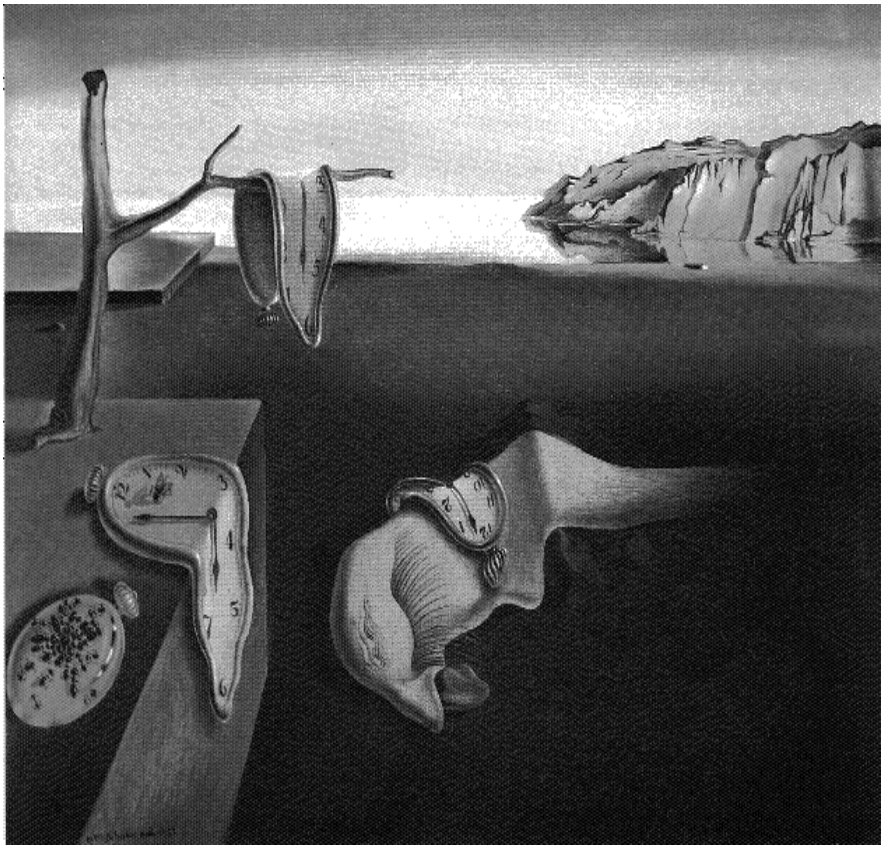
Institut für Automation
Abt. für Automatisierungssysteme

Technische Universität Wien

Projektbericht Nr. 183/1-90
February 1999

i82596 NTI Device-Driver Software Documentation

Gerda Richter, Michael Schmidt and Ulrich Schmid



Salvador Dali, "Die Beständigkeit der Erinnerung"

Contents

1 Contents **5**

1.1 Definitions And Notational Conventions 6

1.1.1 Definitions 6

1.1.2 Notational Conventions 6

2 Environment **7**

2.1 Clients 8

2.1.1 pSOS^{+m} 8

2.1.2 pNA^{+} 12

2.1.3 Clock Synchronization 17

2.2 Network Controller 18

2.2.1 Operational Model 19

2.2.2 i82596 Memory Structure 19

2.3 Network Time Interface (NTI) 23

3 NTI-driver **27**

3.1 Overview 27

3.2 Application Layer 28

3.2.1 Description 28

3.2.2 KI 28

3.2.3 NI 32

3.2.4 CI 36

3.3 Error Handling and KI Transmission Protocol 41

3.3.1 Description 41

3.3.2 Transmission Protocol 41

3.3.3 Error Handler 44

3.4 Management Layer 45

3.4.1 Description 45

3.4.2 Initialization . . . 45
3.4.3 Commands . . . 46
3.4.4 Received Messages . . . 50
3.5 Driver Layer . . . 52
3.5.1 Description . . . 52
3.5.2 Command Handling . . . 65
3.5.3 Receive Handling . . . 70
3.5.4 Initialization . . . 73

4 Programmer's Reference **77**
4.1 Introduction . . . 77
4.2 Structure of NTI-driver . . . 77
4.2.1 Files of the NTI-driver . . . 78
4.3 Constants . . . 81
4.4 Data Structures . . . 97
4.5 Functions . . . 111

5 Usage Considerations **183**
5.1 Installation, Configuration and Usage . . . 183
5.1.1 pSOS Configuration . . . 183
5.1.2 NTI Configuration . . . 185
5.1.3 Miscellaneous Configuration Issues . . . 186
5.1.4 Usage . . . 186
5.1.5 Memory Requirements . . . 186
5.2 Sample Application . . . 187
5.2.1 System Overview . . . 187
5.2.2 Performance Evaluation . . . 189
5.3 Debugging . . . 190
5.4 Shortcomings and Pitfalls . . . 191

A List of Abbreviations **193**

Index **196**

List of Figures

2.1 Architecture of a pSOS^{+m} node using the NTI-driver 7
2.2 Shared memory structure of LANCE i82596 20
2.3 NTI block diagram 23
2.4 Packet timestamping 25
2.5 Structure of a physical node with multiple NTIs 25

3.1 Structure of NTI-driver 28
3.2 an example for resend queues 41
3.3 Partition of the shared memory area on the NTI 65
3.4 a typical example for buffer allocation/deallocation 66
3.5 Returning a set of fixed size buffers associated with RBDs 70
3.6 a typical example for *copy_rbd* 71

4.1 Structure of NTI-driver Functions 79

5.1 Hardware architecture of the evaluation system 187
5.2 Software architecture of the evaluation system 188

Chapter 1

Contents

This report documents the final version of the *NTI-driver*, a pSOS^{+m} device driver for the i82596 Ethernet-coprocessor that also supports our *Network Time Interface* (NTI) M-Module for clock synchronization.[1] Its remaining chapters are organized as follows:

1. The second chapter describes the environment of the NTI-driver, which consist of its clients and the underlying hardware components.

2. Chapter three is a presentation of the NTI-driver itself. It contains a detailed description of its interfaces to the environment and its internal structure. The different layers and their interactions are also explained in detail.

3. Chapter four is a programmer's reference. Naturally, this chapter is rather voluminous. It contains a detailed description of the NTI-driver structure from the programmer's point of view (interaction of the functions, file structure of the NTI-driver, which file keeps which functions etc.) and lists all the constants, data structures, global variables and functions in exhaustive detail.

4. Chapter five finally contains usage issues, along with a brief description of a sample application developed for experimental evaluation of the resulting system's timestamping accuracy.

5. The appendix contains a list of the used abbreviations, the bibliography and an index.

Many people spent considerable effort on the development of this piece of software, and many difficulties due to programming pitfalls, bugs in tools, and hardware problems had to be overcome in the course of the last few years. Starting out from the diploma thesis of Alfred Pusterhofer [Pus95], where the basic concept of the NTI-driver were developed, Gerda Richter developed a prototype implementation for the MVME-162 (without the NTI) documented in her diploma thesis [Ric97]. Michael Schmidt finally performed the major redesign of the NTI-driver, which now runs stable on the complete hardware.

[1]This work is part of our SynUTC-project (*http://www.auto.tuwien.ac.at/Projects/SynUTC/*), which received funding from the Austrian Science Foundation (FWF) grant P10244-ÖMA, the OeNB "Jubiläumsfonds-Projekt" 6454, the BMfWV research contract Zl.601.577/2-IV/B/9/96, and the Austrian START programme Y41-MAT.

1.1 Definitions And Notational Conventions

1.1.1 Definitions

A *message* is the unit of communication between to clients of the NTI-driver.

A *packet* is the means of communication between two drivers. It consists of a higher layer message that is written into a buffer and an envelope containing specific information about the packet.

When a packet is sent over the network medium, it is transformed into an Ethernet *frame* that contains additional data, like the CRC checksum.

An *envelope* is the control part of a packet. It primarily contains source and destination address and a type field. The NTI-driver distinguishes two types of envelopes, timestamped and normal ones. The difference will be explained in the following chapter.

A transmit resp. receive *buffer* is a contigous memory segment holding a single message. It is made up of one or more fixed-size blocks (also called buffers).

1.1.2 Notational Conventions

In order to facilitate finding indexed terms, they are printed *italic* in the text.

Variables are underlined and *functions* as well as *files* are printed italic.

A list of the abbreviations used in this report can be found in the appendix.

Chapter 2

Environment

The goal of the NTI-driver is to transparently add synchronized clocks to pSOS^{+m} systems. For that purpose, it has to multiplex three different client interfaces to the i82596 Ethernet-controller and the Network Time Interface (NTI) M-Module, as shown in Figure 2.1.

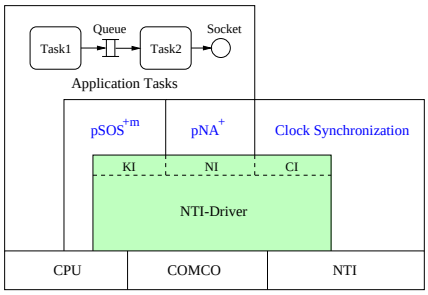


Figure 2.1: Architecture of a pSOS^{+m} node using the NTI-driver

The NTI-driver takes over the clients' messages for transmission, transforms them into packets containing the message and driver-specific information and sends them to the destination. Similarly, when an incoming packet is received, its message is extracted and finally relayed to the appropriate client.

1. *Kernel Interface* (KI): pSOS^{+m} supports multiprocessing by means of remote objects (tasks, queues, semaphores, etc.), which are implemented atop of RPCs. To keep the kernel reasonably independent of the particular communications network, a user-supplied KI is required that maps a simple message-passing interface to the particular network controller.

2. *Network Interface* (NI): In addition to kernel services, application tasks can use TCP/IP sockets for communication with remote sites if the additional software component pNA^{+}is present. Like the pSOS^{+m} kernel, pNA^{+} is kept hardware-independent by means of a user-supplied NI, which is similar to the KI but plugs into a different message-passing interface.

3. *Clock Interface* (CI): The third component that requires network services is the clock synchronization algorithm. Again, a simple message-passing interface CI is sufficient here; it is the only one that uses the timestamping feature of the NTI.

Hence, the environment the NTI-driver has to work in can be neatly parted into an upper (software) and a lower (hardware) half. The upper half is comprised of the clock synchronization and (two of the components of) the pSOS^{+m} real-time operating system *pSOS*. The lower half consists of the MVME-162 CPU, which is equipped with an MC68040 processor and an i86592 Ethernet-controller, and our NTI M-Module plugged into an AcQ i6360 or MEN A203 VME carrier board. Each of the parts relevant for the driver is described in detail in the following sections.

2.1 Clients

Two of the clients shown in Figure 2.1 are part of the pSOS real-time operating system [IS96a] (Integrated Systems, Inc.), which consists of the following basic components:

- **pSOS**$^{+}$ real-time OS
- **pSOS**$^{+m}$ multiprocessor version of pSOS^{+}
- **pNA**$^{+}$ TCP/IP network manager
- **pROBE**$^{+}$ system level debugger for pSOS applications
- **pHILE**$^{+}$ file manager
- **pREPC**$^{+}$ ANSI C Standard libraries for pSOS system
- **XRAY**$^{+}$ source level debugger for pSOS systems

Additional higher-level networking components are available as well.

All pSOS components are available for many different CPUs, including the Motorola M680x0. Apart from being CPU-specific, pSOS is completely hardware independent and can be installed anywhere in RAM or ROM. This hardware independency is accomplished by means of user-supplied *configuration tables*, which specify certain parameters like starting address and size of RAM. The connection between the pSOS components is the so-called Software Component Interconnect Bus (the *node configuration table*), where pointers to all the components can be found. A pointer to this table (called *node anchor*) is located at a fixed memory address.

Booting of the pSOS system is done by calling the pSOS^{+} kernel's startup entry after board initialization and component customization has been performed. The pSOS^{+} startup first uses the node anchor to locate the pSOS^{+} configuration table. It then takes a segment of memory and builds up its data structures and other objects. Next it checks the node configuration table. If other components are present, their startups are called to allow them to set up and initialize. Finally, pSOS^{+} Startup creates and activates the IDLE task and the user ROOT task, to which control is then passed.

2.1.1 pSOS^{+m}

pSOS^{+m} is the multiprocessor version of the real time multitasking operating system kernel pSOS^{+}. As such it performs certain services on demand, schedules, manages and allocates resources and generally coordinates multiple asynchronous activities. Many of the pSOS^{+m} system

calls are extended to operate across multiple processing nodes, which may be connected by any type of connection (shared memory, message-based buses, ...). Following the principle of functionally divided multiprocessing systems, each pSOS^{+m} node manages and executes a certain subset of the application. Typically, decomposition and assignment of functions is done prior to runtime and is thus permanent, as opposed to task reassignment or load balancing.

The pSOS^{+m} kernel supports soft fail and hot swap. Soft fail means that a node can crash and the other nodes keep on running and hot swap that nodes can be inserted into or removed from the system during operation. However, each system must have exactly one special node, called the *master node*, which manages the system and coordinates the activities of all other nodes, consequently called slave nodes. The master node is the only exception to hot swap and soft fail, since it must be present in the system at all times while the others may join and leave any time they please.

Every node is identified by a unique *node number*. This number is assigned by the user, the master node is the one with number 1. The maximum number a node can have (and thus the maximum number of nodes in the system) is specified in the multiprocessor configuration table. Apart from the master node, there may be up to 16382 nodes in the system.

The pSOS^{+m} kernel itself is, due to the configuration tables, entirely hardware independent and requires network communication primitives provided by a user-supplied *Kernel Interface* (*KI*). It contains the following functions:

- **ki_getpkb** for allocation of data buffer
- **ki_init** for initialization of the network driver
- **ki_receive** for receiving a packet that has been announced
- **ki_returnp** for returning data buffers
- **ki_roster** for keeping track of "live" nodes
- **ki_send** for sending a packet
- **ki_time** for allowing the driver to keep track of time
- **ki_announce** for announcing incoming messages to pSOS^{+m}

Viewed from the applications perspective, pSOS^{+m} is an object oriented kernel. The object classes are tasks, memory regions, memory partitions, message queues and semaphores. In a multiprocessor system those objects may be reached or referenced from any node in the system exactly as if they were all located on a single CPU. Each of these objects does have a single node of residence that can be any node in the system. An object can be identified in two ways: by the user assigned 32-bit name and by the unique pSOS-assigned 32-bit run-time ID. The ID of a *global object* (which is declared as such at its creation) is entered into the global object table of the node it resides on and also into the global object table of the master node. The number of global objects is limited to a number that is entered into the pSOS^{+m} configuration table.

When the pSOS^{+m} kernel receives a system call specifying an object which does not reside on the node, a *remote service call* (*RSC*) is executed. Such an RSC follows a certain protocol, depending on whether it is an asynchronous or a synchronous one.

Synchronous Remote Service Calls

Most RSCs are synchronous. This means that the task issuing the call is blocked until the answer to the service call arrives. The steps of a remote *q_send* operation, which is typical for synchronous RSCs, are as follows:

1. pSOS^{+m} kernel receives the call on the source node, extracts the ID and determines that this is a job that calls for an RSC.

2. The kernel allocates a packet buffer via the KI, packs the *q_send* information into it and sends the message to the destination node (again requesting the KI).

3. If the KI transmits successfully, the pSOS^{+m} kernel blocks the calling task and dispatches another one.

4. The network driver of the destination node calls the pSOS^{+m} announce packet entry.

5. pSOS^{+m} calls the KI to receive the packet, extracts its contents and generates an internal *q_send* call to deliver the message to the resident target queue.

6. If the *q_send* call is successful, the pSOS^{+m} kernel uses the packet buffer it received in Step 5 to build a reply packet and calls KI to send this packet to the source node.

7. Back on the source node, the driver announces the incoming packet to the pSOS^{+m} kernel.

8. pSOS^{+m} then calls the KI to receive the packet, extracts its contents, realizes that it is the usual termination of an RSC, returns the packet buffer, unblocks the calling task and executes a normal dispatch to return to the next application task.

Note that steps 1–3 and 7–8 are executed on the source node and 4–6 on the destination node. If an error occurs somewhere in this procedure the results may vary from system shutdowns to an error code returned to the caller. Note that RSCs are supported only if they are sent from tasks. They are forbidden in ISRs, as the overhead associated with internode communications makes them impracticable to use in ISRs.

Asynchronous Remote Service Calls

Some RSCs return nothing but an error code. In such cases it may not be necessary to block the calling task, e.g., if no error is possible. An asynchronous RSC which does not block the calling task can be used instead. The protocol is nearly the same as in the case of a synchronous call, except that the calling task is not blocked and that the destination node does not generate a reply. If an error occurs, the destination node will send an appropriate message to the source node. There, an optional routine *mc_asyncerr* given in the configuration table is called. What this routine does is up to the user; it may perform further error analysis maybe even leading to a system shutdown with the *k_fatal* system call. If this routine is not provided, pSOS^{+m} generates an internal fatal error. Note that an asynchronous operation on a local object is done using the synchronous version, since all the necessary data structures, etc., are easily accessible.

Agents

Certain system calls may require waiting at an object, e.g. *q_receive* or *sm_p*. This is easy to realize in a local operation. The remote call, however, is a bit more tricky. In case of a *q_receive*, the RSC performs almost the same steps as in a *q_send* call. The difference is that a pseudo-task, called an *agent*, is created at the destination node to perform the *q_receive* call at the target queue. The destination node proceeds with executing the ready application tasks, the agent waits on behalf of the remote task that made the RSC.

When a message is put into the target queue, the agent is awakened from the message wait queue. The pSOS^{+m} kernel writes the received message and the other data making up a reply packet into the original RSC packet buffer (the one in which the request arrived on the node). Afterwards it sends the packet via the KI back to the source node. The agent is released into the pool of unused agents. The remaining transactions are performed analogous to a "normal" RSC.

Because of all the hidden transactions, an RSC has a considerable overhead. The exact measure of this overhead is dependent on the network medium used by the system, e.g., shared memory or Ethernet.

System Startup

As has been said previously, the master node must be present in the system at all times. Consequently it is the one that has to be started first. All the other nodes may join and leave as they please after the master node has been started. A *slave node* sends a message to the master node upon joining the system to announce itself. The master performs various coherency checks concerning certain parameters before it decides to let the node join. It then notifies the other nodes in the system that a new one has joined.

Note that if the master is not running or starts running too late, the join request message is lost and the slave node cannot finish its startup procedure (and hangs). Hence, the user has to ensure somehow that no node attempts to join too early. Moreover, the joining node must not post RSCs until sufficient time has elapsed for the master node to announce the new node to all the others. Otherwise it may happen that the destination node of an RSC ignores it, because it does not recognize the sender.

Error Handling

The major distinction is between fatal and non-fatal errors. Non-fatal errors are the usual parametric or resource exhaustion problems. They are reported back to the caller who acts upon them. The other kind are the fatal errors, such as startup configuration defects or other non-recoverable conditions. The user can generate a fatal error with the *k_fatal* system call. Every such error has an associated error code, which must be greater than 0x10000 to signify a user generated error whereas smaller numbers are reserved for pSOS error codes.

When a fatal error occurs, the pSOS^{+m} kernel passes control to a user-supplied fatal error handler in the configuration table. If there is no such error handler, an internal fatal error is generated, shutting down the node or, if the masternode is the affected one, the whole system.

Interrupt Handling

Each hardware interrupt directly activates the related interrupt service routine *ISR*, the address of which is provided by the appropriate entry in the vector table. After execution of the ISR, there are two possibilities to relinquish control: the RTE-instruction and *I_RETURN*. I_RETURN returns control to pSOS^{+m}, while RTE continues at the interruption address. Since the latter may cause static priority scheduling violations, RTE may only be used if the concerned ISR as well as all those of higher priority do not execute system calls that may result in task switches. It has to be noted, though, that I_RETURN cannot be called if pSOS is not running, e.g., during system startup.

Not all pSOS^{+m} system calls are allowed in ISRs. Generally, all those that can cause the calling task to block are forbidden.

Asynchronous Signals

Asynchronous signals can be viewed as a sort of "software interrupt". The ASR executes in the same context as the task, though, so that it may use all system calls that are allowed to a task. Using *asynchronous signals* facilitates the partition of a task into two asynchronous parts: the main body and the ASR (Asynchronous signal Service Routine). Each task has 32 signals at its disposal, which are encoded bitwise in a long word. To receive signals, a task has to establish an ASR using *as_catch*. If another task sends a signal to a so prepared task, the latter is forced to go to its ASR as soon as it is next dispatched. The task returns to its original point of execution when *as_return* is called.

2.1.2 pNA$^+$

pNA$^+$ offers TCP/IP network connectivity to pSOS application tasks. Supported protocols suites include TCP, UDP, IP, ICMP, IGMP and ARP, all accessed through standard socket programming interfaces. It offers its services to application developers and higher pSOS system networking add-ons such as RPC, NFS, FTP and others. In addition, pNA$^+$ supports SNMP and the Management Information Base for Network Management of TCP/IP based Internet standard (*MIB II*).

pNA$^+$ Software Architecture

pNA$^+$ is organized in four layers, namely

- **Socket Layer:** Supplies the application programming interface. It also contains enhancements specifically for embedded real-time applications.
- **Transport Layer:** Supports the two internet transport protocols TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).
- **Internet Protocol:** Responsible for the transmission of datagrams. It provides packet routing, fragmentation and reassembly of long datagrams and support of multicast IPs.
- **Network Interface Layer:** Isolates the IP layer from the physical characteristics of the underlying network medium.

Other protocols supported by pNA$^+$ are ARP (Address Resolution Protocol), ICMP (Internet Control Message Protocol) and IGMP (Internet Group Management Protocol). The latter two are part of the IP, where ICMP is used for error reporting and other network management tasks. IGMP is used by IP nodes to report their host group memberships to any immediately neighbouring multicast routers. ARP is used to map Internet addresses to physical network addresses.

pNA$^+$ provides level 2 IP multicast capability as defined by RFC 1112, which implies support for sending and receiving IP multicast packets and an implementation of the IGMP. Therefore the NTI-driver should support multicasting, too.

Socket Layer

The socket layer is the programmer's interface to the pNA$^+$ component. It is based on the notion of sockets and designed to be syntactically and semantically compatible with UNIX 4.3 BSD networking services. Sockets themselves are end points of communication, i.e., tasks communicate by sending and receiving data through sockets. pNA$^+$ supports three types of sockets, providing three types of service:

- **Stream sockets** supply a connection based communication service. They use TCP for reliable data transmission.
- **Datagram sockets** allow a connectionless communication. They use UDP to exchange data with a minimum of overhead, though at the cost of reliability.
- **Raw sockets** provide user level access to the IP, ICMP and IGMP, enabling the user to implement transport protocols over the IP layer. They allow connectionoriented and datagram communication service.

When two tasks wish to communicate, the first thing to do is to create a socket. In the case of stream sockets a connection must then be established. Such a connection is usually asymmetric, consisting of a client and a server. The server binds its socket to an address and then waits for clients to call. The client must set off a connection request to the address bound to the server. If the server accepts the connection, it creates a new socket with the same properties and connects it to the client's socket. As soon as the connection is established, data can be transferred. It has to be closed after the end of transmission. Note that stream sockets support so-called out-of-band, (i.e., high priority) data.

When using a connectionless or datagram socket, the destination address has to be specified at each send command.

The pNA$^+$ Daemon Task

There are three possible outcomes of pNA$^+$ system calls:

1. The requested service is executed and control returns to the caller.
2. The system call cannot be completed immediately, but it does not require the caller to wait. In this case the pNA$^+$ schedules the necessary operations and returns control to the caller (e.g. if a transmission cannot be executed immediately, due to resource problems).

3. The system call cannot be completed immediately and the caller must wait. In this case, it is blocked and eventually rescheduled by subsequent asynchronous activity.

As indicated above, the Internet protocols are not always synchronous. To handle asynchronous operations, the pNA$^+$ component creates a daemon task called PNAD. This is done during pNA$^+$ initialization. The PNAD's priority has to be higher than that of any tasks calling the pNA$^+$ component. Usually the PNAD is blocked and waiting for one of two events (at the arrival of either it unblocks and preempts the running task):

- The first event is sent upon reception of a pNA$^+$ packet (i.e. upon call of the pNA$^+$ announce packet entry point by the network driver). Further action of the PNAD depends on the contents of the packet (unblocking of a task, forwarding of the packet, etc).
- The other event is sent every 100 ms and causes PNAD to perform time-specific actions for TCP (retries, timeouts). Afterwards it polls each interface that is unable to call the packet announce entry point upon packet reception.

User Signal Handler and Error Handling

pNA$^+$ defines a set of asynchronous signals that indicate unusual conditions arising during normal operation. The handler for these signals can be provided by the user via the pNA configuration table. When called, the handler receives the signal type, the descriptor of the affected socket and the task ID of its owner. If no signal handler is entered into the configuration table the signals are ignored.

Errors are reported according to UNIX BSD 4.3 socket level error reporting mechanisms. Upon an error condition an error code is stored in the variable errno and -1 returned to the caller. To get the errorcode, the caller has to read it prior to making another system call.

Network Interfaces

The pNA$^+$ component accesses the network by calling a user-provided layer of software, the *network interface* (*NI*). This NI is a simple message passing interface (similar but not equivalent) to the KI that is independent of the network's physical characteristics. The NI has to provide the following functions:

- **ni_initial** for initialization of a network interface
- **ni_getpkb** for allocating data buffer
- **ni_retpkb** for deallocating data buffer
- **ni_send** for sending an NI packet
- **ni_broadcast** for broadcasting an NI packet
- **ni_poll** to allow polling for incoming messages
- **ni_ioctl** for performing I/O control operations
- **ni_announce** for letting the NI hand up incoming messages

Apart from the last function, all NI functions called via a dispatch entry point. The messsage announce entry point is handed to the driver during initialization. Note that the NI need not provide all the features of the LLC (logical link control) and ARP, which is done within pNA$^+$. In fact, the network interface is used only as a basic message passing subsystem.

