOnPubRemote onPubRemoteChanged;
```

```
    void * onPubRemoteChangedParam;
    ORTEOnPubDelete onPubDelete;
    void * onPubDeleteParam;
    ORTEOnSubRemote onSubRemoteNew;
    void * onSubRemoteNewParam;
    ORTEOnSubRemote onSubRemoteChanged;
    void * onSubRemoteChangedParam;
    ORTEOnSubDelete onSubDelete;
    void * onSubDeleteParam;
  };
```

Members

onMgrNew

    new manager has been created

onMgrNewParam

    user parameters for *onMgrNew* handler

onMgrDelete

    manager has been deleted

onMgrDeleteParam

    user parameters for *onMgrDelete* handler

onAppRemoteNew

    new remote application has been registered

onAppRemoteNewParam

    user parameters for *onAppRemoteNew* handler

onAppDelete

    an application has been removed

onAppDeleteParam

    user parameters for *onAppDelete* handler

onPubRemoteNew

    new remote publication has been registered

onPubRemoteNewParam

    user parameters for *onPubRemoteNew* handler

onPubRemoteChanged

    a remote publication’s parameters has been changed

onPubRemoteChangedParam

    user parameters for *onPubRemoteChanged* handler

onPubDelete

    a publication has been deleted

onPubDeleteParam

    user parameters for *onPubDelete* handler

onSubRemoteNew

    a new remote subscription has been registered

onSubRemoteNewParam

    user parameters for *onSubRemoteNew* handler

onSubRemoteChanged

    a remote subscription’s parameters has been changed

onSubRemoteChangedParam

    user parameters for *onSubRemoteChanged* handler

onSubDelete

    a publication has been deleted

onSubDeleteParam

    user parameters for *onSubDelete* handler

Description

Prototypes of events handler fucntions can be found in file typedefs_api.h.

struct ORTETasksProp

Name

`struct ORTETasksProp` — ORTE task properties, not supported

Synopsis

```
struct ORTETasksProp {
  Boolean realTimeEnabled;
  int smtStackSize;
  int smtPriority;
  int rmtStackSize;
  int rmtPriority;
};
```

Members

realTimeEnabled
: not supported

smtStackSize
: not supported

smtPriority
: not supported

rmtStackSize
: not supported

rmtPriority
: not supported

struct ORTEDomainProp

Name

`struct ORTEDomainProp` — domain properties

Synopsis

```
struct ORTEDomainProp {
  ORTETasksProp tasksProp;
  ORTEIFProp * IFProp;
  //interface propertiesunsigned char          IFCount;
  //count of interfacesORTEDomainBaseProp     baseProp;
  ORTEDomainWireProp wireProp;
  ORTEMulticastProp multicast;
  //multicast properiesORTEPublProp           publPropDefault;
  //default properties for a Publ/SubORTESubsProp           subsPropDefault;
  char * mgrs;
  //managerschar * keys;
  //keysIPAddress appLocalManager;
  //applicationschar * version;
  //string product versionint                    recvBuffSize;
  int sendBuffSize;
};
```

Members

tasksProp
: task properties

IFProp
: properties of network interfaces

IFCount
: number of network interfaces

baseProp
: base properties (see *ORTEDomainBaseProp* for details)

wireProp
: wire properties (see *ORTEDomainWireProp* for details)

multicast
: multicast properties (see *ORTEMulticastProp* for details)

publPropDefault
: default properties of publiciations (see *ORTEPublProp* for details)

subsPropDefault
: default properties of subscriptions (see *ORTESubsProp* for details)

mgrs
: list of known managers

keys
: access keys for managers

appLocalManager
: IP address of local manager (default localhost)

version
: string product version

recvBuffSize
: receiving queue length

sendBuffSize
: transmitting queue length

1.1.3.2. Functions

IPAddressToString

Name

`IPAddressToString` — converts IP address IPAddress to its string representation

Synopsis

```
char* IPAddressToString (IPAddress ipAddress, char * buff);
```

Arguments

ipAddress
: source IP address

buff
: output buffer

StringToIPAddress

Name

`StringToIPAddress` — converts IP address from string into IPAddress

Synopsis

```
IPAddress StringToIPAddress (const char * string);
```

Arguments

string
: source string

NtpTimeToStringMs

Name

`NtpTimeToStringMs` — converts NtpTime to its text representation in miliseconds

Synopsis

```
char * NtpTimeToStringMs (NtpTime time, char * buff);
```

Arguments

time
: time given in NtpTime structure

buff
: output buffer

NtpTimeToStringUs

Name

`NtpTimeToStringUs` — converts NtpTime to its text representation in microseconds

Synopsis

```
char * NtpTimeToStringUs (NtpTime time, char * buff);
```

Arguments

time
: time given in NtpTime structure

buff
: output buffer

ORTEDomainStart

Name

`ORTEDomainStart` — start specific threads

Synopsis

```
void ORTEDomainStart (ORTEDomain * d, Boolean recvMetatrafficThread, Boolean recvUserDataThread,
Boolean sendThread);
```

Arguments

d

domain object handle

recvMetatrafficThread

specifies whether recvMetatrafficThread should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

recvUserDataThread

specifies whether recvUserDataThread should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

sendThread

specifies whether sendThread should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

Description

Functions *ORTEDomainAppCreate* and *ORTEDomainMgrCreate* provide facility to create an object with its threads suspended. Use function *ORTEDomainStart* to resume those suspended threads.

ORTEDomainPropDefaultGet

Name

`ORTEDomainPropDefaultGet` — returns default properties of a domain

Synopsis

```
Boolean ORTEDomainPropDefaultGet (ORTEDomainProp * prop);
```

Arguments

prop

pointer to struct ORTEDomainProp

Description

Structure ORTEDomainProp referenced by *prop* will be filled by its default values. Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainInitEvents

Name

`ORTEDomainInitEvents` — initializes list of events

Synopsis

```
Boolean ORTEDomainInitEvents (ORTEDomainAppEvents * events);
```

Arguments

events

pointer to struct ORTEDomainAppEvents

Description

Initializes structure pointed by *events*. Every member is set to NULL. Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainAppCreate

Name

`ORTEDomainAppCreate` — creates an application object within given domain

Synopsis

```
ORTEDomain * ORTEDomainAppCreate (int domain, ORTEDomainProp * prop, ORTEDomainAppEvents * events,
Boolean suspended);
```

Arguments

domain

given domain

prop

properties of application

events

events associated with application or NULL

suspended

specifies whether threads of this application should be started as well (ORTE_FALSE) or stay suspended (ORTE_TRUE). See *ORTEDomainStart* for details how to resume suspended threads

Description

Creates new Application object and sets its properties and events. Return handle to created object or NULL in case of any error.

ORTEDomainAppDestroy

Name

`ORTEDomainAppDestroy` — destroy Application object

Synopsis

```
Boolean ORTEDomainAppDestroy (ORTEDomain * d);
```

Arguments

d

domain

Description

Destroys all application objects in specified domain. Returns ORTE_TRUE or ORTE_FALSE in case of any error.

ORTEDomainAppSubscriptionPatternAdd

Name

`ORTEDomainAppSubscriptionPatternAdd` — create pattern-based subscription

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternAdd (ORTEDomain * d, const char * topic, const char * type,
ORTESubscriptionPatternCallBack subscriptionCallBack, void * param);
```

Arguments

d

domain object

topic

pattern for topic

type

pattern for type

subscriptionCallBack

pointer to callback function which will be called whenever any data are received through this subscription

param

user params for callback function

Description

This function is intended to be used in application interesded in more published data from possibly more remote applications, which should be received through single subscription. These different publications are specified by pattern given to `topic` and `type` parameters.

For example suppose there are publications of topics like *temperatureEngine1*, *temperatureEngine2*, *temperatureEngine1Backup* and *temperatureEngine2Backup* somewhere on our network. We can subscribe to each of Engine1 temperations by creating single subscription with pattern for topic set to “temperatureEngine1*”. Or, if we are interested only in values from backup measurements, we can use pattern “*Backup”.

Syntax for patterns is the same as syntax for *fnmatch* function, which is employed for pattern recognition.

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainAppSubscriptionPatternRemove

Name

`ORTEDomainAppSubscriptionPatternRemove` — remove subscription pattern

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternRemove (ORTEDomain * d, const char * topic, const char
* type);
```

Arguments

d

domain handle

topic

pattern to be removed

type

pattern to be removed

Description

Removes subscritions created by *ORTEDomainAppSubscriptionPatternAdd*. Patterns for *type* and *topic* must be exactly the same strings as when *ORTEDomainAppSubscriptionPattern* function was called.

Returns ORTE_TRUE if successful or ORTE_FALSE if none matching record was found

ORTEDomainAppSubscriptionPatternDestroy

Name

`ORTEDomainAppSubscriptionPatternDestroy` — destroys all subscription patterns

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternDestroy (ORTEDomain * d);
```

Arguments

d

domain handle

Description

Destroys all subscription patterns which were specified previously by *ORTEDomainAppSubscriptionPa* function.
Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainMgrCreate

Name

`ORTEDomainMgrCreate` — create manager object in given domain

Synopsis

```
ORTEDomain * ORTEDomainMgrCreate (int domain, ORTEDomainProp * prop, ORTEDomainAppEvents * events,
Boolean suspended);
```

Arguments

domain

-- undescribed --

prop

desired manager's properties

events

manager's event handlers or NULL

suspended

specifies whether threads of this manager should be started as well (ORTE_FALSE) or stay suspended (ORTE_TRUE). See *ORTEDomainStart* for details how to resume suspended threads

Description

Creates new manager object and sets its properties and events. Return handle to created object or NULL in case of any error.

ORTEDomainMgrDestroy

Name

`ORTEDomainMgrDestroy` — destroy manager object

Synopsis

```
Boolean ORTEDomainMgrDestroy (ORTEDomain * d);
```

Arguments

d

manager object to be destroyed

Description

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEPublicationCreate

Name

`ORTEPublicationCreate` — creates new publication

Synopsis

```
ORTEPublication * ORTEPublicationCreate (ORTEDomain * d, const char * topic, const char * typeName,
void * instance, NtpTime * persistence, int strength, ORTESendCallBack sendCallBack, void * sendCallBackParam,
NtpTime * sendCallBackDelay);
```

Arguments

d
: pointer to application object

topic
: name of topic

typeName
: data type description

instance
: output buffer where application stores data for publication

persistence
: persistence of publication

strength
: strength of publication

sendCallBack
: pointer to callback function

sendCallBackParam
: user parameters for callback function

sendCallBackDelay
: periode for timer which issues callback function

Description

Creates new publication object with specified parameters. The *sendCallBack* function is called periodically with *sendCallBackDelay* periode. In strict reliable mode the *sendCallBack* function will be called only if there is enough room in transmitting queue in order to prevent any data loss. The *sendCallBack* function should prepare data to be published by this publication and place them into *instance* buffer.

Returns handle to publication object.

ORTEPublicationDestroy

Name

`ORTEPublicationDestroy` — removes a publication

Synopsis

```
int ORTEPublicationDestroy (ORTEPublication * cstWriter);
```

Arguments

cstWriter
: handle to publication to be removed

Description

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

ORTEPublicationPropertiesGet

Name

`ORTEPublicationPropertiesGet` — read properties of a publication

Synopsis

```
ORTEPublicationPropertiesGet (ORTEPublication * cstWriter, ORTEPublProp * pp);
```

Arguments

cstWriter
: pointer to cstWriter object which provides this publication

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

Description

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

ORTEPublicationPropertiesSet

Name

`ORTEPublicationPropertiesSet` — set properties of a publication

Synopsis

```
int ORTEPublicationPropertiesSet (ORTEPublication * cstWriter, ORTEPublProp * pp);
```

Arguments

cstWriter
: pointer to cstWriter object which provides this publication

pp
: pointer to ORTEPublProp structure containing values of publication's properties

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstWriter` is not valid publication handle.

ORTEPublicationGetStatus

Name

`ORTEPublicationGetStatus` — removes a publication

Synopsis

```
int ORTEPublicationGetStatus (ORTEPublication * cstWriter, ORTEPublStatus * status);
```

Arguments

cstWriter
: pointer to cstWriter object which provides this publication

status
: pointer to ORTEPublStatus structure

Description

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

ORTEPublicationSend

Name

`ORTEPublicationSend` — force publication object to issue new data

Synopsis

```
int ORTEPublicationSend (ORTEPublication * cstWriter);
```

Arguments

cstWriter
: publication object

Description

Returns ORTE_OK if successful.

ORTESubscriptionCreate

Name

`ORTESubscriptionCreate` — adds a new subscription

Synopsis

```
ORTESubscription * ORTESubscriptionCreate (ORTEDomain * d, SubscriptionMode mode, SubscriptionType
sType, const char * topic, const char * typeName, void * instance, NtpTime * deadline, NtpTime
* minimumSeparation, ORTERecvCallBack recvCallBack, void * recvCallBackParam);
```

Arguments

d
: pointer to ORTEDomain object where this subscription will be created

mode
: see enum SubscriptionMode

sType
: see enum SubscriptionType

topic
: name of topic

typeName
: name of data type

instance
: pointer to output buffer

deadline
: deadline

minimumSeparation
: minimum time interval between two publications sent by Publisher as requested by Subscriber (strict - minumSep musi byt 0)

recvCallBack
: callback function called when new Subscription has been received or if any change of subscription's status occures

recvCallBackParam
: user parameters for *recvCallBack*

Description

Returns handle to Subscription object.

ORTESubscriptionDestroy

Name

ORTESubscriptionDestroy — removes a subscription

Synopsis

```
int ORTESubscriptionDestroy (ORTESubscription * cstReader);
```

Arguments

cstReader
: handle to subscriotion to be removed

Description

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

ORTESubscriptionPropertiesGet

Name

`ORTESubscriptionPropertiesGet` — get properties of a subscription

Synopsis

```
int ORTESubscriptionPropertiesGet (ORTESubscription * cstReader, ORTESubsProp * sp);
```

Arguments

cstReader
: handle to publication

sp
: pointer to ORTESubsProp structure where properties of subscrition will be stored

ORTESubscriptionPropertiesSet

Name

`ORTESubscriptionPropertiesSet` — set properties of a subscription

Synopsis

```
int ORTESubscriptionPropertiesSet (ORTESubscription * cstReader, ORTESubsProp * sp);
```

Arguments

cstReader
: handle to publication

sp
: pointer to ORTESubsProp structure containing desired properties of the subscription

Description

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

ORTESubscriptionWaitForPublications

Name

`ORTESubscriptionWaitForPublications` — waits for given number of publications

Synopsis

```
int ORTESubscriptionWaitForPublications (ORTESubscription * cstReader, NtpTime wait, unsigned int
retries, unsigned int noPublications);
```

Arguments

cstReader
: handle to subscription

wait
: time how long to wait

retries
: number of retries if specified number of publications was not reached

noPublications
: desired number of publications

Description

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

ORTESubscriptionGetStatus

Name

`ORTESubscriptionGetStatus` — get status of a subscription

Synopsis

```
int ORTESubscriptionGetStatus (ORTESubscription * cstReader, ORTESubsStatus * status);
```

Arguments

cstReader
: handle to subscription

status
: pointer to ORTESubsStatus structure

Description

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

ORTESubscriptionPull

Name

`ORTESubscriptionPull` — read data from receiving buffer

Synopsis

```
int ORTESubscriptionPull (ORTESubscription * cstReader);
```

Arguments

cstReader
: handle to subscription

Description

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

ORTETypeRegisterAdd

Name

`ORTETypeRegisterAdd` — register new data type

Synopsis

```
int ORTETypeRegisterAdd (ORTEDomain * d, const char * typeName, ORTETypeSerialize ts, ORTETypeDeserialize
ds, unsigned int gms);
```

Arguments

d
: domain object handle

typeName
: name of data type

ts
: pointer to serialization function. If NULL data will be copied without any processing.

ds
: deserialization function. If NULL data will be copied without any processing.

gms
: maximum length of data (in bytes)

Description

Each data type has to be registered. Main purpose of this process is to define serialization and deserialization functions for given data type. The same data type can be registered several times, previous registrations of the same type will be overwritten.

Examples of serialization and deserialization functions can be found if contrib/shape/ortedemo_types.c file.

Returns ORTE_OK if new data type has been succesfully registered.

ORTETypeRegisterDestroyAll

Name

`ORTETypeRegisterDestroyAll` — destroy all registered data types

Synopsis

```
int ORTETypeRegisterDestroyAll (ORTEDomain * d);
```

Arguments

d

domain object handle

Description

Destroys all data types which were previously registered by function *ORTETypeRegisterAdd*. Return ORTE_OK if all data types has been succesfully destroyed.

ORTEVerbositySetOptions

Name

`ORTEVerbositySetOptions` — set verbosity options

Synopsis

```
void ORTEVerbositySetOptions (const char * options);
```

Arguments

options

verbosity options

Description

There are 10 levels of verbosity ranging from 1 (lowest) to 10 (highest). It is possible to specify certain level of verbosity for each module of ORTE library. List of all supported modules can be found in linorte/usedSections.txt file. Every module has been aasigned with a number as can be seen in usedSections.txt file.

For instance

options = “ALL,7” Verbosity will be set to level 7 for all modules.

options = “51,7:32,5” Modules 51 (RTPSCSTWrite.c) will use verbosity level 7 and module 32 (ORTEPublicationTimer.c) will use verbosity level 5.

Maximum number of modules and verbosity levels can be changed in order to save some memory space if small memory footprint is neccessary. These values are defined as macros MAX_DEBUG_SECTIONS and MAX_DEBUG_LEVEL in file *include*/defines.h.

Return ORTE_OK if desired verbosity levels were successfuly set.

ORTEVerbositySetLogFile

Name

`ORTEVerbositySetLogFile` — set log file

Synopsis

```
void ORTEVerbositySetLogFile (const char * logfile);
```

Arguments

logfile
: log file name

Description

Sets output file where debug messages will be writen to. By default these messages are written to stdout.

ORTEInit

Name

`ORTEInit` — initialization of ORTE layer (musi se zavolat)

Synopsis

```
void ORTEInit ( void);
```

Arguments

void
: no arguments

ORTEAppSendThread

Name

`ORTEAppSendThread` — resume sending thread in context of calling function.

Synopsis

```
void ORTEAppSendThread (ORTEDomain * d);
```

Arguments

d
: domain object handle

Description

Sending thread will be resumed. This function never returns.

ORTESleepMs

Name

`ORTESleepMs` — suspends calling thread for given time

Synopsis

```
void ORTESleepMs (unsigned int ms);
```

Arguments

ms

miliseconds to sleep

1.1.3.3. Macros

SeqNumberCmp

Name

`SeqNumberCmp` — comparison of two sequence numbers

Synopsis

```
SeqNumberCmp ( sn1,  sn2);
```

Arguments

sn1

source sequential number 1

sn2

source sequential number 2

Return

1 if sn1 > sn2 -1 if sn1 < sn2 0 if sn1 = sn2

SeqNumberInc

Name

`SeqNumberInc` — incrementation of a sequence number

Synopsis

```
SeqNumberInc ( res,  sn);
```

Arguments

res

result

sn
: sequential number to be incremented

Description

res = sn + 1

SeqNumberAdd

Name

`SeqNumberAdd` — addition of two sequential numbers

Synopsis

```
SeqNumberAdd ( res,  sn1,  sn2);
```

Arguments

res
: result

sn1
: source sequential number 1

sn2
: source sequential number 2

Description

res = sn1 + sn2

SeqNumberDec

Name

`SeqNumberDec` — decrementation of a sequence number

Synopsis

```
SeqNumberDec ( res,  sn);
```

Arguments

res
: result

sn
: sequential number to be decremented

Description

res = sn - 1

SeqNumberSub

Name

`SeqNumberSub` — substraction of two sequential numbers

Synopsis

SeqNumberSub (*res*, *sn1*, *sn2*);

Arguments

res
: result

sn1
: source sequential number 1

sn2
: source sequential number 2

Description

res = sn1 - sn2

NtpTimeCmp

Name

`NtpTimeCmp` — comparation of two NtpTimes

Synopsis

NtpTimeCmp (*time1*, *time2*);

Arguments

time1
: source time 1

time2
: source time 2

Return value

1 if time 1 > time 2 -1 if time 1 < time 2 0 if time 1 = time 2

NtpTimeAdd

Name

`NtpTimeAdd` — addition of two NtpTimes

Synopsis

`NtpTimeAdd ( res, time1, time2);`

Arguments

res
: result

time1
: source time 1

time2
: source time 2

Description

res = time1 + time2

NtpTimeSub

Name

`NtpTimeSub` — substraction of two NtpTimes

Synopsis

`NtpTimeSub ( res, time1, time2);`

Arguments

res
: result

time1
: source time 1

time2
: source time 2

Description

res = time1 - time2

NtpTimeAssembFromMs

Name

`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

Name

`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

Name

`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

Name

`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

Domain2Port

Name

`Domain2Port` — converts Domain value to IP Port value

Synopsis

```
Domain2Port ( d,  p);
```

Arguments

d
: domain

p
: port

Domain2PortMulticastUserdata

Name

`Domain2PortMulticastUserdata` — converts Domain value to userdata IP Port value

Synopsis

```
Domain2PortMulticastUserdata ( d,  p);
```

Arguments

d
: domain

p

port

Domain2PortMulticastMetatraffic

Name

`Domain2PortMulticastMetatraffic` — converts Domain value to metatraffic IP Port value

Synopsis

```
Domain2PortMulticastMetatraffic ( d,  p);
```

Arguments

d

domain

p

port

1.1.4. 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

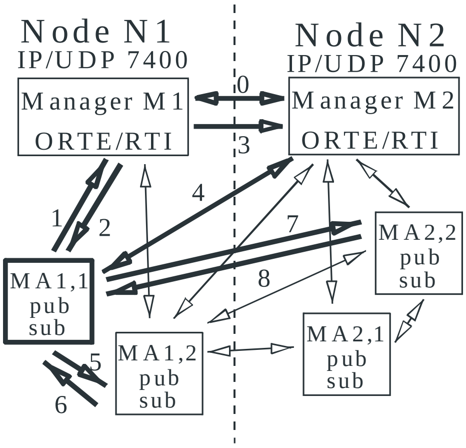

Figure 1-1. Communication among network objects.

1.1.5. 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.

1.1.6. Examples

The skeleton of an ORTE application is very simple:

```
#include <orte_api.h>

ORTEDomain *d = NULL;
char instance2send[64];
int counter = 0;

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

  d = ORTEDomainAppCreate(ORTE_DEFAUL_DOMAIN, NULL, NULL, ORTE_FALSE);
  if (!d)
  {
    printf("ORTEDomainAppCreate failed\n");
    return -1;
  }
  /*
  .....
  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.

```
ORTEPublication *p;
NtpTime persistence, delay;

ORTETypeRegisterAdd(d, "HelloMsg", NULL, NULL, 64);
NTPTIME_BUILD(persistence, 3);  /* this issue is valid for 3 seconds */
NTPTIME_DELAY(delay, 1);        /* a callback function will be called every 1 second */
p = ORTEPublicationCreate( d,
                           "Example HelloMsg",
                           "HelloMsg",
                           &instance2Send,
                           &persistence,
                           1,
                           sendCallBack,
                           NULL,
                           &delay);
```

The callback function will be then called when new issue from publisher has to be sent.

```
void sendCallBack(const ORTESendInfo *info, void *vinstance, void *sendCallBackParam)
{
  char *instance = (char *) vinstance;
  switch (info->status)
  {
    case NEED_DATA:
         printf("Sending publication, count %d\n", counter);
         sprintf(instance, "Hello world (%d)", counter++);
         break;

    case CQL:  //criticalQueueLevel has been reached
         break;
  }
}
```

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.