From pNA$^+$s point of view, there must be one NI for each network connected to a pNA$^+$ node, though this may be realized by a reentrant driver which is handed the number of the required interface. Each of the interfaces is assigned a unique IP address. Each network interface also has a number of associated attributes:

- the address of the NI entry point, which may be the same for different interfaces. pNA$^+$ uses it for calling the functions the interface provides.
- the unique IP address
- the maximum transmission unit (MTU)
- the length of its hardware address
- control flags
- the IP network subnet mask
- for point-to-point links the IP address of the destination node

Most networks are limited in the number of bytes that can be physically transmitted in a single frame. Therefore the *MTU*, which is the maximum packet size that can be sent or received, must be specified for each network interface. The IP layer performs fragmentation and reassembly, if necessary. The minimum MTU allowed by pNA$^+$ is 64 bytes, there is no maximum limit. For Ethernet, the MTU is 1500 bytes.

Every network interface has a *hardware address* in addition to its Internet address. The latter is used by the IP layer, while the former is used by the network driver when transferring the actual packets on the network. The mapping between those addresses is called address resolution and usually done according to the ARP. The length of a hardware address is dependent on the type of the network. For example, Ethernet uses 6-byte addresses. The pNA$^+$ component can support hardware addresses with a length of up to 14 bytes.

The characteristics of a particular network interface are described by a set of *control flags*:

ARP: If this flag is clear (and thus ARP performed by pNA), broadcast must be enabled as well

Broadcast Tells the pNA$^+$ whether broadcasting is supported

Extloopback: If this is disabled, the pNA$^+$ component internally loops back packets that are addressed to itself (the NI is not called)

Multicast: Tells whether multicast is supported

Poll: Driver must be polled for received packets, which is the case if the driver cannot call the announce packet entry upon packet reception (e.g. if the hardware does not provide interrupts)

Pointtopoint: If this is set the connection is a point-to-point interface.

Rawmem: If this is set, the driver understands and provides pNA$^+$s message structure (see Section 2.1.2 for details).

Unnumbered: If this is set, the network connection is an unnumbered point-to-point link.

ARP

As has been said before, every NI has two addresses associated to it: an IP address and a hardware address. The IP layer uses the former, while the network driver uses the latter. The process of mapping the addresses is called address resolution. The pNA$^+$ component goes through the following steps when performing address resolution:

1. It examines the NI control flags to determine whether it should handle address resolution internally or let the driver handle it.
2. If the ARP flag is disabled, the pNA$^+$ component's ARP table is searched for an entry containing the Internet address. If the entry is found, the corresponding hardware address is passed to the NI.
3. If there is no entry, the pNA$^+$ component uses ARP to obtain the hardware address dynamically.

For initialization purposes, an initial ARP table can be supplied. Dynamically determined addresses are written into the table. Additional mappings can also be added dynamically by the appropriate system call.

Memory Management

As packets are handed up and down through the different protocol layers, they are subject to several manipulations, such as addition or deletion of protocol headers, fragmentation or reassembly of packets, copying of packets, etc.

The basic unit of data used internally by the pNA$^+$ component is the *message*, consisting of a linked list of message block triplets. Each of these triplets contains a message block, a data block and a buffer. The message block contains the characteristics of the partial message defined by the message block triplet. A data block contains the characteristics of the buffer to which it points, which is a contiguous block of memory containing data.

The basic unit of transmission used by the protocol layers in the pNA$^+$ component is a packet. It contains a protocol header and the encapsulated data. Each layer adds a header to the packet and passes it to the lower layer for transmission, where the whole is interpreted as data to transmit. Packets are internally stored in the form of messages, which allows to circumvent copying the packet's data when handing it through the protocol layers ("zero-copy feature").

The IFF_RAWMEM flag tells whether the network interface can use those data structures as well. If so, it must understand messages handed over for transmission in message block triplets. Similarly, the NI must accomodate received messages in buffers allocated by pNA$^+$'s allocation functions (supplied via *ni_initial*) before calling *ni_announce*. In this case, the NI-function *ni_getpkb* and *ni_retpkb* are not needed. Those functions are only called if the NI does not have RAWMEM-capabilities, in case of which pNA$^+$ performs the necessary copying of data.

MIB-II Support

The pNA$^+$ component supports the Management Information Base (MIB-II), a tool for managing nodes in a TCP/IP network. It is defined in the internet standard RFC 1213. The *MIB-II* is used by the SNMP but can also be accessed by applications. RFC 1213 groups MIB-II objects into the following categories:

- **System:** Support is provided by the pSOSystem SNMP library
- **Interfaces:** The objects defined in this group are maintained by the NI or by the driver providing the NI. The objects are accessed via the *ni_ioctl* system call. See Section 3.2.3 for details.
- **Address Translation:** This group is being phased out of the MIB-II specification. Its functionality is supported by the IP group.
- **IP:** The support for this group is the responsibility of the pNA$^+$ component
- **ICMP:** like IP
- **TCP:** like IP
- **UDP:** like IP
- **EGP:** Not included in pNA$^+$ and hence not supported.
- **Transmission:** Is not yet defined and hence not supported.
- **SNMP:** Here the support is provided by the pSOSystem SNMP library.

2.1.3 Clock Synchronization

This final client of the NTI-driver is not imposed by the pSOS system, but will eventually provide each node with a highly accurate common time base. Although the NTI-driver does not impose a particular algorithm, we are primarily interested in *interval-based clock synchronization*, which means that what is transmitted, compared and computed are *accuracy intervals* containing (or, to be more correct, assumed to contain) real time. They are built by combining the ordinary clock value with an interval of accuracies that changes with time. If no resynchronization takes place, the accuracy intervals deteriorate according to the maximum drift of a node's clock.

Actually kept independent of the other clients, the clock synchronization algorithm has been placed directly above the NTI-driver for performance reasons (see Figure 2.1). It performs in periodic rounds that involve the following basic steps:

- **Transmission**: Each node periodically broadcasts a clock synchronization message that contains the local time and the current accuracy.
- **Message Reception**: When a clock synchronization message arrives from another node, a tuple consisting of the message and the local time at which the message was received is stored.

- **Synchronization**: An interval-valued convergence function is eventually applied to the received accuracy intervals to compute and subsequently enforce an improved accuracy interval to the local clock.

The clock synchronization requires a simple message passing interface that allows to uni/broadcast timestamped messages. For high accuracy, the sending resp. receiving timestamps have to be generated as close to the transmission resp. reception event as possible. This is acutally done in hardware by our NTI M-Module, see Section 2.3. Moreover, timestamped messages must be sent with high priority w.r.t. ordinary messages queued for transmission. This feature allows to considerably reduce the broadcast latency. In addition to timestamped messages, the NTI-driver must also provide a means to transmit ordinary (= non-timestamped) messages of variable length.

Therefore, the CI contains the following functions:

- **ci_init** for initialization of the driver
- **ci_get_tbuf** for allocation of transmission buffer
- **ci_get_tenv** for allocation of a transmission envelope
- **ci_ret_tbuf** for returning transmission buffer
- **ci_ret_tenv** for returning a transmission envelope
- **ci_ret_rbuf** for returning receive buffer
- **ci_ret_renv** for returning a receive envelope
- **ci_send** for sending a message
- **ci_announce** for being handed a received message by the NTI-driver
- **ci_announce_error** for being notified of a transmit error by the NTI-driver

Regarding the implementation of the software part of the clock synchronization, there are two possibilities. It may be implemented completely as an ISR, or as a combination of a task with an ISR, the latter only storing received messages and notifying the task that something has come in. In the former case the functions of the CI are called in the context of an ISR, in the latter in that of a task.

2.2 Network Controller

The network controller used in this version of the NTI-driver is Intel's i82596CA Ethernet-controller [Cor92], from now on referred to as "*LANCE*" or "*i82596*". It is a 32-bit LAN co-processor which, among other protocols like HDLC, primarily supports IEEE 802.3 CSMA/CD. It can also be configured for full duplex operation and even CSMA/DCR (Deterministic Collision Resolution), thus providing a bounded delay until channel access (a feature important for real-time applications).

The CSMA/CD unit handles all OSI MAC (Medium Access Control) functions (layers 1 and 2), i.e., implements all IEEE 802.3 MAC and channel interface functions like framing, preamble

generation and stripping, source address generation, destination address checking, short frame detection and automatic length field handling. Note that many of the LANCEs parameters can be changed by software, making it flexible and therefore usable in various settings.

Basically, the chip consists of three subsystems: parallel, serial and FIFO.

- The parallel subsystem handles the interface between the coprocessor and its host system. Its components (bus interface unit, micro machine, DMA unit and data interface unit) communicate over an internal 32-bit P-bus controlled by the micro machine.
- The serial subsystem provides the interface to the network. It consists of CSMA/CD units (extra units for reception and transmission), a data encoder and decoder, carrier sense/collision detect logic and HDLC bit insertion and deletion logic.
- The FIFO subsystem is located between the two others. It consists of a 128 byte receive FIFO, a 64 byte transmit FIFO and control logic.

As the LANCE has *DMA* and memory management capabilities, it relieves the host CPU of tasks related to controlling and managing communication via LAN. More specifically, its on-chip DMA channels fetch resp. store data from resp. into memory completely independent of the CPU. It can be configured to use either segmented addresses, adding a 16-bit offset to a 32-bit base address, or full 32-bit linear addresses. The latter mode blows up the size of the data structures considerably, but makes it feasible to use memory areas of more than 64k.

2.2.1 Operational Model

Since the i82596 links the network and the host CPU, it must be able to execute commands simultaneously to receiving incoming frames. Therefore, the chip utilizes two independent units for transmission and reception, the Command Unit (*CU*) and the Receive Unit (*RU*).

Communication between the host CPU the i82596 takes place via shared memory. The CPU places CU commands in the shared memory area, activates *Channel Attention* (*CA*) to notify the LANCE and returns to its other duties. When sensing CA, the coprocessor's CU checks this area and executes the commands without further intervention from the CPU. After command completion, an interrupt is generated to notify the CPU.

The reception of frames also works without CPU supervision. All that has to be done is to setup a chain of receive buffers adequate for the rate of reception and to activate the RU. Packet reception and writing received packets into the buffers is then performed completely transparent by the i82596, apart from the fact that CPU interrupts are raised to signal packet reception.

2.2.2 i82596 Memory Structure

As far as the LANCE is concerned, the shared memory consists of four parts (see Figure 2.2): the initialization root, the system control block, the command block list and the receive frame area.

The *initialization root* is used to relay the address of the system control block to the LANCE. At initialization its address is given to the LANCE via a special port command. If no address comes in, the coprocessor uses a default value and expects the data structures of the initialization root (see Section 3.5.1 for details) at address 0x00FFFFF4.

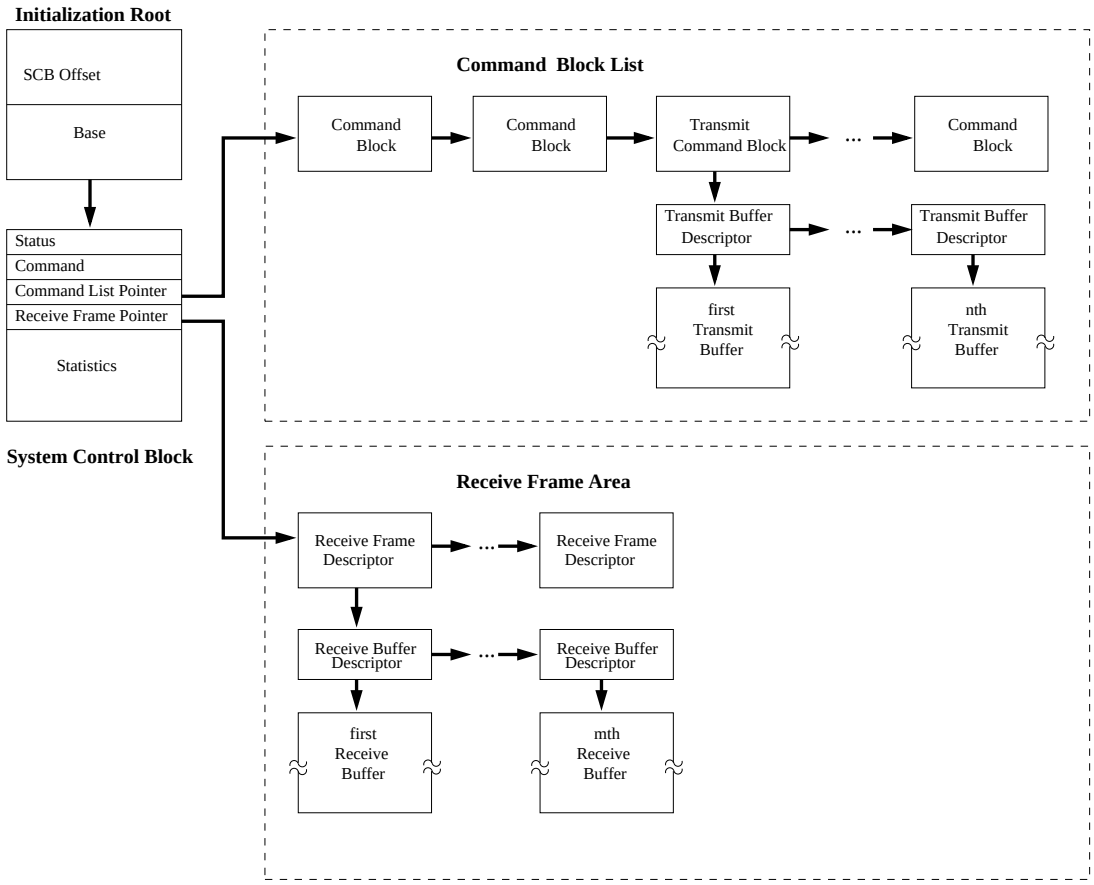


Figure 2.2: Shared memory structure of LANCE i82596

The System Control Block (*SCB*) is the primary means to exchange status information and commands between host CPU and coprocessor. The first part of the SCB contains the bytes for control of CU and RU, the pointer to the command list and the pointer to the receive frame area. The second part contains statistical data and status information.

The command block list (*CBL*) serves as a sort of program for the CU, where each block represents a special command the CU has to execute. The CPU assembles a chain of such commands, enters its address into the SCB and then starts the CU. The CU executes one comman block after the other and stops when it has executed the last one. Having done this it goes to the IDLE state. Appending or inserting a command (in)to the command block list is possible, though one has to bear in mind that all kinds of race conditions can arise since the LANCE cannot be stopped in its execution.

The following different *LANCE commands* exist:

1. **NOP** No operation
2. **IAS** Individual Address Setup, telling the LANCE its hardware address
3. **CFG** Configure, configure LANCE
4. **MCS** Multicast Setup, telling the LANCE the multicast addresses
5. **TRX** Transmit, submitting a message (transmit command block) for transmission

6. **TDR** Time Domain Reflectometry, for detecting open or short circuits on the link and their distance from the diagnosing station

7. **DUMP** for writing the contents of the internal registers into a buffer

8. **DIAG** for triggering an internal self-test procedure

Transmit

A *transmit packet* is represented by a dedicated *transmit command block* (*TCB*), which contains the *envelope* information like destination address, type etc. as well as a pointer to a linked list of transmit buffer descriptors (TBDs). Each TBD contains the address and size of a dedicated (contiguous) data buffer as well as the TBD link field. All transmit buffers (TBs) together contain the whole data message, which hence needs not be stored contiguously to be transmitted correctly by the LANCE.

This way of organizing the packet is called the *flexible memory model*, as opposed to the *simplified model*, where each TCB has the data buffer appended. The problem with the latter is, that either a new TCB of the appropriate size has to be freshly allocated for each transmit request (eliminating the possibility of keeping a pool of TCBs) or that all TCBs have to be of the same size, i.e., that of the largest packet expected for transmission. The latter is only feasible, though, if all packets in the network are of about the same size. Since the sizes of pSOS^{+m} and pNA^{+} messages may differ by a factor of up to 15, this is not the case for the NTI-driver. Hence, the flexible memory model was chosen.

In the flexible memory model, the following steps have to be taken to transmit a message.

1. First a TCB has to be allocated.

2. Then as much TBs (together with their associated TBDs, of course) as necessary to accommodate the message have to be allocated

3. The necessary information (destination address, length field, etc.) must be put into the TCB together with directives for the CU (about the applicable memory model etc).

4. The TCB is put into the CBL and handed to the LANCE, which executes the command and generates an interrupt after completion (if the appropriate flag in the TCB is set or the TCB is the last one in the CBL).

5. The corresponding ISR checks whether the command was executed correctly and no error happened.

6. Further processing depends on the type of the packet.

In the TCB, several bits are reserved for reporting certain events:

- Transmission successful

- Transmission unsuccessful - cause: lost carrier sense

- Transmission unsuccessful - cause: lost clear to send

- Transmission unsuccessful - cause: DMA underrun, system bus did/could not keep up with transmission
- Transmission unsuccessful - cause: number of collisions exceeded the allowed maximum
- The number of collisions that happened during the frame transmission
- Indication of the presence of a heartbeat during the last interframe spacing period after transmission.

Receive

Incoming frames are transparently stored in a certain linked memory structure called *receive frame area* (*RFA*), which must be properly set up and pointed to in the SCB prior to starting the RU. The RFA consists of a linked list of free *receive frame descriptors* (*RFDs*) holding the *envelope* information of a single received packet, and a linked list of free *receive buffer descriptors* (*RBDs*). Each RBD is associated with exactly one *receive buffer* (*RB*) and contains its size, address, the RBD link field and the number of bytes actually written into it during reception. As for transmit commands, this memory organization is called the *flexible memory model*, since it allows reception of packets of varying size. Again, there is also a simplified memory model, where each RFD also contains a maximum size data buffer. For the NTI-driver, however, we use the flexible memory model due to the varying packet sizes.

When reception of a frame begins, the LANCE takes the first free RFD from the RFA and writes the envelope data into it. As packet reception proceeds, it takes successive RBDs from the list of free ones, links it to the already filled one resp. the RFD and writes the data into it. Note that if a bad frame is recognized, then the RFD and RBDs used for it are reclaimed and hence reused for the next incoming packet. Of course, the LANCE will continue to receive as long as RFDs and RBs are available.

After reception of an entire frame, an interrupt is generated to notify the host CPU and the appropriate bits in the SCB set:

- Incorrect CRC in a properly aligned frame
- Incorrect CRC in a misaligned frame
- Frame is shorter than the minimal frame length the LANCE is configured for
- DMA Overrun - Part of the frame was not placed in memory because the system bus did/could not keep up with incoming data
- Part of the frame was discarded because of insufficient storage space
- A collision was detected during reception
- A frame not matching the frame length parameter was detected

Moreover, the SCB contains six 32-bit statistical counters that are updated after each received frame that passes address filtering and is longer than the configured minimum frame length.

- Number of wellaligned frames with CRC errors

- Number of misaligned frames with CRC errors
- Number of frames discarded due to insufficient resources
- Number of frames discarded due to overrun
- Number of collisions detected during receive
- Number of frames discarded due to being shorter than the minimum frame length

2.3 Network Time Interface (NTI)

High-accuracy clock synchronization requires a certain amount of hardware support, which should be compatible with existing systems and network architectures. This is satisfied by our Network Time Interface (*NTI*) M-Module, see [HSS98] for an overview and [MNS99] for a comprehensive description. Fig. 2.3 shows the major components on-board the NTI, which can be accessed from any CPU and network controller with MA-interface via ordinary memory and memory-mapped registers.

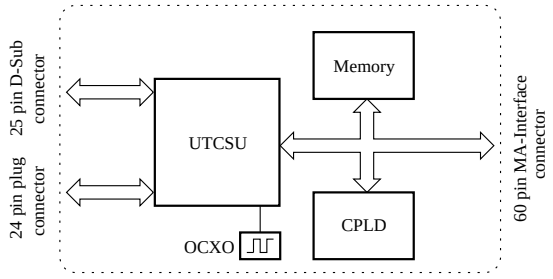


Figure 2.3: NTI block diagram

The major blocks of the NTI are as follows:

- The *shared memory* serves as control and data interface between the CPU and the network controller, providing special functionality for accurate timestamping of network data packets accomodated therein. It consists of up to four 64k x 16 bit SRAM chips and supports byte, word, and longword read/write accesses.
- The *UTCSU-ASIC* (*Universal Time Coordinated Synchronization Unit*) described in [SSHL97], [Loy96] contains most of the dedicated hardware support for clock synchronization. Manufactured as an ASIC in 0.7 μm CMOS technology, the UTCSU accommodates about $80,000$ gates on a 100 mm^2 die packed into a 208-pin PQFP case.

  The centerpiece of the UTCSU is a *local clock unit* (LTU) utilizing a 56 bit NTP-time format. Clock time can be read atomically as a 32 bit *timestamp* with resolution $2^{-24} \approx 60$ ns and a 32 bit *macrostamp* containing the remaining 24 most-significant bits + an 8 bit checksum protecting the entire time information. The LTU employs an unconventional *adder-based* clock design, which uses a large (91 bit), high-speed adder instead of a simple counter for summing up the elapsed time between succeeding oscillator ticks. Owing to this,

the UTCSU can be paced by a quartz oscillator of arbitrary frequency $f_{osc} \in 1 \ldots 20$ MHz; alternatively, an external frequency source like the 10 MHz output of a high-end GPS receiver can be used. Moreover, the local clock is fine-grained rate adjustable in steps of about 10 ns/s and supports state adjustment via continuous amortization as well as (optional) leap second corrections in hardware.

To support interval-based clock synchronization, the UTCSU contains two more adder-based "clocks" in the ACU that are also driven by the oscillator frequency f_{osc}. They are responsible for holding and automatically deteriorating the 16 bit accuracies α^- and α^+, thereby maintaining a bound on the local clock's instantaneous deviation w.r.t. real-time.

A number of external events, supplied to the UTCSU via buffered/opto-decoupled, polarity programmable input lines, can be *time/accuracy-stamped* with local time and accuracy upon the appropriate input transition. Optionally, an interrupt can be raised on such an event as well. Three different functional blocks in the UTCSU utilize this feature: First, trigger signals generated by the decoding logic at CSP transmission and reception sample the current local time/accuracy into dedicated UTCSU registers in one of the six available SSUs. Second, three independent GPUs are provided for timestamping the *one pulse per second* (1pps) signal —indicating the exact beginning of a UTC second— from up to three GPS receivers. Finally, nine independent application time/accuracy-stamping inputs are provided by the APU.

Those timestamping features are complemented by several 48 bit programmable *duty timers*: Whenever local time reaches the programmed time of an armed duty timer, an internal or external signal changes state and an optional interrupt is raised. Duty timers are required for triggering activity of the clock synchronization algorithm, for controlling continuous amortization, inserting/deleting leap seconds, and generating application-related events.

- All decoding and glue logic of the NTI is assembled in a single, in-circuit programmable *complex programmable logic device* (CPLD) programmed in VHDL [NKC+98]. It adapts the UTCSU and the memory to the MA-Module interface, forwards interrupt requests from the UTCSU to the carrier-board, generates the acknowledgement signal terminating a bus cycle, etc.

The primary purpose of the NTI w.r.t. networking is highly accurate timestamping of network packets. In purely software-based clock synchronization, this is done at the sending resp. receiving end by reading the clock when assembling the data packet for transmission resp. in the packet reception interrupt service routine. However, as explained in detail in [HSS98], this implies that the transmission delay uncertainty ε includes both the network channel access uncertainty and the reception interrupt latency, which can be quite large. To get rid of them, the NTI transparently inserts a timestamp on-the-fly into the memory holding a packet in a way that minimizes ε, as outlined in Fig. 2.4.

Whenever the network controller fetches data from the *timestamped transmit envelope* of the data packet, it has to read across the particular offset that causes a special decoding logic to generate the trigger signal TRANSMIT. Upon its occurrence, the UTCSU-clock puts a *transmit timestamp* (TxTS) into a dedicated sample register, which is transparently mapped into a certain portion of the transmit envelope and hence automatically inserted into the outgoing packet. Note that the trigger address and the mapping address are configurable parameters that may be different. By the same token, when the receiving network controller writes a certain offset in the receive

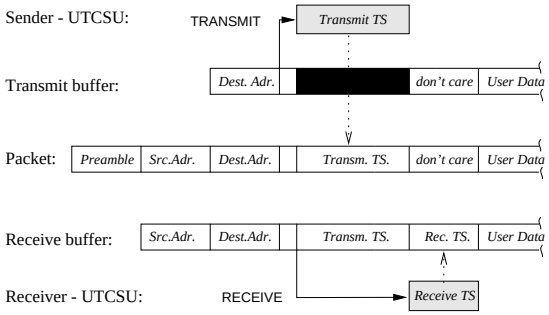


Figure 2.4: Packet timestamping

buffer, the trigger signal RECEIVE is generated by the decoding logic, which causes the UTCSU to sample the *receive timestamp* (RxTS) into a dedicated register. This timestamp is saved subsequently in an unused portion of the receive buffer by an interrupt service routine.

For cost reduction purposes, the NTI-driver must be able to handle more than one NTI per physical node to emulate up to four *virtual nodes*. Figure 2.5 shows its basic structure.

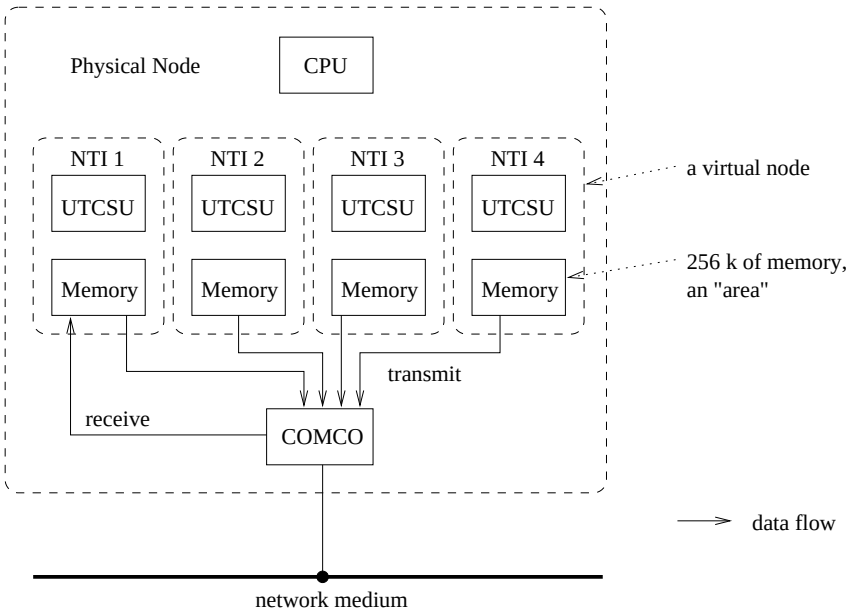


Figure 2.5: Structure of a physical node with multiple NTIs

Outgoing CI-messages "originate" in exactly one of the multiple NTIs. Thus, up to four different sets of transmit buffers and timestamped transmit envelopes have to be managed by the NTI-driver. Since each NTI provides its own shared memory ared, this is relatively easy to manage: The number of the originating *area* has to be specified at the CI's transmit or buffer allocation request.

Reception handling for virtual nodes is different, since the network controller cannot write incoming packets into multiple memory areas at once. Hence, reception is only done into the shared memory area #0, but the receive timestamps are triggered simultaneously at all NTIs

by hard-wiring the RxTS-input of all UTCSUs with area #0's RECEIVE trigger signal.

All software handling of the NTI, including the reception interrupt handling for moving the receive timestamps of all areas into the packet's envelope, is performed by the *NTI-Handler* [SM99]. Therefore, apart from importing the layout of the NTI shared memory (number of timestamped envelopes, timestamp trigger and mapping offsets) and calling the initialization function, the NTI-driver needs not care about the NTI at all. It is important, though, to understand the interrelation in detail if changes are made to the memory layout.

Chapter 3

NTI-driver

This chapter describes in detail the structure of the NTI-driver. First the reader will find an overview, detailing which modules the software consists of and how they interact. He will find a description of each function, of what it has to do and how it goes about this. What was not taken into consideration here are implementation details, mutual exclusion issues etc. Please see the next chapter for such information. Small functions which are of no relevance to the functionality of the NTI-driver (such as clearing a certain memory area) are also not listed here and can be found in the next chapter.
After the functional description follows a list of the parameters, giving their types and names together with a short explanation of their meaning. See Section 4.4 for details on the NTI-driver typical data types. The specification of the return values of the function forms the conclusion. They may either be a direct result of the function or the return value of a routine called during execution. Constants and data structures used in this section are described extensively in the next chapter. See there for additional information, such as the actual values.

3.1 Overview

The driver of the NTI consists of three layers. This partitioning has been undertaken according to the duties of each of these layers. Each layer in turn is parted into two or three modules, again according to the various duties they have to perform. The structure of the NTI-driver can be seen in Figure 3.1, a more detailed one is provided in Figure 4.1. The uppermost layer is responsible for the communication with the applications using the NTI-driver. As there are three clients, pSOS^{+m}, pNA^{+} and the clock synchronization, there are three modules in this layer, each with the necessary functions to provide its interface. The middle layer has to coordinate the data transfer between the upper and the lower layer. It assembles packets from client messages and envelopes or strips packets of the envelope to gain the client message.
The lowest layer handles the data structures for communication with the LANCE. It controls the queue of commands to the network controller and conveys received messages to the middle layer.

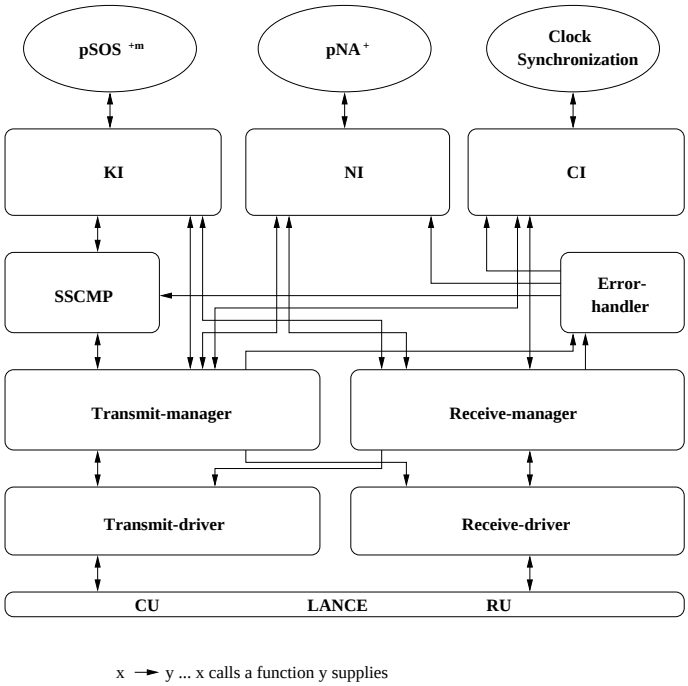


Figure 3.1: Structure of NTI-driver

3.2 Application Layer

3.2.1 Description

The *application layer* is the one communicating with the clients that use the NTI-driver, i.e. the pSOS^{+m} kernel, the pNA+ component and the clock synchronization. Of these only the third, the clock synchronization, has an interface that has been developed more or less concurrently with the NTI-driver. It is called the clock synchronization interface (CI). The others have predefined interfaces: pSOS^{+m} using the KI to communicate with the NTI-driver and pNA^{+} the NI.

3.2.2 KI

The *Kernel Interface* has been defined to regulate communication between network driver and pSOS^{+m} kernel, thus rendering the latter independent of the communication medium. The following services must be provided by the NTI-driver:

- **ki_getpkb** for allocation of a data buffer
- **ki_init** for initialization of the network driver
- **ki_receive** for receiving a packet that has been announced
- **ki_returnp** for returning a data buffer
- **ki_roster** for keeping track of "live" nodes
- **ki_send** for sending a packet

- **ki_time** for allowing the NTI-driver to keep track of time

Essentially, all those routines are called as simple subroutines by the kernel, where the parameters are passed in registers. The return is assumed to be a subroutine return (using RTS), with the return value in register D0. Thus entry and return use the following procedures:

Entry pSOS^{+m} calls the KI with a JSR instruction. It passes a function code in D0.L to select the specific KI service. Additional input parameters are passed in assigned registers.

Return The NTI-driver should use RTS to return to the pSOS^{+m} kernel. An error code must always be returned in D0.L, 0 indicating success. Error codes must be greater than 0x10000000, i.e, a negative number. All registers except D0.L and those used by the return parameters must be restored. When the KI is called by pSOS^{+m} its state is supervisory. The interrupt level must not be lowered but may be raised, provided it is restored prior to returning to pSOS^{+m}.

In turn, there have to be entry points to announce incoming messages or error conditions to the pSOS^{+m} kernel. At initialization the NTI-driver is handed the pointer to a function which serves for announcing an incoming message. It needs no parameters and has no return value. It causes pSOS^{+m} to call *ki_receive* to get the message from the NTI-driver.
Errors result in a *k_fatal* system call, if they happen asynchronously. Synchronous errors are reported directly using the return codes.

The NTI-driver is not a pSOS^{+m} I/O driver. It is therefore subject to certain restrictions concerning the pSOS^{+m} system calls that may be made. Generally all the calls allowed in an ISR can be used. In addition, it can send a system call if the following conditions are met:

1. The call does not generate a recursive request to the KI (e.g. an RSC). This means, that only local objects are available for the NTI-driver!
2. The call does not attempt to block.

The NTI-driver may be called when pSOS^{+m} is not yet active. Consequently the previously stated rule, that any call allowed in an ISR is also allowed in the NTI-driver, has to be revised: The NTI-driver must not use any pSOS^{+m} or pRPC^{+} system calls, since it may be active when pSOS^{+m} is not.

As the NTI-driver has to return immediately after the transmission a failure caused by the other node being unable to receive or by faulty connections cannot be announced as the result of one particular transmit command. If the protocol described in Section 3.3.2 detects such an error it calls the error handler.
A receive error is usually a resource error and not reported to the application.
In the following, the services and certain implementation issues/constraints are described in more detail:

ki_getpkb

obtains a packet buffer from the NTI-driver. The function hands the request down and gets either the pointer to a contiguous data area or NULL. In both cases that value is returned in register A0.

The input parameters are:
D0 = 2
D1... Packet size in bytes.
have a size of exactly 100 (see [IS96b, p.2-23]).
D2... Destination node number, 0 means broadcast
A0... here the address of the allocated packet buffer or NULL respectively is returned.

The return value in D0 is 0 if everything went OK, or
NO_BUF_FREE if there was no buffer space available,
NETWORK_NOT_FUNCTIONAL if the KI was not initialized

ki_init

initializes the NTI-driver. This means initializing the communication hardware as well as the data structures. pSOS^{+m} also hands down the address of the KI message announce entry point, which is kept in a global variable. It is called whenever a message arrives for the KI.
Ki_init is called with the interrupt mask at level 7. The level can be dropped if this proves necessary (contrary to the other functions), provided it is raised prior to return.

The input parameters are:
D0 = 1
D1... Address of pSOS^{+m} Announce packet entry

The return value is 0 if everything went OK, or
INVALID_IF_NUMBER Interface Number not valid.
NETWORK_NOT_FUNCTIONAL if there is a problem with the hardware.

ki_receive

obtains a received packet. This function is called after the message announce entry point has been called or, on a slave node, to poll for the reply to the join request message (see description of pSOS^{+m} in Section 2.1.1) during startup. It must present the messages in FIFO order.

The input parameters are:
D0 = 6
A0... here the address of the waiting packet buffer is returned

The return value is 0 if everything went OK, or
INVALID_IF_NUMBER Interface number not valid.
NO_MESSAGE_THERE no message was waiting (fatal error).

ki_returnp

returns a receive buffer to the NTI-driver. Only the buffer is handed over to the driver, there are no other parameters. This pointer is handed down to the lower layers where the address of the corresponding control structure is computed and then the buffer freed and returned to the NTI-driver. If the return value of this function is not 0, pSOS^{+m} generates an internal error.

The input parameters are:
D0 = 3
A0... pointer to packet buffer

The return value is 0 if everything went OK, or
INVALID_BUF_ADDRESS if the pointer specified a nonexisting buffer
NETWORK_NOT_FUNCTIONAL KI not initialized.

ki_roster

provides roster information for the KI. Each node has a unique node number and gets a sequence number upon joining the system. The roster is a table kept by each of the nodes that lists the nodes currently active in the system. pSOS^{+m} provides a means for the NTI-driver to keep its own roster table, which is updated by this function. The NTI-driver does not need this information so that the function is empty.

The input parameters are:
D0 = 9
D1... type of roster change:
0... initial roster, A0 points to the internal pSOS^{+m} roster
1... a node has joined, D2 contains node number, D3 its sequence number
2... a node has failed, D2 contains node number, D3 its failure code, D4 the number of the node that initiated the removal of the node from the system (may be the failed node itself)
D2... node number
D3... not needed in this implementation
D4... not needed in this implementation

The return value is 0 if everything went OK, or
NETWORK_NOT_FUNCTIONAL KI was not initialized.

ki_send

transmits a packet. The buffer has to be released automatically after transmission. Instead of the real address, the NTI-driver gets the number of the destination node. It has to perform address resolution to get the hardware destination address. The NTI-driver must not block (see below), so it sends the message to its destination and returns immediately. Since pSOS^{+m} expects the transmission to be reliable, though, a suitable protocol has been included in the NTI-driver (See Section 3.3.2). If a transmission error occurs asynchronously, the fatal error handler is called, which results in a *k_fatal* call. See the error handling procedure in Section 2.1.1 for details.
The buffer handed down for transmission may be a reused receive buffer. This possibility has to be considered when releasing the packet.
It is assumed that pSOS^{+m} does not call *ki_send* again while another call is still being processed. If that should happen there would be the distinct possibility of message reordering.

The input parameters are:
D0 = 4
D1... packet size in bytes
D2... destination node number
A0... pointer to packet buffer

The return value is 0 if everything went OK,
NO_ENV_FREE if there was no envelope available,
NETWORK_NOT_FUNCTIONAL KI not initialized.

INVALID_BUF_ADDRESS Address doesn't exist or wrong.
INVALID_ENV_ADDRESS Address doesn't exist.
BUFSIZE_TOO_SMALL allocated a buffer that is too small for size.

ki_time

announces a clock tick to the NTI-driver. The function is called periodically by pSOS^{+m} at every clock tick. It facilitates the implementation of retransmission timers. It is used to call the watchdog used in the sliding window protocol and a function, which asynchronously returns receive buffer descriptors.

The input parameters are:
D0 = 7

The return value is always 0.

3.2.3 NI

As the KI does for pSOS^{+m}, the network interface (*NI*) renders the pNA^{+} independent of the underlying network. The pNA^{+} requires one entry in its table of NIs for each network connected to a pNA^{+} node. Each of those network connections must be assigned a unique IP address. Multiple network controllers, e.g. in gateways can also be serviced by one reentrant driver. The distinction between the different physical devices is made through the numbers they are assigned: With each function call the pNA^{+} specifies which interface the request is intended for. pNA^{+} use other interfaces apart from the NTI-driver, so that the numbers it uses to identify the different ones may be different from the ones used internally in the driver. This makes mapping necessary, which is realised through introduction of two tables, one for translation from pNA^{+} number to NTI-driver number and one for translating in the other direction.

The following services make up the NI:

- **ni_initial** for initialization of a network interface
- **ni_send** for sending an NI packet
- **ni_broadcast** for broadcasting an NI packet
- **ni_poll** to allow polling for incoming messages
- **ni_ioctl** for performing I/O control operations

pNA^{+} calls the NI services as subroutines with a JSR instruction.

Entry pNA^{+} calls the NI with a JSR instruction. The parameters are passed on the stack. There are two of them, one specifying the command, the other being a pointer to the parameter block, which is specific for each function.

Return Having performed the requested service, the NTI-driver returns using RTS. It must return a value in D0. The registers D0.L, D1.L, A0.L and A1.L have to be preserved.

The NTI-driver must not assume automatically that pSOS^{+} is running and in no case is the NI allowed to block.

To transmit data pNA^{+} prepares a packet and passes it to the NI for transmission. The pNA^{+} supports two types of interfaces: pNA^{+} *packet dependent* and pNA^{+} *packet independent*.

- Interfaces of the first type leave the handling of the data structures used for transmission to pNA$^+$. During initialization all drivers are handed the addresses of a set of functions for allocating and releasing the appropriate data structure. Packet dependent interfaces use them, so that the packets they handle have the same message block structure as is used in pNA$^+$.

- Drivers of the second type utilize their own data structures (i.e. data buffers, their descriptors and envelopes). For transmission pNA$^+$ calls the NTI-driver to allocate a certain number of bytes of buffer space. It stores the data that is to be transmitted in there and hands the address of the buffer to the NTI-driver when calling the *ki_send* or the *ki_broadcast* function. Assembly of the transmission packets, i.e. adding an envelope, lies within the responsibility of the NTI-driver.

The NTI-Driver uses pNA's packet dependent interface, where mblk-structures are used for handing over data packets. The NTI-Driver must thus be installed with the IFF_RAWMEM flag set!

As in the KI an *NI announce packet entry point* for announcing incoming messages is necessary. The address of this entry point is provided by pNA$^+$ when calling *ni_initial*. It has to be called whenever a packet arrives. The NTI-driver has to push six parameters onto the stack and then to call the NI announce packet entry via a JSR instruction. pNA$^+$ processes the packet and returns control to the NTI-driver by use of an RTS instruction. Note, that this call returns the status of pSOS$^+$. If its up and running a positive number is returned, 0 otherwise.
The following is a list of the parameters. Each is a 32 bit value.

type is the type of the packet. It must be either 0x00000800 to mark an IP packet or 0x00000806 to signify an ARP packet. Packets with other headers are not processed by the pNA$^+$ and therefore should not be forwarded to it.

buf_addr is the address of the buffer containing the packet.

count is the size of the packet in bytes

if_num is the number of the interface that received the message.

src_addr is the pointer to the address of the source address of the packet

dest_addr is the pointer to the address of the destination address of the packet

There is no entry point to report errors, since pNA$^+$ has its own protocol and therefore need not be informed of asynchronous transmission failures. The ones it can be informed of are synchronous ones via the return values of transmit requests. Because of this protocol it is also not necessary to provide any internal features to ensure reliable transmission. Receive errors are usually resource errors and not reported either.

Apart from NETWORK_NOT_FUNCTIONAL and NO_BUF_FREE the error codes returned by the functions to pNA$^+$ are pNA$^+$ error codes. They are defined in the library file error.h.

The services to be provided by the NTI-driver are as follows:

ni_initial

is called for initialization of an interface. It performs the duties necessary upon initialization of the NTI-driver, such as initialization of internal data structures and the underlying hardware. It also keeps the address of the NI announce packet entry and those for NI message block allocation/deallocation functions in global variables.

The parameter block has the following structure:
int (*ap_addr)()... address of announce packet entry point
int if_num... number of interface to be initialized
int ip_addr... IP address of the network interface
int (*ni_funcs)()... pointer to pNA memory management functions.

The return value is a pointer to the Ethernet (hardware) address of the interface.
INVALID_IF_NUMBER illegal interface number.
NETWORK_NOT_FUNCTIONAL indicates an error during initialization

ni_send

With this call pNA$^+$ sends a packet. Therefore, it first allocates a transmit buffer and then copies the data from the NI internal message structure into this buffer. In NI, each message is kept scattered in a list of so-called message blocks (m_blks). Since the pNA dependent interface is used, the driver is responsible for copying the data. After sending, the packet buffer has to be released automatically. As has been said before, pNA$^+$ does not require the transmission to be reliable since it has its own protocols.

The parameter block has the following structure:
addr *hwa_ptr... pointer to destination HW address
char *buff_add... pointer to a list of message blocks
int count... size of the packet in bytes
int type... packet type distinguishes between ARP packets (0x806) and
IP packets (0x800)
int if_num... number of the network interface

The return value is 0 if the packet is successfully sent, or
EINVALID Illegal packet size.
INVALID_IF_NUMBER Interface number not valid.
NO_ENV_FREE No envelope available, probably all in use.
INVALID_BUF_ADDRESS Address does not exist.
INVALID_ENV_ADDRESS Address does not exist.
BUFSIZE_TOO_SMALL allocated a buffer that is too small for size.

ni_broadcast

With this call pNA$^+$ transmits a message to all nodes in the network. Internally, the function just inserts the hardware broadcast address supported by the LANCE and calls ni_send. After transmission the packet buffer has to be released.

The parameter block has the following structure:
char *buff_addr ... pointer to a list of message blocks
int count... size of the data in bytes
int type... type of the packet distinguishes between ARP packets (0x806) and
IP packets (0x800)
int if_num number of the network interface

The return value is 0 if the packet is successfully sent, or
EINVALID Illegal packet size.
INVALID_IF_NUMBER Interface number not valid.
NO_ENV_FREE No envelope available, probably all in use.
INVALID_BUF_ADDRESS Address does not exist.
INVALID_ENV_ADDRESS Address does not exist.

ni_poll

This call allows pNA$^+$ to poll for incoming packets. If a packet has come in, ni_poll must pass it to pNA$^+$ using the message announce entry point.

The parameter block has the following structure:
int if_num... number of the interface

This function returns always 0.

ni_ioctl

is called by pNA$^+$ to perform various I/O control or MIB-II related operations on the Network Interface (see pNA$^+$ description in Section 2.1.2).

The parameter block has the following structure:
unsigned long command
SIOCSIFADDR ... inform NI of setting of NI's IP Address
SIOCSIFSTADDR ... inform a point-to-point NI of the destination IP address - not needed
SIOCPSOSINIT ... inform NI that pSOS is initialized
SIOCADDMCAST ... add multicast hardware address for packet reception – not supported
SIOCDELMCAST ... delete multicast hardware address for packet reception – not supported
SIOCMAPMCAST ... map a protocol multicast address to a hardware address – not supported
SIOCSGIFDESCR ... get NI descriptor
SIOCGIFTYPE ... get NI type
SIOCGIFMTUNIT ... get maximum transmission unit
SIOCGIFSPEED ... get interface speed
SIOCGIFPHYSADDRESS ... get physical address
SIOCGIFADMINSTATUS ... get administrative status
SIOCGIFOPERSTATUS ... get operational status
SIOCGIFLASTCHANGE ... get time of last change of status
SIOCGIFINOCTETS ... get the number of bytes received
SIOCGIFINUCASTPKTS ... get the number of unicast packets received
SIOCGIFINNUCASTPKTS ... get the number of multi- and broadcast packets received
SIOCGIFINDISCARDS ... get the number of received packets that were discarded
SIOCGIFINERRORS ... get the number of received packets with errors
SIOCGIFINUNKNOWNPROTOS ... get the number of received packets with unknown higher layer protocols
SIOCGIFOUTOCTETS ... get the number of bytes sent

SIOCGIFOUTUCASTPKTS ... get the number of unicast packets sent
SIOCGIFOUTNUCASTPKTS ... get the number of multi- and broadcast packets sent
SIOCGIFOUTDISCARDS ... get the number of transmit packets discarded due to resource problems
SIOCGIFOUTERRORS ... get the number of transmit packets discarded due to errors
SIOCGIFOUTQLEN ... get the length of the outbound queue
SIOCGIFSPECIFIC ... This call is taken for turning on and respectively off the forwarding task (determined via a string as parameter: "tenable" or "tdisable"). If turned on, all KI and CI messages are queued and forwarded up to the specific interfaces in a task context instead of the ISR context, which reduces ISR execution time but increases latency.
SIOCSIFADMINSTATUS ... set administrative status
NI_IOCTRL_PSOSDOWN ... is pSoS running or not?

char *arg... pointer to the argument of the operation.
int if_num ... number of the network interface

This function returns always 0.

3.2.4 CI

The idea behind the clock synchronization interface (*CI*) is to keep the clock synchronization independent of the network hardware. Each network connected to the node needs its own CI. This is realized by providing one reentrant driver which may be used for more than one interface. The different instances are identified by their interface number which must be specified at each call of a CI function.
The packets the clock synchronization transmits and receives consist of an envelope and the data. There are two types of transmission envelopes, timestamped and non timestamped. Since the management of the data structures is the NTI-driver's responsibility, the clock synchronization needs to request buffer space and an envelope for assembling a packet. At requesting an envelope it specifies the type it needs. The format of the data structure that is returned by such a call contains the fields common to both of the two different types, since the timestamp information is filled in automatically upon the actual transmission and thus the specific attributes of the timestamped envelope are of no moment to the NTI-driver:

```
typedef packed struct
{
  INT8  reserved[16];
  addr  da;
  INT16 length_field;
} tenvx;
```

The clock synchronization must fill in the hardware address of the destination node into da. Length_field will contain the type of the packet, in this case CI. It is filled by the NTI-driver. A CI packet may consist of only an envelope without a message to be transmitted. In that case the address of the transmit data buffer (a pointer to a memory space of the required number of

bytes) is NULL.
Having written its data into the buffer the clock synchronization hands the envelope and the data to the NTI-driver, requesting a normal or a high priority send. It can also declare if it wants the NTI-driver to release both buffer and envelope immediately after transmission.

On the receiving side the various timestamp informations - being of vital interest to the clock synchronization - are contained partly in the receive envelope and parly in the buffer. If the message was sent as a timestamped message by the source node, the receive buffer contains four longwords holding the *transmit timestamp information* in the following sequence:

area ... number of the virtual node (see Section 2.3) the message was sent from

ms ... macrostamp

ts ... timestamp information

acc ... accuracy

The receive envelope contains the *receive timestamp information*:

```
typedef packed struct
{
  INT8  reserved[22];
  addr  sa;
  INT16 length_field;
  timex utcsu[MAXASICS];
} renvx;
```

Sa contains the hardware address of the sender, length_field the type of the sender, which must be CI, and utcsu the receive timestamps from each area (NTI) resident on this node. MAXASICS is the maximum number of NTIs a node can be equipped with. The timestamps are stored in the following structure:

```
typedef struct
{
  INT32 rms;
  INT32 rts;
} timex;
```

rms contains the macrostamp, whereas rts stores the receive timestamp.

The following services make up the CI:

- **ci_init** for initialization of the data structures and underlying hardware
- **ci_get_tbuf** for allocation of a transmit buffer
- **ci_get_tenv** for allocation of a transmit envelope
- **ci_ret_tbuf** for returning a transmit buffer

- **ci_ret_tenv** for returning a transmit envelope
- **ci_ret_rbuf** for returning a receive buffer
- **ci_ret_renv** for returning a receive envelope
- **ci_send** for sending a packet

The clock synchronization calls the CI services via a single CI-dispatcher function:

Entry The clock syncronization calls the CI dispatcher with a JSR instruction. Two parameters are passed on the stack: One specifies the command, the other is a pointer to the parameter block, which is specific for each function.

Return Having performed the requested service, the NTI-driver returns using RTS. It must return a value in D0.

As in the other interfaces, there is an entry point for announcing incoming messages in FIFO order. Since the clock synchronization may have to serve more than one hardware interface, the address of this *ci announce packet entry* has to be provided as a parameter to *ci_init*.
ci_announce has four parameters for passing the relevant information:

renv pointer to the envelope

buf pointer to the contiguous data buffer

size size of packet buffer in bytes

if_num number of the interface that received the message

It returns an integer value, which tells the NTI-driver what to do with the packet. If the return value is IMMEDIATE_RELEASE, the whole packet, envelope and buffer, is released.