```
ORTESubscription *s;
NtpTime deadline, minimumSeparation;

ORTETypeRegisterAdd(d, "HelloMsg", NULL, NULL, 64);
NTPTIME_BUILD(deadline, 20);
NTPTIME_DELAY(minimumSeparation, 0);
p = ORTESubscriptionCreate( d,
                            IMMEDIATE,
                            BEST_EFFORTS,
                            "Example HelloMsg",
                            "HelloMsg",
                            &instance2Recv,
                            &deadline,
                            &minimumSeparation,
                            recvCallBack,
                            NULL);
```

The callback function is shown in the following example:

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void *recvCallBackParam)
{
```

```
        char *instance = (char *) vinstance;
        switch (info->status)
        {
          case NEW_DATA:
               printf("%s\n", instance);
               break;

          case DEADLINE:  //deadline occurred
               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.
Spy: Example demonstrating functionality for network analysis and debugging.
Reliable: Example demonstrating functionality reliable communication using ORTE.

1.1.7. 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

1.2. 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.

1.2.1. 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

1.2.2. 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.

1.2.3. API / Compatibility

not applicable

1.2.4. 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.

1.2.5. 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.

1.2.6. Examples

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

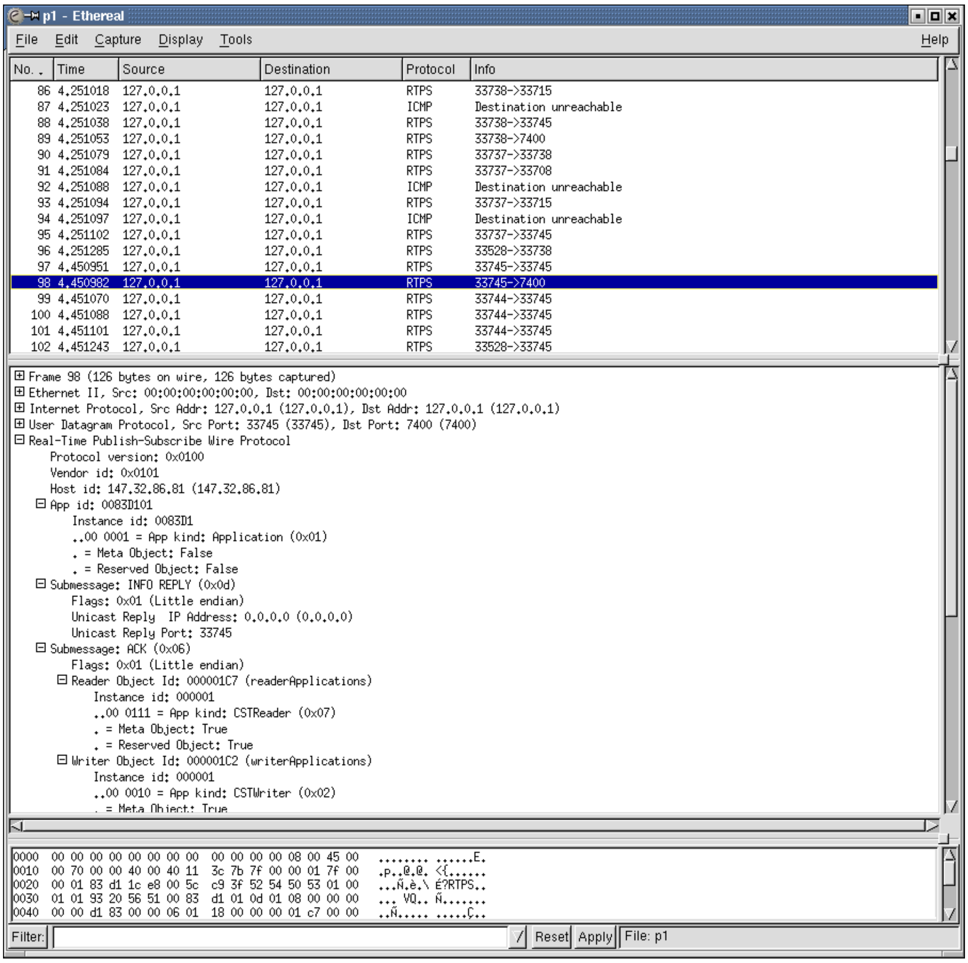

Figure 1-2. Screenshot

1.2.7. 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 2. Linux/RT-Linux CAN Driver (LinCAN)

The LINCAN is an implementation of the Linux device driver supporting more CAN controller chips and many CAN interface boards. Its implementation has long history already. The OCERA version of the driver adds new features, continuous enhancements and reimplementation of structure of the driver. Most important feature is that driver supports multiple open of one communication object from more Linux and even RT-Linux applications and threads. The usage of the driver is tightly coupled to the virtual CAN API interface component which hides driver low level interface to the application programmers.

2.1. LinCAN Summary

2.1.1. Summary

Name of the component

Linux CAN Driver (LINCAN)

Author

Pavel Pisa

Arnaud Westenberg

Tomasz Motylewski

Maintainer

Pavel Pisa

LinCAN Internet resources

http://www.ocera.org OCERA project home page

http://sourceforge.net/projects/ocera OCERA SourceForge project page. The OCERA CVS relative path to LinCAN driver sources is `ocera/components/comm/can/lincan`.

http://cmp.felk.cvut.cz/~pisa/can local testing directory

Reviewer

The previous driver versions were tested by more users. The actual version has been tested at CTU by more OCERA developers, by Unicontrols and by BFAD GmbH, which use pre-OCERA and current version of the driver in their products.

List of the cards tested with latest version of the driver:

- PC104 Advantech PCM3680 dual channel board on 2.4 RT-Linux enabled kernel
- PiKRON ISA card on 2.4.and 2.6 Linux kernels
- BfaD DIMM PC card on 2.4 RT-Linux enabled kernel
- KVASER pcican-q on 2.6 Linux kernel and on 2.4 RT-Linux enabled kernel
- virtual board tested on all systems as well

Supported layers

- High-level available

  Linux device interface available for soft real-time Linux only and for mixed-mode RT-Linux/Linux driver compilation

- Low-level available

RT-Linux device is registered only for mixed-mode RT-Linux/Linux driver compilation. The driver messages transmition and receiption runs in hard real-time threads in such case.

Version
: lincan-0.2

Status
: Beta

Dependencies
: The driver requires CAN interface hardware for access to real CAN bus.
: Driver can be used even without hardware if a virtual board is configured. This setup is useful for testing of interworking of other CAN components.
: Linux kernels from 2.2.x, 2.4.x and 2.6.x series are fully supported.
: The RT-Linux version 3.2 or OCERA RT-Linux enabled system is required for hard real-time use.
: The RT-Linux version requires RT-Linux `malloc`, which is part of OCERA RT-Linux version and can be downloaded for older RT-Linux versions .
: The use of VCA API library is suggested for seamless application transitions between driver kinds and versions.

Supported hardware (some not tested)

- Advantech PC-104 PCM3680 dual channel board
- PiKRON ISA card
- BfaD DIMM PC card
- KVASER PCIcan-Q, PCIcan-D, PCIcan-S
- KVASER PCcan-Q, PCcan-D, PCcan-S, PCcan-F
- MPL pip5 and pip6
- NSI PC-104 board CAN104
- Contemporary Controls PC-104 board CAN104
- Arcom Control Systems PC-104 board AIM104CAN
- IXXAT ISA board PC-I03
- SECO PC-104 board M436
- Board support template sources for yet unsupported hardware
- Virtual board

Release date
: February 2004

2.2. LinCAN Driver Description

2.2.1. Introduction

The LinCAN driver is the loadable module for the Linux kernel which implements CAN driver. The driver communicates and controls one or more CAN controllers chips. Each chip/CAN interface is represented to the applications as one or more CAN message objects accessible as character devices. The application can open the character device and use `read`/`write` system calls for CAN messages transmission or reception through the connected message object. The parameters of the message object can be modified by the `IOCTL` system call. The closing of the character device releases resources allocated by the application. The present version of the driver supports three most common CAN controllers:

- Intel i82527 chips
- Philips 82c200 chips
- Philips SJA1000 chips in standard and PeliCAN mode

The intelligent CAN/CANopen cards should be supported by in the near future. One of such cards is P-CAN series of cards produced by Unicontrols. The driver contains support for more than ten CAN cards basic types with different combinations of the above mentioned chips. Not all card types are held by OCERA members, but CTU has and tested more SJA1000 type cards and will test some i82527 cards in near future.

2.3. LinCAN Driver System Level API

2.3.1. Device Files and Message Structure

Each driver is a subsystem which has no direct application level API. The operating system is responsible for user space calls transformation into driver functions calls or dispatch routines invocations. The CAN driver is implemented as a character device with the standard device node names `/dev/can0`, `/dev/can1`, etc. The application program communicates with the driver through the standard system low level input/output primitives (`open`, `close`, `read`, `write`, `select` and `ioctl`). The CAN driver convention of usage of these functions is described in the next subsection.

The `read` and `write` functions need to transfer one or more CAN messages. The structure `canmsg_t` is defined for this purpose and is defined in include file `can/can.h`. The `canmsg_t` structure has next fields:

```
struct canmsg_t {
    short flags;
    int cob;
    unsigned long id;
    unsigned long timestamp;
    unsigned int length;
    unsigned char
    data[CAN_MSG_LENGTH];
} PACKED;
```

flags

: The flags field holds information about message type. The bit `MSG_RTR` marks remote transmission request messages. Writing of such message into the CAN message object handle results in transmission of the RTR message. The RTR message can be received by the read call if no buffer with corresponding ID is pre-filled in the driver. The bit `MSG_EXT` indicates that the message with extended (bit 29 set) ID will be send or was received. The bit `MSG_OVR` is intended for fast indication of the reception message queue overfill. The transmitted messages could be distributed back to the local clients after transmition to the CAN bus. Such messages are marked by `MSG_LOCAL` bit.

cob

: The field reserved for a holding message communication object number. It could be used for serialization of received messages from more message object into one message queue in the future.

id

: CAN message ID.

timestamp

: The field intended for storing of the message reception time.

length

The number of the data bytes send or received in the CAN message. The number of data load bytes is from 0 to 8.

data

The byte array holding message data.

As was mentioned above, direct communication with the driver through system calls is not encouraged because this interface is partially system dependent and cannot be ported to all environments. The suggested alternative is to use OCERA provided VCA library which defines the portable and clean interface to the CAN driver implementation.

The other issue is addition of the support for new CAN interface boards and CAN controller chips. The subsection Board Support Functions describes template functions, which needs to be implemented for newly supported board. The template of board support can be found in the file `src/template.c`.

The other task for more brave souls is addition of the support for the unsupported chip type. The source supporting the SJA1000 chip in the PeliCAN mode can serve as an example. The full source of this chip support is stored in the file `src/sja1000p.c`. The subsection Chip Support Functions describes basic functions necessary for the new chip support.

2.3.2. CAN Driver File Operations

open

Name

`open` — message communication object open system call

Synopsis

```
int open (const char * pathname, int flags);
```

Arguments

pathname

The path to driver device node is specified there. The conventional device names for Linux CAN driver are `/dev/can0`, `/dev/can1`, etc.

flags

flags modifying style of open call. The standard `O_RDWR` mode should be used for CAN device. The mode `O_NOBLOCK` can be used with driver as well. This mode results in immediate return of `read` and `write`.

Description

Returns negative number in the case of error. Returns the file descriptor for named CAN message object in other cases.

close

Name

`close` — message communication object close system call

Synopsis

```
int close (int fd);
```

Arguments

fd
: file descriptor to opened can message communication object

Description

Returns negative number in the case of error.

read

Name

`read` — reads received CAN messages from message object

Synopsis

```
ssize_t read(int fd, void * buf, size_t count);
```

Arguments

fd
: file descriptor to opened can message communication object

buf
: pointer to array of `canmsg_t` structures.

count
: size of message array buffer in number of bytes

Description

Returns negative value in the case of error else returns number of read bytes which is multiple of `canmsg_t` structure size.

write

Name

`write` — writes CAN messages to message object for transmission

Synopsis

```
ssize_t write(int fd, const void * buf, size_t count);
```

Arguments

fd
: file descriptor to opened can message communication object

buf
: pointer to array of `canmsg_t` structures.

count

size of message array buffer in number of bytes. The parameter informs driver about number of messages prepared for transmission and should be multiple of `canmsg_t` structure size.

Description

Returns negative value in the case of error else returns number of bytes successfully stored into message object transmission queue. The positive returned number is multiple of `canmsg_t` structure size.

struct canfilt_t

Name

`struct canfilt_t` — structure for acceptance filter setup

Synopsis

```
struct canfilt_t {
  int flags;
  int queid;
  int cob;
  unsigned long id;
  unsigned long mask;
};
```

Members

flags

message flags

`MSG_RTR` .. message is Remote Transmission Request,

`MSG_EXT` .. message with extended ID,

`MSG_OVR` .. indication of queue overflow condition,

`MSG_LOCAL` .. message originates from this node.

there are corresponding mask bits `MSG_RTR_MASK`, `MSG_EXT_MASK`, `MSG_LOCAL_MASK`.

`MSG_PROCESSLOCAL` enables local messages processing in the combination with global setting

queid

CAN queue identification in the case of the multiple queues per one user (open instance)

cob

communication object number (not used)

id

selected required value of cared ID id bits

mask

select bits significant for the comparison;

1 .. take care about corresponding ID bit,

0 .. don't care

IOCTL CANQUE_FILTER

Name

`IOCTL CANQUE_FILTER` — Sets acceptance filter for CAN queue connected to client state

Synopsis

```
int ioctl(int fd, int command = CANQUE_FILTER, struct canfilt_t * filt);
```

Arguments

fd

file descriptor to opened can message communication object

command

Denotes CAN queue filter command, `CANQUE_FILTER`

filt

pointer to the `canfilt_t` structure.

Description

The `CANQUE_FILTER` IOCTL invocation sets acceptance mask of associated canqueue to specified parameters. Actual version of the driver changes filter of the default receiption queue. The filed `queid` should be initialized to zero to support compatibility with future driver versions.

The call returns negative value in the case of error.

IOCTL CANQUE_FLUSH

Name

`IOCTL CANQUE_FLUSH` — Flushes messages from receiption CAN queue

Synopsis

```
int ioctl(int fd, int command = CANQUE_FLUSH, int queid);
```

Arguments

fd

file descriptor to opened can message communication object

command

Denotes CAN queue flush command, `CANQUE_FLUSH`

queid

Should be initialized to zero to support compatibility with future driver versions

Description

The call flushes all messages from the CAN queue.

The call returns negative value in the case of error.

2.4. LinCAN Driver Architecture

The LinCAN provides simultaneous queued communication for more concurrent running applications.

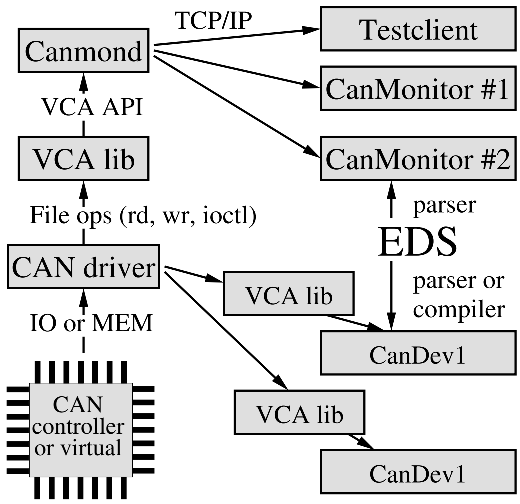

Figure 2-1. LinCAN architecture

Even each of communication object can be used by one or more applications, which connects to the communication object internal representation by means of CAN FIFO queues. This enables to build complex systems based even on card and chips, which provides only one communication objects (for example SJA1000).

The driver can be configured to provide virtual CAN board (software emulated message object) to test CAN components on the Linux system without hardware required to connect to the real CAN bus. The example configuration of the CAN network components connected to one real or virtual communication object of LinCAN driver is shown in figure Figure 2-1. The communication object is used by the CAN monitor daemon and two CANopen devices implemented by OCERA CanDev component. The actual system dependent driver API is hidden to applications under VCA library. The CAN monitor daemon translates CAN messages to TCP/IP network for Java based platform independent CAN monitor and C based test client.

Each communication object is represented as character device file. The devices can be opened and closed by applications in blocking or non-blocking mode. LinCAN client application state, chip and object configurations are controlled by IOCTL system call. One or more CAN messages can be sent or received through write/read system calls. The data read from or written to the driver are formed from sequence of fixed size structures representing CAN messages.

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

The LinCAN driver version 0.2 has rewritten infrastructure based on message FIFOs organized into oriented edges between chip drivers (`structure chip_t`) message objects representations (`structure msgobj_t`) and open device file instances state (`structure canuser_t`). The complete relationship between CAN hardware representation and open instances is illustrated in the figure Figure 2-4.

The message FIFO (`structure canque_fifo_t`) initialization code allocates configurable number of slots capable to hold one message.

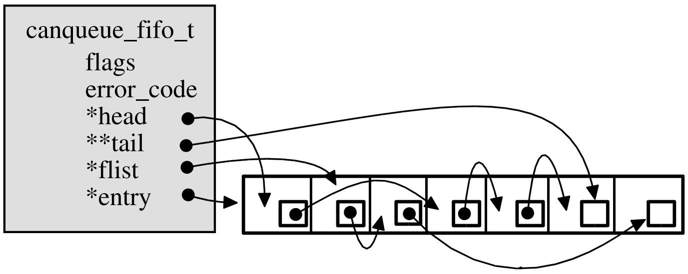

Figure 2-2. LinCAN message FIFO implementation

The all slots are linked to the free list after initialization. The slot can be requested by FIFO input side by function `canque_fifo_get_inslot`. The slot is filled by message data and is linked into FIFO queue by function `canque_fifo_put_inslot`. If previously requested slot is not successfully filled by data, it can be released by `canque_fifo_abort_inslot`. The output side of the FIFO tests presence of ready slots by function `canque_fifo_test_outslot`. If the slot is returned by this function, it is processed and released by function `canque_fifo_free_outs` The processing can be postponed in the case of bus error or higher priority message processing request by `canque_fifo_again_outslot` function. All these functions are optimized to be fast and short, which enables to synchronize them by spin-lock semaphores and guarantee atomic nature of them. The FIFO implementation is illustrated in the figure Figure 2-2.

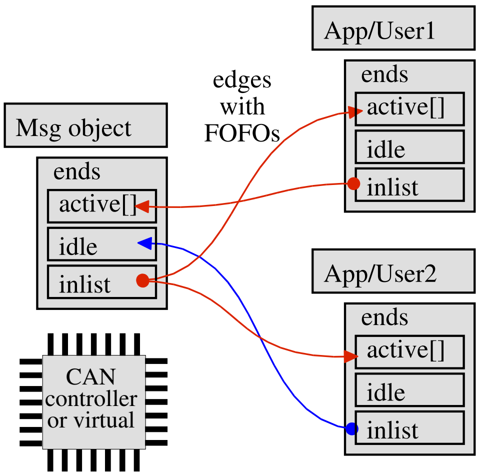

Figure 2-3. LinCAN driver message flow graph edges

The low level message FIFOs are wrapped by CAN edges structures (`canque_edge_t`), which are used for message passing between all components of the driver. The actual version of LinCAN driver uses oriented edges to connect Linux and RT-Linux clients/users with chips and communication objects. Each entity, which is able to hold edge ends, has to be equipped by `canque_ends_t` structure. The input ends of edges/FIFOs are held on `inlist`. The inactive/empty out ends of the edges are held on a `idle` list and active out ends are held on a `active` list corresponding to the edge priority. The `canque_fifo_test_outslot` function can determine by examination of active lists if there is message to accept/process. This concept makes possible to use same type of edges for outgoing and incoming directions. The concept of edges can be even used for message filtering by priority or acceptance masks. It is prepared for future targeting messages to predefined message objects according to their priority or type and for redundant and fault tolerant message distribution into more CAN buses. Message concentration, virtual nodes and other special processing can be implemented above this concept as well. The example of interconnection of one communication object with two users/open instances is illustrated in the picture Figure 2-3. Three edges/FIFOs are in the active state and one edge/FIFO is empty in the shown example.

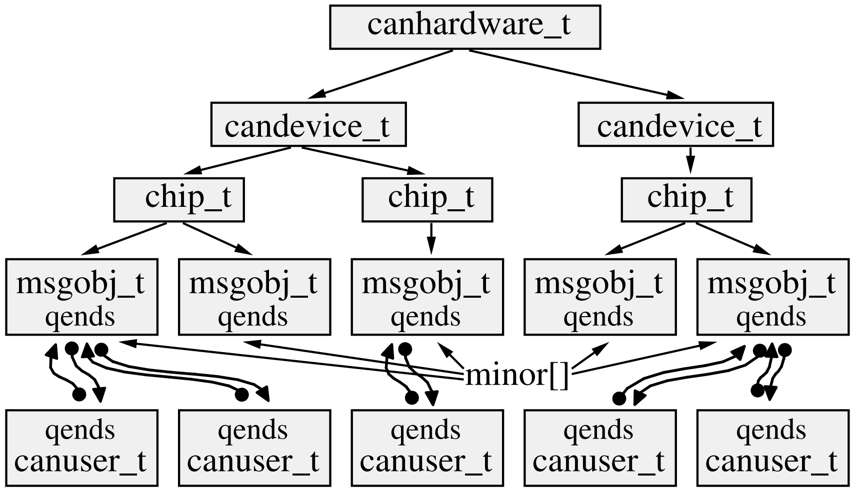

Figure 2-4. CAN hardware model in the LinCAN driver

The figure Figure 2-4 is example of object inside LinCAN driver representing system with two boards, three chips and more communication objects. Some of these objects are used by one or more applications. The object open instances are represented as `canuser_t` structures.

2.5. Driver History and Implementation Issues

The development of the CAN drivers for Linux has long history. We have been faced before two basic alternatives, start new project from scratch or use some other project as basis of our development. The first approach could lead faster to more simple and clean internal architecture but it would mean to introduce new driver with probably incompatible interface unusable for already existing applications. The support of many types of cards is thing which takes long time as well. More existing projects aimed to development of a Linux CAN driver has been analyzed:

Original LDDK CAN driver project

: The driver project aborted on the kernel evolution and LDDK concept. The LDDK tried to prepare infrastructure for development of the kernel version independent character device drivers written in meta code. The goal was top-ranking, but it proves, that well written "C" language driver can be more portable than the LDDK complex infrastructure.

can4linux-0.9 by PORT GmbH

: This is version of the above LDDK driver maintained by Port GmbH. The card type is hard compiled into the driver by selected defines and only Philips 82c200 chips are supported.

CanFestival

: The big advantage of this driver is an integrated support for the RT-Linux, but driver implementation is highly coupled to one card. Some concepts of the driver are interesting but the driver has the hard-coded number of message queues.

can-0.7.1 by Arnaud Westenberg

: This driver has its roots in the LDDK project as well. The original LDDK concept has been eliminated in the driver source and necessary adaptation of the driver for the different Linux kernel versions is achieved by the controllable number of defines and conditional compilation. There is more independent contributors. The main advantages of the driver are support of many cards working in parallel, IO and memory space chip connection support and more cards of different types can be selected at module load time. There exist more users and applications compatible with the driver interface. Disadvantages of the original version of this driver are non-optimal infrastructure, non-portable make system and lack of the select support.

The responsible OCERA developers selected the can-0.7.1 driver as a base of their development for next reasons:

- Best support for more cards in system
- Supports for many types of cards
- The internal abstraction of the peripheral access method and the chip support

The most important features added by OCERA development team are:

- Added the select system call support
- The support for our memory mapped ISA card added, which proved simplicity of addition of the support for new type of CAN cards
- Added devfs support
- Revised and bug-fixed the IRQ support in the first phase
- Added support for 2.6.x kernels
- Rebuilt the make system to compile options fully follow the running kernel options, cross-compilation still possible when the kernel location and compiler is specified. The driver checked with more 2.2.x, 2.4.x and 2.6.x kernels and hardware configurations.
- Cleaned-up synchronization required to support 2.6.x SMP kernels and enhanced 2.4.x kernels performance
- The deeper rebuilt of the driver infrastructure to enable porting to more systems (most important RT-Linux). The naive FIFO implementation has been replaced by robust CAN queues, edges and ends framework. The big advantage of continuous development is ability to keep compatibility with many cards and applications
- The infrastructure rewrite enabled to support multiple opening of the single minor device
- Support for individual queues message acceptance filters added
- The driver setup functions modified to enable PCI and USB hardware hot-swapping and PnP recognition in the future
- Added support for KVASER PCI cards family
- Added support for virtual can board for more CAN/CANopen components interworking testing on single computer without real CAN hardware.
- The conditional compilation mode for Linux/RT-Linux support has been added. The driver manipulates with chips and boards from RT-Linux hard real-time worker threads in that compilation mode. The POSIX device file interface is provided for RT-Linux threads in parallel to the standard Linux device interface.
- Work on support for first of intelligent CAN/CANopen cards has been started

The possible future enhancements

- Cleanup and enhance RTR processing. Add some support for emulated RTR processing for SJA1000 chips
- Enhance clients API to gain full advantages of possibility to connect more CAN queues with different priorities to the one user state structure
- Add support for more CAN cards and chips (82C900 comes to mind)
- Add support for XILINX FPGA based board in development at CTU. There already exists VHDL source for the chip core, connect it to PC-104 bus and LinCAN driver
- Do next steps in the PCI cards support cleanup and add Linux 2.6.x sysfs support

2.6. LinCAN Driver Internals

2.6.1. Basic Driver Data Structures

struct canhardware_t

Name

`struct canhardware_t` — structure representing pointers to all CAN boards

Synopsis

```
struct canhardware_t {
  int nr_boards;
  struct rtr_id * rtr_queue;
  can_spinlock_t rtr_lock;
  struct candevice_t * * candevice;
};
```

Members

nr_boards
: number of present boards

rtr_queue
: RTR - remote transmission request queue (expect some changes there)

rtr_lock
: locking for RTR queue

candevice
: array of pointers to CAN devices/boards

struct candevice_t

Name

`struct candevice_t` — CAN device/board structure

Synopsis

```
struct candevice_t {
  char * hwname;
  int candev_idx;
  unsigned long io_addr;
  unsigned long res_addr;
  unsigned long dev_base_addr;
  unsigned int flags;
  int nr_all_chips;
  int nr_82527_chips;
  int nr_sja1000_chips;
  struct chip_t * * chip;
  struct hwspecops_t * hwspecops;
  struct canhardware_t * hosthardware_p;
  union sysdevptr;
};
```

Members

hwname
: text string with board type

candev_idx
: board index in canhardware_t.candevice[]

io_addr
: IO/physical MEM address

res_addr
: optional reset register port

dev_base_addr
: CPU translated IO/virtual MEM address

flags
: board flags: `PROGRAMMABLE_IRQ` .. interrupt number can be programmed into board

nr_all_chips
: number of chips present on the board

nr_82527_chips
: number of Intel 8257 chips

nr_sja1000_chips
: number of Philips SJA100 chips

chip
: array of pointers to the chip structures

hwspecops
: pointer to board specific operations

hosthardware_p
: pointer to the root hardware structure

sysdevptr
: union reserved for pointer to bus specific device structure (case *`pcidev`* is used for PCI devices)

Description

The structure represent configuration and state of associated board. The driver infrastructure prepares this structure and calls board type specific `board_register` function. The board support provided register function fills right function pointers in *`hwspecops`* structure. Then driver setup calls functions `init_hw_data`, `init_chip_data`, `init_chip_data`, `init_obj_data` and `program_irq`. Function `init_hw_data` and `init_chip_data` have to specify number and types of connected chips or objects respectively. The use of *`nr_all_chips`* is preferred over use of fields *`nr_82527_chips`* and *`nr_sja1000_chips`* in the board non-specific functions. The *`io_addr`* and *`dev_base_addr`* is filled from module parameters to the same value. The request_io function can fix-up *`dev_base_addr`* field if virtual address is different than bus address.

struct chip_t

Name

`struct chip_t` — CAN chip state and type information

Synopsis

```
struct chip_t {
  char * chip_type;
  int chip_idx;
  int chip_irq;
  unsigned long chip_base_addr;
  unsigned int flags;
```