In addition there is an entry point for reporting errors that occurred during transmission. It has the name *ci_announce_error* and the following parameters:

env pointer to the envelope

buf pointer to the (contiguous) data buffer

size size of the packet data

if_num number of the interface

area number of the virtual node (See Section 2.3)

With this function packets that could not be transmitted are passed back to the clock synchronization which then can take the appropriate steps.

The detailed description of the CI services is as follows:

ci_init

Is called for initialization of an interface. It performs the duties necessary upon initialization of the CI part of the NTI-driver.

The function has the following parameters:
int if_num ... number of the interface
int (*ci_announce)()... address of CI message entry point
int (*ci_announce_error)()... address of CI receive error entry point.

The return value is 0 if everything is well or
INVALID_IF_NUMBER if the given interface does not exist
NETWORK_NOT_FUNCTIONAL if a hardware error occurred
INVALID_POINTER if one of the entry point addresses is NULL.

ci_get_tbuf

This function allocates buffer space for transmission

The function has the following parameters:
INT16 size ... requested size of the data buffer
bufx **buf ... pointer for returning pointer to data buffer
int if_num ... number of interface
int area ... number of virtual node the buffer should come from.

The function returns 0 if all went well,
INVALID_IF_NUMBER if the given interface or area doesn't exist,
INVALID_POINTER if buf was NULL or
NO_BUF_FREE if no buffer space was available.
NETWORK_NOT_FUNCTIONAL Interface is not initialized.

ci_get_tenv

Allocates a timestamped or normal envelope in a certain shared memory area.

The function requires the following parameters:
int mode ... determines the type of the envelope:
TS_PACKET ... timestamped
0 ... normal envelope
tenvx **tenv ... pointer for returning pointer to envelope
int if_num ... number of interface
int area ... number of virtual node.

If everything went well the return value is 0. It is
INVALID_IF_NUMBER if the given interface or area doesn't exist,
INVALID_POINTER if buf was NULL or
NETWORK_NOT_FUNCTIONAL Interface not initialized.
NO_ENV_FREE if no envelope was available.

ci_ret_tbuf

Returns transmit- or re-used receive buffer to the NTI-driver.

The function has the following parameters:
bufx *buf ... pointer to buffer

This function returns 0 if everything was OK or
INVALID_BUF_ADDRESS if there is no buffer at the given address.

ci_ret_tenv

Returns a transmit envelope to the NTI-driver

The function has the following parameters:
tenvx *tenv ... pointer to the envelope

This function returns 0 if everything was OK or
INVALID_ENV_ADDRESS if there is no envelope with the given address.

ci_ret_rbuf

Returns receive buffer to the NTI-driver

The function has the following parameters:
bufx *buf ... pointer to buffer

This function returns 0 if everything was OK or
INVALID_BUF_ADDRESS if there is no receive buffer at the given address.

ci_ret_renv

Returns receive envelope to the NTI-driver

The function has the following parameters:
renvx *renv ... pointer to the envelope

This function returns 0 if everything was OK or
INVALID_ENV_ADDRESS if there is no buffer at the given address.

ci_send

Allows the sending of a packet. Broadcasting is realized by stating the broadcast address as the destination. The packet may be a high priority packet, which means that it is going to be placed as far to the front of the queue of outbound messages as possible.

The function has the following parameters:
int mode ... the mode of the transmission
PRIOR means that this is a high priority sending
RET_PKB_IMM means that the buffer and envelope should be returned immediately after transmission
tenvx *tenv ... pointer to envelope
bufx *buf ... pointer to data buffer
INT16 size ... size of buffer in bytes
int if_num ... number of the interface
int area ... number of the virtual node.

The returnvalue is 0 if everything went OK or
INVALID_IF_NUMBER if the given interface or area doesn't exist or
NETWORK_NOT_FUNCTIONAL Interface is not initialized.
BUFSIZE_TOO_SMALL alloc'ed a buffer that is too small for size.
NO_ENVELOPE_ATTACHED if tenv was NULL.

3.3 Error Handling and KI Transmission Protocol

3.3.1 Description

The transmission protocol for KI messages lies between the interface and the management layer, belonging to neither. Together with the error handler it forms a sort of intermediate layer.

3.3.2 Transmission Protocol

The pSOS^{+m} kernel expects its transmissions to be reliable, without duplications and preserved in their order. To manage this, a protocol has to be implemented in the NTI-driver. A suitable protocol is a relatively simple sort of sliding window protocol (used in TCP implementations), which has been employed.

Each node maintains queues, where certain important data of messages that have been sent are stored. The protocol itself is realized with four functions: *put*, *watchdog*, *sscmp_announce* and *sscmp_release*, having the following duties:

- **put** inserts messages into the right queue and calls the transmit function of the lower layer (managment layer).
- **watchdog** controls the timestamps and retransmits messages
- **sscmp_announce** receives messages from other nodes, removes acknowledged messages and sends acknowlegdments.
- **sscmp_release** marks messages (transmission ready) and eventually removes acknowledged messages.

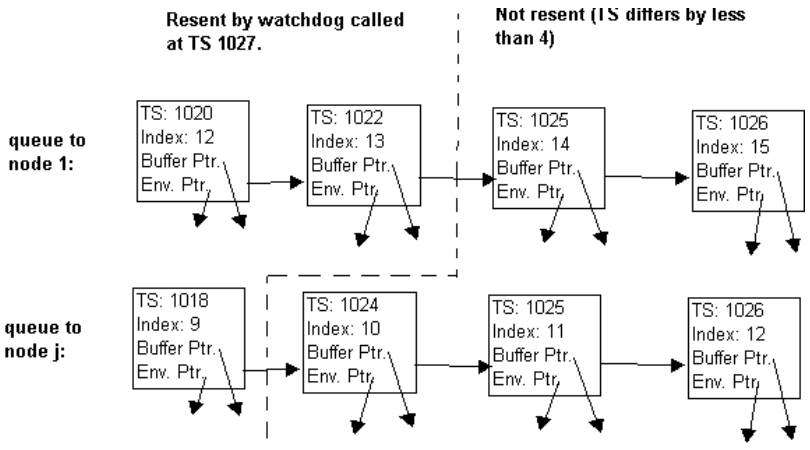


Figure 3.2: an example for resend queues

An ordered linked list (queue) is used for outgoing messages. On each node, there is one queue for each node, to which messages are sent. The function *put* sequences the messages, puts the sequence number into the SSCMP message header and gives them a local timestamp (a *INT32* number increased on each clock tick). Messages to different nodes are sequenced independently.

Then the messages (described by pointers to the envelope and buffer) are put to the appropriate queue together with the timestamp and the number of resends, which is 0.

The *watchdog* is called every few ticks and the first messages in the queue are resent if the actual timestamp differs too much from the message timestamp (see figure 2.3). Not all entries of the queue have to be inspected, because if the difference is small enough for one entry, all later entries will have still smaller differences. For example, after the third entry was inspected, the watchdog returns. Every time a message has to be resent, a counter for this message is updated. If this counter reaches a maximum, the *fatal_error* function is called.

At the receiving node *sscmp_announce* gets messages from the lower layer. It determines the type of the packet via a field in the SSCMP header (acknowledge or data). If the packet in question is assumed to contain data, its sequence number is read from the protocol data field. If that number is correct (i.e. the same as the next expected one), the address of the data buffer is written into a queue where KI messages are kept until they are handed over to pSOS^{+m}. Finally an acknowledge message is generated and sent to the source node, and the kernel is informed that a message came in via the KI message announce entry point.

On reception of a reply message (acknowledgment) *sscmp_announce* removes the first entries from the queue as long as their sequence number is smaller than or equal to the sequence number of the reply. The entry is not removed, if the message is under transmission, but only marked as invalid and removed later (otherwise a message, which is resent but gets a reply for the first transmission later, may loose envelope and buffer while retransmission is under progress!) If the sequence number of one local queue entry is larger than the received sequence number (this means, that for this entry no reply was received yet), this is also true for the following queue entries.

sscmp_release marks messages as beeing already transmitted ("under transmission" flag cleared). If these messages did already receive a reply, *sscmp_release* has to remove the messages from the queue, because sscmp_announce will not be called any more.

init_sscmp

initializes the data structures the protocol needs, mainly the message window.

the function has no parameters

and it has no return value either.

put

this function is handed a message for transmission. It sequences the message, creates an entry in the queue described above and hands the message to the lower layer.

the parameters of this function are:
bufx *buf ... the pointer to the data buffer
addr da ... the destination address of the message
INT16 size ... the size of the data buffer

The return value is always 0.

release_sscmp

this function is called after transmission of a KI message. It clears the flag that marks the message as being in the process of being transmitted. Eventually it has to remove the message from its queue.

the parameters of this function are:
envx *env ... pointer to the envelope

the function has no return value

sscmp_announce

this function receives messages from the manager layer that are destined for the KI. By a field in the header it distinguishes between acknowledge and data messages. Then it handles acknowledgments and data messages as described above.

the parameters of this function are:
renvx *env ... the pointer to the envelope
bufx *buf ... the pointer to the data buffer
INT16 size ... buffer size
addr *sa ... the source address of the message

the function has no return value

sscmp_error_announce

this function is called by the error handler to announce that a message could not be transmitted. The error message is relayed to the KI.

the parameters of this function are:
envx *env ... pointer to the envelope
bufx *buf ... pointer to data buffer

the function has no return value

watchdog

this function is called every few ticks (i.e. every few 10ms) by pSOS^{+m} through the *ki_time* function.
It has the duty to resend messages which were not replied yet by the comparision of timestamps.

this function has no parameters

and no return value

3.3.3 Error Handler

This part of the NTI-driver is responsible for collecting the error messages coming in from the transmit and the receive part of the manager layer. The transmit part error messages are all reports of failed transmissions. The packets are analyzed to determine the client that requested the transmission and an error message is sent to it via its interface (if error handling is supported at all).
The receive error messages are notifications of resource errors. There is a counter for each of the different types of receive errors. Whenever the announcement of such a one comes in the appropriate counter is increased.

trans_fatal_error

the function receives the data of a packet that could not be transmitted. It determines the application interface that requested the transmision and informs it that a send failed. What the application or the interface does with this information is up to it.

the parameters of this function are:
envx *env ... pointer to the envelope of the packet
bufx *buf ... pointer to the data buffer
INT16 size ... size of the data buffer
int if_num ... ID of the interface that tried to send

the function has no return value

rcv_fatal_error

this function gets an error code. It increases the corresponding counter which is ran_out_of_bufs in the case of NO_BUF_FREE and ran_out_of_envs in the case of NO_ENV_FREE. Otherwise the counter other_errors is increased.

the parameters of this function are:
int error ... encoded error. It can be
NO_BUF_FREE meaning that the list of available RBDs is empty, or
NO_ENV_FREE meaning that the list of available RFDs is empty.

the function has no return value

rcv_fatal_error

this function gets an error code. It increases the corresponding counter which is ran_out_of_bufs in the case of NO_BUF_FREE and ran_out_of_envs in the case of NO_ENV_FREE. Otherwise the counter other_errors is increased.

the parameters of this function are:
int error ... encoded error. It can be
NO_BUF_FREE meaning that the list of available RBDs is empty, or
NO_ENV_FREE meaning that the list of available RFDs is empty.

the function has no return value

fatal_error

this function just increases an error counter and can be entered into the pSOS^{+m} configuration table.

the function has no parameters and return value

asynchronous_error

this function just increases an error counter and can be entered into the PNA configuration table.

the function has no parameters and return value

3.4 Management Layer

3.4.1 Description

This layer performs multiplexing activities, on one hand taking commands from the three interfaces and relaying them to the appropriate part of the actual driver (the lowest layer), and on the other hand relaying incoming packets from there to the interfaces they are intended for.
One can distinguish between functions concerned with initialization, those for command handling, and those processing receive structures. Note that the first consists of exactly one function in this layer.
Since the transformations between driver layer data structures and interface layer data structures are performed here, the underlying hardware has some influence on this layer. It has to be checked and probably parts of it redesigned if another network coprocessor is used.

3.4.2 Initialization

If the LANCE has not yet been initialized the appropriate functions of the lowest layer have to be called. The information, which interface has called for an initialization and is therefore active is kept, so that received packets intended for this interface will be handed up instead of being discarded. The following function handles the initialization at manager layer:

init

the function checks whether the LANCE and its data structures are already initialized. If not, certain global variables are set to their initial value. This has to be done explicitly in the NTI-driver since the first inititalization may be called before pSOS^{+m} has set up the initialized data sections properly, so that all memory contents are yet meaningless.
Afterwards functions initializing internal data structures and hardware are called.
Depending on the calling interface additional steps may be necessary such as initialization of the KI message queue.

the parameters for this function are:
int ifc ... the application calling for an initialization
int if_num ... the network interface that is to be initialized

the return value is 0 if everything was OK or
INVALID_IF_NUMBER if the chosen interface does not exist
NETWORK_NOT_FUNCTIONAL if there were hardware problems

3.4.3 Commands

Command handling on this layer consists of transforming the data structures between the formats used in the application interfaces and those used in the driver layer for communication with the LANCE.

alloc_ci_tbuf

this function allocates buffer space for the CI. If no space was available the NULL pointer and an error value are returned to the calling function. Otherwise the address of the buffer is handed up. The calling function gets only the actual buffer without knowing that it is logically parted into 64 byte blocks each of which is controlled by a buffer descriptor.

the parameters of this function are:
INT16 size ... the required size of the data buffer
bufx **buf ... the of the pointer to the buffer
int if_num ... the network the buffer is needed for
int area ... the virtual node to take the buffer from

the return value of this function is 0 if everything is OK or
NO_BUF_FREE if no buffer was available.

alloc_ci_tenv

this function allocates an envelope for the CI. The structure of the envelope is derived from the structure of the driver layer envelope, containing the data relevant for CI. The input parameter mode specifies whether the allocated envelope should be timestamped or not. A timestamped envelope lies in a shared memory area where accessing a certain address triggers a timestamp to be pulled by the NTI and inserted transparently into a certain place in shared memory.

the function has the following parameters:
int mode ... which is:
0 for a normal envelope or
TS_PACKET for a timestamped one
tenvx **tenv ... address of the pointer to the envelope
int if_num ... the required network
int area ... the required virtual node

the return value is 0 if everything was OK or
NO_ENV_FREE if no envelope was available.

alloc_ki_tbuf

this function allocates buffer space for the KI. Analogous to *alloc_ci_tbuf* it returns either the pointer to the buffer space or NULL and an error value. Neither if_num nor area are specified since the KI can only manage one network interface and the virtual nodes are of interest only to the clock synchronization.

the function has the following parameters:
INT16 size ... the required size of the buffer - is usually 100 for KI packets
bufx **buf ... address of the pointer to the buffer

the return value is 0 if everything was OK or
NO_BUF_FREE if no buffer was available.

alloc_ki_tenv

this function allocates an envelope for the KI. It works similar to *alloc_ci_tenv* with the difference that only non timestamped envelopes can be allocated.

the function has the following parameters:
envx **env ... address of the pointer to the envelope

the return value is 0 if everything was OK or
NO_ENV_FREE if no envelope was available.

alloc_ni_tbuf

this function allocates buffer space for the NI. Analogous to *alloc_ci_tbuf* it returns either the pointer to the buffer space or NULL and an error value. Note that area is not of relevance, even though the NI can handle different network interfaces because the various NTIs are intended solely for testing the clock synchronisation.

the function has the following parameters:
INT16 size ... the required size of the data buffer
bufx **buf ... address of the pointer to the buffer
int if_num ... the required network interface

the return value is 0 if everything was OK or
NO_BUF_FREE if no buffer was available.

release_cb

this function gets the command blocks that have been executed by the LANCE. It reinitializes them or, in the case of transmit commands, checks if an error occurred during execution. If not, the handling of the command depends on the client having sent it. If it came from the KI it is immediately released if it was an acknowledgement packet; otherwise *release_sscmp* is called. An NI message is also immediately released. The treatment of a CI transmit command depends on a parameter the clock synchronisation specified when ordering a transmit, which has been stored in the TCB. If IMMEDIATE_RELEASE has been requested, the TCB as well as the buffer is released here, otherwise nothing is done with it. In that case the responsibility of returning it lies with the clock synchronization.
If an error occured during execution, *trans_fatal_error* is called and the packet is returned afterwards. On occurence of DMA *underrun* errors,an error counter is increased instead of calling *trans_fatal_error*, because at the moment this error can happen sometimes in connection with the NTI hardware.

the function has the following parameters:
cbx *cb ... pointer to the command block
int if_num ... interface the command was executed on

the function has no return value.

ret_ci_tbuf

first the function determines which interface and area the buffer belongs to. If buf is recognized as being invalid an error message is returned. Otherwise the address of the buffer descriptor of the first buffer block is computed and the whole structure is freed by a driver layer call. The function has to distinguish between transmit and receive buffers, since receive buffers may have been reused for transmission.

the function has the following parameters:
bufx *buf ... pointer to the buffer

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the given address does not belong to any buffer pool

ret_ci_tenv

the function checks the validity of the envelope address it has received. It then either returns an errorcode or calls the appropriate function to release the envelope.

the function has the following parameters:
envx *env ... pointer to the envelope

the return value is 0 if everything was OK or
INVALID_ENV_ADDRESS if the given address does not belong to any envelope pool

ret_ki_tbuf

the function determines whether the address is valid. If the buffer does exist, the address of its descriptor is computed and the buffer released. The function has to distinguish between transmit and receive buffers, since receive buffers may have been reused for transmission.

the function has the following parameters:
bufx *buf ... pointer to the buffer

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the given address does not belong to any buffer pool

ret_ki_tenv

first the function checks whether the given envelope exists in the pool. If the address is valid, i.e. refers to an existing envelope, that envelope is released.

the function has the following parameters:
envx *env ... pointer to the envelope

the return value is 0 if everything was OK or
INVALID_ENV_ADDRESS if the address does not belong to the envelope pool

ret_ki_tmess

this function is for returning a whole packet consisting of envelope and buffer. First the buffer is returned, using the same algorithm as in *ret_ki_tbuf*, after this the envelope, analogous to *ret_ki_tenv*.

the function has the following parameters:
envx *env ... the pointer to the envelope

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address does not belong to any buffer pool, or
INVALID_ENV_ADDRESS if the address does not belong to any envelope pool

trans_ci

out of the data given to it by the CI it is determined, if the envelope is of timestamped type, specific actions are taken (e.g. setting the area). Finally the function hands the envelope to the driver layer function responsible for submitting commands to the LANCE. This may be *put_cb* in case of a normal transmit or *put_cb_urgent* in case of a high priority transmit.

the function has the following parameters:
int mode ... telling the NTI-driver what to do with the TCB after transmission:
RET_PKB_IMM ... the packet is released after transmission,
0 ... nothing is done .
tenvx *tenv ... pointer to the envelope of the message or TCB
bufx *buf ... pointer to the buffer
INT16 size ... of the buffer
int if_num ... number of the required network
int area ... number of the required virtual node

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address does not belong to any buffer pool, or
INVALID_ENV_ADDRESS if the address does not belong to any envelope pool
BUFSIZE_TO_SMALL allocated a buffer, which is too small for the size parameter

trans_ki

gets a nearly complete packet in driver layer format from the SSCMP transmission protocol. It prepares the buffer for KI messages (additional bytes for control information) and calls transm for assembling the transmit structure. Then the envelope is handed to the lower layer.

the function has the following parameters:
envx *env ... pointer to the envelope
bufx *buf ... pointer to the buffer
INT16 size ... its size

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address does not belong to any buffer pool, or
INVALID_ENV_ADDRESS if the address does not belong to any envelope pool
BUFSIZE_TO_SMALL allocated a buffer, which is too small for the size parameter

trans_ni

the function allocates an envelope for transmission at the required network interface. Then it calls transm for assembling the packet and later hands the envelope to the lower layer.

the function has the following parameters:
addr da ... destination address
int type ... type of the message
bufx *buf ... pointer to the buffer

INT16 size ... its size
int if_num ... number of the required network

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address could not be allocated
INVALID_ENV_ADDRESS if the envelope could not be allocated.
BUFSIZE_TO_SMALL allocated a buffer, which is too small for the size parameter

transm

This function compiles a transmit command: It writes the pointer to the buffer descriptor associated with the buffer and also the mode and the length field into their appropriate places in the envelope. It also sets certain flags in the buffer descriptors. If the buffer is so small, that the resulting packet would stay under the minimum length, the size has to be increased artificially.

the function has the following parameters:
int mode ... telling the NTI-driver what to do with the TCB after transmission:
RET_PKB_IMM ... the packet is released after transmission,
0 ... nothing is done .
tcbx *trans ... pointer to the envelope of the message or TCB
bufx *buf ... pointer to the buffer
INT16 size ... of the buffer
int if_num ... number of the required network
int *netbytes ... returned number of payload bytes sent in frame

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address could not be allocated
INVALID_ENV_ADDRESS if the envelope could not be allocated.
BUFSIZE_TO_SMALL allocated a buffer, which is too small for the size parameter

3.4.4 Received Messages

Received messages are handed up from the driver layer. Further processing depends on the application the message is intended for. If it is for KI or NI the message is copied into a contiguous shared memory area and the appropriate function is called to relay the message to "its" interface. A CI message is simply handed up.
The application will process the messages (maybe reusing the receive buffer for transmission) and then release the data structures it has received.

forward_rpk

this function is called from the driver layer. If pSOS is up and the relevant ioctl-call turned this functionality on, *forward_rpk* executes in its own context, not in the ISR! It gets a pointer to a received message, the size of the message and a flag saying if the message is contigous. If it is not, *copy_rbd* is called. Then it is dispatched to the right part (SSCMP, NI module or CI module) of the upper layer.

the function has the following parameters:
rfdx *rfd ... pointer to the message
INT16 bsize ... size of the data buffer
int if_num ... number of the network interface

INT16 contigous ... flag containing 0 for contigous messages
INT16 no_rbds ... number of fixed size buffers used by the frame.

the function has no return value

ki_get_rbuf

this function is called by the KI to fetch a received message that has been announced via the announce entry point and hand it up to the application.

the function has the following parameter:
bufx **buf ... address of the pointer to the message

the return value is always 0

poll_ni

this function is called from the NI to inquire whether a message has come in if the NTI-driver is unable to call the message announce entry point by itself. The query is performed by calling the driver layer routine that is responsible for taking packets out of the receive shared memory area and handing them up to *forward_rpk*.

the function has the following parameters:
int if_num ... number of the required network interface

the return value is always 0

rcv_error

if an error is detected in the receive part of the driver layer that cannot be reported otherwise (e.g. by returning an error code) This function is called with the encoded error as a parameter. It is immediately relayed to the error handler. Usually those errors are notifications of resource shortages.

the function has the following parameters:
int error ... errorcode, which can be
NO_BUF_FREE ... the list of free RBDs is empty, or
NO_ENV_FREE ... the list of free RFDs is empty.
int if_num ... number of the network interface

the function has no return value

ret_ci_rbuf

first the validity of the buffer address is checked. If it is OK, the address of its buffer descriptor is computed and the buffer freed, otherwise an error code is returned. With this function only receive buffers may be returned, transmit buffers are counted as invalid.

the function has the following parameters:
bufx *buf ... pointer to the buffer

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address does not belong to any buffer pool

ret_ci_renv

The function checks whether the envelope belongs to the pool. If it does it is freed.

the function has the following parameters:
renvx *renv ... pointer to the envelope

the return value is 0 if everything was OK or
INVALID_ENV_ADDRESS if the address does not belong to any pool

ret_ki_rbuf

first the validity of the buffer address is checked. If it is OK, the address of its buffer descriptor is computed and the buffer freed, otherwise an error code is returned. With this function only receive buffers may be returned, transmit buffers are counted as invalid.

the function has the following parameters:
bufx *buf ... pointer to the buffer

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address does not belong to any buffer pool

ret_ki_renv

The function checks whether the envelope belongs to the pool. If it does it is freed.

the function has the following parameters:
renvx *renv ... pointer to the envelope

the return value is 0 if everything was OK or
INVALID_ENV_ADDRESS if the address does not belong to any pool

3.5 Driver Layer

3.5.1 Description

The lowest layer is called the *driver layer* since it represents the actual driver of the communication coprocessor. It handles the communication with the network and timestamping hardware, relaying commands to it and receiving messages from it.
The LANCE is parted into a receive and a command unit. Since this layer is rather closely related to the coprocessor it was only natural to carry this partition from the hardware on to the software.

This section first describes how the LANCE is to be initialized and what its requirements are on a driver. That part is followed by a description of the DMA area used for communication between CPU and i82596, and a description of how to program the LANCE. The chapter is concluded with a list of the functions belonging to reception, transmission and initialization.

There are two ways of accessing the network coprocessor directly: the channel attention (*CA*) signal and the PORT. Both of them are realized by memory mapped I/O at certain addresses. When a read or write is performed on these addresses, the channel attention respectively port command is executed.
A problem may arise when a cache is active in the system. The LANCE may not realize that CA was accessed since that action remained in the cache. Luckily the MVME162 provides bus snooping, allowing to recognize an access to an address even if it lies in the cache.

Channel Attention signals the LANCE it should take a look at the SCB, it carries no additional information.
PORT offers more possibilities. With a port access one can cause a reset, a self test or a dump, or issue the address of the SCP to the LANCE. A PORT access (where 32 bits of data are handed over) must be undertaken in two steps. First the upper command word (16 bits) is written and then the lower. Since the original value has to be swapped (see Page 57) the first step is to write the lower word of the port command into the upper word of the address mapped to PORT, and the second to write the upper word of the command into the lower word of the register. What action is to be performed by the LANCE is encoded in the bits 0-3 of the port command, the rest is either all "0"s or contains an address. Not that any address issued to the LANCE in this way has to be 16-byte aligned (i.e. the last four bits are 0).