```
    long clock;
    long baudrate;
    void (* write_register (unsigned char data,unsigned long address);
    unsigned (* read_register (unsigned long address);
    unsigned short sja_cdr_reg;
    unsigned short sja_ocr_reg;
    unsigned short int_cpu_reg;
    unsigned short int_clk_reg;
    unsigned short int_bus_reg;
    struct msgobj_t * * msgobj;
    struct chipspecops_t * chipspecops;
    struct candevice_t * hostdevice;
    int max_objects;
    can_spinlock_t chip_lock;
    #ifdef CAN_WITH_RTLpthread_t worker_thread;
    unsigned long pend_flags;
  };
```

Members

chip_type
: text string describing chip type

chip_idx
: index of the chip in candevice_t.chip[] array

chip_irq
: chip interrupt number if any

chip_base_addr
: chip base address in the CPU IO or virtual memory space

flags
: chip flags: `CHIP_CONFIGURED` .. chip is configured, `CHIP_SEGMENTED` .. access to the chip is segmented (mainly for i82527 chips)

clock
: chip base clock frequency in Hz

baudrate
: selected chip baudrate in Hz

write_register
: write chip register function copy -

read_register
: read chip register function copy

sja_cdr_reg
: SJA specific register - holds hardware specific options for the Clock Divider register. Options defined in the sja1000.h file: `CDR_CLKOUT_MASK`, `CDR_CLK_OFF`, `CDR_RXINPEN`, `CDR_CBP`, `CDR_PELICAN`

sja_ocr_reg
: SJA specific register - hold hardware specific options for the Output Control register. Options defined in the sja1000.h file: `OCR_MODE_BIPHASE`, `OCR_MODE_TEST`, `OCR_MODE_NORMAL`, `OCR_MODE_CLOCK`, `OCR_TX0_LH`, `OCR_TX1_ZZ`.

int_cpu_reg
: Intel specific register - holds hardware specific options for the CPU Interface register. Options defined in the i82527.h file: `iCPU_CEN`, `iCPU_MUX`, `iCPU_SLP`, `iCPU_PWD`, `iCPU_DMC`, `iCPU_DSC`, `iCPU_RST`.

int_clk_reg
: Intel specific register - holds hardware specific options for the Clock Out register. Options defined in the i82527.h file: `iCLK_CD0`, `iCLK_CD1`, `iCLK_CD2`, `iCLK_CD3`, `iCLK_SL0`, `iCLK_SL1`.

int_bus_reg
: Intel specific register - holds hardware specific options for the Bus Configuration register. Options defined in the i82527.h file: `iBUS_DR0`, `iBUS_DR1`, `iBUS_DT1`, `iBUS_POL`, `iBUS_CBY`.

msgobj
: array of pointers to individual communication objects

chipspecops
: pointer to the set of chip specific object filled by `init_chip_data` function

hostdevice
: pointer to chip hosting board

max_objects
: maximal number of communication objects connected to this chip

chip_lock
: reserved for synchronization of the chip supporting routines (not used in the current driver version)

worker_thread
: chip worker thread ID (RT-Linux specific field)

pend_flags
: holds information about pending interrupt and `tx_wake` operations (RT-Linux specific field). Masks values: `MSGOBJ_TX_REQUEST` .. some of the message objects requires `tx_wake` call, `MSGOBJ_IRQ_REQUEST` .. chip interrupt processing required `MSGOBJ_WORKER_WAKE` .. marks, that worker thread should be waked for some of above reasons

Description

The fields *`write_register`* and *`read_register`* are copied from corresponding fields from *`hwspecops`* structure (chip->hostdevice->hwspecops->write_register and chip->hostdevice->hwspecops->read_register) to speedup `can_write_reg` and `can_read_reg` functions.

struct msgobj_t

Name

`struct msgobj_t` — structure holding communication object state

Synopsis

```
struct msgobj_t {
  unsigned long obj_base_addr;
  unsigned int minor;
  unsigned int object;
  unsigned long obj_flags;
  int ret;
  struct canque_ends_t * qends;
  struct canque_edge_t * tx_qedge;
  struct canque_slot_t * tx_slot;
  int tx_retry_cnt;
  struct timer_list tx_timeout;
  struct canmsg_t rx_msg;
  struct chip_t * hostchip;
  atomic_t obj_used;
  struct list_head obj_users;
};
```

Members

obj_base_addr

minor
: associated device minor number

object
: object number in chip_t structure +1

obj_flags
: message object specific flags. Masks values: `MSGOBJ_TX_REQUEST` .. the message object requests TX activation `MSGOBJ_TX_LOCK` .. some IRQ routine or callback on some CPU is running inside TX activation processing code

ret
: field holding status of the last Tx operation

qends
: pointer to message object corresponding ends structure

tx_qedge
: edge corresponding to transmitted message

tx_slot
: slot holding transmitted message, slot is taken from `canque_test_outslot` call and is freed by `canque_free_outslot` or rescheduled `canque_again_outslot`

tx_retry_cnt
: transmission attempt counter

tx_timeout
: can be used by chip driver to check for the transmission timeout

rx_msg
: temporary storage to hold received messages before calling to `canque_filter_msg2edges`

hostchip
: pointer to the &chip_t structure this object belongs to

obj_used
: counter of users (associated file structures for Linux userspace clients) of this object

obj_users
: list of user structures of type &canuser_t.

struct canuser_t

Name

`struct canuser_t` — structure holding CAN user/client state

Synopsis

```
struct canuser_t {
  unsigned long flags;
  struct list_head peers;
  struct canque_ends_t * qends;
  struct msgobj_t * msgobj;
  struct canque_edge_t * rx_edge0;
  union userinfo;
  int magic;
};
```

Members

flags
: used to distinguish Linux/RT-Linux type

peers
: for connection into list of object users

qends
: pointer to the ends structure corresponding for this user

msgobj
: communication object the user is connected to

rx_edge0
: default receive queue for filter IOCTL

userinfo
: stores user context specific information. The field *fileinfo*.file holds pointer to open device file state structure for the Linux user-space client applications

magic
: magic number to check consistency when pointer is retrieved from file private field

struct hwspecops_t

Name

`struct hwspecops_t` — hardware/board specific operations

Synopsis

```
struct hwspecops_t {
  int (* request_io (struct candevice_t *candev);
  int (* release_io (struct candevice_t *candev);
  int (* reset (struct candevice_t *candev);
  int (* init_hw_data (struct candevice_t *candev);
  int (* init_chip_data (struct candevice_t *candev, int chipnr);
  int (* init_obj_data (struct chip_t *chip, int objnr);
  int (* program_irq (struct candevice_t *candev);
  void (* write_register (unsigned char data,unsigned long address);
  unsigned (* read_register (unsigned long address);
};
```

Members

request_io
: reserve io or memory range for can board

release_io
: free reserved io memory range

reset
: hardware reset routine

init_hw_data
: called to initialize &candevice_t structure, mainly *res_add*, *nr_all_chips*, *nr_82527_chips*, *nr_sja1000_chips* and *flags* fields

init_chip_data
: called initialize each &chip_t structure, mainly *chip_type*, *chip_base_addr*, *clock* and chip specific registers. It is responsible to setup &chip_t->*chipspecops* functions for non-standard chip types (type other than “i82527”, “sja1000” or “sja1000p”)

init_obj_data
: called initialize each &msgobj_t structure, mainly *obj_base_addr* field.

program_irq
: program interrupt generation hardware of the board if flag `PROGRAMMABLE_IRQ` is present for specified device/board

write_register
: low level write register routine

read_register
: low level read register routine

struct chipspecops_t

Name

`struct chipspecops_t` — can controller chip specific operations

Synopsis

```
struct chipspecops_t {
  int (* chip_config (struct chip_t *chip);
  int (* baud_rate (struct chip_t *chip, int rate, int clock, int sjw,int sampl_pt, int flags);
  int (* standard_mask (struct chip_t *chip, unsigned short code,unsigned short mask);
  int (* extended_mask (struct chip_t *chip, unsigned long code,unsigned long mask);
  int (* message15_mask (struct chip_t *chip, unsigned long code,unsigned long mask);
  int (* clear_objects (struct chip_t *chip);
  int (* config_irqs (struct chip_t *chip, short irqs);
  int (* pre_read_config (struct chip_t *chip, struct msgobj_t *obj);
  int (* pre_write_config (struct chip_t *chip, struct msgobj_t *obj,struct canmsg_t *msg);
  int (* send_msg (struct chip_t *chip, struct msgobj_t *obj,struct canmsg_t *msg);
  int (* remote_request (struct chip_t *chip, struct msgobj_t *obj);
  int (* check_tx_stat (struct chip_t *chip);
  int (* wakeup_tx (struct chip_t *chip, struct msgobj_t *obj);
  int (* enable_configuration (struct chip_t *chip);
  int (* disable_configuration (struct chip_t *chip);
  int (* set_btregs (struct chip_t *chip, unsigned short btr0,unsigned short btr1);
  int (* start_chip (struct chip_t *chip);
  int (* stop_chip (struct chip_t *chip);
  can_irqreturn_t (* irq_handler (int irq, void *dev_id, struct pt_regs *regs);
};
```

Members

chip_config
: CAN chip configuration

baud_rate
: set communication parameters

standard_mask
: setup of mask for message filtering

extended_mask
: setup of extended mask for message filtering

message15_mask
: set mask of i82527 message object 15

clear_objects
: clears state of all message object residing in chip

config_irqs
: tunes chip hardware interrupt delivery

pre_read_config
: prepares message object for message reception

pre_write_config
: prepares message object for message transmission

send_msg
: initiate message transmission

remote_request
: configures message object and asks for RTR message

check_tx_stat
: checks state of transmission engine

wakeup_tx
: wakeup TX processing

enable_configuration
: enable chip configuration mode

disable_configuration
: disable chip configuration mode

set_btregs
: configures bitrate registers

start_chip
: starts chip message processing

stop_chip
: stops chip message processing

irq_handler
: interrupt service routine

2.6.2. Board Support Functions

The functions, which should be implemented for each supported board, are described in the next section. The functions are prefixed by boardname. The prefix `template` has been selected for next description.

template_request_io

Name

`template_request_io` — reserve io or memory range for can board

Synopsis

```
int template_request_io (struct candevice_t * candev);
```

Arguments

candev
: pointer to candevice/board which asks for io. Field *io_addr* of *candev* is used in most cases to define start of the range

Description

The function `template_request_io` is used to reserve the io-memory. If your hardware uses a dedicated memory range as hardware control registers you will have to add the code to reserve this memory as well. `IO_RANGE` is the io-memory range that gets reserved, please adjust according your hardware. Example: #define IO_RANGE 0x100 for i82527 chips or #define IO_RANGE 0x20 for sja1000 chips in basic CAN mode.

Return Value

The function returns zero on success or `-ENODEV` on failure

File

src/template.c

template_release_io

Name

`template_release_io` — free reserved io memory range

Synopsis

```
int template_release_io (struct candevice_t * candev);
```

Arguments

candev

pointer to candevice/board which releases io

Description

The function `template_release_io` is used to free reserved io-memory. In case you have reserved more io memory, don't forget to free it here. IO_RANGE is the io-memory range that gets released, please adjust according your hardware. Example: #define IO_RANGE 0x100 for i82527 chips or #define IO_RANGE 0x20 for sja1000 chips in basic CAN mode.

Return Value

The function always returns zero

File

src/template.c

template_reset

Name

`template_reset` — hardware reset routine

Synopsis

```
int template_reset (struct candevice_t * candev);
```

Arguments

candev

Pointer to candevice/board structure

Description

The function `template_reset` is used to give a hardware reset. This is rather hardware specific so I haven't included example code. Don't forget to check the reset status of the chip before returning.

Return Value

The function returns zero on success or `-ENODEV` on failure

File

src/template.c

template_init_hw_data

Name

`template_init_hw_data` — Initialize hardware cards

Synopsis

```
int template_init_hw_data (struct candevice_t * candev);
```

Arguments

candev

Pointer to candevice/board structure

Description

The function `template_init_hw_data` is used to initialize the hardware structure containing information about the installed CAN-board. `RESET_ADDR` represents the io-address of the hardware reset register. `NR_82527` represents the number of Intel 82527 chips on the board. `NR_SJA1000` represents the number of Philips sja1000 chips on the board. The flags entry can currently only be `CANDEV_PROGRAMMABLE_IRQ` to indicate that the hardware uses programmable interrupts.

Return Value

The function always returns zero

File

src/template.c

template_init_chip_data

Name

`template_init_chip_data` — Initialize chips

Synopsis

```
int template_init_chip_data (struct candevice_t * candev, int chipnr);
```

Arguments

candev

Pointer to candevice/board structure

chipnr

Number of the CAN chip on the hardware card

Description

The function `template_init_chip_data` is used to initialize the hardware structure containing information about the CAN chips. `CHIP_TYPE` represents the type of CAN chip. `CHIP_TYPE` can be "i82527" or "sja1000". The *chip_base_addr* entry represents the start of the 'official' memory map of the installed chip. It's likely that this is the same as the *io_addr* argument supplied at module loading time. The *clock* entry holds the chip clock value in Hz. The entry *sja_cdr_reg* holds hardware specific options for the Clock Divider register. Options defined in the `sja1000`.h file: `CDR_CLKOUT_MASK`, `CDR_CLK_OFF`, `CDR_RXINPEN`, `CDR_CBP`, `CDR_PELICAN` The entry *sja_ocr_reg* holds hardware specific options for the Output Control register. Options defined in the `sja1000`.h file: `OCR_MODE_BIPHASE`, `OCR_MODE_TEST`, `OCR_MODE_NORMAL`, `OCR_MODE_CLOCK`, `OCR_TX0_LH`, `OCR_TX1_ZZ`. The entry *int_clk_reg* holds hardware specific options for the Clock Out register. Options defined in the `i82527`.h file: `iCLK_CD0`, `iCLK_CD1`, `iCLK_CD2`, `iCLK_CD3`, `iCLK_SL0`, `iCLK_SL1`. The entry *int_bus_reg* holds hardware specific options for the Bus Configuration register. Options defined in the `i82527`.h file: `iBUS_DR0`, `iBUS_DR1`, `iBUS_DT1`, `iBUS_POL`, `iBUS_CBY`. The entry *int_cpu_reg* holds hardware specific options for the cpu interface register. Options defined in the `i82527`.h file: `iCPU_CEN`, `iCPU_MUX`, `iCPU_SLP`, `iCPU_PWD`, `iCPU_DMC`, `iCPU_DSC`, `iCPU_RST`.

Return Value

The function always returns zero

File

src/template.c

template_init_obj_data

Name

`template_init_obj_data` — Initialize message buffers

Synopsis

```
int template_init_obj_data (struct chip_t * chip, int objnr);
```

Arguments

chip

Pointer to chip specific structure

objnr

Number of the message buffer

Description

The function `template_init_obj_data` is used to initialize the hardware structure containing information about the different message objects on the CAN chip. In case of the sja1000 there's only one message object but on the i82527 chip there are 15. The code below is for a i82527 chip and initializes the object base addresses The entry *`obj_base_addr`* represents the first memory address of the message object. In case of the sja1000 *`obj_base_addr`* is taken the same as the chips base address. Unless the hardware uses a segmented memory map, flags can be set zero.

Return Value

The function always returns zero

File

src/template.c

template_program_irq

Name

`template_program_irq` — program interrupts

Synopsis

```
int template_program_irq (struct candevice_t * candev);
```

Arguments

`candev`

Pointer to candevice/board structure

Description

The function `template_program_irq` is used for hardware that uses programmable interrupts. If your hardware doesn't use programmable interrupts you should not set the *`candevices_t`*->flags entry to `CANDEV_PROGRAMMABLE_IRQ` and leave this function unedited. Again this function is hardware specific so there's no example code.

Return value

The function returns zero on success or `-ENODEV` on failure

File

src/template.c

template_write_register

Name

`template_write_register` — Low level write register routine

Synopsis

```
void template_write_register (unsigned char data, unsigned long address);
```

Arguments

data
: data to be written

address
: memory address to write to

Description

The function `template_write_register` is used to write to hardware registers on the CAN chip. You should only have to edit this function if your hardware uses some specific write process.

Return Value

The function does not return a value

File

src/template.c

template_read_register

Name

`template_read_register` — Low level read register routine

Synopsis

```
unsigned template_read_register (unsigned long address);
```

Arguments

address
: memory address to read from

Description

The function `template_read_register` is used to read from hardware registers on the CAN chip. You should only have to edit this function if your hardware uses some specific read process.

Return Value

The function returns the value stored in *address*

File

src/template.c

2.6.3. Chip Support Functions

The controller chip specific functions are described in the next section. The functions should be prefixed by chip type. Because documentation of chip functions has been retrieved from the actual SJA1000 PeliCAN support, the function prefix is *`sja1000p`*.

sja1000p_enable_configuration

Name

`sja1000p_enable_configuration` — enable chip configuration mode

Synopsis

`int` **`sja1000p_enable_configuration`** `(struct chip_t *` *`chip`*`);`

Arguments

`chip`

pointer to chip state structure

sja1000p_disable_configuration

Name

`sja1000p_disable_configuration` — disable chip configuration mode

Synopsis

`int` **`sja1000p_disable_configuration`** `(struct chip_t *` *`chip`*`);`

Arguments

`chip`

pointer to chip state structure

sja1000p_chip_config

Name

`sja1000p_chip_config` — can chip configuration

Synopsis

`int` **`sja1000p_chip_config`** `(struct chip_t *` *`chip`*`);`

Arguments

`chip`

pointer to chip state structure

Description

This function configures chip and prepares it for message transmission and reception. The function resets chip, resets mask for acceptance of all messages by call to `sja1000p_extended_mas` function and then computes and sets baudrate with use of function `sja1000p_baud_rate`.

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_extended_mask

Name

`sja1000p_extended_mask` — setup of extended mask for message filtering

Synopsis

```
int sja1000p_extended_mask (struct chip_t * chip, unsigned long code, unsigned long mask);
```

Arguments

chip
: pointer to chip state structure

code
: can message acceptance code

mask
: can message acceptance mask

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_baud_rate

Name

`sja1000p_baud_rate` — set communication parameters.

Synopsis

```
int sja1000p_baud_rate (struct chip_t * chip, int rate, int clock, int sjw, int sampl_pt, int flags);
```

Arguments

chip
: pointer to chip state structure

rate
: baud rate in Hz

clock
: frequency of sja1000 clock in Hz (ISA osc is 14318000)

sjw
: synchronization jump width (0-3) prescaled clock cycles

sampl_pt
: sample point in % (0-100) sets (TSEG1+1)/(TSEG1+TSEG2+2) ratio

flags
: fields `BTR1_SAM`, `OCMODE`, `OCPOL`, `OCTP`, `OCTN`, `CLK_OFF`, `CBP`

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_read

Name

`sja1000p_read` — reads and distributes one or more received messages

Synopsis

```
void sja1000p_read (struct chip_t * chip, struct msgobj_t * obj);
```

Arguments

chip
: pointer to chip state structure

obj
: pinter to CAN message queue information

File

src/sja1000p.c

sja1000p_pre_read_config

Name

`sja1000p_pre_read_config` — prepares message object for message reception

Synopsis

```
int sja1000p_pre_read_config (struct chip_t * chip, struct msgobj_t * obj);
```

Arguments

chip
: pointer to chip state structure

obj
: pointer to message object state structure

Return Value

negative value reports error. Positive value indicates immediate reception of message.

File

src/sja1000p.c

sja1000p_pre_write_config

Name

`sja1000p_pre_write_config` — prepares message object for message transmission

Synopsis

```
int sja1000p_pre_write_config (struct chip_t * chip, struct msgobj_t * obj, struct canmsg_t * msg);
```

Arguments

chip
: pointer to chip state structure

obj
: pointer to message object state structure

msg
: pointer to CAN message

Description

This function prepares selected message object for future initiation of message transmission by `sja1000p_send_msg` function. The CAN message data and message ID are transfered from *msg* slot into chip buffer in this function.

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_send_msg

Name

`sja1000p_send_msg` — initiate message transmission

Synopsis

```
int sja1000p_send_msg (struct chip_t * chip, struct msgobj_t * obj, struct canmsg_t * msg);
```

Arguments

chip
: pointer to chip state structure

obj
: pointer to message object state structure

msg
: pointer to CAN message

Description

This function is called after `sja1000p_pre_write_config` function, which prepares data in chip buffer.

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_check_tx_stat

Name

`sja1000p_check_tx_stat` — checks state of transmission engine

Synopsis

```
int sja1000p_check_tx_stat (struct chip_t * chip);
```

Arguments

chip
: pointer to chip state structure

Return Value

negative value reports error. Positive return value indicates transmission under way status. Zero value indicates finishing of all issued transmission requests.

File

src/sja1000p.c

sja1000p_set_btregs

Name

`sja1000p_set_btregs` — configures bitrate registers

Synopsis

```
int sja1000p_set_btregs (struct chip_t * chip, unsigned short btr0, unsigned short btr1);
```

Arguments

chip
: pointer to chip state structure

btr0
: bitrate register 0

btr1
: bitrate register 1

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_start_chip

Name

`sja1000p_start_chip` — starts chip message processing

Synopsis

```
int sja1000p_start_chip (struct chip_t * chip);
```

Arguments

chip
: pointer to chip state structure

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_stop_chip

Name

`sja1000p_stop_chip` — stops chip message processing

Synopsis

```
int sja1000p_stop_chip (struct chip_t * chip);
```

Arguments

chip

pointer to chip state structure

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_remote_request

Name

`sja1000p_remote_request` — configures message object and asks for RTR message

Synopsis

```
int sja1000p_remote_request (struct chip_t * chip, struct msgobj_t * obj);
```

Arguments

chip

pointer to chip state structure

obj

pointer to message object structure

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_standard_mask

Name

`sja1000p_standard_mask` — setup of mask for message filtering

Synopsis

```
int sja1000p_standard_mask (struct chip_t * chip, unsigned short code, unsigned short mask);
```

Arguments

chip

pointer to chip state structure

code

can message acceptance code

mask
can message acceptance mask

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_clear_objects

Name

`sja1000p_clear_objects` — clears state of all message object residing in chip

Synopsis

```
int sja1000p_clear_objects (struct chip_t * chip);
```

Arguments

chip
pointer to chip state structure

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_config_irqs

Name

`sja1000p_config_irqs` — tunes chip hardware interrupt delivery

Synopsis

```
int sja1000p_config_irqs (struct chip_t * chip, short irqs);
```

Arguments

chip
pointer to chip state structure

irqs
requested chip IRQ configuration

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_irq_write_handler

Name

`sja1000p_irq_write_handler` — part of ISR code responsible for transmit events

Synopsis

```
void sja1000p_irq_write_handler (struct chip_t * chip, struct msgobj_t * obj);
```

Arguments

chip

pointer to chip state structure

obj

pointer to attached queue description

Description

The main purpose of this function is to read message from attached queues and transfer message contents into CAN controller chip. This subroutine is called by `sja1000p_irq_write_handle` for transmit events.

File

src/sja1000p.c

sja1000p_irq_handler

Name

`sja1000p_irq_handler` — interrupt service routine

Synopsis

```
can_irqreturn_t sja1000p_irq_handler (int irq, void * dev_id, struct pt_regs * regs);
```

Arguments

irq

interrupt vector number, this value is system specific

dev_id

driver private pointer registered at time of `request_irq` call. The CAN driver uses this pointer to store relationship of interrupt to chip state structure - *struct* chip_t

regs

system dependent value pointing to registers stored in exception frame

Description

Interrupt handler is activated when state of CAN controller chip changes, there is message to be read or there is more space for new messages or error occurs. The receive events results in reading of the message from CAN controller chip and distribution of message through attached message queues.

File

src/sja1000p.c

sja1000p_wakeup_tx

Name

`sja1000p_wakeup_tx` — wakeups TX processing

Synopsis

```
int sja1000p_wakeup_tx (struct chip_t * chip, struct msgobj_t * obj);
```

Arguments

chip
: pointer to chip state structure

obj
: pointer to message object structure

Return Value

negative value reports error.

File

src/sja1000p.c

2.6.4. CAN Queues Common Structures and Functions

This part of the driver implements basic CAN queues infrastructure. It is written as much generic as possible and then specialization for each category of CAN queues clients is implemented in separate subsystem. The only synchronization mechanism required from target system are spin-lock synchronization and atomic bit manipulation. Locked sections are narrowed to the short operations. Even can message 8 bytes movement is excluded from the locked sections of the code.

struct canque_slot_t

Name

`struct canque_slot_t` — one CAN message slot in the CAN FIFO queue

Synopsis

```
struct canque_slot_t {
  struct canque_slot_t * next;
  unsigned long slot_flags;
  struct canmsg_t msg;
};
```

Members

next

pointer to the next/younger slot

slot_flags

space for flags and optional command describing action associated with slot data

msg

space for one CAN message

Description

This structure is used to store CAN messages in the CAN FIFO queue.

struct canque_fifo_t

Name

`struct canque_fifo_t` — CAN FIFO queue representation

Synopsis

```
struct canque_fifo_t {
  unsigned long fifo_flags;
  unsigned long error_code;
  struct canque_slot_t * head;
  struct canque_slot_t ** tail;
  struct canque_slot_t * flist;
  struct canque_slot_t * entry;
  can_spinlock_t fifo_lock;
  int slotsnr;
};
```

Members

fifo_flags

this field holds global flags describing state of the FIFO. `CAN_FIFOF_ERROR` is set when some error condition occurs. `CAN_FIFOF_ERR2BLOCK` defines, that error should lead to the FIFO block state. `CAN_FIFOF_BLOCK` state blocks insertion of the next messages. `CAN_FIFOF_OVERRUN` attempt to acquire new slot, when FIFO is full. `CAN_FIFOF_FULL` indicates FIFO full state. `CAN_FIFOF_EMPTY` indicates no allocated slot in the FIFO. `CAN_FIFOF_DEAD` condition indication. Used when FIFO is beeing destroyed.

error_code

futher description of error condition

head

pointer to the FIFO head, oldest slot

tail

pointer to the location, where pointer to newly inserted slot should be added

flist

pointer to list of the free slots associated with queue

entry
: pointer to the memory allocated for the list slots.

fifo_lock
: the lock to ensure atomicity of slot manipulation operations.

slotsnr
: number of allocated slots

Description

This structure represents CAN FIFO queue. It is implemented as a single linked list of slots prepared for processing. The empty slots are stored in single linked list (*`flist`*).

canque_fifo_get_inslot

Name

`canque_fifo_get_inslot` — allocate slot for the input of one CAN message

Synopsis

`int` **`canque_fifo_get_inslot`** `(struct canque_fifo_t *` *`fifo`*`, struct canque_slot_t **` *`slotp`*`, int` *`cmd`*`);`

Arguments

`fifo`
: pointer to the FIFO structure

`slotp`
: pointer to location to store pointer to the allocated slot.

`cmd`
: optional command associated with allocated slot.

Return Value

The function returns negative value if there is no free slot in the FIFO queue.

canque_fifo_put_inslot

Name

`canque_fifo_put_inslot` — releases slot to further processing

Synopsis

`int` **`canque_fifo_put_inslot`** `(struct canque_fifo_t *` *`fifo`*`, struct canque_slot_t *` *`slot`*`);`

Arguments

`fifo`
: pointer to the FIFO structure

`slot`
: pointer to the slot previously acquired by `canque_fifo_get_inslot`.

Return Value

The nonzero return value indicates, that the queue was empty before call to the function. The caller should wake-up output side of the queue.

canque_fifo_abort_inslot

Name

`canque_fifo_abort_inslot` — release and abort slot

Synopsis

```
int canque_fifo_abort_inslot (struct canque_fifo_t * fifo, struct canque_slot_t * slot);
```

Arguments

fifo
: pointer to the FIFO structure

slot
: pointer to the slot previously acquired by `canque_fifo_get_inslot`.

Return Value

The nonzero value indicates, that fifo was full

canque_fifo_test_outslot

Name

`canque_fifo_test_outslot` — test and get ready slot from the FIFO

Synopsis

```
int canque_fifo_test_outslot (struct canque_fifo_t * fifo, struct canque_slot_t ** slotp);
```

Arguments

fifo
: pointer to the FIFO structure

slotp
: pointer to location to store pointer to the oldest slot from the FIFO.

Return Value

The negative value indicates, that queue is empty. The positive or zero value represents command stored into slot by the call to the function `canque_fifo_get_inslot`. The successfully acquired FIFO output slot has to be released by the call `canque_fifo_free_outslot` or `canque_fifo_again_outslot`.

canque_fifo_free_outslot

Name

`canque_fifo_free_outslot` — free processed FIFO slot

Synopsis

```
int canque_fifo_free_outslot (struct canque_fifo_t * fifo, struct canque_slot_t * slot);
```

Arguments

fifo
: pointer to the FIFO structure

slot
: pointer to the slot previously acquired by `canque_fifo_test_outslot`.

Return Value

The returned value informs about FIFO state change. The mask `CAN_FIFOF_FULL` indicates, that the FIFO was full before the function call. The mask `CAN_FIFOF_EMPTY` informs, that last ready slot has been processed.

canque_fifo_again_outslot

Name

`canque_fifo_again_outslot` — interrupt and postpone processing of the slot

Synopsis

```
int canque_fifo_again_outslot (struct canque_fifo_t * fifo, struct canque_slot_t * slot);
```

Arguments

fifo
: pointer to the FIFO structure

slot
: pointer to the slot previously acquired by `canque_fifo_test_outslot`.

Return Value

The function cannot fail..

struct canque_edge_t

Name

`struct canque_edge_t` — CAN message delivery subsystem graph edge

Synopsis

```
struct canque_edge_t {
  struct canque_fifo_t fifo;
  unsigned long filtid;
  unsigned long filtmask;
  struct list_head inpeers;
  struct list_head outpeers;
  struct list_head activepeers;
  struct canque_ends_t * inends;
  struct canque_ends_t * outends;
  atomic_t edge_used;
  int edge_prio;
  int edge_num;
  #ifdef CAN_WITH_RTLstruct list_head pending_peers;
  unsigned long pending_inops;
  unsigned long pending_outops;
};
```

Members

fifo
: place where primitive `struct` canque_fifo_t FIFO is located.

filtid
: the possible CAN message identifiers filter.

filtmask
: the filter mask, the comparison considers only *`filtid`* bits corresponding to set bits in the *`filtmask`* field.

inpeers
: the lists of all peers FIFOs connected by their input side (*`inends`*) to the same terminal (`struct` canque_ends_t).

outpeers
: the lists of all peers FIFOs connected by their output side (*`outends`*) to the same terminal (`struct` canque_ends_t).

activepeers
: the lists of peers FIFOs connected by their output side (*`outends`*) to the same terminal (`struct` canque_ends_t) with same priority and active state.

inends
: the pointer to the FIFO input side terminal (`struct` canque_ends_t).

outends
: the pointer to the FIFO output side terminal (`struct` canque_ends_t).

edge_used
: the atomic usage counter, mainly used for safe destruction of the edge.

edge_prio
: the assigned queue priority from the range 0 to `CANQUEUE_PRIO_NR-1`

edge_num
: edge sequential number intended for debugging purposes only

pending_peers
: edges with pending delayed events (RTL->Linux calls)

pending_inops
: bitmask of pending operations

pending_outops
: bitmask of pending operations

Description

This structure represents one direction connection from messages source (*inends*) to message consumer (*outends*) fifo ends hub. The edge contains &struct canque_fifo_t for message fifo implementation.

struct canque_ends_t

Name

`struct canque_ends_t` — CAN message delivery subsystem graph vertex (FIFO ends)

Synopsis

```
struct canque_ends_t {
  unsigned long ends_flags;
  struct list_head * active;
  struct list_head idle;
  struct list_head inlist;
  struct list_head outlist;
  can_spinlock_t ends_lock;
  void (* notify (struct canque_ends_t *qends, struct canque_edge_t *qedge, int what);
  void * context;
  union endinfo;
  struct list_head dead_peers;
};
```

Members

ends_flags

  this field holds flags describing state of the ENDS structure.

active

  the array of the lists of active edges directed to the ends structure with ready messages. The array is indexed by the edges priorities.

idle

  the list of the edges directed to the ends structure with empty FIFOs.

inlist

  the list of outgoing edges input sides.

outlist

  the list of all incoming edges output sides. Each of there edges is listed on one of *active* or *idle* lists.

ends_lock

  the lock synchronizing operations between threads accessing same ends structure.

notify

  pointer to notify procedure. The next state changes are notified. `CANQUEUE_NOTIFY_EMPTY` (out->in call) - all slots are processed by FIFO out side. `CANQUEUE_NOTIFY_SPACE` (out->in call) - full state negated => there is space for new message. `CANQUEUE_NOTIFY_PROC` (in->out call) - empty state negated => out side is requested to process slots. `CANQUEUE_NOTIFY_NOU` (both) - notify, that the last user has released the edge usage called with some lock to prevent edge disappear. `CANQUEUE_NOTIFY_DEAD` (both) - edge is in progress of deletion. `CANQUEUE_NOTIFY_ATACH` (both) - new edge has been attached to end. `CANQUEUE_NOTIFY_FILTCH` (out->in call) - edge filter rules changed `CANQUEUE_NOTIFY_ERROR` (out->in call) - error in messages processing.

context

  space to store ends user specific information

endinfo
: space to store some other ends usage specific informations mainly for waking-up by the notify calls.

dead_peers
: used to chain ends wanting for postponed destruction

Description

Structure represents place to connect edges to for CAN communication entity. The zero, one or more incoming and outgoing edges can be connected to this structure.

canque_notify_inends

Name

`canque_notify_inends` — request to send notification to the input ends

Synopsis

```
void canque_notify_inends (struct canque_edge_t * qedge, int what);
```

Arguments

`qedge`
: pointer to the edge structure

`what`
: notification type

canque_notify_outends

Name

`canque_notify_outends` — request to send notification to the output ends

Synopsis

```
void canque_notify_outends (struct canque_edge_t * qedge, int what);
```

Arguments

`qedge`
: pointer to the edge structure

`what`
: notification type

canque_notify_bothends

Name

`canque_notify_bothends` — request to send notification to the both ends

Synopsis

```
void canque_notify_bothends (struct canque_edge_t * qedge, int what);
```

Arguments

qedge
: pointer to the edge structure

what
: notification type

canque_activate_edge

Name

`canque_activate_edge` — mark output end of the edge as active

Synopsis

```
void canque_activate_edge (struct canque_ends_t * inends, struct canque_edge_t * qedge);
```

Arguments

inends
: input side of the edge

qedge
: pointer to the edge structure

Description

Function call moves output side of the edge from idle onto active edges list.

canque_filtid2internal

Name

`canque_filtid2internal` — converts message ID and filter flags into internal format

Synopsis

```
unsigned int canque_filtid2internal (unsigned long id, int filtflags);
```

Arguments

id
: CAN message 11 or 29 bit identifier

filtflags
: CAN message flags

Description

This function maps message ID and `MSG_RTR`, `MSG_EXT` and `MSG_LOCAL` into one 32 bit number

canque_fifo_flush_slots

Name

`canque_fifo_flush_slots` — free all ready slots from the FIFO

Synopsis

```
int canque_fifo_flush_slots (struct canque_fifo_t * fifo);
```

Arguments

fifo
: pointer to the FIFO structure

Description

The caller should be prepared to handle situations, when some slots are held by input or output side slots processing. These slots cannot be flushed or their processing interrupted.

Return Value

The nonzero value indicates, that queue has not been empty before the function call.

canque_fifo_init_slots

Name

`canque_fifo_init_slots` — initializes slot chain of one CAN FIFO

Synopsis

```
int canque_fifo_init_slots (struct canque_fifo_t * fifo);
```

Arguments

fifo
: pointer to the FIFO structure

Return Value

The negative value indicates, that there is no memory to allocate space for the requested number of the slots.

canque_get_inslot

Name

`canque_get_inslot` — finds one outgoing edge and allocates slot from it

Synopsis

```
int canque_get_inslot (struct canque_ends_t * qends, struct canque_edge_t ** qedgep, struct canque_slot_t
** slotp, int cmd);
```

Arguments

qends
: ends structure belonging to calling communication object

qedgep
: place to store pointer to found edge

slotp
: place to store pointer to allocated slot

cmd
: command type for slot

Description

Function looks for the first non-blocked outgoing edge in *qends* structure and tries to allocate slot from it.

Return Value

If there is no usable edge or there is no free slot in edge negative value is returned.

canque_get_inslot4id

Name

`canque_get_inslot4id` — finds best outgoing edge and slot for given ID

Synopsis

```
int canque_get_inslot4id (struct canque_ends_t * qends, struct canque_edge_t ** qedgep, struct
canque_slot_t ** slotp, int cmd, unsigned long id, int prio);
```

Arguments

qends
: ends structure belonging to calling communication object

qedgep
: place to store pointer to found edge

slotp
: place to store pointer to allocated slot

cmd
: command type for slot

id
: communication ID of message to send into edge

prio
: optional priority of message

Description

Function looks for the non-blocked outgoing edge accepting messages with given ID. If edge is found, slot is allocated from that edge. The edges with non-zero mask are preferred over edges open to all messages. If more edges with mask accepts given message ID, the edge with highest priority below or equal to required priority is selected.

Return Value

If there is no usable edge or there is no free slot in edge negative value is returned.

canque_put_inslot

Name

`canque_put_inslot` — schedules filled slot for processing

Synopsis

```
int canque_put_inslot (struct canque_ends_t * qends, struct canque_edge_t * qedge, struct canque_slot_t
* slot);
```

Arguments

qends
: ends structure belonging to calling communication object

qedge
: edge slot belong to

slot
: pointer to the prepared slot

Description

Puts slot previously acquired by `canque_get_inslot` or `canque_get_inslot4id` function call into FIFO queue and activates edge processing if needed.

Return Value

Positive value informs, that activation of output end has been necessary

canque_abort_inslot

Name

`canque_abort_inslot` — aborts preparation of the message in the slot

Synopsis

```
int canque_abort_inslot (struct canque_ends_t * qends, struct canque_edge_t * qedge, struct canque_slot_t
* slot);
```

Arguments

`qends`
: ends structure belonging to calling communication object

`qedge`
: edge slot belong to

`slot`
: pointer to the previously allocated slot

Description

Frees slot previously acquired by `canque_get_inslot` or `canque_get_inslot4id` function call. Used when message copying into slot fails.

Return Value

Positive value informs, that queue full state has been negated.

canque_filter_msg2edges

Name

`canque_filter_msg2edges` — sends message into all edges which accept its ID

Synopsis

```
int canque_filter_msg2edges (struct canque_ends_t * qends, struct canmsg_t * msg);
```

Arguments

`qends`
: ends structure belonging to calling communication object

`msg`
: pointer to CAN message

Description

Sends message to all outgoing edges connected to the given ends, which accepts message communication ID.

Return Value

Returns number of edges message has been send to

canque_test_outslot

Name

`canque_test_outslot` — test and retrieve ready slot for given ends

Synopsis

```
int canque_test_outslot (struct canque_ends_t * qends, struct canque_edge_t ** qedgep, struct canque_slot_t
** slotp);
```

Arguments

qends
: ends structure belonging to calling communication object

qedgep
: place to store pointer to found edge

slotp
: place to store pointer to received slot

Description

Function takes highest priority active incoming edge and retrieves oldest ready slot from it.

Return Value

Negative value informs, that there is no ready output slot for given ends. Positive value is equal to the command slot has been allocated by the input side.

canque_free_outslot

Name

`canque_free_outslot` — frees processed output slot

Synopsis

```
int canque_free_outslot (struct canque_ends_t * qends, struct canque_edge_t * qedge, struct canque_slot_t
* slot);
```

Arguments

qends
: ends structure belonging to calling communication object

qedge
: edge slot belong to

slot
: pointer to the processed slot

Description

Function releases processed slot previously acquired by `canque_test_outslot` function call.

Return Value

Return value informs if input side has been notified to know about change of edge state

canque_again_outslot

Name

`canque_again_outslot` — reschedule output slot to process it again later

Synopsis

```
int canque_again_outslot (struct canque_ends_t * qends, struct canque_edge_t * qedge, struct canque_slot_t
* slot);
```

Arguments

qends
: ends structure belonging to calling communication object

qedge
: edge slot belong to

slot
: pointer to the slot for re-processing

Description

Function reschedules slot previously acquired by `canque_test_outslot` function call for second time processing.

Return Value

Function cannot fail.

canque_set_filt

Name

`canque_set_filt` — sets filter for specified edge

Synopsis

```
int canque_set_filt (struct canque_edge_t * qedge, unsigned long filtid, unsigned long filtmask,
int filtflags);
```

Arguments

qedge
: pointer to the edge

filtid
: ID to set for the edge

filtmask
: mask used for ID match check

filtflags
: required filer flags

Return Value

Negative value is returned if edge is in the process of delete.

canque_flush

Name

`canque_flush` — fluesh all ready slots in the edge

Synopsis

```
int canque_flush (struct canque_edge_t * qedge);
```

Arguments

qedge

pointer to the edge

Description

Tries to flush all allocated slots from the edge, but there could exist some slots associated to edge which are processed by input or output side and cannot be flushed at this moment.

Return Value

The nonzero value indicates, that queue has not been empty before the function call.

canqueue_ends_init_gen

Name

`canqueue_ends_init_gen` — subsystem independent routine to initialize ends state

Synopsis

```
int canqueue_ends_init_gen (struct canque_ends_t * qends);
```

Arguments

qends

pointer to the ends structure

Return Value

Cannot fail.

canqueue_connect_edge

Name

`canqueue_connect_edge` — connect edge between two communication entities

Synopsis

```
int canqueue_connect_edge (struct canque_edge_t * qedge, struct canque_ends_t * inends, struct
canque_ends_t * outends);
```

Arguments

qedge
: pointer to edge

inends
: pointer to ends the input of the edge should be connected to

outends
: pointer to ends the output of the edge should be connected to

Return Value

Negative value informs about failed operation.

canqueue_disconnect_edge

Name

`canqueue_disconnect_edge` — disconnect edge from communicating entities

Synopsis

```
int canqueue_disconnect_edge (struct canque_edge_t * qedge);
```

Arguments

qedge
: pointer to edge

Return Value

Negative value means, that edge is used by somebody other and cannot be disconnected. Operation has to be delayed.

canqueue_block_inlist

Name

`canqueue_block_inlist` — block slot allocation of all outgoing edges of specified ends

Synopsis

```
void canqueue_block_inlist (struct canque_ends_t * qends);
```

Arguments

qends
: pointer to ends structure

canqueue_block_outlist

Name

`canqueue_block_outlist` — block slot allocation of all incoming edges of specified ends

Synopsis

```
void canqueue_block_outlist (struct canque_ends_t * qends);
```

Arguments

qends
: pointer to ends structure

canqueue_ends_kill_inlist

Name

`canqueue_ends_kill_inlist` — sends request to die to all outgoing edges

Synopsis

```
int canqueue_ends_kill_inlist (struct canque_ends_t * qends, int send_rest);
```

Arguments

qends
: pointer to ends structure

send_rest
: select, whether already allocated slots should be processed by FIFO output side

Return Value

Non-zero value means, that not all edges could be immediately disconnected and that ends structure memory release has to be delayed

canqueue_ends_kill_outlist

Name

`canqueue_ends_kill_outlist` — sends request to die to all incoming edges

Synopsis

```
int canqueue_ends_kill_outlist (struct canque_ends_t * qends);
```

Arguments

qends
: pointer to ends structure

Return Value

Non-zero value means, that not all edges could be immediately disconnected and that ends structure memory release has to be delayed

2.6.5. CAN Queues Kernel Specific Functions

canqueue_notify_kern

Name

`canqueue_notify_kern` — notification callback handler for Linux userspace clients

Synopsis

```
void canqueue_notify_kern (struct canque_ends_t * qends, struct canque_edge_t * qedge, int what);
```

Arguments

qends
: pointer to the callback side ends structure

qedge
: edge which invoked notification

what
: notification type

Description

The notification event is handled directly by call of this function except case, when called from RT-Linux context in mixed mode Linux/RT-Linux compilation. It is not possible to directly call Linux kernel synchronization primitives in such case. The notification request is postponed and signaled by *pending_inops* flags by call `canqueue_rtl2lin_check_and_pen` function. The edge reference count is increased until until all pending notifications are processed.

canqueue_ends_init_kern

Name

`canqueue_ends_init_kern` — Linux userspace clients specific ends initialization

Synopsis

```
int canqueue_ends_init_kern (struct canque_ends_t * qends);
```

Arguments

qends
: pointer to the callback side ends structure

canque_get_inslot4id_wait_kern

Name

`canque_get_inslot4id_wait_kern` — find or wait for best outgoing edge and slot for given ID

Synopsis

```
int canque_get_inslot4id_wait_kern (struct canque_ends_t * qends, struct canque_edge_t ** qedgep,
struct canque_slot_t ** slotp, int cmd, unsigned long id, int prio);
```

Arguments

qends
: ends structure belonging to calling communication object

qedgep
: place to store pointer to found edge

slotp
: place to store pointer to allocated slot

cmd
: command type for slot

id
: communication ID of message to send into edge

prio
: optional priority of message

Description

Same as `canque_get_inslot4id`, except, that it waits for free slot in case, that queue is full. Function is specific for Linux userspace clients.

Return Value

If there is no usable edge negative value is returned.

canque_get_outslot_wait_kern

Name

`canque_get_outslot_wait_kern` — receive or wait for ready slot for given ends

Synopsis

```
int canque_get_outslot_wait_kern (struct canque_ends_t * qends, struct canque_edge_t ** qedgep,
struct canque_slot_t ** slotp);
```

Arguments

qends
: ends structure belonging to calling communication object

qedgep
: place to store pointer to found edge

slotp

place to store pointer to received slot

Description

The same as `canque_test_outslot`, except it waits in the case, that there is no ready slot for given ends. Function is specific for Linux userspace clients.

Return Value

Negative value informs, that there is no ready output slot for given ends. Positive value is equal to the command slot has been allocated by the input side.

canque_sync_wait_kern

Name

`canque_sync_wait_kern` — wait for all slots processing

Synopsis

```
int canque_sync_wait_kern (struct canque_ends_t * qends, struct canque_edge_t * qedge);
```

Arguments

qends

ends structure belonging to calling communication object

qedge

pointer to edge

Description

Functions waits for ends transition into empty state.

Return Value

Positive value indicates, that edge empty state has been reached. Negative or zero value informs about interrupted wait or other problem.

canque_fifo_init_kern

Name

`canque_fifo_init_kern` — initialize one CAN FIFO

Synopsis

```
int canque_fifo_init_kern (struct canque_fifo_t * fifo, int slotsnr);
```

Arguments

fifo

pointer to the FIFO structure

slotsnr
number of requested slots

Return Value

The negative value indicates, that there is no memory to allocate space for the requested number of the slots.

canque_fifo_done_kern

Name

`canque_fifo_done_kern` — frees slots allocated for CAN FIFO

Synopsis

```
int canque_fifo_done_kern (struct canque_fifo_t * fifo);
```

Arguments

fifo
pointer to the FIFO structure

canque_new_edge_kern

Name

`canque_new_edge_kern` — allocate new edge structure in the Linux kernel context

Synopsis

```
struct canque_edge_t * canque_new_edge_kern (int slotsnr);
```

Arguments

slotsnr
required number of slots in the newly allocated edge structure

Return Value

Returns pointer to allocated slot structure or `NULL` if there is not enough memory to process operation.

canqueue_ends_dispose_kern

Name

`canqueue_ends_dispose_kern` — finalizing of the ends structure for Linux kernel clients

Synopsis

```
int canqueue_ends_dispose_kern (struct canque_ends_t * qends, int sync);
```

Arguments

qends
: pointer to ends structure

sync
: flag indicating, that user wants to wait for processing of all remaining messages

Return Value

Function should be designed such way to not fail.

2.6.6. CAN Queues RT-Linux Specific Functions

canqueue_rtl2lin_check_and_pend

Name

`canqueue_rtl2lin_check_and_pend` — postpones edge notification if called from RT-Linux

Synopsis

```
int canqueue_rtl2lin_check_and_pend (struct canque_ends_t * qends, struct canque_edge_t * qedge,
int what);
```

Arguments

qends
: notification target ends

qedge
: edge delivering notification

what
: notification type

Return Value

if called from Linux context, returns 0 and lefts notification processing on caller responsibility. If called from RT-Linux contexts, schedules postponed event delivery and returns 1

canque_get_inslot4id_wait_rtl

Name

`canque_get_inslot4id_wait_rtl` — find or wait for best outgoing edge and slot for given ID

Synopsis

```
int canque_get_inslot4id_wait_rtl (struct canque_ends_t * qends, struct canque_edge_t ** qedgep,
struct canque_slot_t ** slotp, int cmd, unsigned long id, int prio);
```

Arguments

qends
: ends structure belonging to calling communication object

qedgep
: place to store pointer to found edge

slotp
: place to store pointer to allocated slot

cmd
: command type for slot

id
: communication ID of message to send into edge

prio
: optional priority of message

Description

Same as `canque_get_inslot4id`, except, that it waits for free slot in case, that queue is full. Function is specific for Linux userspace clients.

Return Value

If there is no usable edge negative value is returned.

canque_get_outslot_wait_rtl

Name

`canque_get_outslot_wait_rtl` — receive or wait for ready slot for given ends

Synopsis

```
int canque_get_outslot_wait_rtl (struct canque_ends_t * qends, struct canque_edge_t ** qedgep,
struct canque_slot_t ** slotp);
```

Arguments

qends
: ends structure belonging to calling communication object

qedgep
: place to store pointer to found edge

slotp
: place to store pointer to received slot

Description

The same as `canque_test_outslot`, except it waits in the case, that there is no ready slot for given ends. Function is specific for Linux userspace clients.

Return Value

Negative value informs, that there is no ready output slot for given ends. Positive value is equal to the command slot has been allocated by the input side.

canque_sync_wait_rtl

Name

`canque_sync_wait_rtl` — wait for all slots processing

Synopsis

```
int canque_sync_wait_rtl (struct canque_ends_t * qends, struct canque_edge_t * qedge);
```

Arguments

qends
: ends structure belonging to calling communication object

qedge
: pointer to edge

Description

Functions waits for ends transition into empty state.

Return Value

Positive value indicates, that edge empty state has been reached. Negative or zero value informs about interrupted wait or other problem.

canque_fifo_init_rtl

Name

`canque_fifo_init_rtl` — initialize one CAN FIFO

Synopsis

```
int canque_fifo_init_rtl (struct canque_fifo_t * fifo, int slotsnr);
```

Arguments

fifo
: pointer to the FIFO structure

slotsnr
: number of requested slots

Return Value

The negative value indicates, that there is no memory to allocate space for the requested number of the slots.

canque_fifo_done_rtl

Name

`canque_fifo_done_rtl` — frees slots allocated for CAN FIFO

Synopsis

```
int canque_fifo_done_rtl (struct canque_fifo_t * fifo);
```

Arguments

`fifo`
: pointer to the FIFO structure

canque_new_edge_rtl

Name

`canque_new_edge_rtl` — allocate new edge structure in the RT-Linux context

Synopsis

```
struct canque_edge_t * canque_new_edge_rtl (int slotsnr);
```

Arguments

`slotsnr`
: required number of slots in the newly allocated edge structure

Return Value

Returns pointer to allocated slot structure or `NULL` if there is not enough memory to process operation.

canqueue_notify_rtl

Name

`canqueue_notify_rtl` — notification callback handler for Linux userspace clients

Synopsis

```
void canqueue_notify_rtl (struct canque_ends_t * qends, struct canque_edge_t * qedge, int what);
```

Arguments

`qends`
: pointer to the callback side ends structure

`qedge`
: edge which invoked notification

what

notification type

canqueue_ends_init_rtl

Name

`canqueue_ends_init_rtl` — RT-Linux clients specific ends initialization

Synopsis

`int **canqueue_ends_init_rtl** (struct canque_ends_t * *qends*);`

Arguments

qends

pointer to the callback side ends structure

canqueue_ends_dispose_rtl

Name

`canqueue_ends_dispose_rtl` — finalizing of the ends structure for Linux kernel clients

Synopsis

`int **canqueue_ends_dispose_rtl** (struct canque_ends_t * *qends*, int *sync*);`

Arguments

qends

pointer to ends structure

sync

flag indicating, that user wants to wait for processing of all remaining messages

Return Value

Function should be designed such way to not fail.

canqueue_rtl_initialize

Name

`canqueue_rtl_initialize` — initialization of global RT-Linux specific features

Synopsis

`void **canqueue_rtl_initialize** ( *void*);`

Arguments

void

no arguments

canqueue_rtl_done

Name

`canqueue_rtl_done` — finalization of glopal RT-Linux specific features

Synopsis

```
void canqueue_rtl_done ( void);
```

Arguments

void

no arguments

2.6.7. CAN Queues CAN Chips Specific Functions

canqueue_notify_chip

Name

`canqueue_notify_chip` — notification callback handler for CAN chips ends of queues

Synopsis

```
void canqueue_notify_chip (struct canque_ends_t * qends, struct canque_edge_t * qedge, int what);
```

Arguments

qends

pointer to the callback side ends structure

qedge

edge which invoked notification

what

notification type

Description

This function has to deal with more possible cases. It can be called from the kernel or interrupt context for Linux only compilation of driver. The function can be called from kernel context or RT-Linux thread context for mixed mode Linux/RT-Linux compilation.

canqueue_ends_init_chip

Name

`canqueue_ends_init_chip` — CAN chip specific ends initialization

Synopsis

```
int canqueue_ends_init_chip (struct canque_ends_t * qends, struct chip_t * chip, struct msgobj_t
* obj);
```

Arguments

qends
: pointer to the ends structure

chip
: pointer to the corresponding CAN chip structure

obj
: pointer to the corresponding message object structure

canqueue_ends_done_chip

Name

`canqueue_ends_done_chip` — finalizing of the ends structure for CAN chips

Synopsis

```
int canqueue_ends_done_chip (struct canque_ends_t * qends);
```

Arguments

qends
: pointer to ends structure

Return Value

Function should be designed such way to not fail.

2.6.8. CAN Boards and Chip Setup specific Functions

can_checked_malloc

Name

`can_checked_malloc` — memory allocation with registering of requested blocks

Synopsis

```
void * can_checked_malloc (size_t size);
```

Arguments

size

size of the requested block

Description

The function is used in the driver initialization phase to catch possible memory leaks for future driver finalization or case, that driver initialization fail.

Return Value

pointer to the allocated memory or NULL in the case of fail

can_checked_free

Name

`can_checked_free` — free memory allocated by `can_checked_malloc`

Synopsis

```
int can_checked_free (void * address_p);
```

Arguments

address_p

pointer to the memory block

can_del_mem_list

Name

`can_del_mem_list` — check for stale memory allocations at driver finalization

Synopsis

```
int can_del_mem_list ( void);
```

Arguments

void

no arguments

Description

Checks, if there are still some memory blocks allocated and releases memory occupied by such blocks back to the system

can_request_io_region

Name

`can_request_io_region` — request IO space region

Synopsis