The main data structure for comunication with the LANCE is the *SCB*, the *System Control Block*. Whenever a signal comes in at the CA pin the LANCE has a look at the command word in the SCB and acts upon it. The SCB has the following structure:

```
typedef volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 cl;
  INT32 rfa;
  INT32 crc_errors;
  INT32 alignment_errors;
  INT32 resource_errors;
  INT32 overrun_errors;
  INT32 rcv_errors;
  INT32 frame_errors;
  INT16 timer_off;
  INT16 timer_on;
} scbx;
```

sw is the status word of the SCB, cw its command word.
The bitwise description of the status word is as follows:

- **CX** bit 15; "1" indicates that the CU finished execution of a command which had the interrupt bit set.
- **FR** bit 14; "1" indicates that the RU finished receiving a frame
- **CNA** bit 13; "1" indicates that the CU went not active, i.e. left the "Active" state.
- **RNR** bit 12; "1" indicates that the RU left the "Ready" state
- **CUS** bits 8-10 (bit 11 is always 0); indicate the state of the CU

  0 ... Idle

  1 ... Suspended

  2 ... Active

  3-7 ... not needed

- **RUS** bits 4-7; indicate the state of the RU

  0x0 ... Idle

  0x1 ... Suspended

  0x2 ... No Resources

  0x3 ... (not used)

  0x4 ... Ready

  0xA ... No resources due to no more RBDs

  0xC ... Ready with no more RBDs present

- **T** bit 3; a 1 indicates that the bus throttle timers have been loaded

- **Zero** bits 0-2; are always 0

The bitwise description of the command word is as follows:

- **ACK_CX** bit 15; "1" means the acknowledge of the CX (Command Completed) interrupt indicated in the status word

- **ACK_FR** bit 14; "1" means the acknowledge of the FR (Frame Received) interrupt indicated in the status word

- **ACK_CNA** bit 13; "1" means the acknowledge of the CNA (CU Went Inactive) interrupt indicated in the status word

- **ACK_RNR** bit 12; "1" means the acknowledge of the RNR (Receive Not Ready) interrupt indicated in the status word

- **CUC** bits 8-10 (bit 11 is always 0); the encoded command to the CU:

  0 ... no operation, does not affect the current state of the CU

  1 ... start execution of the command list. If a command is currently executed it is finished before the new list is started. The address of the new list is fetched from the SCB in field cl.

  2 ... resume the operation of the command unit, execute the next command. It is assumed that the CU is in the suspended state.

  3 ... suspend execution of commands after finishing the current one

  4 ... abort current command immediately

  5 ... load the bus throttle timers so they will be initialized with their new values after the active timer reaches terminal count.

  6 ... load and immediately restart the bus throttle timers with their new values

  7 ... reserved

- **RE** bit 7; "1" causes a chip reset which is logically the same as a hardware reset.

- **RUC** bits 4-6; the encoded command to the RU

  0 ... no operation, does not affect the current state of the unit

1 ... start reception of frames. If a frame is being received complete that action before starting. Get the address of the RFA from the SCB, field rfa.

2 ... resume frame reception. It is assumed that the RU is in the suspended state.

3 ... suspend frame reception. If a frame is currently being received complete that action.

4 ... abort receive operation immediately.

5-7 ... reserved

- **Zero** bits 0-3; are always 0

Cl is the address of the first command in the list that is to be executed by the LANCE. It is read when the command CU_START is encoded in the cw.
Rfa is the pointer to the first RFD. It is read when the LANCE reads the command RU_START. Afterwards both pointers or rather the pointers to the current command respectively RFD are kept in on-chip registers.
The next six fields in the SCB are statistical counters. They are only increased by the LANCE, initializing them is the duty of the NTI-driver during initialization. They are wraparound counters. It is possible that more than one counter is increased for one frame if it falls into more than one of the counter categories. The read modify write operation is kept atomic by locking the bus during execution.
Crc_errors contains the numbers of aligned frames discarded because of a CRC error. It is updated regardless of the RU state.
Alignment_errors contains the number of frames that are both misaligned and contain a CRC error. It, too, is updated regardless of the RU state.
Resource_errors contains the number of good frames discarded because there were no resources available. Clearly this counter is updated only when the RU is in "No Resources" state.
Overrun_errors contains the number of frames lost because the local system bus was not available.
Rcv_errors contains the number of collisions detected during frame reception.
Frame_errors contains the number of received frames that are shorter than the minimum frame length.
The last two fields of the SCB are timer_off and timer_on. The former contains the number of clock cycles the LANCE has to stay off the bus once it has been forced away from it and the latter the maximum number of clock cycles it may stay on it.

Initialization

The Initialization Root (see Section 2.2.2) consists of a *System Configuration Pointer* or SCP, and an *Intermediate System Configuration Pointer* or ISCP. Initialization starts if a channel attention signal is issued after a reset, upon which the LANCE accesses the initialization root. The SCP is read first; it is structured as follows:

```
volatile packed struct
{
  INT32 sysbus;
  INT32 unused;
  INT32 iscp_address;
} scpx;
```

Of sysbus the bits 8-31 are always zero. 0-7 encode interrupt polarity, operational mode, bus throttle triggering method and the internal bus DMA channel switching mechanism.
Note that locked cycles (used by the LANCE for buffer prefetching and statistical counter updates) must not be enabled in hardware architectures that do not support read-modify-write cycles, as the MVME162 over the VME-Bus!
Iscp_address is the address of the *ISCP*, which is accessed next in the initialization sequence.

```
volatile packed struct
{
  INT8  zero1;
  INT8  busy;
  INT16 zero2;
  INT32 scb_address;
} iscpx;
```

Busy is cleared by the LANCE to signal the end of the initialization sequence. Consequently it has to be set by the NTI-driver before starting LANCE initialization.
scb_address is the pointer to the SCB, which is accessed last.

Initialization of the LANCE thus takes the following steps:

1. Before sending any signal to the LANCE, all data structures have to be properly initialized. Especially the ISCP, SCP, SCB and the commands needed for configuration of the processor, such as CFG, IAS and maybe MCS.
2. Reset i82596 using the appropriate PORT access.
3. Issue the address of the SCP to the LANCE with another PORT access. Otherwise it would look for the SCP at the default address 0x00FFFFF4. From now on, no other PORT access must be made, lest the address of the SCP be lost. The SCP address has to be 16 byte aligned, for the reasons discussed in the previous section.
4. The SCB command word is cleared, so that no action of CU or RU is performed upon the next step:
5. Issue a channel attention. It is the first after a reset and thus causes the LANCE to go through the initialization root.
6. Wait for the field busy of the ISCP to clear.
7. In the SCB status word the flags CX and CNA have been set. Acknowlege them by setting the corresponding bit in the command word.
8. Issue a channel attention.
9. Wait for the command word to be cleared, which signals that the LANCE has read the command word and hence found the acknowledgement.
10. Configure the interrupt and set the ISR.
11. Issue the following commands to the LANCE: CFG, IAS, maybe MCS and wait for completion of their execution.

12. Write the address of the first free RFD into rfa in the SCB.

13. Write RU_START into the command word of the SCB.

14. Issue a channel attention.

Programming of LANCE i82596CA

According to Figure 2.2 in Section 2.2.2, the LANCE is programmed by handing it a list of command blocks. Each command contains the following sequence, thus providing a common structure for handling the different command types:

```
volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
} cbx;
```

Sw holds the status word which is set by the LANCE only and read by the NTI-driver only. It shows the current status of the command.
The actual commands are encoded in cw, the command word. It tells the LANCE how it has to interpret the bytes following link. The word also contains various flags, two of which have the same meaning for every command: EL marks the last command in the list and I tells the LANCE to generate an interrupt after executing the command.
Link contains the address of the next CB in the list. The LANCE provides two possibilities for link addresses. Either linear addressing using 32-bit addresses or segmented mode using 16-bit offsets to a common base address. Since the second mode would limit the internal data structures to a maximum size of 64k, the linear mode is used by the NTI-driver.
Because of the special hardware architecture (combining an Intel chip with a Motorola CPU) the high and low word of 32-bit entities always have to be switched. Thus, the address 0xFFFFAAAA has to be written as 0xAAAAFFFF to be read correctly by the LANCE. To tell the LANCE that a link does not point to a valid address the value 0xFFFFFFFF has to be written into it. This is the LANCE's equivalent to the NULL pointer.

Now we will briefly describe the different commands the LANCE can execute. Each of those is represented by a command block containing the information necessary for the LANCE to execute the command.

NOP-command

```
volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
} nopx;
```

The fields of the NOP command are the same as those of the common structure `cbx` detailed above.

IAS-command

```
typedef volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  addr  ia;
} iasx;
```

The command for setting the individual address of the LANCE has the additional field ias, where the address is stored. The field is an array of six bytes, the format of a hardware Ethernet address.

Configuration-command

```
typedef volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT8  cfgn[CFG_CNT];
} cfgx;
```

The configuration command is called during initialization. The array cfgn contains 14 bytes of flags. To list them all in detail is neither necessary nor practicable, so I'll keep to those that deviate from the default value in this implementation or are otherwise of importance. See [Cor92, p.4-126ff] for exact bit and byte positions and meanings. By default the LANCE is configured to comply to the IEEE 802.3 standard:

- **PREFETCH:** if this flag is set, the LANCE marks the RFDs and RBDs it is going to fill next by setting a flag. (See description of those data structures in Section 3.5.1 for details).
- **SAVE_BAD_FRAMES:** this feature allows the user to receive frames that otherwise would be discarded due to errors, e.g. an invalid CRC. By default this feature is turned off anyway, but it is mentioned here, because it should always be turned off due to difficulties that could arise otherwise when using the NTI-driver.
- **LOOPBACK:** The loopback mode has been set to external loopback with the external loopback pin deactivated, meaning that the coprocessor transmits and receives simultaneously at full rate without being aware of it. By default loopback is disabled, but it is in the case of multiple areas/virtual nodes. Messages have to be looped back to let the other virtual nodes receive messages originating from virtual nodes on the same physical node.
- **BROADCAST DISABLE:** if this flag is enabled broadcast messages, identifiable by their destination addresses (all "1"s) are not received. This feature is turned off by default and should stay that way since broadcast is needed by the clock synchronization and the pNA$^+$.

- **MINIMUM FRAME LENGTH:** This value specifies the minimum number of bytes a frame must have to be received. By default it is 64, but it had to be turned down to 16 since SSCMP sends protocol messages that would be too short and consequently not received otherwise.

MC-setup-command

```
typedef volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT16 mc_count;
  INT8  mc_adr[MAX_ADDR_NO][ADDRCNT];
} mcsx;
```

The multicast setup command submits a list of multicast addresses to the LANCE.
Mc_count is the number of multicast addresses in the list,
mc_adr the address of the list itself. LANCE reads the addresses, hashes them and sets the according slots in its internal hash table. an incoming frame is accepted if it has a destination address with "1" as its least significant bit and which, after hashing, points to a "1" in the hash table.

Transmit commands

```
typedef volatile packed struct
{
  volatile INT16 sw;
  volatile INT16 cw;
  volatile INT32 link;
  volatile INT32 tbd_add;
  volatile INT16 eof_tcbcount;
  volatile INT32 packet_info;
  volatile addr  da;
  volatile INT16 length_field;
  volatile INT8 env_data[TSTIMEOFF];
  volatile INT32 area;
  volatile ti_acx tst;
  volatile IN16 unused[INFOCNT];
} ts_tenvx;
typedef ts_tenvx ts_tcbx;

typedef volatile packed struct
{
  volatile INT16 sw;
  volatile INT16 cw;
  volatile INT32 link;
  volatile INT32 tbd_add;
```

```
  volatile INT16 eof_tcbcount;
  volatile INT32 packet_info;
  volatile INT16 zero;
  volatile addr  da;
  volatile INT16 length_field;
} tenvx;
typedef tenvx tcbx;
```

The first structure, `ts_tcbx` represents a timestamped transmit command, the second a non timestamped one. Most of the fields are the same for both of them.
Cw contains an additional flag apart from I, namely SF, which sets the flexible memory mode (see Section 2.2.2) on or off.
eof_tcbcount specifies the number of bytes of the frame that should be transmitted as data. Thus it is possible to add envelope data to the byte stream that is sent on the medium. The count starts at a fixed offset within the TCB. The value is set to include the destination address and the length field for the normal TCB and additionally the area and the timestamp information for the timestamped TCBs.
tbd_add is the address of the first of the TBDs belonging to the TCB (or TSTCB).
da contains the destination address of the packet.
length_field is used to store the type of the packet (KI for kernel messages, CI for clock synchronisation messages and IP or ARP for pNA$^+$ messages).
packet_info is used to hold local information.
The primary difference between TCB and TSTCB lies in the fields area and tst. The latter is a structure reserving the space where the *transmit timestamp* information will be inserted automatically upon transmission of the packet:

```
typedef struct
{
  INT32 ts;
  INT32 ms;
  INT32 acc;
} ti_acx;
```

The first field contains the timestamp, the next the macrostamp and the last the accuracy. All those values are automatically written into the data section of the outgoing frame upon transmission.

A TBD has the following format:

```
typedef volatile packed struct tb
{
  INT16 eof_size;
  INT32 n_tbd;
  INT32 tb_addr;
} tbdx;
```

where tb_addr is the address of the buffer belonging to the TBD. It has space for 64 bytes of data.

N_tbd is the address of the next TBD in the list of a multi-buffer message. This field is used only in allocated TBDs.
The flag EOFR in the field eof_size should be set to 1 to indicate the last TBD in a message. This field also contains the number of bytes in the associated buffer that are actually filled with data. This, too, is set by the NTI-driver.

TDR-command

```
typedef volatile packed struct
{
  volatile INT16 sw;
           INT16 cw;
           INT32 link;
           INT16 result;
           INT16 zero;
} tdrx;
```

Time domain reflectometry is a mechanism to detect open or short circuits on the link and their distance from the diagnosing node. The command causes the LANCE to send a special bit stream of the following format on the net:

- **Preamble** (8 bits)
- **Source address** (48 bits)
- **Source address** (48 bits) so that the frame will be received again by the station
- **Data field** containing a pattern of 0x7E
- **Jam Pattern** which is the inverse CRC of the transmitted frame.

The field result contains the outcome of the test. That can be a notification that there is a problem with the transceiver or that the transmission line is either not properly terminated or has a short circuit.

DUMP-command

```
typedef volatile packed struct
{
  volatile INT16 sw;
           INT16 cw;
           INT32 link;
           INT32 buffer;
} dumpx;
```

The DUMP command causes the contents of various i82596 registers to be written into a shared memory area specified by buffer. This dump area is 304 bytes large and need not be detailed here. Suffice it to say that it contains the current configuration of the LANCE, its individual address, its hash table, the SCB address and lots of additional information. See [Cor92, p.4-155ff] for details.

Test-command

```
typedef volatile packed struct
{
  volatile INT16 sw;
           INT16 cw;
           INT32 link;
} diagx;
```

This command triggers a self test, checking internal hardware. Passing or failing of the test is indicated in the status word of the command.

Programming the LANCE takes the following steps:

1. One or more commands have to be compiled to a command list. The EL flag in the command word of the last command has to be set. The link addresses have to be swapped before being written into the field, so that they will be read correctly by the LANCE.
2. The address of the first command in the list is written (also swapped, of course) into the field cl of the SCB. The command word of the SCB is set to CU_START.
3. A channel attention is issued to notify the LANCE.

The rest is up to the LANCE. After it has executed the whole list or a command that had the I flag in its cw set, an interrupt is generated. The ISR now has to do the following:

1. Wait until the command word is cleared, as this signifies that the LANCE is done with the command.
2. Acknowledge the interrupts announced in the status word by setting the corresponding bits in the command word of the SCB.
3. Notify LANCE of this by issuing a channel attention
4. Again wait for the command word to be cleared, signalling that the LANCE has processed the acknowledgement.
5. Clear the Interrupt Pending bit in the Interrupt control register of the board.
6. Process the executed commands, return them to the pool.
7. If necessary generate a new command list and hand it to the LANCE as described above.

The interrupt handling of the MVME162 requires acknowledgement of an interrupt by setting the appropriate flag in the interrupt control register. The LANCE requires acknowledgement of interrupts by setting the appropriate flags in the SCB. If the interrupt on the board is cleared after clearing it with the LANCE, an interrupt may occur in the meantime which would not be realized and processed until the next one came in. The correct procedure is to clear the interrupt on the board and then handle the SCB.
It may be the case that an interrupt condition arises between clearing the bit on the board and handling the SCB. In that case the interrupt flag on the board remains set. If the ISR does not provide for this case and clear the interrupt flag even if there was apparently no interrupt from the LANCE, the ISR will be called again and again, prohibiting execution of anything else if the ISR is of sufficient priority.

Receive Handling

As has been described in Figure 2.2 in Section 2.2.2 the Receive Frame Area (*RFA*) consists of a list of Receive Frame Desriptors (RFDs), connected to a list of Receive Buffer Descriptors (RBDs) with the associated receive buffers. Like the transmit buffers each of those can contain up to 64 bytes of data.
The *RFD* is the data structure containing general information about the packet that has come in. For the NTI-driver it has the following format:

```
typedef PACKED struct
{
  volatile INT16 sw;
  volatile INT16 cw;
  volatile INT32 link;
  volatile INT32 rbd_add;
  volatile INT16 eof_full_actc;
  volatile INT16 size;
  volatile addr  da;
  volatile addr  sa;
  volatile INT16 length_field;
  volatile INT16 free;
  volatile timex utcsu[MAXASICS];
} renvx;
typedef renvx rbdx;
```

Sw indicates the result of the receive operation.
By setting the appropriate flags in cw, the command word, the NTI-driver can mark the last RFD in the list of available RFDs, suspend the RU after reception of this frame, and select the memory mode (SF - see Section 2.2.2).
link is used to connect the available RFDs together. The LANCE will follow those links when having received a message and looking for a new RFD.
Eof_full_actc indicates whether this is the last frame in the received message (it has to be, since the flexible memory model is used), whether the frame is full (i.e. the whole frame has been received) and how many bytes of the incoming data have been written into the RFD.
Size specifies the number of bytes that are to be included in the frame. This value is set appropiately so, that source and destination address together with length_field are read in from the data portion of the incoming message.
Da and sa contain the source respectively destination hardware address.
The type of the packet is encoded in the length_field. It can be CI for a clock synchronization message, KI for a kernel message or ARP or IP for a pNA$^+$ message. (See Section 4.3 for the actual values of constants).
Utcsu is an array of MAXASICS entries of the following format:

```
typedef struct
{
  INT32 rms;
  INT32 rts;
} timex;
```

where rts is the *receive timestamp* of the NTI (written into the RFD by *nti_isr*, see Section **??**) and rms is the *macrostamp* (see Section 2.3.). MAXASICS may at the most be four since that is the maximum size the array can have and still fit into the receive frame which has a size of 64 bytes.
Note that an RFD has to be 64 bytes long, so that a reduction of MAXASICS to a number smaller than 4 has to be balanced by an expansion of the field free in rfdx.

The *RBD*s are structures controlling the data buffers themselves:

```
typedef volatile packed struct
{
  INT16 eof_full_actc;
  INT16 zero1;
  INT32 n_rbd;
  INT32 rb_addr;
  INT16 el_pref_size;
  INT16 zero;
} rbdx;
```

Eof_full_actc contains the flag telling whether this RBD is the last one in a received packet, the flag telling whether the RBD is full, i.e. has been written by the LANCE, and the number of bytes that have been stored in the buffer.
N_rbd is the address of the next RBD in a multibuffer packet. The LANCE follows those links when it has written a buffer and now looks for a new one. The RBDs and their buffers do not have to be directly behind one another physically for the LANCE to receive a message correctly. Thus a message may be contained in a set of buffers that do not form a contiguous memory space.
Rb_addr is the address of the buffer belonging to the RBD.
The field el_pref_size keeps the flag which marks the last RBD in the list of all the free ones, another flag which signals that this RBD has been prefetched by the LANCE and finally the size of the buffer that is attached to the RBD.

Shared Memory

The shared memory area on the NTI, being the means of communication between the host CPU and the LANCE, has a size of 256k. Of those 64k are decoded in such a way that a timestamp is being triggered when a certain offset within a 64 byte block is read or written by the LANCE, though not at a CPU access. 4k of those are reserved for timestamped transmit commands (TSTCBs) and 60k for receive frames (RFDs), see Figure 3.3.

Given that TCBs are 24 bytes and TSTCBs are 128 bytes long, there are 32 timestamped envelopes and 960 receive frames available. Proceeding from this precondition the 184k of Header and Buffer space have been distributed among the non-timestamped TCBs, TBDs, RBDs and transmit- and receive-buffers.
The remaining 12k of the DMA area are reserved for system structures. The initialization root, the SCB and the remaining commands for the LANCE (NOP, MCS, TDR, IAS, DIAG, DUMP, CFG) have been placed there. The command structures are provided only once since they won't be needed again after initialization of the i82596, whereas transmit command blocks will most

Figure 3.3: Partition of the shared memory area on the NTI

probably be requested rather frequently and hence have to be provided in multiple instances to avoid unnecessary contention. Not all of the 12 KB is used for system structures, therefore there is a field between TCBs and system structures used for byte-padding.
The shared memory is mapped into two linear address spaces of equal structure but starting at different base addresses, one for CPU and one for LANCE accesses. This is done, because LANCE accesses to certain addresses trigger the timestamping hardware, which causes timestamp writes to certain memory cells. CPU accesses must not release this effect. Two macros are used for address calculation: One calculates the CPU addresses from pointer fields in the shared memory, which always have to contain LANCE addresses, since they may be accessed and used by the LANCE. The other one calculates LANCE addresses from CPU addresses for insertion into the mentioned shared memory pointer fields.

3.5.2 Command Handling

The functions listed here handle the command list for the LANCE, the command data structures and the executed commands.

Those command data structures, of which multiple copies exist and consequently must be handled, are the TCBs, the TSTCBs, the TBDs and their buffers. Each TBD is assigned one buffer block of 64 bytes.

When allocating a buffer for a data packet of a certain size, it has to be assured that the space it occupies is contigous. Linked lists are used to handle buffer allocation. Each entry describes a continous space of free buffers: start index, length of the block (number of fixed-size buffers contained) and a next-pointer. List entries are ordered by buffer addresses. Two following entries will always have a contigous block of allocated (reserved) buffers between them. When buffers are returned there are a few possibilities:

1. new list entries have to be generated;

2. old entries are extended (if the returned block borders to a block of free space);

3. entries are put together and one entry is removed (if the returned block bordes to two blocks of free space on both sides and therefore connects them).

The big advantage here is, that no series of buffer descriptors have to be scanned to find a block, which is large enough to hold the frame. In reality, not many buffers are in use mostly, so the receive buffer managment is not stressed very much.

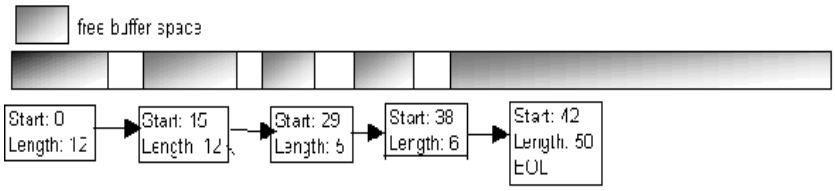


Figure 3.4: a typical example for buffer allocation/deallocation

example

(see figure 3.4): There are five contigous blocks of free buffers (with 12, 12, 5, 8 and 50 buffers). If a space of 350 Bytes (6 buffers) is allocated, the list is searched for a contigous block large enough to hold the whole space. The first entry (12 buffers) is large enough. Therefore, a pointer to the first buffer descriptor in this block is returned. Before, the list is updated: The start entry is set from 0 to 6 and the length is decreased by 6. If the length would decrease to 0, the entry would be deleted and the start of the list updated.

example

(see figure 3.4): Assume, the reserved block containing buffers 13 and 14 is returned. Firstly, the list is searched to find the right place.In this case, two entries of the existing list have to be merged, since they are connected by the returned buffer (list element one and two). Therefore, the first element is updated(start:0, length: 26), list element two is deleted and the next pointer from element one is set to element three.

Since each packet uses exactly one envelope there is no need for special structures for organizing TCBs and TSTCBs. They are simply inserted into linked lists, one for the free TCBs and one for the TSTCBs. An envelope returned to the pool is appended to its list, the first one is taken from it at allocation.

As has been said in the previous section, the LANCE is programmed by handing it a list of command blocks (CBL). This list is built by appending newly submitted command blocks to

it. One has to bear in mind, though, that the LANCE cannot be stopped in its execution of the command blocks. Appending a new command makes sense only if the LANCE has not yet started execution of the last CB. If adding to the list is performed when the LANCE is about to execute the last command block (having its EL-bit set) it cannot be guaranteed that the LANCE reads the modified pointer - it might as well have read the old one, causing the CU to stop. In this case the ISR processing the executed commands has to restart the LANCE with the new (still unexecuted) part of the command list.
If the last command is definitely being executed at the time of adding a new command, appending to the list is useless. An alternative list (ACBL) is started, respectively the new command is appended to it. The ISR will make a new list out of the formerly mentioned rest of the old CBL and the ACBL and restart the CU.