```
int can_request_io_region (unsigned long start, unsigned long n, const char * name);
```

Arguments

start
: the first IO port address

n
: number of the consecutive IO port addresses

name
: name/label for the requested region

Description

The function hides system specific implementation of the feature.

Return Value

returns positive value (1) in the case, that region could be reserved for the driver. Returns zero (0) if there is collision with other driver or region cannot be taken for some other reason.

can_release_io_region

Name

`can_release_io_region` — release IO space region

Synopsis

```
void can_release_io_region (unsigned long start, unsigned long n);
```

Arguments

start
: the first IO port address

n
: number of the consecutive IO port addresses

can_request_mem_region

Name

`can_request_mem_region` — request memory space region

Synopsis

`int` **`can_request_mem_region`** `(unsigned long` *`start`*`, unsigned long` *`n`*`, const char *` *`name`*`);`

Arguments

`start`

the first memory port physical address

`n`

number of the consecutive memory port addresses

`name`

name/label for the requested region

Description

The function hides system specific implementation of the feature.

Return Value

returns positive value (1) in the case, that region could be reserved for the driver. Returns zero (0) if there is collision with other driver or region cannot be taken for some other reason.

can_release_mem_region

Name

`can_release_mem_region` — release memory space region

Synopsis

`void` **`can_release_mem_region`** `(unsigned long` *`start`*`, unsigned long` *`n`*`);`

Arguments

`start`

the first memory port physical address

`n`

number of the consecutive memory port addresses

can_base_addr_fixup

Name

`can_base_addr_fixup` — relocates board physical memory addresses to the CPU accessible ones

Synopsis

`int` **`can_base_addr_fixup`** `(struct candevice_t *` *`candev`*`, unsigned long` *`new_base`*`);`

Arguments

candev

pointer to the previously filled device/board, chips and message objects structures

new_base

candev new base address

Description

This function adapts base addresses of all structures of one board to the new board base address. It is required for translation between physical and virtual address mappings. This function is prepared to simplify board specific `xxx_request_io` function for memory mapped devices.

register_obj_struct

Name

`register_obj_struct` — registers message object into global array

Synopsis

```
int register_obj_struct (struct msgobj_t * obj, int minorbase);
```

Arguments

obj

the initialized message object being registered

minorbase

wanted minor number, if (-1) automatically selected

Return Value

returns negative number in the case of fail

register_chip_struct

Name

`register_chip_struct` — registers chip into global array

Synopsis

```
int register_chip_struct (struct chip_t * chip, int minorbase);
```

Arguments

chip

the initialized chip structure being registered

minorbase

wanted minor number base, if (-1) automatically selected

Return Value

returns negative number in the case of fail

init_hw_struct

Name

`init_hw_struct` — initializes driver hardware description structures

Synopsis

```
int init_hw_struct ( void);
```

Arguments

`void`
: no arguments

Description

The function `init_hw_struct` is used to initialize the hardware structure.

Return Value

returns negative number in the case of fail

init_device_struct

Name

`init_device_struct` — initializes single CAN device/board

Synopsis

```
int init_device_struct (int card, int * chan_param_idx_p, int * irq_param_idx_p);
```

Arguments

`card`
: index into *`hardware_p`* HW description

`chan_param_idx_p`
: pointer to the index into arrays of the CAN channel parameters

`irq_param_idx_p`
: pointer to the index into arrays of the per CAN channel IRQ parameters

Description

The function builds representation of the one board from parameters provided

in the module parameters arrays

`hw`[card] .. hardware type, `io`[card] .. base IO address, `baudrate`[chan_param_idx] .. per channel baudrate, `minor`[chan_param_idx] .. optional specification of requested channel minor base, `irq`[irq_param_idx] .. one or more board/chips IRQ parameters. The indexes are advanced after consumed parameters if the registration is successful.

The hardware specific operations of the device/board are initialized by call to `init_hwspecops` function. Then board data are initialized by board specific `init_hw_data` function. Then chips and objects representation is build by `init_chip_struct` function. If all above steps are successful, chips and message objects are registered into global arrays.

Return Value

returns negative number in the case of fail

init_chip_struct

Name

`init_chip_struct` — initializes one CAN chip structure

Synopsis

```
int init_chip_struct (struct candevice_t * candev, int chipnr, int irq, long baudrate);
```

Arguments

candev
: pointer to the corresponding CAN device/board

chipnr
: index of the chip in the corresponding device/board structure

irq
: chip IRQ number or (-1) if not appropriate

baudrate
: baudrate in the units of 1Bd

Description

Chip structure is allocated and chip specific operations are filled by call to board specific `init_chip_data` function and generic `init_chipspecops` function. The message objects are generated by calls to `init_obj_struct` function.

Return Value

returns negative number in the case of fail

init_obj_struct

Name

`init_obj_struct` — initializes one CAN message object structure

Synopsis

```
int init_obj_struct (struct candevice_t * candev, struct chip_t * hostchip, int objnr);
```

Arguments

candev
: pointer to the corresponding CAN device/board

hostchip
: pointer to the chip containing this object

objnr
: index of the builded object in the chip structure

Description

The function initializes message object structure and allocates and initializes CAN queue chip ends structure.

Return Value

returns negative number in the case of fail

init_hwspecops

Name

`init_hwspecops` — finds and initializes board/device specific operations

Synopsis

```
int init_hwspecops (struct candevice_t * candev, int * irqnum_p);
```

Arguments

candev
: pointer to the corresponding CAN device/board

irqnum_p
: optional pointer to the number of interrupts required by board

Description

The function searches board *hwname* in the list of supported boards types. The board type specific `board_register` function is used to initialize *hwspecops* operations.

Return Value

returns negative number in the case of fail

init_chipspecops

Name

`init_chipspecops` — fills chip specific operations for board for known chip types

Synopsis

```
int init_chipspecops (struct candevice_t * candev, int chipnr);
```

Arguments

candev
: pointer to the corresponding CAN device/board

chipnr
: index of the chip in the device/board structure

Description

The function fills chip specific operations for next known generic chip types “i82527”, “sja1000”, “sja1000p” (PeliCAN). Other non generic chip types operations has to be initialized in the board specific `init_chip_data` function.

Return Value

returns negative number in the case of fail

can_chip_setup_irq

Name

`can_chip_setup_irq` — attaches chip to the system interrupt processing

Synopsis

```
int can_chip_setup_irq (struct chip_t * chip);
```

Arguments

chip
: pointer to CAN chip structure

Return Value

returns negative number in the case of fail

can_chip_free_irq

Name

`can_chip_free_irq` — unregisters chip interrupt handler from the system

Synopsis

```
void can_chip_free_irq (struct chip_t * chip);
```

Arguments

chip

pointer to CAN chip structure

2.6.9. CAN Boards and Chip Finalization Functions

msgobj_done

Name

`msgobj_done` — destroys one CAN message object

Synopsis

```
void msgobj_done (struct msgobj_t * obj);
```

Arguments

obj

pointer to CAN message object structure

canchip_done

Name

`canchip_done` — destroys one CAN chip representation

Synopsis

```
void canchip_done (struct chip_t * chip);
```

Arguments

chip

pointer to CAN chip structure

candevice_done

Name

`candevice_done` — destroys representation of one CAN device/board

Synopsis

```
void candevice_done (struct candevice_t * candev);
```

Arguments

candev

pointer to CAN device/board structure

canhardware_done

Name

`canhardware_done` — destroys representation of all CAN devices/boards

Synopsis

```
void canhardware_done (struct canhardware_t * canhw);
```

Arguments

canhw

pointer to the root of all CAN hardware representation

2.7. LinCAN Usage Information

2.7.1. Installation Prerequisites

The next basic conditions are necessary for the LinCAN driver usage

- some of supported types of CAN interface boards (high or low speed). Not required for *virtual* board setup.
- cables and at least one device compatible with the board or the second computer with an another CAN interface board. Not required for *virtual* board setup. Even more clients can communicate each with another if *process local* is enabled for real chip driver.
- working Linux system with any recent 2.6.x, 2.4.x or 2.2.x kernel (successfully tested on 2.4.18, 2.4.22, 2.2.19, 2.2.20, 2.2.22, 2.6.0 kernels) or working setup for kernel cross-compilation
- installed native and or target specific development tools (GCC and binutils) and pre-configured kernel sources corresponding to the running kernel or intended target for cross-compilation

Every non-archaic Linux distribution should provide good starting point for the LinCAN driver installation.

If mixed mode compilation for Linux/RT-Linux is required, additional conditions has to be fulfilled:

- RT-Linux version 3.2 or higher is required and RT-Linux enabled Linux kernel sources and configuration has to be prepared. The recommended is use of OCERA Linux/RT-Linux release (http://www.ocera.org).
- RT-Linux real-time `malloc` support. It is already included in the OCERA release. It can be downloaded from OCERA web site for older RT-Linux releases as well (http://www.ocera.org/dow

The RT-Linux specific Makefiles infrastructure is not distributed with the current standard LinCAN distribution yet. Please, download full OCERA-CAN package or retrieve sources from CVS by next command:

```
cvs -d:pserver:anonymous@cvs.ocera.sourceforge.net:/cvsroot/ocera login
```

```
cvs -z3 -d:pserver:anonymous@cvs.ocera.sourceforge.net:/cvsroot/ocera co ocera/components/comm/can
```

2.7.2. Quick Installation Instructions

Change current directory into the LinCAN driver source root directory

```
cd lincan-dir
```

invoke make utility. Just type '**make**' at the command line and driver should compile without errors

```
make
```

If there is problem with compilation, look at first lines produced by 'make' command or store make output in file. More about possible problems and more complex compilation examples is in the next subsection.

Install built LinCAN driver object file (`can.o`) into Linux kernel loadable module directory (`/lib/modules/2.x.y/kernel/drivers/char`). This and next commands needs root privileges to proceed successfully.

```
make install
```

If device filesystem (devfs) is not used on the computer, device nodes have to be created manually.

```
mknod -m666 /dev/can0 c 91 0
mknod -m666 /dev/can1 c 91 1
...
mknod -m666 /dev/can7 c 97 7
```

The parameters, IO address and interrupt line of inserted CAN interface card need to be determined and configured. The manual driver load can be invoked from the command line with parameters similar to example below

```
insmod can.o hw=pip5 irq=4 io=0x8000
```

This commands loads module with selected one card support for PIP5 board type with IO port base address `0x8000` and interrupt line `4`. The full description of module parameters is in the next subsection. If module starts correctly utilities from `utils` subdirectory can be used to test CAN message interchange with device or another computer. The parameters should be written into file `/etc/modules.conf` for subsequent module startup by modprobe command.

Line added to file `/etc/modules.conf` follows

```
options can hw=pip5 irq=4 io=0x8000
```

The module dependencies should be updated by command

```
depmod -a
```

The driver can be now stopped and started by simple **modprobe** command

```
modprobe -r can modprobe can
```

2.7.3. Installation instructions

The LinCAN make solutions tries to fully automate native kernel out of tree module compilation. Make system recurses through kernel `Makefile` to achieve selection of right preprocessor, compiler and linker directives. The description of make targets after make invocation in driver top directory follows

lincan-drv/Makefile (all)

LinCAN driver top makefile

lincan-drv/src/Makefile (default or all -> make_this_module)

Needs to resolve target system kernel sources location. This can be selected manually by uncommenting the `Makefile` definition **KERNEL_LOCATION=/usr/src/linux-2.2.22**. The default behavior is to find the running kernel version and look for path to sources of found kernel version in `/lib/modules/2.x.y/build` directory. If no such directory exists, older version of kernel is assumed and makefile tries the `/usr/src/linux` directory.

lib/modules/2.*x*.*y*/build/Makefile SUBDIRS=.../lincan-drv/src (modules)

The kernel supplied `Makefile` is responsible for defining of right defines for preprocessor, compiler and linker. If the Linux kernel is cross-compiled, Linux kernel sources root `Makefile` needs be edited before Linux kernel compilation. The variable CROSS_COMPILE should contain development tool-chain prefix, for example **arm-linux-**. The Linux kernel make process recurses back into LinCAN driver `src/Makefile`.

lincan-drv/src/Makefile (modules)

This pass starts real LinCAN driver build actions.

If there is problem with automatic build process, the next commands can help to diagnose the problem.

```
make clean make >make.out 2>&1
```

The first lines of file `make.out` indicates auto-detected values and can help with resolving of possible problems.

```
make -C src default ;
make -C utils default ;
make[1]: /scripts/pathdown.sh: Command not found
make[1]: Entering directory '/usr/src/can-0.7.1-pi3.4/src'
echo >.supported_cards.h echo \#define ENABLE_CARD_pip 1 >>.supported_cards.h ; ...
Linux kernel version 2.4.19
echo Linux kernel sources /lib/modules/2.4.19/build
Linux kernel sources /lib/modules/2.4.19/build
echo Module target can.o
Module target can.o
echo Module objects proc.o pip.o pccan.o smartcan.o nsi.o ...
make[2]: Entering directory '/usr/src/linux-2.4.19'
```

The driver size can be decreased by restricting of number of supported types of boards. This can be done by editing of definition for SUPPORTED_CARDS variable.

There is complete description of driver supported parameters.

```
insmod can.o hw='your hardware' irq='irq number' io='io address' <more options>
```

The more values can be specified for *hw*, *irq* and *io* parameters if more cards is used. Values are separated by commas in such case. The *hw* argument can be one of:

- `pip5`, for the pip5 computer by MPL
- `pip6`, for the pip6 computer by MPL
- `pccan-q`, for the PCcan-Q ISA card by KVASER
- `pccan-f`, for the PCcan-F ISA card by KVASER
- `pccan-s`, for the PCcan-S ISA card by KVASER
- `pccan-d`, for the PCcan-D ISA card by KVASER
- `pcican-q`, for the PCIcan-Q PCI card by KVASER (4x SJA1000)
- `pcican-d`, for the PCIcan-D PCI card by KVASER (2x SJA1000)
- `pcican-s`, for the PCIcan-S PCI card by KVASER (1x SJA1000)
- `nsican`, for the CAN104 PC/104 card by NSI
- `cc104`, for the CAN104 PC/104 card by Contemporary Controls
- `aim104`, for the AIM104CAN PC/104 card by Arcom Control Systems
- `pc-i03`, for the PC-I03 ISA card by IXXAT

- `pcm3680`, for the PCM-3680 PC/104 card by Advantech
- `m437`, for the M436 PC/104 card by SECO
- `bfadcan` for sja1000 CAN embedded card made by BFAD GmbH
- `pikronisa` for ISA memory mapped sja1000 CAN card made by PiKRON Ltd.
- `template`, for yet unsupported hardware (you need to edit `src/template.c`)
- `virtual`, virtual/dummy board support for testing of driver and software devices and applications

The lists of values for board hardware type (*hw*) and board base IO address (*io*) parameters have to contain same number of values. If the value of *io* has no meaning for specified hardware type (`virtual` or PCI board), it has to be substituted by `0`.

The number of required *irq* values per board is variable. The `virtual` and PCI board demands no value, most of the other boards requires one *irq* value per each chip/channel.

The <*more options*> can be one or more of:

- *major*=<*nr*>, major specifies the major number of the driver. Default value is 91
- *minor*=<*nr*>, you can specify which base minor number the driver should use for each can channel/chip. Consecutive numbers are taken in the case, that chip supports more communication objects. The values for channels are separated by comas
- *extended*=[1|0], enables automatic switching to extended format if ID>2047, selects extended frames reception for i82527
- *pelican*=[1|0], unused parameter, PeliCAN used by default for sja1000p chips now
- *baudrate*=<*nr*>, baudrate for each channel in step of 1kBd
- *clock_freq*=<*nr*>, the frequency of the CAN quartz for BfaD board
- *stdmask*=<*nr*>, default standard mask for some (i82527) chips
- *extmask*=<*nr*>, default extended mask for some (i82527) chips
- *mo15mask*=<*nr*>, sets the mask for message object 15 (i82527 only)
- *processlocal*=<*nr*>, select post-processing/loop-back of transmitted messages

  0 .. disabled

  1 .. can be enabled by application by FIFO filter setup

  2 .. enabled by default

- *can_rtl_priority*=<*nr*>, select priority of chip worker thread for driver compiled with RT-Linux support

Actual list of supported CAN module parameters and short description can be reached by invocation of the command

```
modinfo can
```

.

2.7.4. Simple Utilities

The simple test utilities can be found in the `utils` subdirectory of the LinCAN driver source subtree. These utilities can be used as base for user programs directly communicating with the LinCAN driver. We do not suggest to build applications directly dependent on the driver operating system specific interface. We suggest to use the VCA API library for communication with the driver which brings higher level of system interface abstraction and ensures compatibility with the future versions of LinCAN driver and RT-Linux driver clone versions. The actual low level RT-Linux API to LinCAN driver closely matches `open`/`close`, `read`/`write` and `ioctl` interface. Only `select` cannot be provided directly by RT-Linux API.

The basic utilities provided with LinCAN driver are:

rxtx

: the simple utility to receive or send message which guides user through operation, the message type, the message ID and the message contents by simple prompts

send
: even more simplistic message sending program

readburst
: the utility for continuous messages reception and printing of the message contents. This utility can be used as an example of the `select` system call usage.

sendburst
: the periodic message generator. Each message is filled by the constant pattern and the message sequence number. This utility can be used for throughput and message drops tests.

can-proxy
: the simple TCP/IP to CAN proxy. The proxy receives simple commands from IP datagrams and processes command sending and state manipulations. Received messages are packed into IP datagrams and send back to the client.

readburst

Name

`readburst` — the utility for continuous messages reception and printing of the message contents

Synopsis

readburst [-d *candev*][-m *mask*][-i *id*][-f *flags*][-w *sec*][-p *prefix*][-V][-h]

Description

The utility **readburst** can be used to monitor or log CAN messages received by one CAN message communication object. Even outgoing transmitted messages can be logged if *process local* is globally or explicitly enabled.

OPTIONS

`-d --device`
: This options selects **readburst** target CAN device. If the option is not specified, default device name `/dev/can0` is used.

`-m --mask`
: This option enables to change default mask accepting all messages to the specified CAN message id mask. The hexadecimal value has to be prefixed by prefix `0x`. Numeric value without any prefix is considered as decimal one.

`-i --id`
: This option specifies CAN message identifier in the acceptance mask. The accepted CAN messages are then printed by **readburst** command. Only bits corresponding to the non-zero bits of acceptance mask are compared. Hexadecimal value has to be prefixed by any prefix `0x`. Numeric value without prefix is considered as decimal one.

`-f --flags`
: Specification of modifiers flags of receiption CAN queur. Hexadecimal value has to be prefixed by prefix `0x`. Numeric value without any prefix is considered as decimal one.

Bit name	Bit number	Mask	Description
`MSG_RTR`	0	0x1	Receive RTR or non-RTR messages

Bit name	Bit number	Mask	Description
MSG_EXT	2	0x4	Receive extended/standard messages
MSG_LOCAL	3	0x8	Receive local or external messages
MSG_RTR_MASK	8	0x100	Take care about MSG_RTR bit else RTR and non-RTR messages are accepted
MSG_EXT_MASK	10	0x400	Take care about MSG_EXT bit else extended and standard messages are accepted
MSG_LOCAL_MASK	11	0x800	Take care about MSG_LOCAL bit else both local and external messages are accepted
MSG_PROCESSLOCAL	9	0x200	Enable processing of the local messages if not explicitly enabled globally or disabled globally.

`-w --wait`
: The number of second the **readburst** waits in the `select` call.

`-p --prefix`
: The *prefix* string can is added at beginning of each printed line. The format specifies `%s` could be used to add device name into prefix.

`-V --version`
: Print command version.

`-h --help`
: Print command usage information

sendburst

Name

`sendburst` — the utility for continuous messages reception and printing of the message contents

Synopsis

sendburst [-d *candev*][-i *id*][-s][-f *flags*][-w *sec*][-b *blocksize*][-c *count*][-p *prefix*][-V][-h]

Description

The utility **sendburst** generates blocks of messages with specified CAN message ID. The burst block of *blocksize* messages is generated and pushed into can device. If *count* is specified, the command stops and exits after *count* of message blocks send.

OPTIONS

`-d --device`
: This options selects **sendburst** target CAN device. If the option is not specified, default device name `/dev/can0` is used.

`-i --id`
: This option specifies which CAN message ID is used for transmitted blocks of messages. Hexadecimal value has to be prefixed by prefix `0x`. Numeric value without any prefix is considered as decimal one.

`-f --flags`

Specification of modifiers flags of the send message. Hexadecimal value has to be prefixed by prefix `0x`. Numeric value without prefix is considered as decimal one.

Bit name	**Bit number**	**Mask**	**Description**
`MSG_RTR`	0	0x1	Generate RTR messages if specified
`MSG_EXT`	2	0x4	Use extended messages identifiers if specified

`-s --sync`

Open device in the synchronous mode. The `send` and `close` blocks until message is sent to to CAN bus.

`-w --wait`

The number of second the **sendburst** waits between sending burst blocks.

`-b --block`

The number of messages in the one burst block. Default value is `10`.

`-c --count`

The number of block send after command invocation. If specified, command finishes and returns after specified number of blocks. If unspecified, the **sendburst** runs for infinite time.

`-p --prefix`

The *`prefix`* string can is added at beginning of each printed line. The format specifies `%s` could be used to add device name into prefix.

`-V --version`

Print command version.

`-h --help`

Print command usage information

Chapter 3. CAN/CANopen

3.1. Virtual CAN API (VCA)

The virtual CAN API is an 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 words we can say that VCA is a common API between the CAN driver and the application threads.

3.1.1. Summary

Name of the component
: Virtual CAN API (VCA)

Author
: Pavel Pisa, Frantisek Vacek

Reviewer
: not validated

Layer
: Low-level, High-level

Version
: 0.2 Beta

Status
: Beta

Dependencies
: Needs CAN driver module for used level.

Release date
: February 2004

3.1.2. Description

A virtual CAN API is an interface used to connect the application threads either with 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.

3.1.3. API / Compatibility

3.1.3.1. VCA API

struct canmsg_t

Name

`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

Name

`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

Name

`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

Name

`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

Name

`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

Name

`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

Name

`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

Name

`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

Name

`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

3.1.4. 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.) (see LinCan CAN driver), but VCA remains always the same. Actually `libvca` does not contain functions like `select` or other functions which can suspend calling thread. This approach makes `libvca` independent on used RT OS synchronization mechanisms.

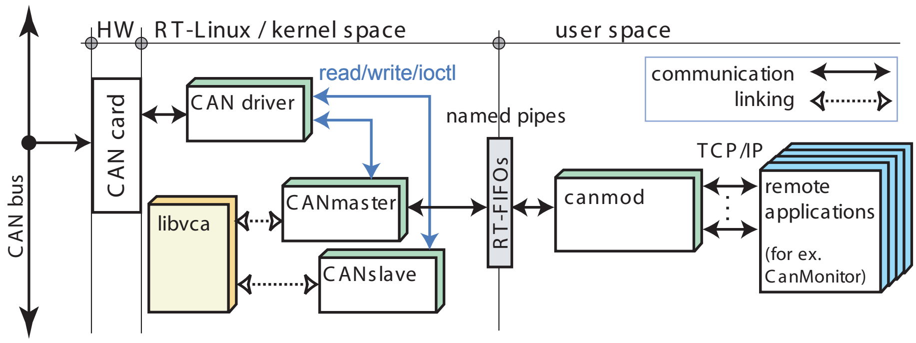

Figure 3-1. Hard real time CAN driver usage example

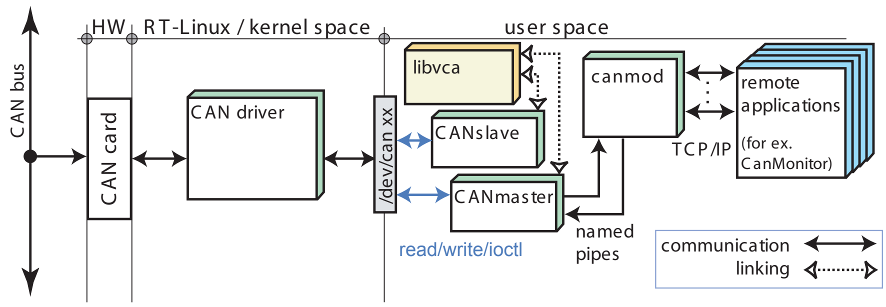

Figure 3-2. Soft real time CAN driver usage example

On figures above we can find, two possible examples of CAN usage in hard real time and also in soft real time. Both figures are describing CANopen VCA exploitation, for more information (see Section 3.2, *CAN device* and canmond).

3.1.5. Tests

Only soft real-time solution was tested yet. Only functionality was tested, no heavy tests were made. All tests were performed during CANmaster and CANslave testing (see CanMonitor tests).

All VCA sources were compiled by GNU C ver. 3.2 and linked with glibc ver. 2.2.5. All VCA sources can be compiled by GNU C ver. 2.96 and above

3.1.6. Examples

Directory `ocera/components/comm/can/canvca/cantest` contains two example programs - *sendcan.c* and *readcan.c*. First one shows the simplest way to send CAN message via VCA. The second one shows, how to read CAN message.

sendcan invocation: `sendcan id byte_1 ... byte_n` n <= 8

Note: If you communicate with CANopen device, you do not forget restart CAN device before communication (`sendcan 0 1 0`).

3.1.7. Installation instructions

All communication components can be compiled issuing `make` command in their directory. Compiled programs can be found in `ocera/components/comm/can/_compiled/bin_utils`.

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

3.2. CAN device

3.2.1. Summary

Name of the component
: CANopen device

Author
: Pavel Pisa
: Frantisek Vacek

Reviewer
: not validated

Layer
: Low-level, High-level, libraries are layer independent

Version
: 0.1 Alfa

Status
: Alfa

Dependencies
: CANmaster and CANslave need CAN driver and libvca installed.

Release date
: February 2004

3.2.2. Description

CANopen device component consists of two programs, `CANmaster` and `CANslave`. Both of them are software solutions based on PDO processor (see Section 3.2.3.1, *PDO processor API*), SDO FSM (Finite State Machine) (see Section 3.2.3.2, *SDO FSM API*), OD (Object dictionary) generated from EDS file (see Section 3.2.3.3, *Object Dictionary API*) and HDS (Hardware Definition Sheet) file.

3.2.3. API / Compatibility

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

3.2.3.1. PDO processor API

This library supports PDO messages processing.

struct vcapdo_mapping_t

Name

`struct vcapdo_mapping_t` — structure representing mapping of sigle object in PDO

Synopsis

```
struct vcapdo_mapping_t {
  vcaod_object_t * object;
  unsigned char   start;
  unsigned char   len;
  sui_dinfo_t * dinfo;
};
```

Members

object

pointer to the mapped object

start

bit offset of object value in PDO

len

bit length of object value in PDO

dinfo

pointer to object data source. Every PDO can be read/written through *`dinfo`* to the OD or to hardware. Actualy there is no other way for PDO object to do that.

struct vcapdolst_object_t

Name

`struct vcapdolst_object_t` — structure representing single PDO object

Synopsis

```
struct vcapdolst_object_t {
  gavl_node_t my_node;
  struct vcaPDOProcessor_t * pdo_processor;
  unsigned long cob_id;
  unsigned char transmition_type;
  unsigned flags;
```

```
    unsigned char sync_every;
    unsigned char sync_counter;
    __u16 inhibit_time;
    __u16 event_timer;
    int mapped_cnt;
    vcapdo_mapping_t * mapped_objects;
    evc_rx_hub_t rx_hub;
};
```

Members

my_node
: structure necessary for storing node in GAVL tree

pdo_processor
: pointer to PDO processor servicing this PDO

cob_id
: COB ID of PDO

transmition_type
: type of PDO transmission according to DS301 table 55

flags
: PDO characteristics and parsed transmission_type

sync_every
: synchronous PDO will be processed every n-th SYNC message

sync_counter
: auxiliary variable for `sync_every`

inhibit_time
: minimum gap between two PDO transmissions (multiples of 100 us)

event_timer
: if nonzero, PDO is transmitted every *event_timer* ms. Valid only in transmission modes 254, 255. (!vcapdoFlagSynchronous && !vcapdoFlagRTROnly)

mapped_cnt
: number of mapped objects in OD

mapped_objects
: array to structures describing mapping details for all mapped objects

rx_hub
: If PDO communication is event driven, appropriate events are connected to this hub

pdo_buff

buffer for received/transmitted PDO

struct vcapdolst_root_t

Name

`struct vcapdolst_root_t` — structure representing root of OD

Synopsis

```
struct vcapdolst_root_t {
  gavl_node_t * my_root;
};
```

Members

my_root
: object dictionary GAVL tree root

struct vcaPDOProcessor_t

Name

`struct vcaPDOProcessor_t` — structure used for PDO communication

Synopsis

```
struct vcaPDOProcessor_t {
  vcapdolst_root_t pdolst_root;
  vcapdo_send_to_can_fnc_t * send_to_can_fnc;
  vcaod_root_t * od_root;
  //vcaDinfoManager_t * dinfo_mgr;
  int node_id;
};
```

Members

pdolst_root
: GAVL containing all defined &vcapdolst_object_t structures

send_to_can_fnc
: PDOProcessor should use this function if it needs to send CAN message during processing

od_root
: pointer to used OD (necessary for PDOs creation and initialization in `vcaPDOProcessor_createP`

dinfo_mgr
: pointer to used DinfoManager (providing HW dinfos during initialization)

node_id
: Node number, optional parameter, if it is specified, default PDO COB-IDs can be assigned if they are not specified in EDS. If *`node_id`* is 0, then it is ignored.

Description

vcaPDOProcessor is responsible for all PDO related tasks in CANopen device

vcaPDOProcessor_init

Name

`vcaPDOProcessor_init` — vcaPDOProcessor constructor

Synopsis

```
void vcaPDOProcessor_init (vcaPDOProcessor_t * proc);
```

Arguments

proc

pointer to PDO processor to work with

vcaPDOProcessor_destroy

Name

`vcaPDOProcessor_destroy` — vcaPDOProcessor destructor

Synopsis

```
void vcaPDOProcessor_destroy (vcaPDOProcessor_t * proc);
```

Arguments

proc

pointer to PDO processor to work with

Description

It releases all PDO objects

vcaPDOProcessor_setOD

Name

`vcaPDOProcessor_setOD` — assign OD to PDOProcessor

Synopsis

```
void vcaPDOProcessor_setOD (vcaPDOProcessor_t * proc, vcaod_root_t * od_root);
```

Arguments

proc

pointer to PDO processor to work with

od_root

assigned root of Object Dictionary

vcaPDOProcessor_createPDOList

Name

`vcaPDOProcessor_createPDOList` — scans OD and creates all valid PDO structures.

Synopsis

```
int vcaPDOProcessor_createPDOList (vcaPDOProcessor_t * proc);
```

Arguments

proc
: pointer to PDO processor to work with

Description

It also deletes previously created PDO structures (if any).

Return

0 or negative number in case of an error

_vcaPDOProcessor_disconnectDinfoLinks

Name

`_vcaPDOProcessor_disconnectDinfoLinks` — disconnect all PDOs and their dinfo structures

Synopsis

```
void _vcaPDOProcessor_disconnectDinfoLinks (vcaPDOProcessor_t * proc);
```

Arguments

proc
: pointer to PDO processor to work with

Description

Actualy it only decrements RefCnt, so only dinfos with RefCnt==1 will be deleted

vcaPDOProcessor_makeDinfoLinks

Name

`vcaPDOProcessor_makeDinfoLinks` — scans defined PDOs and makes necessary data links from PDOs to OD and HW

Synopsis

```
void vcaPDOProcessor_makeDinfoLinks (vcaPDOProcessor_t * proc);
```

Arguments

proc
: pointer to PDO processor to work with

Description

Disconnect all connected dinfos. For each mapped object tries to find appropriate dinfo asking DinfoManager. If DinfoManager returns NULL, thats means, that no HW is connected to this object. In such case function creates dbuff_dinfo for data stored in OD and connect it to mapped PDO.

vcaPDOProcessor_processMsg

Name

`vcaPDOProcessor_processMsg` — tries to process *`msg`*

Synopsis

```
int vcaPDOProcessor_processMsg (vcaPDOProcessor_t * proc, canmsg_t * msg);
```

Arguments

`proc`

pointer to PDO processor to work with

`msg`

CAN msg to proceed

Return

zero if msg is processed

3.2.3.2. SDO FSM API

This library should be used for SDO FSM implementation.

struct vcasdo_fsm_t

Name

`struct vcasdo_fsm_t` — structure representing SDO FSM

Synopsis

```
struct vcasdo_fsm_t {
  unsigned srvcli_cob_id;
  unsigned clisrv_cob_id;
  unsigned node;
  unsigned index, subindex;
  struct timeval last_activity;
  int bytes_to_load;
  unsigned char toggle_bit;
  char is_server;
  char is_uploader;
  int state;
  vcasdo_fsm_state_fnc_t * statefnc;
  int err_no;
  ul_dbuff_t data;
  canmsg_t out_msg;
};
```

Members

srvcli_cob_id
: SDO server-client COB_ID (default is 0x580 + node), port on which master listen

clisrv_cob_id
: SDO client-server COB_ID (default is 0x600 + node), port on which slave listen

node
: CANopen 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)

is_server
: type of FSM client or server (Master or Slave) (internal use)

is_uploader
: processing upload/download in state sdofsmRun, sdofsmDone

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

statefnc
: pointer to the state function (internal use)

err_no
: error number in state `sdofsmError`.

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

out_msg
: if `vcasdo_taste_msg` generates answer, it is stored in the `out_msg`

vcasdo_fsm_upload1

Name

`vcasdo_fsm_upload1` — starts SDO upload using parameters set by previous calling `vcasdo_init_fsm`

Synopsis

```
int vcasdo_fsm_upload1 (vcasdo_fsm_t * fsm);
```

Arguments

`fsm`
: FSM to work with

vcasdo_fsm_download1

Name

`vcasdo_fsm_download1` — starts SDO download using parameters set by previous calling `vcasdo_init_fsm`

Synopsis

```
int vcasdo_fsm_download1 (vcasdo_fsm_t * fsm, ul_dbuff_t * data);
```

Arguments

fsm

FSM to work with

data

pointer to &ul_dbuff_t structure where downloaded data will be stored

vcasdo_read_multiplexor

Name

`vcasdo_read_multiplexor` — reads index and subindex from multiplexor part of CANopen mesage

Synopsis

```
void vcasdo_read_multiplexor (const byte * mult, unsigned * index, unsigned * subindex);
```

Arguments

mult

pointer to the multiplexor part of CANopen mesage

index

pointer to place to store read index

subindex

pointer to place to store read subindex

vcasdo_error_msg

Name

`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

Name

`vcasdo_init_fsm` — init SDO FSM

Synopsis

```
void vcasdo_init_fsm (vcasdo_fsm_t * fsm, unsigned srvcli_cob_id, unsigned clisrv_cob_id, unsigned node);
```

Arguments

fsm
: fsm to init

srvcli_cob_id
: port to use for server->client communication (default 0x850 used if `srvcli_cob_id`==0)

clisrv_cob_id
: port to use for client->server communication (default 0x600 used if `clisrv_cob_id`==0)

node
: number of node on CAN bus to communicate with

vcasdo_destroy_fsm

Name

`vcasdo_destroy_fsm` — frees all SDO FSM resources (destructor)

Synopsis

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

Arguments

fsm
: fsm to destroy

vcasdo_fsm_idle

Name

`vcasdo_fsm_idle` — sets SDO FSM to idle state

Synopsis

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

Arguments

fsm
SDO FSM

vcasdo_fsm_run

Name

`vcasdo_fsm_run` — starts SDO communication protocol for this FSM

Synopsis

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

Arguments

fsm
SDO FSM

vcasdo_fsm_abort

Name

`vcasdo_fsm_abort` — aborts SDO communication for this FSM, fill abort out_msg

Synopsis

```
void vcasdo_fsm_abort (vcasdo_fsm_t * fsm, __u32 abort_code);
```

Arguments

fsm
SDO FSM

abort_code
code to fill to out_msg

vcasdo_fsm_upload

Name

`vcasdo_fsm_upload` — starts upload SDO communication protocol for this FSM

Synopsis

`int vcasdo_fsm_upload (vcasdo_fsm_t * fsm, int node, unsigned index, byte subindex, unsigned srvcli_cob_id, unsigned clisrv_cob_id);`

Arguments

fsm
: SDO FSM

node
: CANopen device node to upload from

index
: uploaded object index

subindex
: uploaded object subindex

srvcli_cob_id
: port to use for server->client communication (default 0x850 used if `srvcli_cob_id`==0)

clisrv_cob_id
: port to use for client->server communication (default 0x600 used if `clisrv_cob_id`==0)

Description

Returns not 0 if `fsm`->out_msg contains CAN message to sent

vcasdo_fsm_download

Name

`vcasdo_fsm_download` — starts download SDO communication protocol for this FSM

Synopsis

`int vcasdo_fsm_download (vcasdo_fsm_t * fsm, ul_dbuff_t * dbuff, int node, unsigned index, byte subindex, unsigned srvcli_cob_id, unsigned clisrv_cob_id);`

Arguments

fsm
: SDO FSM

dbuff
: pointer to a ul_dbuff structure to store received/transmitted data

node
: CANopen device node to upload from

index
: uploaded object index

subindex
: uploaded object subindex

srvcli_cob_id
: port to use for server->client communication (default 0x850 used if `srvcli_cob_id`==0)

clisrv_cob_id
: port to use for client->server communication (default 0x600 used if `clisrv_cob_id`==0)

Description

Returns not 0 if `fsm`->out_msg contains CAN message to sent

vcasdo_fsm_taste_msg

Name

`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

zero if msg is not eatable for FSM

vcasdo_abort_msg

Name

`vcasdo_abort_msg` — translates SDO abort_code to the string message

Synopsis

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

Arguments

abort_code
: abort code

Header

`vcasdo_msg.h`

3.2.3.3. Object Dictionary API

This library supports object values storing and retrieving to/from Object Dictionary.

struct vcaod_root_t

Name

`struct vcaod_root_t` — structure representing root of OD

Synopsis

```
struct vcaod_root_t {
  gavl_node_t * my_root;
};
```

Members

my_root
    object dictionary GAVL tree root

Header

vca_od.h

struct vcaod_object_t

Name

`struct vcaod_object_t` — structure representing single object in OD

Synopsis

```
struct vcaod_object_t {
  gavl_node_t my_node;
  unsigned index;
  int subindex;
  unsigned char data_type;
  unsigned object_type;
  int access;
  unsigned flags;
  char * name;
  struct vcaod_object_t * subobjects;
  int subcnt;
  vcaod_dbuff_t value;
  int valcnt;
  sui_dinfo_t * dinfo;
};
```

Members

my_node
    structure neccessary for storing node in GAVL tree, is NULL for subindicies

index
    index of object

subindex
    subindex of subobject or -1 if object is not subobject

data_type
    can be one of (BOOLEAN, INTEGER8, ...)

object_type
    type of object (DOMAIN=2, DEFTYPE=5, DEFSTRUCT=6, VAR=7, ARRAY=8, RECORD=9)

access
    access attributes (RW, WO, RO, CONST)

flags

flags can be: `VCAOD_OBJECT_FLAG_MANDATORY` object is mandatory/optional, `VCAOD_OBJECT_FLA` object is supposed to be PDO mapped, `VCAOD_OBJECT_FLAG_WEAK_DINFO` *`dinfo`* is weak pointer

name

textual name of object

subobjects

pointer to array of subobjects (definition==DEFSTRUCT, RECORD) or NULL

subcnt

number of subobjects

value

object values (definition==ARRAY) or single value (other definitions). If definition==ARRAY all values have the same length and they are stored sequently in `value`

valcnt

number of values (definition==ARRAY)

dinfo

If object is PDO mapped or coming from HW, PDOProcessor holds reference to dinfo object used for data transfer. In such a case only weak pointer to dinfo is stored in OD object dinfo parameter. Weak in this context means that dinfo object clears this reference Weak pointer is in OD to provide also SDO accesibility to such an object. There are two possibilities on the SDO request. 1. object is PDO mapped, so it is acessed using weak_dinfo, 2. object is not PDO mapped, so it is accesed using functions `vcaod_get_value` and `vcaod_set_value`

Header

vca_od.h

_vcaod_find_object

Name

`_vcaod_find_object` — finds object in OD. This function is not a part of the SDO API

Synopsis

```
vcaod_object_t* _vcaod_find_object (vcaod_root_t * odroot, unsigned ix, unsigned subix, __u32 *
abort_code);
```

Arguments

`odroot`

object dictionary

`ix`

object index

`subix`

object subindex, ignored if object does not have subobjects

`abort_code`

Pointer to the abort code in case of an ERROR. It can be NULL, than it is ignored. Abort codes are defined in CANopen standart 301 and can be translated to text calling `vcasdo_abort_msg`.

Returns

found object or NULL

Header

vca_od.h

vcaod_get_value

Name

`vcaod_get_value` — reads object value from Object Dictionary and copies them to caller buffer

Synopsis

```
int vcaod_get_value (vcaod_root_t * odroot, unsigned ix, unsigned subix, void * buff, int len,
__u32 * abort_code);
```

Arguments

odroot
: object dictionary

ix
: object index

subix
: object subindex, ignored if object does not have subobjects

buff
: buffer to write requested data

len
: length of the buffer

abort_code
: Pointer to the abort code in case of an ERROR. It can be NULL, than it is ignored. Abort codes are defined in CANopen standart 301 and can be translated to text calling `vcasdo_abort_msg`.

Returns

actual length of object in bytes negative value in case of an error

Header

vca_od.h

vcaod_set_value

Name

`vcaod_set_value` — copies object value from caller's buffer to Object Dictionary

Synopsis

```
int vcaod_set_value (vcaod_root_t * odroot, unsigned ix, unsigned subix, const void * buff, int
len, __u32 * abort_code);
```

Arguments

odroot
: object dictionary

ix
: object index

subix
: object subindex, ignored if object does not have subobjects

buff
: buffer containing written data

len
: length of the data

abort_code
: area to fill the abort code in case of an ERROR. It can be NULL, than it is ignored. Abort codes are defined in CANopen standart 301 and can be translated to text calling `vcasdo_abort_msg`.

Returns

actual length of object in bytes negative value in case of an error

Header

vca_od.h

vcaod_od_free

Name

`vcaod_od_free` — release all OD memory

Synopsis

```
void vcaod_od_free (vcaod_root_t * odroot);
```

Arguments

odroot
: pointer to the object dictionary root

Header

vca_od.h

vcaod_load_eds

Name

`vcaod_load_eds` — opens file and create new OD acording to its contens

Synopsis

```
int vcaod_load_eds (vcaod_root_t * odroot, const char* eds_file_name);
```

Arguments

odroot
: root, which will contain loaded EDS

eds_file_name
: name of file to load

Returns

zero in case of success

Header

vca_od.h

vcaod_dump_od

Name

`vcaod_dump_od` — debug function, dumps OD to log

Synopsis

```
void vcaod_dump_od (vcaod_root_t * odroot);
```

Arguments

odroot
: root, which contains OD

Header

vca_od.h

vcaod_get_dinfo_ref

Name

`vcaod_get_dinfo_ref` — returns reference to dinfo corresponting to *obj*

Synopsis

```
sui_dinfo_t * vcaod_get_dinfo_ref (vcaod_object_t * obj, int create_weak);
```

Arguments

`obj`

object from OD

`create_weak`

if there is no HW dinfo for object, creates temporary dbuff dinfo

Description

If *`obj`* allready has its &dinfo assigned `vcaod_get_dinfo_ref` returns this pointer, if it is not function creates new &dinfo object.

Returns

pointer to associated dinfo with reference count increased or NULL if creation fails

Header

vca_od.h

3.2.3.4. canslave command line parameters

canslave command line arguments:

```
USAGE:
canslave [OPTION]

OPTIONS:
 -h
 --help this help screen
 -d
 --dump dumps loaded EDS back to log (debugging purposes)
 -n
 --node set node ID to n
 -e
 --eds EDS file name to load
 -g
 --log_level [n] sets how many log messages you will see.
  0 - fatal errors only
  1 - level 0 + errors
  2 - level 1 + messages
  3 - level 2 + info messages
  4 - level 3 + debug messages
 -v --verbose same as --log_level 3
```

3.2.3.5. CANmaster command line parameters

canmaster command line arguments:

```
CANMASTER - CANopen master
USAGE:
canmaster [OPTION]

OPTIONS:
 -h
 --help this help screen
 --sync n
 -i named pipe --in_pipe named pipe
 -o named pipe --out_pipe named pipe
  name of pipe for communication with monitoring program
  default names are /tmp/canmond/candev-in and /tmp/canmond/candev-out
  in_pipe is name of pipe where monitor feeds my input
  out_pipe is name of pipe where i send answer to the monitoring application
 -g
 --log_level [n] sets how many log messages you will see.
  0 - fatal errors only
  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
 -v --verbose same as --log_level 3
```

3.2.4. Implementation issues

3.2.4.1. Architecture overview

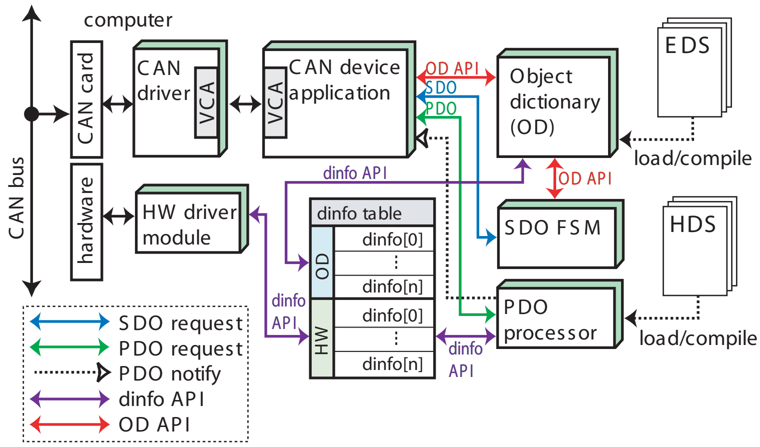

Figure 3-3. RT-CANopen device architecture

CANopen device components description

CAN driver

This part of CANopen device is different in hard and soft real-time spaces. If one can use CAN device in RT-linux he should have CAN driver for RT-Linux. In soft real-time space (also called user space) common CAN driver can be used. Naturally every used CAN driver in any space should provide VCA interface facility.

CAN device application

This application actually makes CANmaster or CANslave. It also encapsulates all threads and synchronization. Libraries alone are thread safe and also without any synchronization inside. This approach gives us opportunity not to change libraries when we migrate between soft and hard real-time spaces.

SDO FSM

SDO FSM (Service Data Object Finite State Machine) is a library providing us SDO FSM data structure `vcasdo_fsm_t` and basic set of functions to proceed SDO communication messages. (see Section 3.2.3.2, *SDO FSM API*).

PDO Processor

PDO Processor is responsible for proper PDO processing. Every PDO data are stored/retrieved through `sui_dinfo_t`(see `can/utils/suiut/sui_dinfo.h`) structures in `dinfo table`. So PDO processor don't know where processed data originates. This makes it simpler and safer.

PDO processor also generates `dinfo table` when CAN device comes to preoperational state.

Object Dictionary OD

Object Dictionary is a place where all device data are stored. Every object can be stored/loaded using its index and subindex (see Section 3.2.3.3, *Object Dictionary API*).

HW driver module

This module shield HW dependent tasks from rest of CAN device. Every HW objects should be exported and accessed through dinfo structures.

In user space is such a module realized as a dynamic link library, in kernel space developer have to write kernel module.

dinfo table

This is not really continuous piece of memory. During CAN device initialization some dinfo structures are allocated. There are two kind of them. HW dinfos resists in `HW driver module` and there are pointers to them from PDO processor and also from OD. OD dinfos are used when OD object PDO mapping exists and there is not HW dinfo to provide its value. In such a case OD dinfo is created for PDO processor. All dinfo structures are reference counted, so they are destroyed automatically.

EDS

EDS means the *Electronic Data Sheet*, text file describing all objects in the slave object dictionary and its mapping into the PDOs. It has normalized form according to `CiA Draft Standard 301`. EDS is parsed in order to create slave OD representation in CANopen device.

HDS

HDS means the *Hardware Definition Sheet*, a text file describing linking of HW dinfos, from HW driver module, with appropriate object index and subindex in OD. It grants 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 device OD. The slave designer have to write this handler dinfo code while the CAN device source code remains always the same.

3.2.4.2. CANopen slave

As can be seen on Figure 3-3 CAN driver sends the CAN messages to device application via VCA. Messages of two main categories are handled in application, process data objects (PDO) and service data objects (SDO, NMT, SFO).

The process data (PDO objects) are handled separately of the SDO with higher priority.

RPDOs (Receive PDO objects) are sent to PDO processor immediately after arriving. It sets appropriate value in OD and also in hardware, if it is defined in HDS. In case of RTR processor add response PDO to waiting PDO list and notify application by calling registered callback function.

TPDOs (Transmit PDO objects) are sent as response to the SYNC object or other device specific event such timer or object value change. In such a case PDO processor add response PDO to waiting PDO list and notify application by calling registered callback function. Application is responsible for sending CAN messages from waiting list to CAN.

All process object values, coming from hardware or not, are accessed via dinfo structures. This gives us uniform interface to get or set its value. Only if non process data object (not PDO mapped or not coming from HW) is accessed via SDO, than it is retrieved by SDO FSM directly from OD.

SDO objects are sent to the SDO FSM. It communicates with OD and prepares response messages for SDO requests. CAN device application only send SDO response to CAN bus if SDO FSM returns any.

Object Dictionary shields user from internal data structures by introducing functions `vcaod_get_value()` and `vcaod_set_value()`. This way can be any object changed.

OD objects can be loaded onto OD from EDS file calling `vcaod_load_eds()`. For kernel space solutions OD with its content should be compiled from C code. This is prepared but not implemented yet.

3.2.4.3. CANopen master

CANopen master architecture is very similar to the CANopen clave one. The main difference lies in OD. CANmaster can have copy of all slaves OD in its memory. This copies can be loaded from slaves issuing SDO commands in preoperational state. This feature is not implemented yet.

Second difference lies in fact that CANmaster can communicate with hierarchically higher application via named pipes. This gives us opportunity to communicate from application in user space with master in RT-linux space through /dev/rtfxx/. See Section 3.3, *CAN monitor*.

3.2.5. Tests

Only soft real-time solution was tested yet. Only functionality was tested, no heavy tests were made. All tests were performed during CanMonitor testing (see CanMonitor tests).

3.2.6. Examples

To make candevice programs type `make` in `ocera/components/comm/can/candev` directory.

3.2.7. Installation instructions

To install `canmaster` and `canslave` simply copy this two files from `ocera/components/comm/can/_c` to desired directory. Do not forget valid *.EDS file to test `canslave` properly.

3.3. CAN monitor

CAN monitor is a component used to monitor CAN/CANopen traffic and also to give user opportunity to involve to it. This component consist of three programs canmond, testclient and Section 3.3, *CAN monitor*.

3.3.1. Summary

Name of the component
: CanMonitor

Author
: Frantisek Vacek

Reviewer
: not validated

Layer
: High-level

Version
: 0.2 Beta

Status
: Beta

Dependencies
: `canmaster` and TCP/IP.

Release date
: February 2004

3.3.2. Description

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

3.3.2.1. 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. It needs running `canmaster` connected to it. Canmond communicates with canmaster via named pipes, so canmaster can be placed in kernel space and use `/dev/rtfxx` or in user space and use arbitrary couple of named pipes.

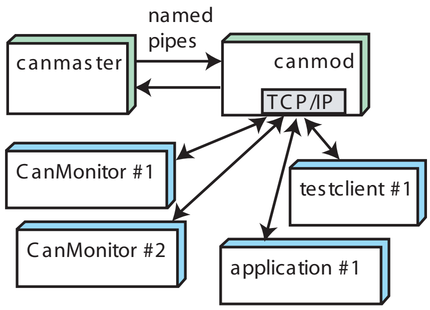

Figure 3-4. Connecting canmond

3.3.2.2. testclient

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

3.3.2.3. 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 by clicking on the mouse.

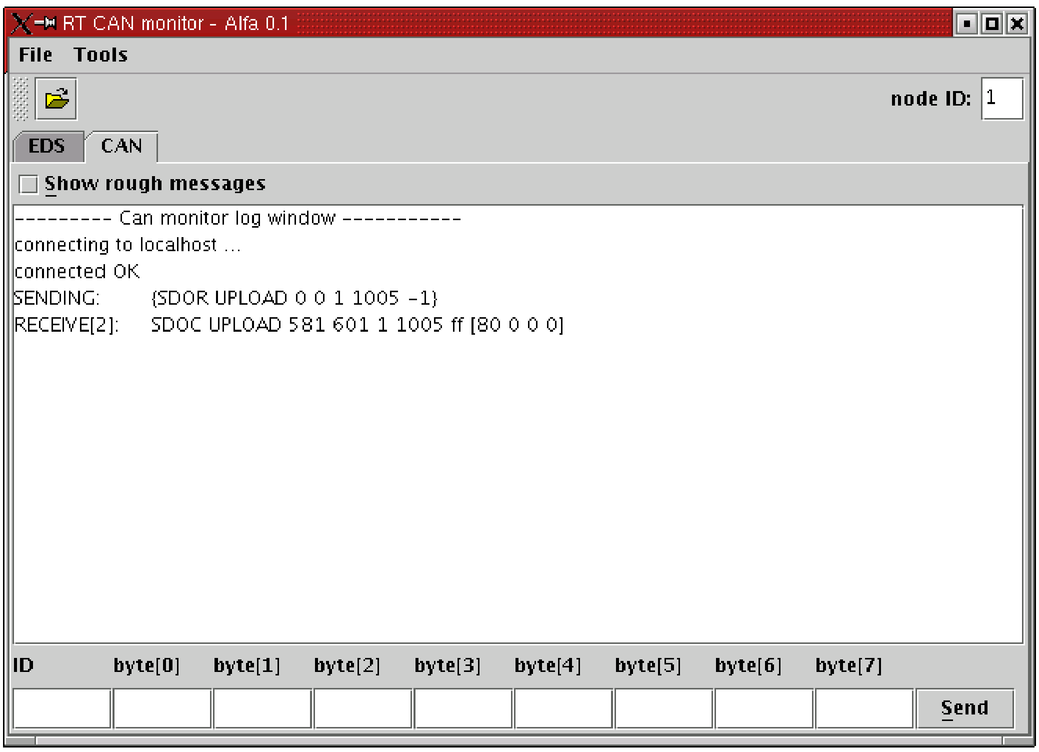

Figure 3-5. CAN monitor CAN messages window

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.

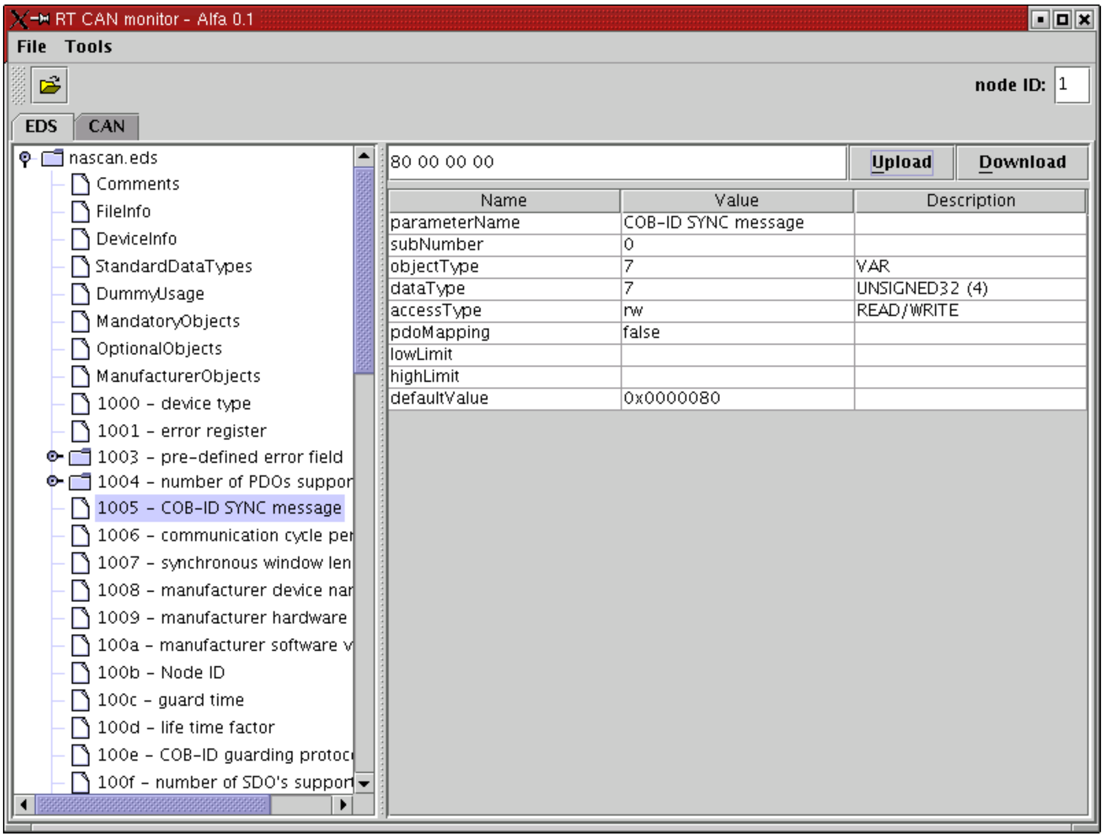

Figure 3-6. The Object Dictionary tree view

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

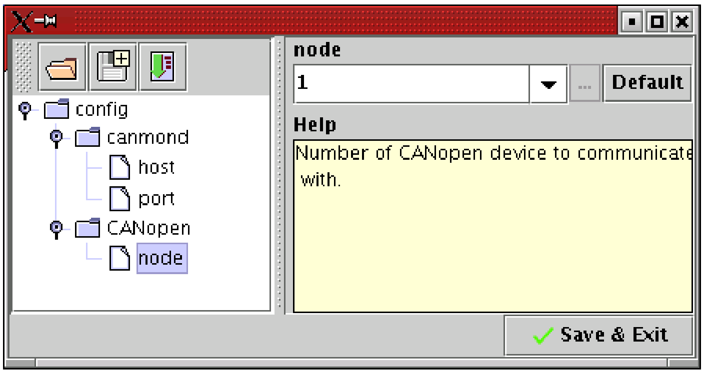

Figure 3-7. The CanMonitor configuration dialog

CanMonitor havs GUI configuration dialog. It can be also configured from command line.

3.3.3. API / Compatibility

3.3.3.1. 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 10001)
 -i named pipe --candev_in named pipe
 -o named pipe --candev_out named pipe
  name of pipe for communication with monitored CAN device
  default names are /tmp/canmond/candev-in and /tmp/canmond/candev-out
  if pipes don't exist, canmond creates the new ones
  canmond writes data to the candev-in pipe
 -g
 --log_level [n] sets how many log messages you will see.
  0 - fatal errors only
  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
```