The clock synchronization may want to send packets with high priority. Those commands are not appended to the CBL or the ACBL but must be inserted as far to the front of the CBL as possible. The usual approach is to mark the CB after the one being currently executed as the last one in the CBL. If, after marking it, the CB is in the process of being executed (as can be seen by a flag in the status word of the CB) it cannot be assured that the LANCE has read the new value of the EL bit. Therefore the NTI-driver tries the same procedure with the next CB in the CBL. This goes on until either the LANCE did not keep up and consequently will read the flag correctly, or the end of the CBL is reached. In the former case the new CB is inserted into the CBL. The ISR processing executed commands will clear the EL bits of those commmands that are not last in the CBL and restart the CU, causing the not yet executed ones to be processed.
In the latter case the command block is appended to a list of urgent CBs (UCBL). The ISR will append the rest of the old CBL to it, add the ACBL and then restart the CU.
The following functions are responsible for transmit command handling.

get_tbd

this function is used to reserve a certain size of contiguous buffer space.
First n, the number of buffers necessary to accomodate the requested number of bytes, is computed. Then a chain of n TBDs is allocated as described above. A pointer to the first TBD is returned, and the PREF-bit (EOTBD constant) of the last TBD is set for later return via ret_tbd.

the parameters of this function are:
INT16 size ... the requested size of the buffer in bytes.
int if_num ... number of the requested network interface
int area ... number of the requested virtual node (see Section 2.3).

the return value of this function is the address of the TBD or
NULL if none was available.

get_tcb

this function takes the first TCB from the list of free ones and returns it to the caller.

the parameters of this function are:
int if_num ... number of the requested network interface

the return value of this function is the address of the TCB or
NULL if none was available.

get_ts_tcb

this function takes the first TSTCB out of the list of free ones and returns it to the caller.

the parameters of this function are:
int in ... number of the requested network interface
int area ... number of the requested virtual node (see Section 2.3).

the return value of this function is the address of the TSTCB or
NULL if none was available

put_cb

this function is handed a command for the LANCE. It tries to append it to the list of commands the LANCE is working on. If there is no such list it creates a new one out of the command in question and starts the LANCE with it. If, on the other hand, the last command in the existing list is already being executed, appending the new command does not make sense since the LANCE has already read the EL flag. In that case a new list is started, called the alternative list, that will eventually be submitted to the LANCE upon completion of the CBL.

the parameters of this function are:
cbx *cb ... address of the command
int if_num ... number of the requested network interface

the return value of this function is always 0

put_cb_urgent

this function gets a command for the LANCE. It tries to insert it as far to the head of the list of commands the LANCE is currently working on as possible. If there is no such list it creates a new one containing only the command in question and starts the LANCE with it. If, on the other hand, the last command in the existing list is already being executed appending the new command does not make sense since the LANCE has already read the EL flag. In that case it is put into the urgent list (see beginning of this section).

the parameters of this function are:
cbx *cb ... address of the command
int if_num ... number of the requested network interface

the return value of this function is always 0

release_cb_up

this function is the ISR for the LANCE command unit. It forwards executed commands to the manager layer to be processed further. If the command unit is idle (e.g. has completed the CBL) the function attempts to generate a new command list out of the urgent list, the remainder of the old command list and the alternative list. The command unit is started again with the new CBL.

the parameters of this function are:
INT16 state ... status word of the SCB
int if_num ... number of the network interface

this function has no return value

ret_tbd

this function reinitializes a chain of TBDs and their buffers and inserts the chain into the free list like described above. The last TBD in the chain is recognized by the set PREF-bit, which was done during get_tbd.

the parameters of this function are:
tbdx *tbd ... address of the first TBD of those to be returned
int if_num ... number of the requested network interface
int area ... number of the requested virtual node (see Section 2.3).

the return value of this function is always 0

ret_tcb

this function initializes a TCB and appends it to the list of free ones. If the list is empty it creates a new one.

the parameters of this function are:
tcbx *tcb ... address of the TCB
int if_num ... number of the requested network interface

the return value of this function is always 0

ret_ts_tcb

this function reinitializes a TSTCB and appends it to the list of free ones. If that list was empty it starts a new one with this TSTCB.

the parameters of this function are:
ts_tcbx *ts_tcb ... address of the TSTCB
int if_num ... number of the requested network interface
int area ... number of the requested virtual node (see Section 2.3).

the return value of this function is always 0

3.5.3 Receive Handling

The functions listed here handle the Receive Frame Area and the receive data structures.

The receive data structures existing in multiple copies are the RFDs and the RBDs and their buffers. Each RBD is assigned one buffer block of 64 bytes (see figure 2.1). This assignment is done so that if two RBDs border in memory (are contigous), the associated buffers will border as well.

Each packet needs exactly one envelope, hence the RFDs are organized in a linked list, similar to the TCBs and TSTCBs. The difference is, that this list is handed to the LANCE, which proceeds filling it with incoming packets. In the earlier version [Ric97] of the driver a complicated Ring Buffer Algorithm was used to return buffers associated with a frame. This was done to avoid unnessary calls to the copy-function: The LANCE itself does not require the entire space it writes the data into to be contiguous. The organization of the RBDs is a bit more complex. It is handed a list of free RBDs, which are connected to each other, and writes the data it receives from the network into their buffers, block by block, according to the sequence in the list. Contrary to that, the clients usually need the data in a contiguous portion of memory. To avoid buffer copies, the LANCE should write to a contigous buffer space within a frame as well. Therefore the connected RBDs should also border in memory. Therefore the driver has to insert returned RBDs in such a way, that logical connections (pointers in RBDs) and physical bordering are identical, which was done by the Ring Buffer Algorithm as far as possible. The disadvantage: The algorithm was slow and complex. The new implementation dispenses with the algorithm and in fact does not induce more copies. The new function ret_rbd() works as follows:

It just remembers the 'last' RBD address (*last_received*) to which the LANCE has written and which was handled by the ISR. In reality the LANCE will mostly be already ahead, because it is receiving new messages, which were not reported to the interrupt handler yet. Another address, which is always remembered and kept up-to-date (in *last_free*), is the last adress of a free buffer before *last_received*. Between *last_free* and *last_received* there are only used RBDs (see figure 3.5). The RBD *last_free* is pointing to, is set to the last one in the list of free RBDs (`EOL` flag) and if it is reached by the LANCE (should never happen, because the receive buffer space is large enough!), no free buffers are available. Then the LANCE has to stop and trigger the ISR. Inside the ISR-code it is resumed.

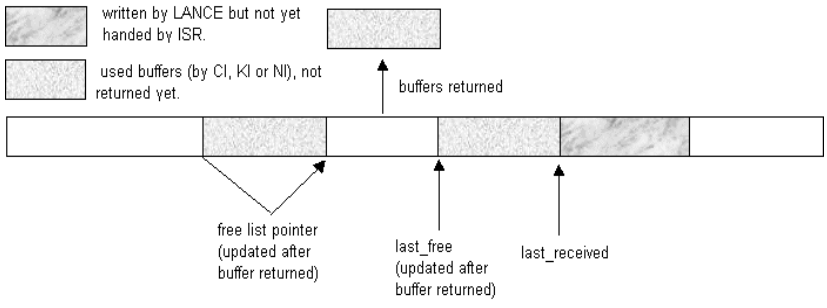


Figure 3.5: Returning a set of fixed size buffers associated with RBDs

When a contigous RBDs is returned (there is another function ret_rbd_nc(), which secures that only contigous buffers are returned by partitioning not contigous buffers into smaller packets),the RBDs immediatly preceding the frame are scanned until a free buffer descriptor is found. Then

it inserts itself into the list of free buffers (the *next*-entry of last buffer in the returned frame is only set, if there is at least one free RBD between the last buffer in the returned frame and *last_received*; otherwise the *EOL*-entry is set). If there is a conflict with the LANCE (because it is already so far ahead, when buffers are tried to be returned), the buffers are not returned but put to a linked list. A periodic watchdog then re-tries to return all elements in the list by calling the described function. The watchdog is called only if pSoS is up, so there could be a few entries collected in the list, until the watchdog is running. In fact, it is very improbable that there will be any entries.

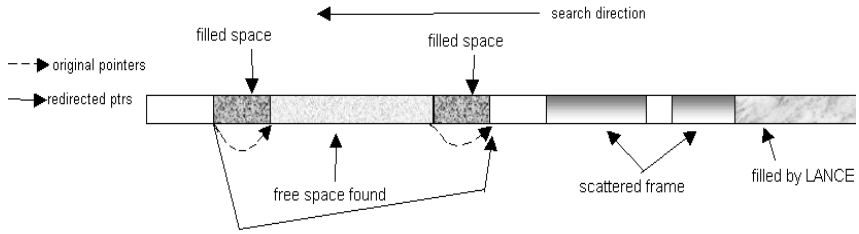


Figure 3.6: a typical example for *copy_rbd*

copy_rbd

This function copies the data from an incontiguous buffer into a contiguous one. It is only called by forward_rpk() for CI- or KI-packets with an uncontigous buffer space. The address of the new chain of RBDs is returned to the caller, the original chain is released. Since *copy_rbd* can take some time (up to 6 ms), the need of disabling interrupts during complete function execution time is avoided: Firstly a space large enough to hold the whole frame is seeked. Then the fields in this space are marked as filled and the pointers of the list containing the free buffers are redirected (look at the found space and the redirected pointers in the picture). Concurrency problems with the LANCE have to be considered here. Such problems may arise, if the LANCE is writing to the free space found, before the pointers can be redirected. Then the interrupts are turned on again and the buffers are copied into the new space, which is reserved now and cannot be touched by the LANCE any more. This part takes most of the function's execution time.

the parameters of this function are:
rbdx *rbd ... address of the RBD starting the maybe not contiguous list.
int if_num ... number of the requested network interface
int no_rbs ... number of RBDs in chain to be copied

the return value of this function is the address of the RBD starting the contiguous list or NULL if the copy could not be performed.

ret_rbd_nc

This function takes a not contigous list of RBDs, portions this list into contigous blocks of RBDs and hands them to ret_rbd.

the parameters of this function are:
rbdx *rbd ... address of the RBD starting the not contiguous list.
int if_num ... number of the requested network interface

the function has no return value.

ret_daemon

This function is called periodically by *ki_timer* and tries to return blocks of RBDs associated with frames that could not be returned by calling *ret_rbd* due to conflicts with the LANCE.
These RBD blocks are kept in lists until it is possible to return them.

there are no parameters and no return value.

forward_rpk_up

this function is the ISR for the LANCE receive unit. It takes the received packets, i.e., the RFDs and the associated RBDs from the list and hands them up to *forward_rpk*, which is responsible for processing and distributing them. The packets are handed up directly (by calling *forward_rpk*) or by sending a message to the task FRPK (implemented in *forward_rpk_stub* and started via an ioctl-call), which itself calls *forward_rpk*. This is done for decoupling the forwarding process, which can contain calls to *copy_rbd*, but also protocol management and complete resends of messages for the Kernel communication, from the ISR. Therefore the ISR execution time is reduced significantly. Clearly, RFDL_BEGIN points to the first received message. The FREE_SLOTS array is updated: FREE_SLOTS contains a flag for each RBD and the RBDs used for the received frames are marked as "full". Furthermore the total number of RBDs, the size of the frame and if it is contigous are determined. Then the packet is handed up to the manager layer together with this information in the way described above. RFDL_BEGIN is set to the next frame in the list and the whole procedure started again, until either the end of the RFDL is reached or a frame that is not (yet) full.
Finally the status of the receive unit is checked. If it has stopped due to resource problems, it is provided with a new RFA, or, should this prove to be impossible, an error message is generated.

the parameters of this function are:
int if_num ... number of the network interface
INT16 state ... contains the status word of the SCB

the function has no return value.

lance_isr

the real ISR does not belong to solely the receive part or the transmit part, but to both. It is called in response to an interrupt from any of the network interfaces serviced by the NTI-driver. It checks which interface caused the interrupt, acknowledges it and calls the appropriate function: *release_cb_up* for the command unit and *forward_rpk_up* for the receive unit.

this function has no parameters

and no return value

ret_rbd

this function receives a chain of contigous RBDs to return to the list of free RBDs. After reinitialization it inserts them into the list according to the algorithm described at the beginning of this section.

the parameters of this function are:
rbdx *rbd ... address of the first RBD in the BDL
int if_num ... number of the network interface

the return value of this function is always 0

ret_rfd

this function is responsible for reinitializing an RFD and appending it to the list of free ones.

the parameters of this function are:
rfdx *rfd ... address of the RFD
int if_num ... number of the network interface

the return value of this function is always 0

3.5.4 Initialization

Initialization of the lowest driver layer is more complex than in the upper layers since the whole shared memory area and the hardware have to be set to their initial values or states. As a consequence the workload has been splitted between various functions. Some of those functions are called not only during initialization but also later on to reinitialize data structures.

Initialization of Receive Data Structures

The function responsible for the first initialization of the receive part of the shared memory is called *init_receive*. It calls the other functions handling receive initializations, which are also called later by other functions during the process of returning data structures to the various pools.

init_rbd

all fields of the RBD are set to zero with exception of el_pref_size, which is set to the size of a buffer and n_rbd and rb_addr, which are not touched at all. The buffer belonging to the descriptor is cleared.

the parameters of this function are:
rbdx *rbd ... address of the RBD

the function has no return value

init_receive

this function has the responsibility of initializing all the data structures that are used for receive purposes. Those are the RFDs, the RBDs and the buffers. The RFDs are organized in a linked list where link is the address of the next one. The RBDs are stored in a circular buffer, see Section 3.5.3. Apart from that buffer the free RBDs are also linked together through the field n_rbd. The global pointers to begins and ends of those lists are set.

the parameters of this function are:
int if_num ... number of the interface that is being initialized.

the function returns always 0.

init_rfd

all fields of an RFD are set to zero with exception of cw in which the flag SF is set to indicate flexible mode, rbd_add which is set to EMPTY (being the LANCE's equivalent to the NULL pointer) and size which is set to 14 since that is the number of bytes the LANCE should write from the incoming data stream into the frame and not the buffer, see Section 3.5.1

the parameters of this function are:
rfdx *rfd ... address of the RFD

the function has no return value

Initialization of Transmit Data Structures

Here the "main" function is *init_trans*. The others may be called again lateron, during return of data structures.

init_cfg

link of the cfg data structure is set to NULL, sw to zero and cw to CFG to indicate a Configuration command to the LANCE. The array handed down as a parameter, containing the configuration of the LANCE, is copied into the according area in the CFG command.

the parameters of this function are:
INT8 cfga[CFG_CNT] ... an array of CFG_CNT bytes containing the configuration values for the LANCE. CFG_CNT is the number of configuration bytes the LANCE accepts
int if_num ... number of the network interface that is to be configured.

the function has no return value

init_ias

link of the ias data structure is set to NULL, sw to zero and cw to IAS to indicate a Set Individual Address command to the LANCE. ias is set to the address handed down as a paramter.

the parameters of this function are:
addr ich ... Ethernet address of this network interface
int if_num ... number of the network interface that is to be configured.

the function has no return value

init_mcs

link of the mcs data structure is set to NULL, sw to zero and cw to MCS to indicate a Multicast Setup command to the LANCE. Mc_count and mc_adr are initialized with the values handed down as parameters.

the parameters of this function are:
int adrc ... the number of multicast addresses
valx *adrc ... the list of multicast addresses
int if_num ... number of the network interface that is to be configured

the function has no return value

init_scb

this function initializes not only the SCB but the ISCP, SCP, TDR, NOP, DUMP and DIAG.
All fields of the SCB but timer_off and timer_on are set to zero. Those two fields are set to their initial values.
Of the SCP and the ISCP the fields that are not needed, i.e. SCP's unused and ISCP's zero1 and zero2 are set to zero.
The fields cw of all commands are set to the appropriate value to indicate that command to the LANCE. Link is set to EMPTY and any remaining fields to zero. An exception is DUMP's buffer field which is set to the address of the area the resulting data are to be written in.

the parameters of this function are:
int if_num ... number of the network interface that is to be configured.

the function has no return value

init_tbd

all fields of the TBD are set to zero with exception of n_tbd and tb_addr which are not touched at all.

the parameters of this function are:
tbdx *tbd ... address of the TBD.

the function has no return value

init_tcb

all fields of the TCB are set to zero with exception of cw where the command TRANSMIT and the flags indicating flexible mode and interrupt generation after command execution are set, tbd_add which is set to EMPTY to indicate a non valid link to the LANCE, eof_tcbcount which is set to 8 so that the LANCE takes the destination address and length_field from the TCB during transmission and link which is not touched.

the parameters of this function are:
tcbx *tcb ... address of the TCB

the function has no return value

init_trans

this function has the responsibility of initializing the transmit part of the shared memory. The affected data structures are the TCBs, TSTCBs, TBDs and transmit buffers. The TCBs and TSTCBS are organized in linked lists. The linked lists for TBD management are initialized: One list entry for each interface, containing all buffers.

the parameters of this function are:
int if_num ... number of the network interface that is to be configured

the function always returns 0

init_ts_tcb

all fields of the TSTCB are set to zero with exception of cw where the command TRANSMIT and the flags indicating flexible mode and interrupt generation after command execution are set, tbd_add which is set to EMPTY to indicate a non valid link to the LANCE, eof_tcbcount which is set to 24 so that the LANCE takes the destination address, length_field, area and the timestamp from the TSTCB during transmission and link which is not touched.

the parameters of this function are:
ts_tcbx *ts_tcb ... address of the TSTCB

the function has no return value

Initialization of the Hardware

init_lance

this function is responsible for initializing the part of the shared memory remaining after initialization of the receive and transmit parts. This comprises the data structures for initialization of the hardware (SCB, ISCP, SCP, CFG, IAS) and the remaining commands (DUMP, MCS, TDR, DIAG, NOP).
Next the ISR servicing LANCE interrrupts and the one inserting the receive timestamps into the RFD are installed.
The LANCE is reset, given the address of the SCP (via a PORT access) and then started with a channel attention signal, thus causing it to step through the initialization root, see Section 3.5.1.
When it is initialized the commands CFG, IAS and MCS are issued, the first RFD entered into SCB's rfa field and thus the receive unit started as well.

the parameters of this function are:
int if_num ... number of the network interface that is to be configured

the function returns 0 if everything is OK, or
NETWORK_NOT_FUNCTIONAL if a hardware error was detected

Chapter 4

Programmer's Reference

4.1 Introduction

The Programmer's Reference is intended for those having to maintain the driver or redesign it, should the hardware environment change. It contains a detailed description of the driver structure, the interaction of the modules and functions and the employed constants and structures. For working with this chapter one should at least have studied the previous ones and be well versed in C. An idea of how the driver works and what it does, together with knowledge of the applications and the hardware are, after all, prerogatives for maintaining it.
The first section describes the constants used in the implementation, gives their current values and what they stand for.
The next one details the data structures used for memory maps etc.
Last follows the biggest section, the description of the functions. It does not reexplain data structures etc already detailed in the previous sections, so please see there for additional information.

4.2 Structure of NTI-driver

As has already been described in Chapter 3, the NTI-driver consists of three layers. The uppermost is the application layer, handling the communication with the clients using the NTI-driver. The middle layer is the manager, assuming a sort of multiplexing function between the highest layer and the lowest, the driver layer, which controls the LANCE and the NTI. Because of the amount of functions necessary for each of the layers they had to be accomodated in different files, producing a file structure similar to the rough structure of the NTI-driver.

4.2.1 Files of the NTI-driver

In Figure 3.1 one can see which files together build which layer and how they all interact. In this figure, a few simple C-functions and all assembler functions are excluded for simplicity reasons: *memcompp (libfunctions.c), memcopp (libfunctions.c), memclear(libfunctions.c), mmalloc (libfunctions.c), mfree (libfunctions.c), initmem (libfunctions.c), block_copy (lancerec.c), get_no (sscmp.c), GetEtherAddr0 (interf.c), intlanceon0 (interf.c), asynchronous_error (error.c), fatal_error (error.c), tbdindex (lancetr.c), tbindex (lancetr.c), rbdindex (lancerec.c)* and *rbindex (lancerec.c).*

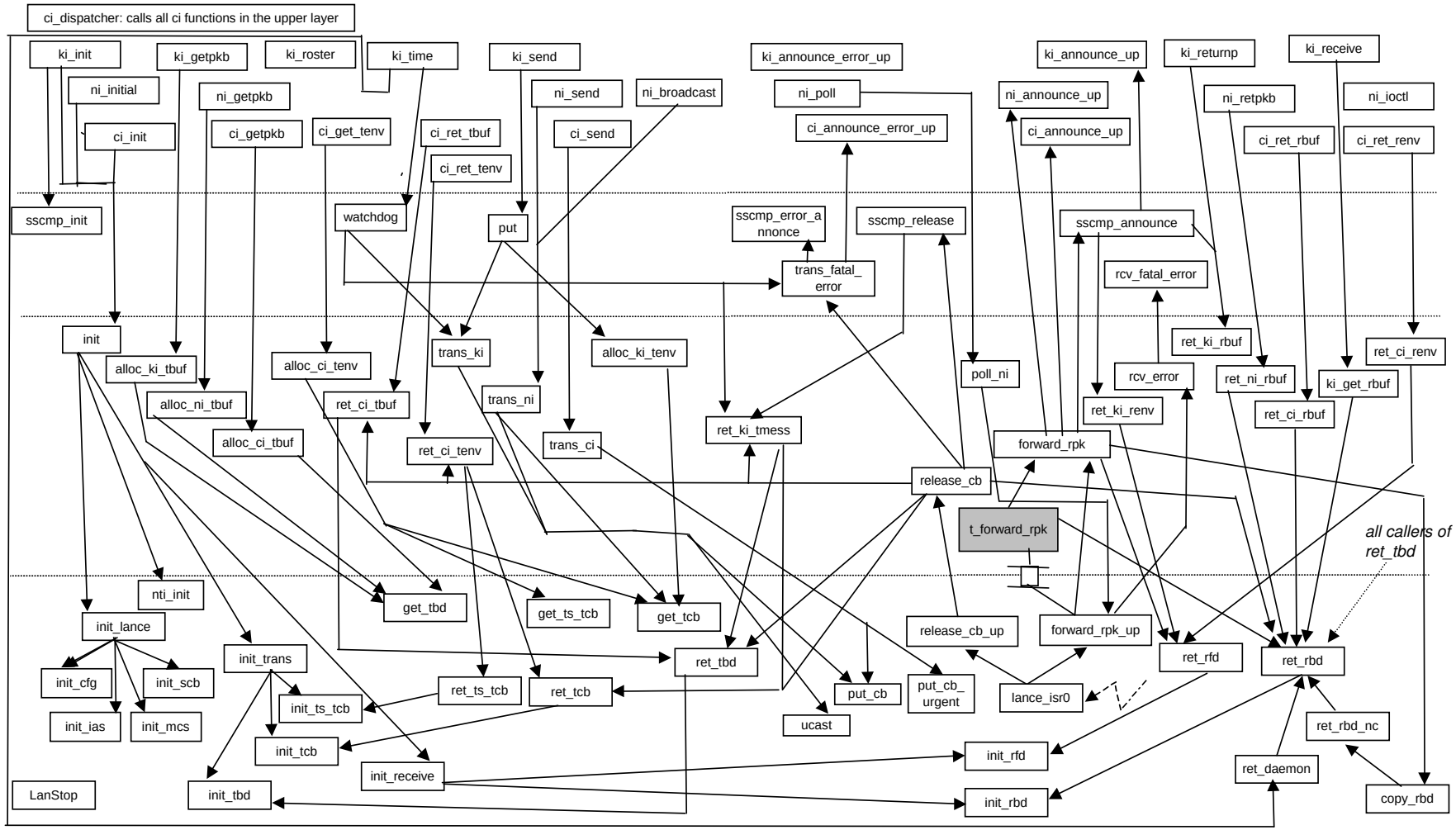


Figure 4.1: Structure of NTI-driver Functions

The uppermost layer consists of the following files:

- ki.c communication with pSOS+m
- ni.c communication with pNA+
- ci.c communication with clock synchronization

The intermediate layer consists of the following files:

- sscmp.c KI network protocol
- error.c error handler

The management layer consists of the following files:

- appltr.c command handling between application and driver layer
- applrec.c receive handling between application and driver layer

And the driver layer consists of the following files

- lancetr.c command handling between LANCE and multiplexer
- lancerec.c receive handling between LANCE and multiplexer
- interf.c functions, that are NTI hardware specific (coupling the driver to the NTI)

Another module contains some C library functions, since the standard library cannot be used before pSOS is up:

- libfunctions.c

In addition to those functions a few assembler procedures were required to handle the special hardware/software issues. The dispatcher functions called by pSOS^{+m} and pNA^{+}, the real ISR that handles the different situations arising when pSOS^{+m} is active or not and the functions raising and lowering the interrupt mask.

- dispatcher.s contains the functions that are the go-betweens between the NTI-driver's KI and NI on one side and pSOS^{+m} and pNA^{+} on the other. Each is adapted to the special calling conventions of the interface.
- lance.s contains the real ISR, the interrupt mask handlers and a function for atomic test-and-set

4.3 Constants

Due to the sheer mass of constants necessary and to the different scope of constants with respect to the various source files, the definitions have been divided unto multiple files.
The different include files are:

- **all.h** definitions common for the driver modules
- **appl.h** definitions of middle layer data structures and constants
- **ci.h** definitions between clock synchronization and driver
- **error.h** definitions of the error codes
- **driver.h** definitions for the bottom layer (hardware level)
- **ki.h** codes for interface between pSOS and driver
- **lance.h** structures for the use of the LANCE 82596 chip and the other hardware on an MVME162 board
- **sscmp.h** data structures and constants for sliding window protocol
- **interf.h** structures necessary for the use of the LANCE 82596 NTI-Driver in conjunction with the NTI M-Module

One of those, CI.H, has to be included by the clock synchronization so that it can use the NTI-driver. It contains the necessary constants and function prototypes.
The file ALL.H has to be included by all of the NTI-driver files, though they may additionally include other files, according to their needs.

ALL.H

IFCNT

Value: 1

Description: The number of network interfaces serviced by the NTI-driver, i.e., the number of LANCEs on board.

DEFI

Value: 0

Description: The default network interface number. This is the network interface all pSOS^{+m} messages pass through.