3.3.3.2. 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 10001

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
 sdo  toggles SDO datagrams only, default is OFF
 q quits
```

3.3.3.3. CanMonitor

Command line arguments

```
loading config from '/home/fanda/.canmonitor/CanMonitor.conf.xml'
USAGE: cammonitor -a host -n node -e EDS_file_name
```

3.3.4. Implementation issues

3.3.4.1. canmond

Any application can attach itself to the `canmond`. It works like TCP server listening on port 10001. If an application opens socket to the server, it can send/receive text messages described in following section.

`canmond` has simple text API to communicate with its clients. API consist of following structures:

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 to fit single CAN message.

`Example: {CANDTG 0 0 0 189 [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 + node_id` for the server_port and `0x600 + node_id` for the client_port are used. If object on desired index do not have sub-indexes, subindex parameter is ignored.

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 if uploaded object is larger than 8 bytes. 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.'}`

3.3.5. Tests

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

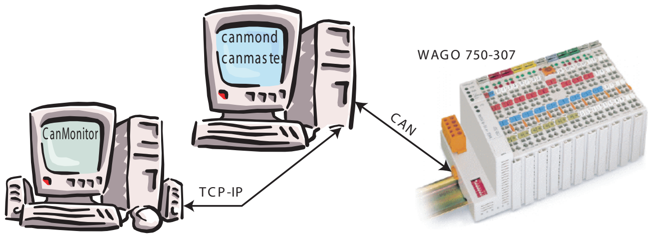

Figure 3-8. CanMonitor testing

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

All components were also tested with `canmaster` and `canslave` components. In following example is written how.

3.3.6. Examples

3.3.6.1. Example 1 - connecting to real CANopen device

Make sure, that CAN driver is installed and works properly. Check that real CANopen device is connected to your CAN card.

Type `make` in `ocera/components/comm/can` directory to make all necessary programs. Than open two terminal windows.

In first window launch canmaster by typing `canmaster`.

You should see something like this

```
[fanda@mandrake bin]$ ./canmaster
CANMASTER - CANopen master
canmaster: entering state STATE_INITIALIZING
canmaster: entering state STATE_PREOPERATIONAL
canmaster: entering state STATE_OPERATIONAL
```

Than you should launch `canmond` on the same machine.

```
[fanda@mandrake bin]$ canmond
CANMOND - CAN monitor server
```

If you have a graphical environment with Java installed, you can launch CanMonitor issuing:

```
[fanda@mandrake bin]$ canmonitor -e nascan.eds
loading config from '/home/fanda/.canmonitor/CanMonitor.conf.xml'
connecting to localhost/127.0.0.1
connected OK
```

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

Instead or in addition to CanMonitor you also launch `testclient` program either on the same machine or on other one. With testclient you can't use EDS file but in other hand you don't need graphical environment.

```
[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` to give the `testclient` readline facility like command history, BASH like line editing etc..

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

3.3.6.2. Example 2

In this example canslave is tested, that means that you do not need any real CANopen device. Tested canslave can resist on same computer as canmaster on can be on other computer connected by CAN bus. If both programs resist on same computer make sure that CAN driver `lincan` was configured to make echo of sent CAN messages to all other who have open CAN driver on same computer.

Type `make` in `ocera/components/comm/can` directory to make all necessary programs. Than open four terminal windows (Four windows using is just for demonstration purposes).

In first window launch `canslave`. You can launch more canslaves with different node numbers. Do not forget introduce *.EDS file name after -e switch in command line.

You should see something like this

```
[fanda@mandrake bin]$ canslave -e nascan.eds
CANSLAVE - CAN slave
canslave: Opening CAN driver: /dev/can0
canslave: Opening EDS: nascan.eds
canslave: entering state STATE_INITIALIZING
canslave: SYNC COB_ID: 0, SYNC period: 0
canslave: entering state STATE_PREOPERATIONAL
canslave: entering state STATE_OPERATIONAL
```

In second window launch `canmaster`.

You should see something like this

```
[fanda@mandrake bin]$ ./canmaster
CANMASTER - CANopen master
canmaster: entering state STATE_INITIALIZING
canmaster: entering state STATE_PREOPERATIONAL
canmaster: entering state STATE_OPERATIONAL
```

In third window launch `canmond`.

You should see something like this

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

Than you can launch `testclient` or `CanMonitor` or both of them like in previous example to work with canslave OD or to see CAN traffic.

3.3.7. 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/canmond` directory. And copy desired files from`ocera/components/comm/can/_compiled` 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.

Java SDK ver. 1.4 or above is recommended.

Chapter 4. Verifications

4.1. CAN model by timed automata

4.1.1. Sumary

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

Alfa

Dependencies

Not validated

Release date

2003-04-07

4.1.2. Description

4.1.2.1. Problem statement

This section deals with a design conception of theoretical study offering methodology tool supporting analysis of distributed Real Time (RT) systems. Figure 4-1 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 futher 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

Coresponding three sub models can be futher 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, ...).

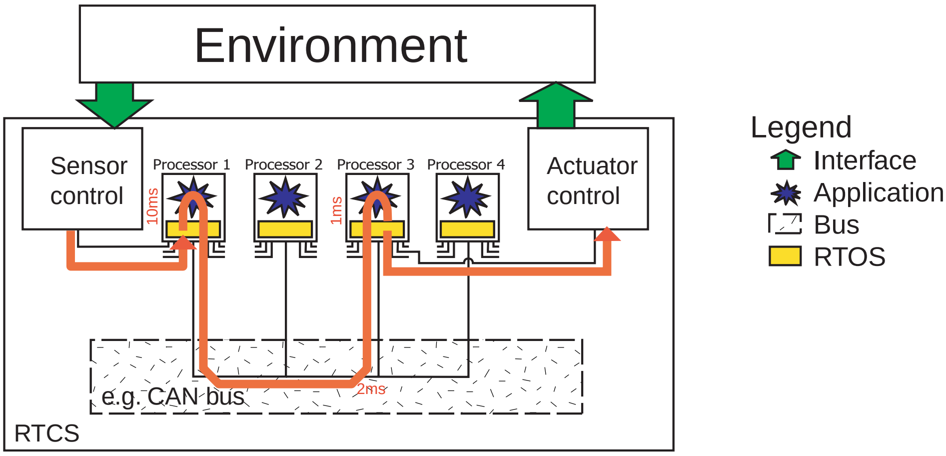

Figure 4-1. Real time control system structure with denotation of computation/communication times

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.

4.1.2.2. CAN bus description

This section introduces a basic terminology futher used in CAN model. It can be skiped by the reader familier with this technology.

Controller Area Network (CAN) [CAN01] is a serial bus system especially suited to interconnect smart devices to build smart systems or sub-systems.

4.1.2.2.1. 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.

4.1.2.2.2. 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).

4.1.2.2.3. Detecting and signalling errors

Unlike other bus systems, the CAN protocol does not use acknowledgement 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 acknowledgement. If no acknowledgement 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 .

4.1.3. API/Compatibility

Not applicable.

4.1.4. Implementation issues

4.1.4.1. Bit-wise arbitration model

The model of CAN arbitration designed in timed automata [UPPAAL00] is shown in Figure 4-3. The model describes MAC mechanism for one message accessing the bus. The location *no_trans_needed* represents a situation when the arbitration model is waiting for *trans_request* from the application process. The locations *send_bit_to_bus*, *listen_bus*, *check_next_bit* represent the arbitration process. The locations *request_denied* and *request_success* give result of the arbitration process.

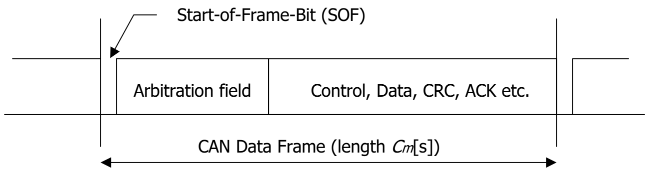

Figure 4-2. CAN message frame format

After processing of the Start of Frame Bit (SOF) (see the CAN message frame format in Figure 4-2) the first bit from the arbitration filed is sent to the bus (transition *send_bit_to_bus* -> *listen_bus*). At the same time the transmitting processor senses the bus and both transmitted bit (local variable *id*) and sensed bit (global variable *signal*) are compared. If they are identical and the end of the Arbitration field (*nsigi* states for the length of the Arbitration field) was not reached the next bit is proceeded (*check_next_bit* location) when nominal bit-time elapses (deterministically given as *tbit* constant). If the sensed bit is not identical to the transmitted one, the transmission is denied (*request_denied* location). If they are identical and the end of the Arbitration field was reached the processor wins the arbitration (*request_success* location). The CAN Arbitration model includes the information about the duration of each bit-time given by invariant $t <= tbit$ in *listen_bus* location and guards $t >= tbit$, $t>= 0$ on outgoing transitions. When *tbit* is not deterministic, i.e. *tbit* is bounded on interval <*tbit_l,tbit_u*>, then the duration of each bit-time given by invariant $t <= tbit\_u$ and guard $t >= tbit\_l$.

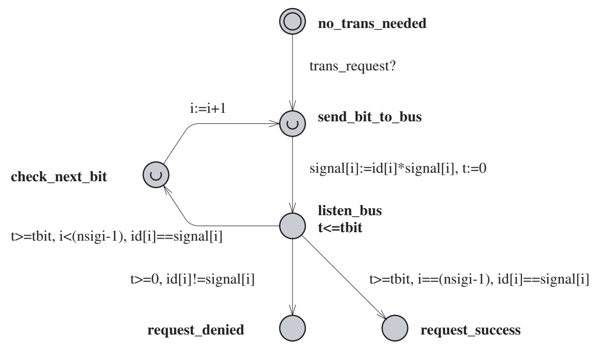

Figure 4-3. Arbitration model (in UPPAAL like notation)

4.1.4.2. Transceiver model

Above explained bit-wise arbitration is a part of the transceiver model.

The implementation of the complete transceiver model is depicted in Figure 4-4 , and its interconnection with other automata is shown in Figure 4-6 . It is composed of the three sections:

- the arbitration section described already in Figure 4-3

- synchronisation section (*waiting_for_free_bus->send_bit_to_bus* transition) that is used to synchronize all transmitting processors prior to arbitration (this part realises broadcast communication) and
- data transmission section given by *trans_section*, *trans_section_finished* and *trans_finished* locations.

The function of transceiver is the following: after receiving the transmission request, the processor is in the waiting state (*waiting_for_free_bus*) until the bus is free. Bus becomes idle, the arbitration processes start (synchronization by urgent *broadcast_synch* channel). If the transmission was denied (*trans_denied* location), the transmission request is immediately repeated and the processor is waiting for free bus again (*waiting_for_free_bus* location). Otherwise the processor message is sent. The duration of message is given by deterministic time *Cm*. When the transmission is finished (*trans_section_finished*) the bus becomes idle (*bus_trans_finished* channel) and the application process is informed about the end of transmission (*trans_compl_status* channel).

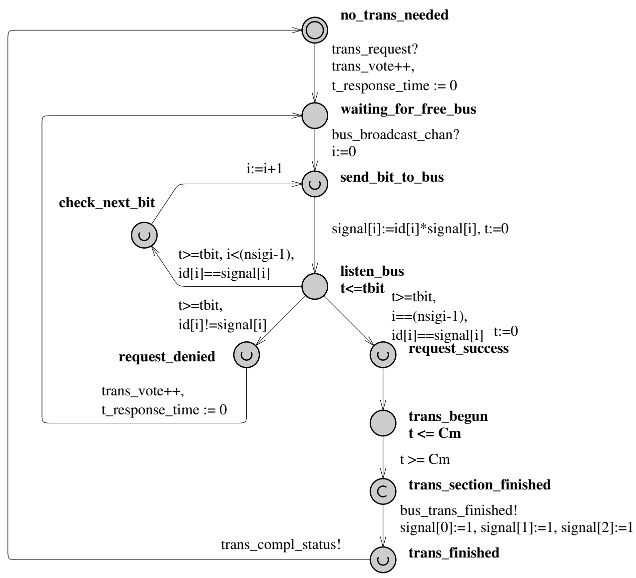

Figure 4-4. Transceiver model

4.1.4.3. Bus model

Figure 4-5 depicts the physical bus model. The model is in idle location when there is no activity on the bus and it is in busy location when any processor transmits. The *trans_vote* global variable is used to detect that at least one processor is willing to start the transmission. If this is the case than the global synchronization is realized via *bus_broadcast_chan* from the bus model.

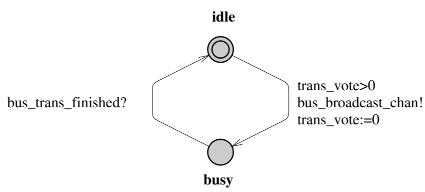

Figure 4-5. Bus model

4.1.5. Tests

Not applicable.

4.1.6. Examples

4.1.6.1. Case study 1 - Application process model

As seen from Figure 4-6the case study assumes 4 processors to be connected via CAN. Each processor is running one application process transmitting the messages of the same identifier. The application processes 1, 2, and 3 are periodic processes transmitting messages with identifier 1, 2, and 3 respectively. The application process 4 is a sporadic process transmitting the lowest priority message with identifier 4.

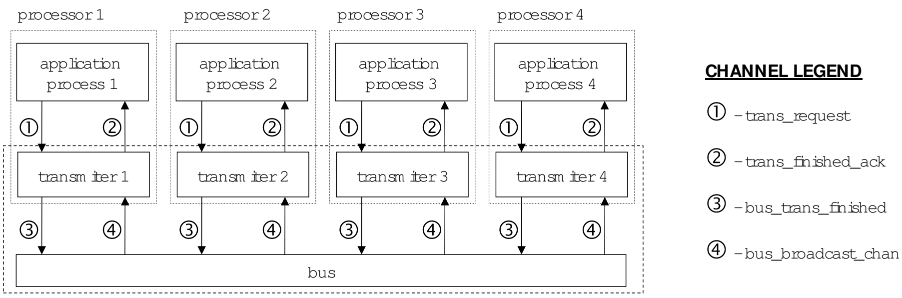

Figure 4-6. Case study system configuration

The periodic application process, with period *Tm*, is depicted in Figure 4-7. Afterwards each message is delayed by an operating system delay, the time between zero and *Ji* (called jitter in [Tindell94]). Then the transmission request is done by *trans_request* channel. When the message is transmitted the process is informed by *trans_compl_status* channel. Location *no_transmission_activity* represents a situation when the process does not perform transmission, i.e. it performs for example computations. Location *init_location* starts the first task period, delayed by time between zero and *Tm* in order to represent the phase shift of the task.

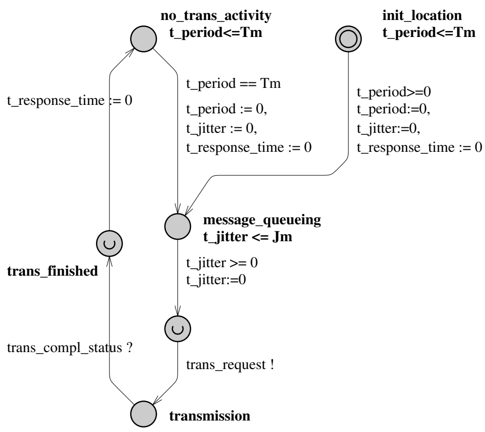

Figure 4-7. Periodic application process model

The sporadic process model is depicted in Figure 4-8 . Locations *no_trans_activity_1* and *no_trans_activity_2* represent a situation when the process does not perform any transmission. The process resides an arbitrary time in location *no_trans_activity_1*, then the transmission request is generated and when the message is transmitted the process is informed by *trans_compl_status* channel, and then the process has no influence on the bus. Local variable *t_response_time* in both models is used in properties to be verified.

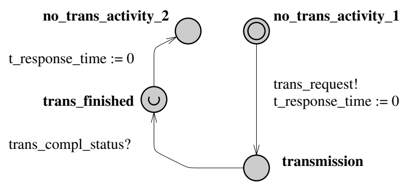

Figure 4-8. Sporadic application process model

4.1.6.1.1. Comparison with traditional approach

The section presents the case study with periodic and sporadic processes including comparison with Tindell's approach (assuming 125kbps baudrate).

Timing and logical properties to be verified can be for example the following ones:

1. Is the system deadlock free?
2. Is there any state in which processor 1 and processor 2 are in the data transmission section?
3. Is there any situation in which the highest priority message does not win the arbitration?
4. Are all periodic messages transmitted prior to their deadlines?
5. What is the worst-case response time *Rm* of the message with identifier *m* (for *m*=1, 2 or 3)?

These properties are formulated in the temporal logic based formalism used in the UPPAAL verification tool UPPAAL [UPPAAL00] as follows:

1. A [] (not deadlock)
2. E <> (Transceiver_1.request_success and Transceiver_2.request_success)
3. E <> (Transceiver_1.request_denied)
4. A [] (Process_m.trans_finished) & (Process_m.t_response_time < Deadline)
5. A [] (Process_m.trans_finished) & (Process_m.t_response_time < *Rm*)

The verification results of timed automata tool are as follows:

1. Property is satisfied
2. Property is not satisfied
3. Property is not satisfied
4. See the section bellow
5. *Rm* found by iteration (using bisection) see the section bellow

We assume the configuration depicted in Figure 4-6 where each processor is running one application process transmitting one type of message (the message *ID* is equal to the application *ID* is equal to the processor *ID*). Table 4-1, *Process parameters table* shows parameters of three periodic and one sporadic process. The aim of the case study is twofold:

- to verify whether the response time satisfies a given deadline of the message (corresponding to property 4)
- to find the worst-case response time *Rm* iteratively by repeating the verification for different values of deadline (corresponding to property 5).

Table 4-1. Process parameters table

Msg. ID	Type	Period Tm[usec]	Deadline[usec]	Cm[usec]
1	periodic	2000	2000	504
2	periodic	3000	3000	504
3	periodic	5000	4000	504
4	sporadic	-	-	1040

Table 4-2, *Results of the experiment related to property 4 and 5* shows verification results of the experiment related to property 4 and 5 without the operating system delay. The response time of each periodic message is shorter then corresponding deadline assuming also relatively long sporadic message.

Table 4-2. Results of the experiment related to property 4 and 5

Msg. ID	Jm	formula 4 result	Rm
1	0	satisfied	1544
2	0	satisfied	2048
3	0	satisfied	3056
4	-	-	-

Table 4-3, *Results of the experiment related to property 4 and 5 with the operating system delay* shows results of the experiment related to property 4 and 5 with the operating system delay *Jm*.

Table 4-3. Results of the experiment related to property 4 and 5 with the operating system delay

Msg. ID	Jm	formula 4 result	Rm
1	456	satisfied	2000
2	0	satisfied	2552
3	0	satisfied	3056
4	-	-	-

Values of *Rm* in Table 4-2, *Results of the experiment related to property 4 and 5*, Table 4-3, *Results of the experiment related to property 4 and 5 with the operating system delay* are identical to those calculated by iterative algorithm [Tindell94] based on equation

$$R_i = J_i + w_i + C_i$$

Figure 4-9. Worst-case response time equation

where

$$w_m = B_m + \sum_{\forall j \in hp(m)} \lceil \frac{w_m + J_j + \tau_{bit}}{T_j} \rceil C_j$$

Figure 4-10. Worst-case queueing delay equation

The term *Bm* presents the longest time that the given message *m* can be delayed by lower priority messages, the *taubit* is the bit time of the bus. The set *hp(m)* is the set of messages of higher priority then message *m*.

4.1.6.2. Case study 2 - Anti-lock Braking System

This case study is example of distributed system containing timed automata models, including the CAN model, the RT operating system model [Wasznio03] and the Anti-lock Brake System [Kerim00] realised as application process model (see Figure 4-11). The system consists of two processors (i.e. MCUs) with pre-emptive RTOS (e.g. OSEK), communicating via CAN. The first processor (see Figure 4-14), connected to the brake pedal (see Figure 4-12), detects the pedal position and transmits corresponding messages to the second processor. The second one (see Figure 4-14) acquires information about acceleration/deceleration from an acceleration sensor, it receives messages from the first processor, and calculates and accomplishes a control action following rules of ABS.

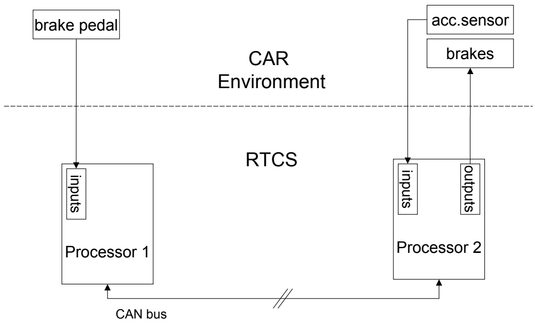

Figure 4-11. Structure of the distributed system - ABS control

The timed automaton in Figure Figure 4-13 models typical situation - the pedal is pressed and then it is released after some undefined time. Model consists of three locations. Variable *BPPEvent* is set when the pedal is pressed (*bPedalInit* -> *bPedalPressed*) and variable *BPREvent* is set when when it is released (*bPedalPressed* -> *bPedalReleased*). Variable *BPPEvent* is read by read by a timed automaton model of *Task1* (see Figure Figure 4-15) and variable *BPREvent* is read by read by a timed automaton model of *Task2* (see Figure Figure 4-16).