DEFA

Value: 0

Description: The number of the virtual node taken as default.

ADDRCNT

Value: 6

Description: The maximum number of multicast addresses that can be entered. This number is limited by the available space in shared memory.

MAXASICS

Value: 4

Description: The masimum number of virtual nodes a hardware interface can support.

BROADCAST_ADDRESS

Value: 0xFF,0xFF,0xFF,0xFF,0xFF,0xFF

Description: The hardware address for broadcast.

CI

Value: 0x5000

Description: ID of the clock synchronization.

KI

Value: 0x4000

Description: ID of the kernel.

NI1

Value: 0x0800

Description: Identifies an IP message. IP messages belong to pNA$^+$.

NI2

Value: 0x0806

Description: Identifies an ARP message. ARP messages also belong to the NI.

B_SIZE

Value: 64

Description: Number of bytes in a buffer.

TSFILLCNT

Value: 20

Description: Number of bytes the TSTCB is padded with to blow its size up to the required 64 bytes.

APPL.H

KQ_LENGTH

Value: 200

Description: Length of the array where the messages waiting to be received by pSOS^{+m} are stored.

CI.H

IMMEDIATE_RELEASE

Value: 22

Description: When the function used by the CI to announce an incoming message to the clock synchronization returns this value, the message is to be released immediately.

PRIOR

Value: 0x08

Description: Marks the message as high priority, and thus as to be transmitted as soon as possible.

RET_PKB_IMM

Value: 0x04

Description: Marks the message as to be returned immediately after transmission.

TS_PACKET

Value: 0x01

Description: For telling the function allocating an envelope for the clock synchronization, that a timestamp envelope is required.

CI_INIT

Value: 0

Description: Code for CI function ci_init.

CI_GTB

Value: 1

Description: Code for CI function ci_get_tbuf.

CI_GTE

Value: 2

Description: Code for CI function ci_get_tenv.

CI_RTB

Value: 3

Description: Code for CI function ci_ret_tbuf.

CI_RTE

Value: 4

Description: Code for CI function ci_ret_tenv.

CI_RRB

Value: 5

Description: Code for CI function ci_ret_rbuf.

CI_RRE

Value: 6

Description: Code for CI function ci_ret_renv.

CI_SEND

Value: 7

Description: Code for CI function ci_send.

ERROR.H

NETWORK_NOT_FUNCTIONAL

Value: -1

Description: Initialization of NI failed.

NO_BUF_FREE

Value: -1

Description: Failed to allocate buffer, since none are free.

NO_ENV_FREE

Value: -2

Description: Failed to allocate envelope, since none is free.

COULD_NOT_BE_MADE_CONTIGUOUS

Value: -3

Description: Failed to render an incontiguous chain of buffers contiguous. Usually this is a resource problem.

NO_ENVELOPE_ATTACHED

Value: -5

Description: The pointer to the envelope of a to be transmitted clock synchronization message was NULL.

INVALID_BUF_ADDRESS

Value: -7

Description: The given address that identifies a buffer does not belong to any of the buffer areas on the node.

INVALID_ENV_ADDRESS

Value: -8

Description: The given address that identifies an envelope does not belong to any of the envelope areas on the node.

INVALID_IF_NUMBER

Value: -9

Description: The given interface does not exist.

INVALID_POINTER

Value: -10

Description: The pointer where an address has to be written to be returned to the caller is NULL.

TX_FAILED

Value: -11

Description: A transmit error occurred in the KI.

INVALID_COMMAND

Value: -12

Description: The CI was given a non-existant function code.

INVALID_PROTO

Value: -13

Description: Protocol invalid. At initialisation.

TASK_STAT_INVALID

Value: -14

Description: Task/queue creation not successful. Driver does not tasks.

LANCE.H

address

Value: INT32

Description: The type of the addresses of the shared memory areas.

TADDSIZE

Value: 8

Description: Number of bytes the LANCE takes from the TCB upon transmission.

RADDSIZE

Value: 14

Description: Number of bytes the LANCE takes from the buffer and writes into the RFD upon reception.

TSADDSIZE

Value: 24

Description: Number of bytes the LANCE takes from the TSTCB upon transmission.

swap(x)

Value: (((INT32)x≪16)—((INT32)x≫16))

Description: This macro swaps the high and the low word of a 4 byte long word.

LANCE_CA(x)

Value: *ca[x]=1

Description: This macro accesses the register mapped to the CA pin of the LANCE and thus causes a CA signal.

CFG_CNT

Value: 14

Description: Number of LANCE Configuration bytes.

MAX_ADDR_CNT)

Value: 2

Description: Maximum number of Multicast Addresses storable.

COMMANDS

Value: 0x0007

Description: The bits of the command word containing the encoded command.

NOP

Value: 0x0000

Description: The code for the NOP command.

IAS

Value: 1

Description: The code for the IAS command.

CFG

Value: 2

Description: The code for the CFG command.

MCS

Value: 3

Description: The code for the MCS command.

TRANSMIT

Value: 4

Description: The code for the TRANSMIT command.

TDR

Value: 5

Description: The code for the TDR command.

DUMP

Value: 6

Description: The code for the DUMP command.

DIAG

Value: 7

Description: The code for the DIAG command.

CU_STATES

Value: 0x0700

Description: Bits in the status word of the SCB containing the state of the CU.

CU_IDLE

Value: 0x0000

Description: State of the CU is IDLE.

CU_SUSPENDED

Value: 0x0100

Description: State of the CU is SUSPENDED.

CU_ACTIVE

Value: 0x0200

Description: State of the CU is ACTIVE.

RU_STATES

Value: 0x00F0

Description: Bits in the status word of the SCB containing the state of the RU.

RU_READY

Value: 0x0040

Description: State of the RU is READY. It is waiting for messages.

RU_SUSPENDED

Value: 0x0010

Description: State of the RU is SUSPENDED.

RU_IDLE

Value: 0x0000

Description: State of the RU is IDLE.

RU_NORESOURCES

Value: 0x0020

Description: State of the RU is NORESOURCES, meaning that there are no RFDs left.

RU_NORBDS

Value: 0x00A0

Description: State of the RU is NO_RBDs, meaning that there are no RBDs left.

RU_R_WITHOUT_RBDS

Value: 0x00C0

Description: RU Ready with no RBDs filled.

RESET

Value: 0x0080

Description: The bit in the command word of the SCB causing a reset of the LANCE.

PORT_RESET

Value: 0xFFFFFFF0

Description: If this value is put to the PORT pins, the LANCE is reset.

PORT_DUMP

Value: 0x03

Description: The address of the area where the data should be written, combined with this value, will case the LANCE to perform a DUMP, when the value is put to the PORT pins.

PORT_SELF_TEST

Value: 0x01

Description: The address of the area where the data should be written, combined with this value, will case the LANCE to perform a DIAG, when the value is put to the PORT pins.

PORT_SCP

Value: 0x02

Description: The address of the SCP combined with this value will cause the LANCE to load the address, when put to the port pins.

STATES

Value: 0xF000

Description: These bits in the status word of the SCB encode the reason for the interrupt.

RNR

Value: 0x1000

Description: The interrupt occurred, because the RU left the READY state.

FR

Value: 0x4000

Description: The interrupt occurred, because the RU finished receiving a frame.

CX

Value: 0x8000

Description: The interrupt occurred because the CU completed a command that had itts interrupt bit set.

CNA

Value: 0x2000

Description: The interrupt occurred because the CU left the ACTIVE state.

CU_INTS

Value: (CX—CNA)

Description: The CU interrupts.

RU_INTS

Value: (RNR—FR)

Description: The RU interrupts.

RUC

Value: 0xFF0F

Description: The mask for the RU commands in the command word of the SCB.

CUC

Value: 0xF0FF

Description: The mask for the CU commands in the command word of the SCB.

RU_RESUME

Value: 0x0020

Description: Order the RU to resume receiving.

RU_START

Value: 0x0010

Description: Order the RU to start receiving (with a new receive frame area).

RU_SUSPEND

Value: 0x0030

Description: Order the RU to suspend receiving after the next incoming frame.

CU_START

Value: 0x0100

Description: Order the CU to start execution of the new command list.

LOAD_BT

Value: 0x0500

Description: Order the CU to load the bus throttle timers.

BUSYCOMPL

Value: 0xC000

Description: Bits in the status word of the command block telling that LANCE is just working on this command block, or has already finished it.

BUSY

Value: 0x4000

Description: Bit in the status word of the command block telling that LANCE is working on this block.

COMPL

Value: 0x8000

Description: Bit in the status word of the command block telling that LANCE has already finished this block.

OK

Value: 0x2000

Description: Bit in the status word of the command block telling that the block was successfully executed.

G_INT

Value: 0x2000

Description: This bit in the command word of the command block tells the LANCE to generate an interrupt after execution of the block.

EL

Value: 0x8000

Description: This bit marks the data structure for the LANCE as the last in the list of RFDs, command blocks or free RBDs.

EOFR

Value: 0x8000

Description: This bit marks the data structure as the last of its type to belong to this message. Used in envelopes and buffer descriptiors.

ERROR

Value: 0x0720

Description: This is the mask for those bits that mark the happening of an error during transmission.

FULL

Value: 0x4000

Description: Marks the data structure as full, i.e. done with, ready for further processing.

PREF

Value: 0x4000

Description: Marks a receive data structure as having been prefetched by the LANCE.

SIZE

Value: 0x0FFF

Description: In the message buffer these bits contain the size of the appended buffer (reserved or actual).

TYPES

Value: 0xF000

Description: The bits where the message type is encoded.

FLEX_MODE

Value: 0x0008

Description: This bit in the command word of a TCB or an RFD tells the LANCE that the flexible memory model is used in combination with this envelope.

DMA_UNDERRUN

Value: 0x100

Description: Underrun error may occurs on NTI Hardware.

ACK_REC

Value: 0x5000

Description: Set bits for RU ints to 1 (Acknowledge).

ACK_TR

Value: 0xA000

Description: set bits for CU ints to 1 (Acknowledge)

EMPTY

Value: 0xFFFFFFFF

Description: The LANCE's equivalent of the NULL pointer.

SSCMP.H

MAX_NODES

Value: 25

Description: The maximum number of pSOS nodes supported by the reliable KI protocol.

COUNTDOWN

Value: 0x20

Description: The number of ticks a kernel message waits for its acknowledgement before it is being retransmitted.

MAX

Value: 50

Description: maximum number of retransmits that is performed on a kernel message. After that the transmission is considered failed.

MULT_TICK_SSCMP

Value: 0x3

Description: SSCMP watchdog called every MULT_TICK pSOS tick.

SSCMPDS

Value: 16

Description: Length of SSCMP protocol header.

SSCMP_PACKET

Value: 0x93

Description: SSCMP protocol-internal packet type.

LIBFUNCTIONS.H

MAX_BLOCKS

Value: 255

Description: Maximum number of blocks, that can be allocated from dynamic memory management.

BLOCK_SIZE

Value: 30

Description: The size of one block in bytes.

INTERF.H

RB_COUNT

Value: 1186

Description: Number of RBDs and buffers on this node.

RFD_COUNT

Value: 960

Description: Number of RFDs on this node.

TC_COUNT

Value: 501

Description: Number of TCBs on this node.

TCTS_COUNT

Value: 64

Description: Number of TSTCBs on this node.

TB_COUNT

Value: 1046

Description: Number of TBDs and buffers on this node.

FREECNT

Value: ((sizeof(ts_tcbx)*TCTS_COUNT+sizeof(rfdx)*RFD_COUNT+ sizeof(bufx)*RB_COUNT+sizeof(rbdx)*RB_COUNT+sizeof(bufx)*TB_COUNT+ sizeof(tbdx)*TB_COUNT+sizeof(tcbx)*TC_COUNT)

Description: Calculates the number of bytes used by these structures and from this the number of bytes needed to let a following structure start at a 16 byte aligned address. The number must not be zero, therefore 16 is added to it.

T_ON

Value: 330

Description: The maximum time the LANCE is allowed to stay on the bus.

T_OFF

Value: 33

Description: The minimum time the LANCE has to stay off the bus, once it has been forced off.

C1

Value: 0x8E

Description: The first byte of the LANCE configuration. It means, that the setting of the prefetch bit has been set on and that the number of configuration bytes is 14.

C2

Value: 0xC8

Description: The second byte of the LANCE configuration. It disables monitoring and sets the FIFO limit as defaulted.

C3

Value: 0x40

Description: The third byte of the LANCE configuration. It disables Save Bad Frames.

C4

Value: 0xA6

Description: The fourth byte of the LANCE configuration. It sets the loopback mode to external loopback without use of the loopback pin, the length of the preamble to 8 bit, enables source address insertion and sets the hardware address length to 6 bytes.

C5

Value: 0x00

Description: The fifth byte of the LANCE configuration. It sets the backoff mode to immediate backoff and the linear and the exponential priority to 0.

C6

Value: 0x60

Description: The sixth byte of the LANCE configuration. It sets the interframe spacing to 96 transmit clocks.

C7

Value: 0x00

Description: The seventh byte of the LANCE configuration. It is the low byte of the value used for the slot time.

C8

Value: 0xF2

Description: The eighth byte of the LANCE configuration. The high byte of the slot time and the maximum number of retries.

C9

Value: 0x00

Description: The ninth byte of the LANCE configuration. It disables padding, bit stuffing, enables CRC32, CRC insertion, says that there is to be no transmission if Carrier Sense signal is missing, sets the encoding to NRZ, enables broadcast and disables promiscuous mode.

C10

Value: 0x00

Description: The tenth byte of the LANCE configuration. It sets collision detection to external, the collision detect filter to 0, the carrier sense to external and the carrier sense filter to 0.

C11

Value: 0x40

Description: The eleventh byte of the LANCE configuration. The minimum frame length is set to 64.

C12

Value: 0xEF

Description: The twelfth byte of the LANCE configuration. It sets monitoring to off, enables multicasting, turns off collision detection by source address comparison, turns on automatic retransmissions, tells the LANCE not to write the CRC into memory, disables padding and the sending of a preamble before assertion of the carrier sense signal.

C13

Value: 0x00

Description: The thirteenth byte of the LANCE configuration. It disables full duplex operation.

C14

Value: 0x3F

Description: The fourteenth byte of the LANCE configuration. No multiple individual addresses are allowed, backoff is enabled, the last 6 bits have to be 1.

DESCR0

Value: "MVME162_512A(CE)"

Description: This is a descriptive string, given to the pNA$^+$ whenever it requests it through use of an *ni_ioctl* call.

TYPE0

Value: 17

Description: Specifies an ethernet medium. This number is handed up to pNA$^+$ whenever requested through use of an *ni_ioctl* call.

MTU0

Value: 1500

Description: The maximum number of bytes transmissible per message. In Ethernet this is limited to 1500.

SPEED0

Value: 3200

Description: The speed of the medium in bytes per second. This number is handed up to pNA$^+$ whenever requested through use of an *ni_ioctl* call.

INT_LEVEL0

Value: 0x05

Description: Interrupt mask for the LANCE ISR. Interrupts up to level 5 are masked and therefore are permitted during ISR execution.

LANCE_INT_VEC_OFFSET

Value: 0x07

Description: The offset the LANCE interrupt has from the basis value entered on the board. This vlaue is entered during initialization, before the initialization of any of the application interfaces is called. Hence only the offset is needed to calculate the correct interrupt vector for the LANCE.

DRIVER.H

FWDPRIO

Value: 239

Description: Priority of the task *t_forward_rpk*, which decouples the ISR from forwarding packets to the specific Interfaces.

MULT_TICK_RETDAEMON

Value: 10

Description: Number of ticks between two periodic calls of *ret_daemon*, which asynchronoulsy tries to return RBD chains.

MACROS.H

CBASE(x)

Value: (((x)==EMPTY)?((INT32)(x)):((INT32)((INT32)(x)
+NTI_MEMORY_WINDOW)))

Description: Calculates Offset for CPU accesses to the shared memory.

LBASE(x)

Value: (((x)==EMPTY)?((INT32)(x)):((INT32)((INT32)(x)
-NTI_MEMORY_WINDOW)))

Description: Calculates Offset for LANCE accesses to the shared memory.

NI.H

MAXpNAIF

Value: 32

Description: Maximum number of interfaces supported by the driver for the PNA module.

NI_IOCTRL_PSOSDOWN

Value: 0x8004731A

Description: NI io_ctrl function code signalling pSOS down.

4.4 Data Structures

In this section one will find the data structures designed for the NTI-driver. They are, of course, contained in the include files. Since the most logical procedure is to list the structures per file, the author will do just that. Naturally the meaning of each structure is explained.

ALL.H

ti_acx

```
typedef struct ti_acx
{
  INT32 ms;
  INT32 ts;
  INT32 acc;
} ti_acx;
```

This is the structure of a transmit timestamp provided by the NTI. First 4 bytes of timesstamp information in ts, followed by another 4 bytes of macro timestamp information in ms. The last 4 bytes, acc, are the accuracy information.

timex

```
typedef struct
{
  INT32 rms;
  INT32 rts;
} timex;
```

The receive timestamp information contains no accuracy but only macrostamp and timestamp.

ts_tenvx

```
typedef packed struct
{
  volatile INT16 sw;
  volatile INT16 cw;
  volatile INT32 link;
  volatile INT32 tbd_add;
  volatile INT16 eof_tcbcount;
  volatile INT16 packet_info;
  volatile addr  da;
  volatile INT16 length_field;
#if TSTIMEOFF > 0
  volatile INT8 env_data[TSTIMEOFF];
#endif
  volatile INT32 area;
  volatile ti_acx tst;
#if TSFILLCNT > 0
  volatile INT8  unused[TSFILLCNT];
#endif
} ts_tenvx;
```

bytes between length field and timestamp in TCB Ts_tenvx is the memory structure of a TSTCB. The first three fields are as in any command block. Tbd_add is the address of the chain of TBDs belonging to the message. Eof_tcbcount contains the flag telling whether another TCB belongs to the message (always no) and the number of bytes that should be taken from this command block before transmitting the buffer. The next field, Packet_info is not transmitted and only used for storing information of what to do with the TSTCB after transmission.Da is the destination address, lenght_field the type of the message. Since the number of bytes between length field and timestamp in TCB can vary for experimental reasons (depends on relative speed of LANCE versus the NTI timestamping hardware), the next field is required. The LANCE will get the timestamp data to insert instead of the real data when it attempts to read tst. Area will contain the area (or virtual node) the message originates from. The last array has been added to blow the size of a TSTCB up to 64 bytes, which is necessary because of NTI hardware requirements. the remaining size TSFILLCNT depends on TSTIMEOFF.

There are multiple copies of this command block in shared memory, since it is used fairly often.

tenvx

```
typedef packed struct
{
  INT16 cw;
  INT16 sw;
  INT32 link;
  INT32 tbd_add;
  INT16 eof_tcbcount;
  INT16 packet_info;
  addr  da;
  INT16 length_field;
} tenvx;
```

Tenvx is the memory structure of a TCB. The first three fields are as in any command block. Tbd_add is the address of the chain of TBDs belonging to the message. Eof_tcbcount contains the flag telling whether another TCB belongs to the message (always no) and the number of bytes that should be taken from this command block before transmitting the buffer. Da is the destination address, lenght_field the type of the message. The field, packet_info is not transmitted and only used for storing information of what to do with the TCB after transmission. There are multiple copies of this command block in shared memory, since it is used fairly often.

renvx

```
typedef volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT32 rbd_add;
  INT16 eof_full_actc;
  INT16 size;
  addr  da;
  addr  sa;
  INT16 length_field;
  INT16 free;
  timex utcsu[AREACNT];
} renvx;
```

The receive envelope can be described with this structure. The first three fields are analoguous to the first three of a command block. The fourth, rbd_add is the address of the chain of RBDs belonging to the message. Eof_tcb_count contains the flag telling that this RFD is the last of this message (always true), whether it is full and the number of bytes

written into it from the buffer. How many bytes should be taken from the buffer can be specified in size. The next two fields, da and sa, contain the source respectively the destination address, length_field the type of the message, as usual. Utcsu reserves the space for the timestamp information. There is enough room to accomodate the timestamps of the maximum allowed number of areas in the system. The field free has been added only to make the RFD exactly 64 bytes in size, which is necessary due to NTI hardware requirements.

CI.H

ciparam

```
typedef union
{
  struct inip
  {
    int         if_num;
    ci_fn       ci_announce_address;
    ci_efn      ci_announce_err_add;
  } inip;
  struct gtbp
  {
    int         if_num;
    int         area;
    INT16       size;
    bufx        **buf;
  } gtbp;
  struct gtep
  {
    int         if_num;
    int         area;
    int         mode;
    tenvx       **tenv;
  } gtep;
  struct rtbp
  {
    bufx        *buf;
  } rtbp;
  struct rtep
  {
    tenvx       *tenv;
  } rtep;
  struct rrbp
  {
    bufx        *buf;
  } rrbp;
  struct rrep
  {
    renvx       *renv;
```

```
  } rrep;
  struct sendp
  {
    int         if_num;
    int         area;
    int         mode;
    INT16       size;
    tenvx       *tenv;
    bufx        *buf;
  } sendp;
  struct conp
  {
    int         if_num;
    bufx        *buf;
    int         size;
    bufx        **nbuf;
    char        *memory;
  } conp;
} ciparam;
```

This union serves as a parameter block for the different types of CI functions.

ci_fn

```
typedef int (*ci_fn)(renvx *renv,bufx *buf,INT16 size,int if_num);
```

The type of the function used to announce an incoming message to the clock synchronization.

ci_efn

```
typedef void (*ci_efn)(tenvx *env,bufx *buf,INT16 size,int if_num,int area);
```

The type of the function used to announce an error to the clock synchronization.

LANCE.H

cbx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
} cbx;
```

This is the smallest common denominator of all command blocks. It serves for inserting a block into the command block list or taking an executed one out of it. The first field, sw, is the status word, the next, cw, is the command word. Link is the address of the next command block in the list.

dump_areax

```
typedef packed struct
{
  INT8  sav_buf;
  INT8  conf[11];
  INT16 ffs;
  addr  ia;
  INT16 last_tx_status;
  INT16 ts_crc[2];
  INT16 rx_crc[2];
  INT16 rx_temp_mem[3];
  INT16 last_rx_status;
  INT16 hash_registers[4];
  INT16 slot_time_counter;
  INT16 wait_time_counter;
  INT16 rx_frame_length;
  INT32 reserved1;
  INT32 cb_in3;
  INT32 cb_in2;
  INT32 cb_in1;
  INT32 la_cb_address;
  INT32 rbd_ptr;
  INT32 int_mem;
  INT32 rfd_size;
  INT32 tbd_ptr;
  INT32 base_add;
  INT32 ru_temp_reg;
  INT32 tcb_count;
  INT32 next_rb_size;
  INT32 next_rb_addr;
  INT32 curr_rb_size;
  INT32 la_rbd_add;
  INT32 next_rbd_add;
  INT32 curr_rbd_add;
  INT32 curr_rbcnt;
  INT32 next_fd_add;
  INT32 curr_fd_add;
  INT32 temp_cu_reg;
  INT32 next_tb_count;
  INT32 buffer_add;
  INT32 la_tbd_add;
  INT32 next_tbd_add;
  INT32 cb_command;
```

```
  INT32 next_cb_add;
  INT32 curr_cb_add;
  INT32 scb_cmd_word;
  INT32 scp_ptr;
  INT32 cb_stat_word;
  INT32 mm_lfsr;
  INT8  mm_bit_array[28];
  INT8  cu_port[16];
  INT32 mm_alu;
  INT32 reserved2;
  INT32 mm_tempa_rr;
  INT32 mm_tempa;
  INT32 tmx_dma_bcnt;
  INT32 mm_in_port_add_reg;
  INT32 tmx_dma_add;
  INT32 mm_port_reg1;
  INT32 rx_dma_bcnt;
  INT32 mm_port_reg2;
  INT32 rx_dma_add;
  INT32 reserved3;
  INT32 bus_ttimers;
  INT16 diu_ctrl_low;
  INT16 diu_ctrl_high;
  INT32 reserved4;
  INT16 fragezeichen;
  INT8  irgendwas;
  INT8  sysbus_byte;
  INT32 biu_ctrl_reg;
  INT32 mm_disp_reg;
  INT32 mm_status_reg;
  INT16 dump_command_status_word
} dump_areax;
```

This structure maps the area where the LANCE dumps the values of the internal registers when it executes a DUMP command. See [Cor92, p. 4-153ff] for details on the various fields.

nopx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
} nopx;
```

This command does nothing. The command block contains only the fields necessary for all commands: The status and command word and the pointer to the next command block.