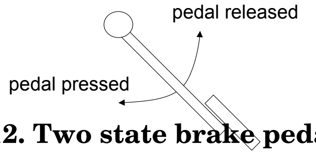

Figure 4-12. Two state brake pedal

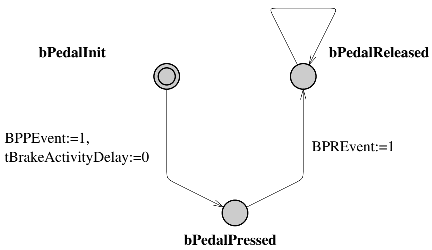

Figure 4-13. Brake pedal model

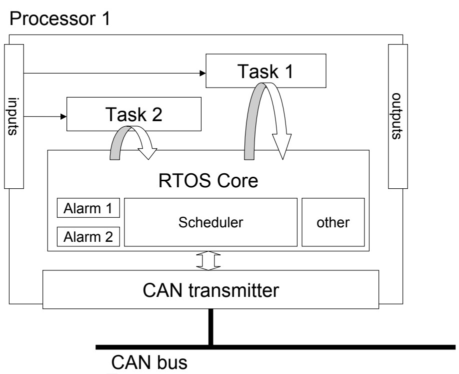

Figure 4-14. Processor 1 structure

The model of the first processor is described in Figure Figure 4-14 . The model consists of two automata, modelling the application tasks (*Task1*,*Task2*) periodically triggering the inputs, group of automata modelling RTOS including pre-emptive scheduler and periodic alarms [Wasznio03], and group of automata modelling CAN as explained in previous sections. As shown below, the first task detects if the brake pedal is pressed and the second one if the pedal is released.

```
Task1RTOS1(void) {
 // send message when the pedal is pressed
 if (BPPEvent) sendMsg(PedalPressed);
 // otherwise terminate the task - wait for the next activation
 TerminateTask();
};


Task2RTOS1(void) {
 // send message when the pedal is released
 if (BPPEvent) sendMsg(PedalReleased);
 // otherwise terminate the task - wait for the next activation
 TerminateTask();
};
```

Timed automaton model of *Task1* is depicted in Figure Figure 4-15. *Task1* becomes ready (location *Ready1*) when it is triggered by an alarm (variable *nActivateBPPMT* and channel *wQuch*) and further it is executed (location *Comp1*) when it is the highest priority task in the OS queue (array *Q1*). *if* statement (location *Comp1*) has execution time bounded by its lower (constant *L1*) and upper bound (constant *U1*). When *BPPevent* is not set then the task terminates. Otherwise *sendMsg* function (locations *Waiting*, *Ready2*, *Comp2*), with execution time bounded by its lower (constant *L2*) and upper bound (constant *U2*), is executed. Similarly the timed automaton model of *Task2* is depicted in Figure Figure 4-15.

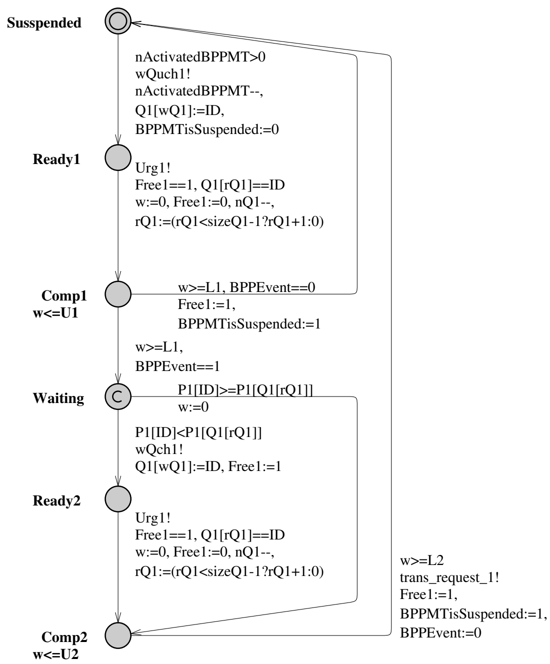

Figure 4-15. Timed automaton model of Task1

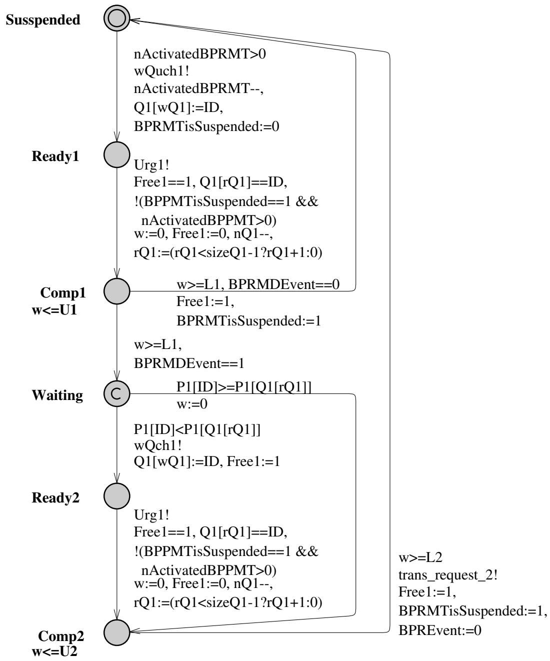

Figure 4-16. Timed automaton model of Task2

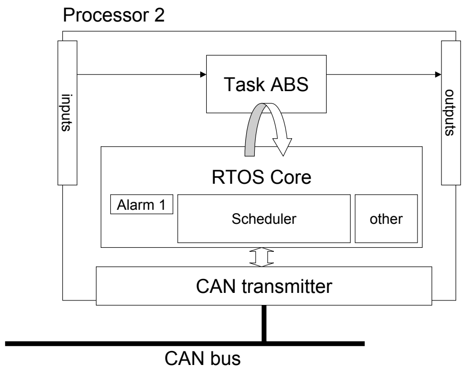

Figure 4-17. Processor 2 structure

The second processor, depicted in the Figure Figure 4-17, includes *TaskABS* which receives the messages (the pedal position), it reads local input (acceleration sensor), it calculates ABS controller and accomplishes a control action (brake shoes). When braking (variable *BPPMEvent == 1* in Figure Figure 4-18), the ABS controller is looking for decelerations in the wheel that are out of the ordinary (guard *acceleration <= MAXdec*). Right before wheel locks up, it will experience a rapid deceleration. If left unchecked, the wheel would stop much more quickly than any car could. The ABS controller knows that such a rapid deceleration is impossible, so it reduces the pressure to that brake (location *brake_shoes_released*) until it sees an acceleration (guard *acceleration > 0*), then it increases the pressure until it sees the deceleration again. It can do this very quickly, before the tire can actually significantly change speed. The result is that the tire slows down at the same rate as the car, with the brakes keeping the tires very near the point at which they will start to lock up. Corresponding detailed models (in UPPAAL notation) of *ABSTask*, brakes and acceleration sensor are depicted in Figure Figure 4-19, FigureFigure 4-20 and Figure Figure 4-21.

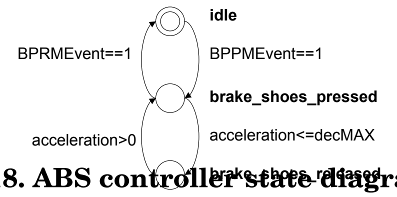

Figure 4-18. ABS controller state diagram

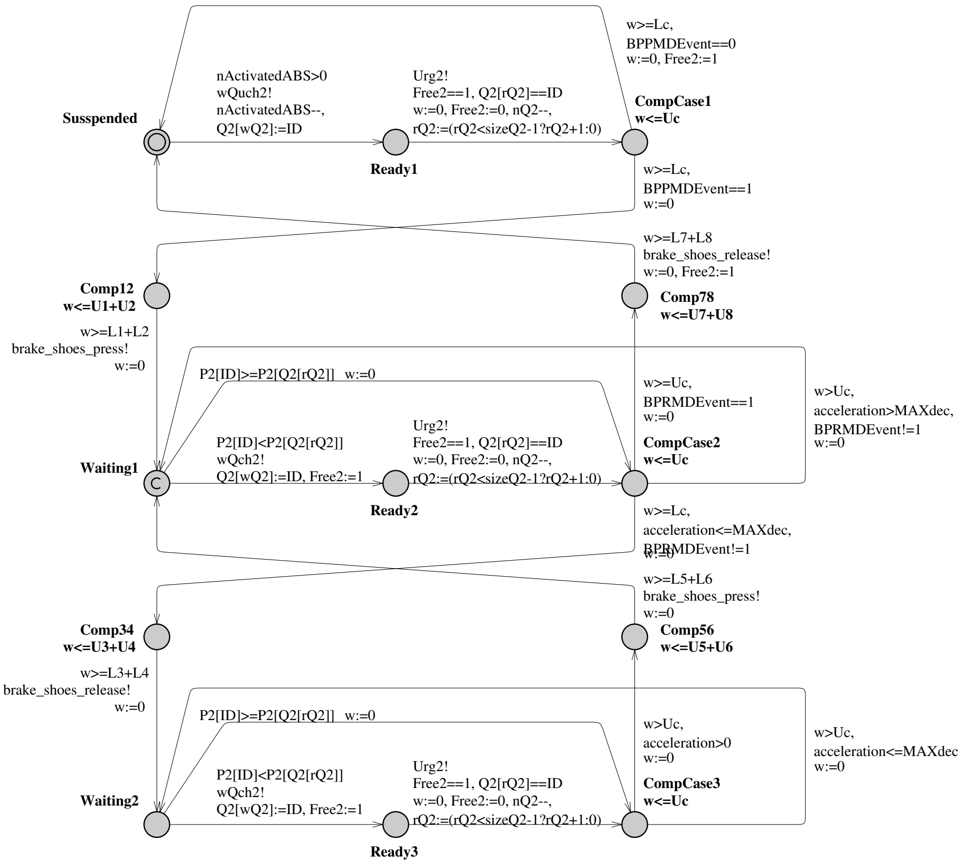

Figure 4-19. Timed automaton model of ABSTask algorithm

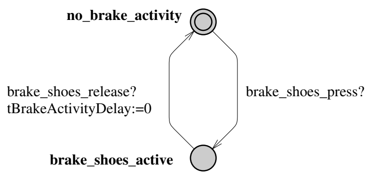

Figure 4-20. Timed automaton model of brakes

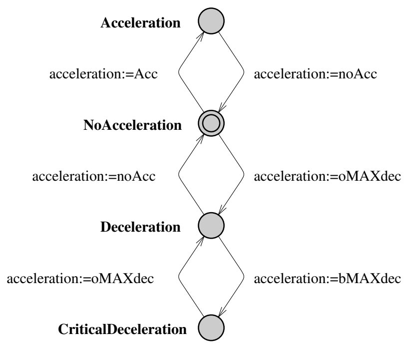

Figure 4-21. Timed automaton model of acceleration sensor

4.1.6.2.1. Verification

The system parameters are shown in Table 4-4, *Processor 1 RTOS system parameters*, Table 4-5, *Processor 2 RTOS system parameters* and Table 4-6, *Message Parameters* for this case study.

Table 4-4. Processor 1 RTOS system parameters

Task name	Task period [usec]	U1,U2,L1,L2[usec]
Task1	5000	1
Task2	5000	1

Table 4-5. Processor 2 RTOS system parameters

Task name	Task period [usec]	U1...U8,L1...L8[usec]
TaskABS	5000	1

Table 4-6. Message Parameters

Message ID	Cm [usec]	bit time[usec]
1	504	8
2	504	8

Timing and logical properties to be verified can be for example the following ones:

1. Is the system deadlock free?
2. When the pedal was pressed and not released, the *PedalReleased* message would not be received.
3. Message *PedalPressed* is received at least 2ms after the pedal has been pressed.
4. What is the worst case receive time for message *PedalPressed*?
5. Will be ever the ABS active?
6. What is the worst-case time for activation of brake shoes?

These properties are formulated in the temporal logic based formalism used in the UPPAAL verification tool UPPAAL [UPPAAL00] as follows:

1. A [] (not deadlock)
2. (BrakePedal.bPedalPressed and not BrakePedal.bPedalStillReleased) --> (not BPRMDEvent==1)
3. A [] (tBrakeActivityDelay>2000 and not BrakePedal.bPedalReleased) imply (BPPMDEvent==1)
4. A [] (tBrakeActivityDelay>X and not BrakePedal.bPedalReleased) imply (BPPMDEvent==1)
5. E <> (R2T1.Sheduled2)
6. A [] (Brake.brake_shoes_active and not BrakePedal.bPedalReleased and not BrakePedal.bPedalStill imply (tBrakeActivityDelay<X)

The verification results of timed automata tool are as follows:

1. Property is satisfied
2. Property is satisfied
3. Property is not satisfied
4. X=5507 - found by iteration (using bisection)
5. Property is satisfied
6. X=10007 - found by iteration (using bisection)

4.1.7. 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*.

[UPPAAL00] Paul Pettersson and Kim Guldstrand Larsen, 2000, *UPPAAL2k: http://www.uppaal.com*.

[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)* .

[CAN01] K. Etschberger, 2001, *Controller Area Network : Basics, Protocols, Chips and Applications*.

[Alur94] Rajeev Alur and David Dill, 1994, *A theory of timed automata*.

[Katoen99] Joost-Pieter Katoen, 1999, *Concepts, Algorithms, and Tools for Model Checking*.

[Wasznio03] Libor Waszniowski and Zdenek Hanzalek, 1994, *Analysis of Real-Time Operating System Based Applications*.

[Corbett96] James C. Corbett, 1996, *Timing Analysis of {A}da Tasking Programs: IEEE Transactions on Software Engineering*.

[Berard01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, 2001, *Systems and Software Verification: Model-Checking Techniques and Tools*.

[Klein93] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonza, and L. Harbour, 1993, *Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real Time Systems*.

[Osek00] 2000, *OSEK/VDX: Specification 2.1*.

[Liu00] Jane Liu, 2000, *Real-Time Systems*.

[Kerim00] Nice Kerim, *How Anti lock Brakes works: http://www.howstuffworks.com*.

4.2. Verification of cooperative scheduling and interrupt handlers

4.2.1. Sumary

Name of the component
: Verification of cooperative scheduling and interrupt handlers

Description

This component is 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

Alfa

Dependencies

Not validated

Release date

N/A

4.2.2. Description

4.2.2.1. Abstract

This chapter is dedicated to modelling of real-time applications running under multitasking operating system. Theoretical background is based on timed automata by Alur and Dill. As this approach is not suited for modelling 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 the 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 since their reachability problem and model checking of TCTL problem is decidable. The approach is illustrated in the form of examples in the real-time verification tool UPPAAL.

4.2.2.2. Introduction

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 premption 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 4-22 representing both, cooperative scheduling (*"yield control"* on *Deschedule* transition) or preemptive scheduling (*"preempted"* on *Deschedule* transition).

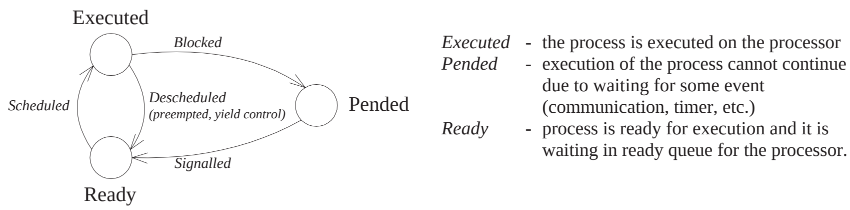

Figure 4-22. State transition diagram of the process in the multitasking operating system

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 4-23 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 4-23 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 4-23 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
}                       }
```

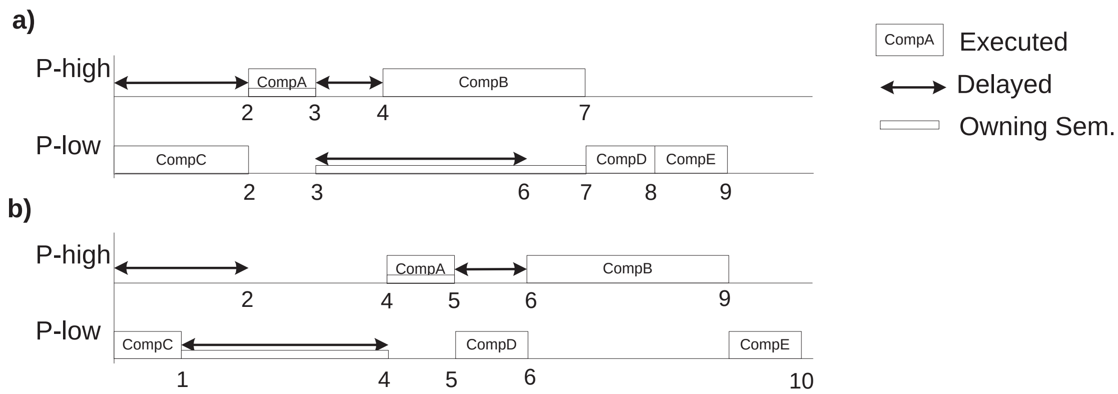

Figure 4-23. Example of monoprocessor scheduling anomaly

4.2.3. API/Compatibility

Not applicable.

4.2.4. Implementation issuses

4.2.4.1. 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 4-24. 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.

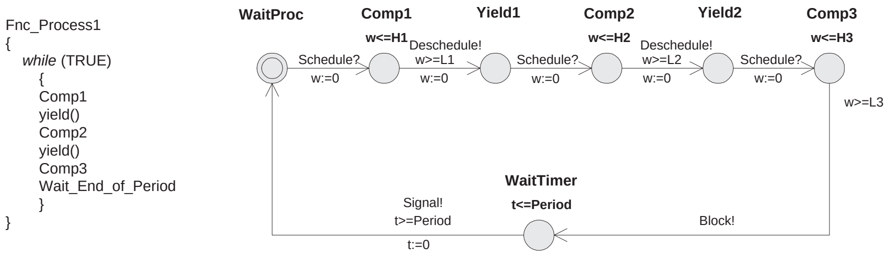

Figure 4-24. Model of the application process executed under cooperative scheduling policy

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 4-25. *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.

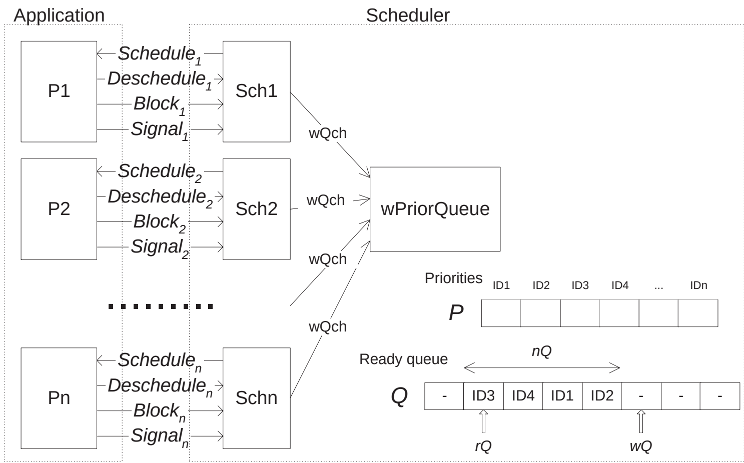

Figure 4-25. Synchronisation of cooperative scheduler with application processes

One automaton of the cooperative scheduler model (Sch_i) is depicted in Figure 4-26.

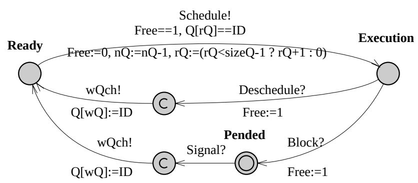

Figure 4-26. One automaton (*Schi*) of the cooperative scheduler in Figure 4-25

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 4-25. 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 4-26 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*.

4.2.4.2. 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 4-27. 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 4-28). 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 4-28 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 4-28). 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 4-29).

Figure 4-30 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).

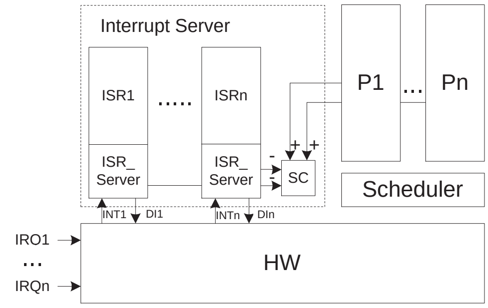

Figure 4-27. System architecture with interrupt server

```
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 4-28. *Interrupt server* routines

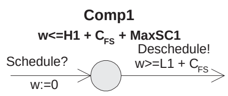

Figure 4-29. *Computation* location considering interrupts

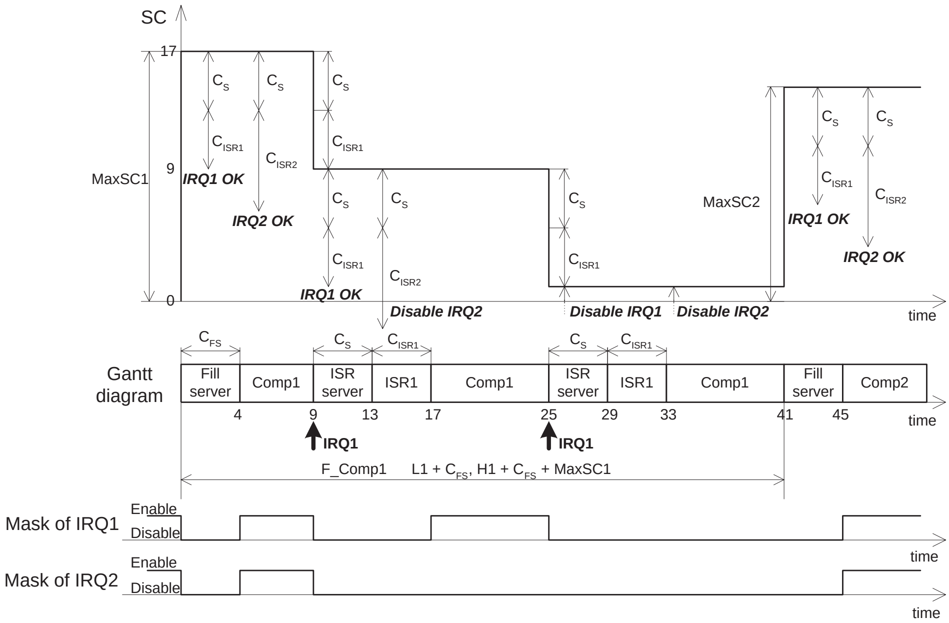

Figure 4-30. Time diagram of ISR execution within *interrupt server*

Choice of MaxCS value for different locations depends on application requirements and it is specified at the design stage. Section 4.2.7, *Examples* section shows an example application with one IRQ, two processes of different priority and one semaphore (semaphore is discussed in Section 4.2.4.3.1, *Semaphore*).

4.2.4.3. 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.

4.2.4.3.1. 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 4-31. 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
    }
}
```

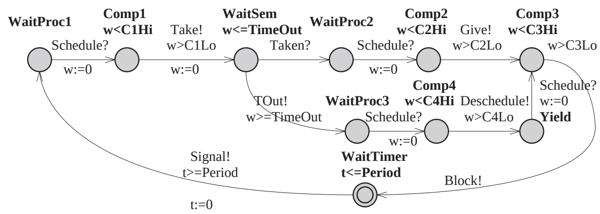

Figure 4-31. Model of process containing Take and Give one semaphore

Scheduler model for application with one semaphore is depicted in Figure 4-32. 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.

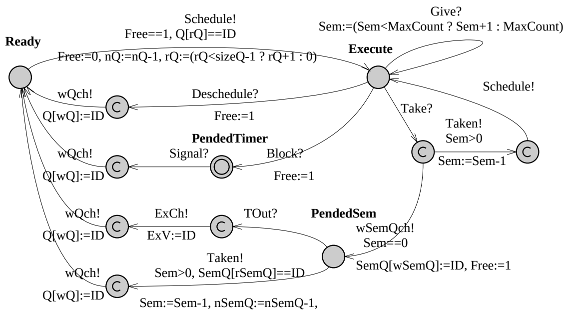

Figure 4-32. Scheduler model containing Take and Give of one semaphore (extension of Figure 4-26)

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 4-32 the automaton *wFifoQueue* is used in Figure 4-33.

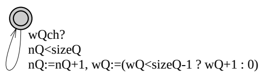

Figure 4-33. Automaton wFifoQueue providing writing to the FIFO queue

4.2.4.4. 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 approche 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 faul-tolerant applications (see workpackage 6 - Fault Tolerant component). In order to reach full compatibility with RTLinux it is needed to study the Kernel intervals and to use different tools (e.g. Hytech) so that the preemptive can be modelled.

4.2.5. Implementation issuses

Not applicable.

4.2.6. Tests

Not applicable.

4.2.7. Examples

4.2.7.1. Example of system with interrupt

Consider application depicted in Figure 4-34. 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 4-32). First process *Proc_Period* is periodically executed with low priority (Figure 4-39). The second process *Proc_Int* with high priority is intended for handling external aperiodic events (Figure 4-38). It is waiting for semaphore that is given within interrupt service routine. Interrupt requests (*IRQ*) are generated by model of *Environment* (Figure 4-35). If the interrupt request is enabled (*EN>0*), hardware interrupt controller *InterruptCtrl* (Figure 4-36) 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 4-37). 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*).

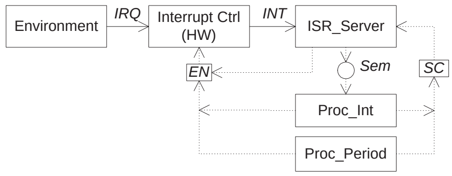

Figure 4-34. Interconnection of sample automata

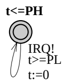

Figure 4-35. Model of Environment generating IRQ

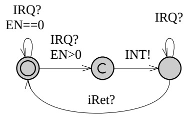

Figure 4-36. Model of hardware interrupt controller

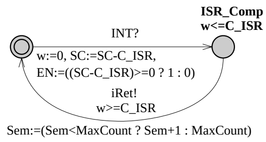

Figure 4-37. *ISR_Server* model

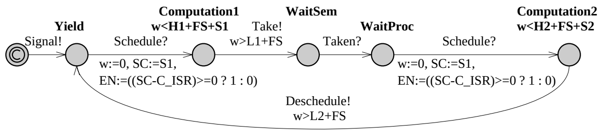

Figure 4-38. Model of high-priority process *Proc_Int*

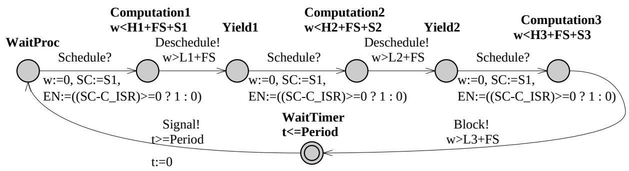

Figure 4-39. Model of low-priority periodic process *Proc_Period*

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)*.