There exists only one NOP command block in shared memory.

iasx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  addr  ia;
} iasx;
```

The first three fields are as in all commands. The last is for the Ethernet hardware address of this node. The command tells the LANCE its individual address. There exists only one copy of it in shared memory.

cfgx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT8  cfgn[CFG_CNT];
} cfgx;
```

As this is the command block for configuring the LANCE, the last field contains the bytes with the configuration parameters (listed in detail in [Cor92, p.4-126ff]). The first three fields are as discribed for cbx. There exists only one copy of this command in shared memory.

mcsx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT16 mc_count;
  INT8  mc_adrl[MAX_ADDR_NO][ADDRCNT];
} mcsx;
```

This command block is used to hand the LANCE the multicast addresses. Again, the first three entries are as described for cbx. The fourth field is mc_count. It is the number of addresses written into mc_adrl, which contains the actual addresses. There is also only one copy of this command block in shared memory.

tbdx

```
typedef packed struct
{
  INT16 eof_size;
  INT16 zero;
  INT32 n_tbd;
  INT32 tb_addr;
} tbdx;
```

This structure is not that of a command, but of a transmit buffer descriptor (TBD). Eof_size is the field where the flag marking the last TBD belonging to a message is located. The rest of the field serves for telling the LANCE the size of the data in the associated buffer. It should not be larger than B_SIZE. The next field is always 0. The one after it is the pointer to the next TBD and the last one the address of the associated buffer. There are as many TBDs in shared memory as transmit buffers.

ts_tcbx

```
typedef typedef ts_tenvx ts_tcbx;
```

See section ALL.H, *ts_tenvx* for more information.

rfdx

```
typedef typedef renvx rfdx;
```

See section ALL.H, *renvx* for more information.

tcbx

```
typedef typedef tcbx tenvx;
```

See section ALL.H, *tenvx* for more information.

tdrx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT16 result;
  INT16 zero;
} tdrx;
```

This is another solitary command block existing only once. It is used for the TDR command. The field result contains, as the name tells us, the result of its execution. See [Cor92, p. 4-151f] for details. The last field is always 0 and the first three fields are as in the other commands and described for cbx.

dumpx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT32 buffer;
} dumpx;
```

Another command block existing only once in shared memory. The first three fieldsare the same as for any of the other command blocks. Buffer is the address where the LANCE should dump the data to. The structure of the dumped data is mirrored in dump_areax.

diagx

```
typedef packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 link;
} diagx;
```

This command block is used for having the LANCE perform a self test. The result of this test can be read from the status word. This is the last of the command blocks that exist only once in shared memory.

rbdx

```
typedef volatile packed struct
{
  INT16 eof_full_actc;
  INT16 zero1;
  INT32 n_rbd;
  INT32 rb_addr;
  INT16 el_pref_size;
  INT16 zero2;
} rbdx;
```

Rbdx is the data structure of an RBD. The first field contains the flag marking the last one in a chain of RBDs, the one for telling whether the RBD is full and the number of

bytes the LANCE has written into the associated buffer. The next field is always 0. N_rbd is the address of the next RBD and rb_addr the address of the buffer associated to the RBD. The next field is el_pref_size. Here the flag marking the RBD as the last in the RBDL given to the LANCE, the one telling that the RBD has been prefetched for reception by the LANCE and the maximum number of bytes that may be written into the associated buffer are located.

scbx

```
typedef volatile packed struct
{
  INT16 sw;
  INT16 cw;
  INT32 cl;
  INT32 rfa;
  INT32 crc_errors;
  INT32 alignment_errors;
  INT32 resource_errors;
  INT32 overrun_errors;
  INT32 rcv_errors;
  INT32 frame_errors;
  INT16 timer_off;
  INT16 timer_on;
} scbx;
```

The data structure describes the SCB. It exists only once in shared memory, being the main means of communicating with the LANCE. The command word, cw, encodes commands for the LANCE, sw as the status word its status. The address of a new command list has to be entered into cl before starting the CU while the address of a new receive area has to be written into rfa prior to starting the RU. The following 6 fields are error counters. If their names are not descriptive enough, more details can be found in [Cor92, p.4-118]. The last two fields are timer_off and timer_on. In here the values of the bus throttle timers can be written.

iscpx

```
typedef volatile packed struct
{
  INT8  zero1;
  INT8  busy;
  INT16 zero2;
  INT32 scb_address;
} iscpx;
```

This is the structure of the ISCP. The fields zero1 and zero2 are always 0, nomen est omen. As long as busy stays $\neq 0$, the LANCE has not finished its initialization sequence and not read the address of the SCB, contained in scb_address. There is only one ISCP in shared memory.

scpx

```
typedef packed struct
{
  INT32 sysbus;
  INT32 unused;
  INT32 iscp_address;
} scpx;
```

The SCP contains as its first field the one for preconfiguring the LANCE. The next is unused and always 0, the last one, iscp_address is the pointer to the ISCP. There is also only one SCP in shared memory.

t_tfreelist

```
typedef PACKED struct
{
  struct t_tfreelist *next;
  INT32 length;
  INT32 start;
  char reserved[2]; /*for own memory management!*/
} t_tfreelist;
```

The structure describes free list entries for the transmit buffer management. Each entry describes a contigous chain of free TBDs. It contains the length of an entry (number of TBDs in chain), the start index (shared memory index of the first TBD in the chain) and a next pointer to the following entry.

SSCMP.H

awx

```
typedef struct
{
  unsigned char repeats;
  INT32 stamp;
  envx  *env;
  bufx  *buf;
  unsigned char index;
  unsigned char  flag;
  INT16 size;
  struct awx *next;
} awx;
```

This structure contains the data necessary to store a message waiting for its acknowledgement. That includes first the count of retransmits in repeats. stamp contains a timestamp

increased on each pSOS clock tick (e.g. every 10 ms). It is compared with the actual timestamp by *watchdog* and if it differs too much from it, the message is resent.
Env is the pointer to the envelope of the message, buf the one to the data buffer. The number of bytes in the buffer is contained in size. flag tells whether the message is at the moment in the process of being transmitted. If it is 0, the message is free, otherwise it is being retransmitted and its entries should not be touched. Index is an increasing counter, which is used to sequence messages. Next points to the next message waiting for retransmission to the same node. The queue is ordered, so the next message was sent later.

waitx

```
typedef struct
{
  char ack_index;
  char last_sent;
  char rec_index;
} waitx;
```

There is one copy of this structure for each node in the net. It is the control or rather protocol structure for kernel messages between this node and one other. Ack_index is the index of the messages that should be acknowledged next. last_sent is the index of the message that has last been entered into the control structure and rec_index is the number a message that wants to be received on this node and originates from the other node should have to get through. Otherwise it is discarded.

INTERF.H

valx

```
typedef packed struct
{
  addr ad;
  char active;
} valx;
```

Each node in the net needs a table containing the Ethernet hardware addresses of all the nodes in the net, since pSOS^{+m} only specifies the node number but not the address. This table has been expanded to keep the roster infromation provided by pSOS^{+m} as well. Therefore an entry in the table no longer contains only the hardware address (ad), but also a flag telling whether the node is active or not (active). The node number is defined by the slot the address occupies in the table.

confx

```
typedef packed struct
{
  INT8 cfga[CFG_CNT];
  char *descr;
  long type;
  long mtu;
  long speed;
  addr ich;
  valx madrl[MAX_ADDR_NO];
  int  maxnodes;
} confx;
```

This is the data structure where all the parameters for the NTI-driver are stored. One is needed for each hardware interface the NTI-driver services. It contains the configuration for the LANCE (in cfga), certain information needed for the support of MIB-II, such as the address of a string with a short description of the NTI-driver (in descr), the type of the network (in type), the maximum transmission unit (in mtu) and the speed of the network (in speed). The remaining fields are the Ethernet hardware address of this interface (in ich), an array containing the multicast table (madrl, supplied by the user) and the maximum number of active nodes in the net (maxnodes). The Ethernet hardware address is also used for calculating the hardware addresses of the other nodes in the system, since the last byte is equal to the node number.

sh_memx

```
typedef packed struct
{
  ts_tcbx ts_tcbs[TCTS_COUNT];
  rfdx rfds[RFD_COUNT];
  bufx rbs[RB_COUNT];
  rbdx rbds[RB_COUNT];
  bufx tbs[TB_COUNT];
  tbdx tbds[TB_COUNT];
  tcbx tcbs[TC_COUNT];
  INT8 free[16-FREECOUNT];
  scpx scp;
  dump_areax dump_area;
  iscpx iscp;
  scbx scb;
  nopx nop;
  iasx ias;
  cfgx cfg;
  mcsx mcs;
  tdrx tdr;
  dumpx dump;
  diagx diag;
```

```
} sh_memx;
```

This is the map of a shared memory area. This layout is the same on all NTIs supported by the NTI-Driver. At the lowest address the timestamped envelopes are located. Directly behind them the RFDs. Afterwards follow the receive buffers, the RBDs, the transmit buffers, the TBDs and the TCBs. The structures existing only once in memory are set at the highest addresses. Those are the SCP, the area for dumping the registers, the ISCP, the SCB and the remaining commands. Between the TCBs and the SCP it may be necessary to insert up to 16 bytes that remain free, since the address of the SCP has to be 16 byte aligned. The number of bytes is calculated automatically, under the assumption, that the shared memory starts at a 16 byte aligned address (i.e., the last four bits of the address are 0).

4.5 Functions

We start with a list of the files the NTI-driver consists of, together with the functions each of them contains. Some of the functions listed here were not mentioned in chapter 3, because they only serve for simplifying procedures or for replacing necessary library functions such as memset or memcpy. When pSOS^{+m} is not active, e.g. during startup, the library functions are inaccessible, so that the NTI-driver has been kept independent of them.

- KI.C: The functions are required by pSOS^{+m}. They are called via an entry point that also acts as a dispatcher. Its address is told to pSOS^{+m} by entering it into the multiprocessor configuration table. The other way round the communication works via aKI announce packet entry point, the address of which is handed to the NTI-driver at initialization.
  Functions
  - ki_announce_error_up
  - ki_announce_up
  - ki_dummy
  - ki_getpkb
  - ki_init
  - ki_receive
  - ki_returnp
  - ki_roster
  - ki_send
  - ki_time
- NI.C: The functions are required by pNA^{+}. They are called through use of an entry point the address of which is entered into the pNA^{+}s table of network interfaces. Communication from the NTI-driver to pNA^{+} goes through the NI message announce entry point, the address of which is handed to the NTI-driver during initialization.
  Functions
  - ni_announce_up

  - ni_broadcast
  - ni_initial
  - ni_ioctl
  - ni_poll
  - ni_send

- CI.C: The functions required from the NTI-driver by the clock synchronization are accessible via an entry point. The driver announces incoming messages and transmission errors through call of respective announce entry points, the addresses of which are handed to the NTI-driver at initialization.
  Functions

  - ci_announce_error_up
  - ci_announce_up
  - ci_dispatcher
  - ci_get_tbuf
  - ci_get_tenv
  - ci_init
  - ci_ret_rbuf
  - ci_ret_renv
  - ci_ret_tbuf
  - ci_ret_tenv
  - ci_send

- SSCMP.C contains the functions constituting the transmission protocol necessary for the kernel messages
  Functions

  - get_no
  - init_sscmp
  - put
  - sscmp_error_announce
  - sscmp_announce
  - watchdog
  - release_sscmp

- ERROR.C is very short since it contains nothing but the two error handling functions.
  Functions

  - rcv_fatal_error
  - trans_fatal_error
  - fatal_error
  - asynchronous_error

- APPLTR.C contains the transmit part manager layer functions. For the most part they do nothing but check addresses for their validity (if they lie within the correct areas) and hand commands and/or messages up and down.
  Functions
  - alloc_ki_tbuf
  - alloc_ki_tenv
  - alloc_ni_tbuf
  - alloc_ci_tbuf
  - alloc_ci_tenv
  - transm
  - trans_ki
  - trans_ni
  - trans_ci
  - release_cb
  - ret_ki_tbuf
  - ret_ki_tenv
  - ret_ki_tmess
  - ret_ci_tbuf
  - ret_ci_tenv
- APPLREC.C contains the receive part middle layer functions. They also mainly check addresses and call the appropriate functions.
  Functions
  - forward_rpk
  - t_forward_rpk
  - ki_get_rbuf
  - poll_ni
  - rcv_error
  - ret_ki_rbuf
  - ret_ki_renv
  - ret_ci_rbuf
  - ret_ci_renv
- LANCETR.C contains the functions responsible for handling data structures of the transmit part and serving the CU.
  Functions
  - tbd_index
  - tb_index
  - get_tbd
  - get_tcb

  - get_ts_tcb
  - init_cfg
  - init_ias
  - init_lance
  - init_mcs
  - init_scb
  - init_tbd
  - init_tcb
  - init_trans
  - init_ts_tcb
  - put_cb
  - put_cb_urgent
  - release_cb_up
  - ret_tbd
  - ret_tcb
  - ret_ts_tcb
  - ucast

- LANCEREC.C contains functions responsible for the receive data structures and also for serving the RU
  Functions

  - rbd_index
  - rb_index
  - copy_block
  - copy_rbd
  - forward_rpk_up
  - init_rbd
  - init_receive
  - init_rfd
  - ret_rbd
  - ret_rbd_nc
  - ret_daemon
  - ret_rfd

- DISPATCHER.S is an assembler file. The function that serves as the entry point for pSOS^{+m} and the one for pNA^{+} are contained in here. They act as dispatchers and call the appropriate function depending on the command code.
  Functions

  - ki_dispatcher
  - ni_dispatcher

- LANCE.S: The assembler functions contained here were written for the lowest layer. They are for setting the interrupt mask, the wrapper for the ISR and a test-and-set function for atomic reading and setting of a value
  Functions
  - disable
  - enable
  - lancedrv
  - mask_drop

- LIBFUNCTIONS.C contains functions used instead of C standard library functions. Due to system calls in the existing library, the functions cannot be used before pSoS is up, although the driver already has to operate.
  Functions
  - memcompp
  - memcopp
  - memclear
  - mmalloc
  - mfree
  - meminit

- INTERF.C contains functions for operating the NTI. Here and only here the interface between the driver and the NTI is implemented. Changing the NTI requires only changes to this module.
  Functions
  - lance_isr0
  - LanStop
  - init
  - getEtherAddress0
  - intlanceon0

And now follows the detailed description of the functions. They are ordered by file, within those by alphabet. The first thing given for each function is the prototype. Under "Description" the reader will find a short resume of what the function does and how it goes about it. "Parameters" lists the parameters of the function, their type and not to forget, the semantics. Under "Return" the return type is specified. If there are any special values, they are, of course, listed. "Caller" lists all the functions calling this one with a short description of what they do. "Calls" lists the functions that are called by this one, again with a description of what they do. Under "Global Variables" the esteemed reader can find a list of the global variables that are used in this function toghether with a description, their types and whether they are only read ("R"), only written ("W") or both ("RW").

CI.C

ci_announce_error_up

Prototype:
void ci_announce_error_up(tenvx *env,bufx *buf,INT16 size,int if_num,int area)

Description:
Reports a transmission error to the clock synchronization unit. It hands up the data of the message so that the clock synchronization may act upon the error.

Parameters:
tenvx *env... pointer to transmission envelope
bufx *buf... pointer to data buffer
INT16 size... size of data buffer
int if... number of the interface the message was to be transmitted from
int area... number of the virtual node the message was to be transmitted from

Return:
void

Callers:
trans_fatal_error... forwards error reports to the concerned interface

Calls:
ci_announce_error... function of the clock synchronization

Global Variables:
ci_efn ci_announce_error[IFCNT]... R, address of the error announce entry point

ci_announce_up

Prototype:
void ci_announce_up(renvx *renv,bufx *buf,INT16 size,int if_num)

Description:
Forwards an incoming message to the clock synchronization. The clock synchronization returns an integer value to the NTI-driver, telling it whether it should discard the message or leave it. In the former case the appropriate management layer procedures are called. There is no parameter for specifying the virtual node a message was received on, since there is only one where messages can be received as well as sent.

Parameters:
renvx *renv... address of the receive envelope containing the receive timestamp
bufx *buf... address of the not necessarily contiguous buffer
INT16 size... number of bytes in the buffer
int if_num... number of the interface the message came in through

Return:
void

Callers:
forward_rpk... forwards incoming messages to the interfaces they are intended for.

Calls:
ci_announce... function of the clock synchronization
ret_ci_renv... returns a receive envelope to the pool
ret_ci_rbuf... returns a receive buffer to the pool

Global Variables:
ci_fn ci_announce[IFCNT]... R, address of the message announce entry point

ci_dispatcher

Prototype:
int ci_dispatcher(int fc,ciparam *param)

Description:
This function is called by the clock synchronization and handed a function code and the address of a parameter block. This block is interpreted according to the function code and the appropriate CI function called. The return value of this function is then returned to the clock synchronization.

Parameters:
int fc... the function code. Possible values are:
CI_INIT... initialization of NTI-driver
CI_GTB... get transmit buffer
CI_GTE... get transmit envelope
CI_RTB... return transmit buffer
CI_RTE... return transmit envelope
CI_RRB... return receive buffer
CI_RRE... return receive envelope
CI_SEND... transmit a message
ciparam *param... address of the parameterblock for the function. This block is a union consisting of the following parts:
inip ... the parameters for initialization
int if_num... number of the interface
ci_fn ci_announce_address ... address of announce message entry point
ci_efn ci_announce_err_add... address of announce error entry point
gtbp ... the parameters for getting transmission buffer
int if_num... number of the interface
int area... number of the virtual node
INT16 size... required size of the buffer
bufx **buf... address to write address of the buffer to
gtep ... the parameters for getting transmission envelope
int if_num... number of the interface
int area... number of the virtual node
int mode... type of the envelope
tenvx **tenv... address to write address of the envelope to
rtbp ... the parameters for returning transmit buffer
bufx *tbuf... address of the buffer
rtep ... the parameters for returning a transmit envelope
tenvx *tenv... address of the envelope
rrbp ... the parameters for returning receive buffer
bufx *rbuf... address of the buffer
rrep ... the parameters for returning a receive envelope
renvx *renv... address of the envelope
sendp ... the parameters for sending a message
int if_num... number of the interface

int area... number of the virtual node
int mode... type of transmission
INT16 size... size of buffer
tenvx *tenv... address of the envelope
bufx *buf... address of the buffer
conp ... the parameters for rendering a buffer contiguous
int if_num... number of the interface
bufx *buf... address of possibly incontiguous buffer
int size... size of buffer
bufx **nbuf... address to write address of contiguous buffer into
char *memory... alternatively, space to copy buffer into

Return:

int... error code, the rturn value of the called function.
It is 0 if everything was OK, or
INVALID_IF_NUMBER if that parameter was invalid
INVALID_POINTER if buf was NULL
NO_BUF_FREE if not enough buffer space was available
NO_ENV_FREE if no envelope was available
NETWORK_NOT_FUNCTIONAL if a hardware error was detected
INVALID_ENV_ADDRESS if the address does not indicate an existing envelope
INVALID_BUF_ADDRESS if the address designates a space not in any pool
COULD_NOT_BE_MADE_CONTIGUOUS if rendering contiguous was not possible
NO_ENVELOPE_ATTACHED if the address of the envelope is NULL
INVALID_COMMAND if the function code does not exist

Callers:

clock synchronization

Calls:

ci_init ... initializes the driver
ci_get_tbuf ... allocates transmit buffer of the required size
ci_get_tenv ... allocates transmit envelope
ci_ret_tbuf ... returns transmit buffer
ci_ret_tenv ... returns transmit envelope
ci_ret_rbuf ... returns receive buffer
ci_ret_renv ... returns receive envelope
ci_send ... transmits a message
ci_mcon ... renders a buffer contiguous

Global Variables:

none

ci_get_tbuf

Prototype:

int ci_get_tbuf(INT16 size,bufx **buf,int if_num,int area)

Description:

Takes a contiguous buffer of the requested size from the pool of the requested interface and virtual node. If none is available buf points to NULL and an error is returned.

Parameters:
INT16 size... size the buffer should have
bufx **buf... address where the address of the buffer is to be written in
int if_num... interface the buffer is intended for
int area... virtual node the buffer is intended for

Return:
int... error code, is 0 if everything was OK, or
INVALID_IF_NUMBER if that parameter was invalid
INVALID_POINTER if buf was NULL
NO_BUF_FREE if not enough buffer space was available

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
alloc_ci_tbuf... allocates buffer space

Global Variables:
int ActiveAs[IFCNT]... R, tells the number of virtual nodes on an interface
int ci_inited[IF_CNT]... R, TRUE if the CI is initialized

ci_get_tenv

Prototype:
int ci_get_tenv(int mode,tenvx **tenv,int if_num,int area)

Description:
Takes an envelope of the requested type (timestamped or not) from the pool of the requested interface and virtual node. If none is available, tenv points to NULL and an error is returned.

Parameters:
int mode... specifies the type of the envelope
tenvx **tenv... address where the address of the envelope is to be written to
int if_num... interface the envelope is intended for
int area... virtual node the envelope is intended for

Return:
int... error code, is 0 if everything was OK, or
INVALID_IF_NUMBER if that parameter was invalid
INVALID_POINTER if buf was NULL
NO_ENV_FREE if no envelope was available

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
alloc_ci_tenv... allocates an envelope

Global Variables:
int ActiveAs[IFCNT]... tells the number of virtual nodes on an interface
int ci_inited[IF_CNT]... R, TRUE if the CI is initialized

ci_init

Prototype:
int ci_init(int if_num,ci_fn ci_announce_address, ci_efn ci_announce_err_ad)

Description:
This function calls the management layer procedure for initialization of the NTI-driver and the hardware. The driver also gets the address of the CI announce message entry point and stores it.

Parameters:
int if_num... number of the interface that should be initialized
ci_fn ci_announce_address... address of message announce entry point
ci_efn ci_announce_err_ad... address of error announce entry point

Return:
int... error code, is 0 if everything was OK, or
INVALID_IF_NUMBER if the parameter was invalid
NETWORK_NOT_FUNCTIONAL if a hardware error was detected
INVALID_POINTER... if an address is NULL

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
init... initializes hardware and sets certain variables

Global Variables:
ci_fn ci_announce[IFCNT]... W, address of the message announce entry point
ci_efn ci_announce_error[IFCNT]... W, address of the error announce entry point
int ci_inited[IF_CNT]... R, TRUE if the CI is initialized

ci_ret_tbuf

Prototype:
int ci_ret_tbuf(bufx *buf)

Description:
Returns transmit buffer space to the pool.

Parameters:
bufx *buf... address of the buffer

Return:
int... error code, is 0 if everything is OK, or
INVALID_BUF_ADDRESS if the address indicates a buffer not in any pool

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
ret_ci_tbuf... calculates address of the first buffer descriptor and releases it

Global Variables:
none

ci_ret_tenv

Prototype:
int ci_ret_tenv(tenvx *tenv)

Description:
Returns transmit envelope (regardless of its type) to the pool

Parameters:
tenvx *tenv... address of the envelope

Return:
int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if the address does not indicate an existing envelope

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
ret_ci_tenv... determines the type of the envelope and releases it

Global Variables:
none

ci_ret_rbuf

Prototype:
int ci_ret_rbuf(bufx *buf)

Description:
Returns receive buffer to the pool.

Parameters:
bufx *buf... address of the buffer

Return:
int... error code, is 0 if everything is OK, or
INVALID_BUF_ADDRESS if the address designates a space not in any pool

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
ret_ci_rbuf... calculates address of first buffer descriptor and releases it

Global Variables:
none

ci_ret_renv

Prototype:
int ci_ret_renv(renvx *renv)

Description:
Returns receive envelope to the pool.

Parameters:
renvx *renv... address of the envelope

Return:
int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if the address does not indicate an existing envelope

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
ret_ci_renv... returns the envelope after checking the validity of its address

Global Variables:
none

ci_send

Prototype:
int ci_send(int mode,tenvx *tenv,bufx *buf,INT16 size,int if_num,int area)

Description:
The clock synchronization calls this function and hands it a lot of parameters. From these a message is compiled and sent to the given address. The message may be sent with priority, meaning that it is to be inserted as far to the front of the outbound queue as possible.

Parameters:
int mode... marks priority and handling of the packet:
0... no priority, no immediate release
RET_PKB_IMM... release buffer and envelope immediately after sending
PRIOR... this packet has high priority
tenvx *tenv... address of the envelope
bufx *buf... address of the data buffer
INT16 size... size of the data buffer
int if_num... number of the interface the message is to be sent through
int area... number of the virtual node the message "comes from"

Return:
int... error code, is 0 if everything is OK, or
INVALID_IF_NUMBER if the specified interface does not exist or is not active
NO_ENVELOPE_ATTACHED if the address of the envelope is NULL

Callers:
ci_dispatcher ... entry point for commands from clock synchronization

Calls:
trans_ci... compiles the message and puts it into the queue

Global Variables:
int ActiveAs[IFCNT]... tells the number of virtual nodes on an interface
int ci_inited[IF_CNT]... R, TRUE if the CI is initialized

KI.C

ki_announce_error_up

Prototype:
void ki_announce_error_up(envx *env,bufx *buf)

Description:
This function is intended to report transmit error conditions to pSOS^{+m}. Unfortunately pSOS^{+m} does not support the reporting of transmit errors. It expects all messages to be transmitted in their correct order and without fault. In the case of an error within the NTI-driver that cannot be reported directly by return code, the driver is to react to it by calling the *k_fatal* routine. As has been described in Chapter 2, the user can install a routine of his choice as a fatal error handler by entering it into the pSOS^{+m} configuration table. This may be a function that tries to recover such an error. A message that could not be transmitted may not simply be discarded, since the correct sequence of the messages is crucial and pSOS^{+m} has no way of knowing that it should retransmit that particular message. The reporting of error conditions has not been made a feature on the grounds that if a transmisssion that has already been retried, etc, fails, there is some serious fault either in the own network interface or in the one of the other node, that cannot be recovered by the software alone.
The call of *k_fatal* does not return.

Parameters:
envx *env... address of the envelope of the unsent message
bufx *buf... address of its buffer space

Return:
void (the function does not return)

Callers:
watchdog... handles retransmissions and releases, counts down timers
sscmp_error_announce... gets error reports from the error handler

Calls:
none

Global Variables:
none

ki_announce_up

Prototype:
void ki_announce_up()

Description:
Announces to pSOS^{+m} that a KI message has come in. The message itself is not handed up.

Parameters:
none

Return:
void

Callers:
sscmp_announce... checks incoming message, acknowledges and announces it

Calls:
ki_announce... the message announce entry point of the pSOS^{+m} kernel

Global Variables:
int (*ki_announce)()... R, address of the message announce entry point

ki_dummy

Prototype:
INT32 ki_dummy()

Description:
One number was left free in the specification of the KI. This function has been entered there for filling the free slot.

Parameters:
none

Return:
INT32... is always 0

Callers:
pSOS^{+m} kernel

Calls:
nothing

Global Variables:
none

ki_init

Prototype:
INT32 ki_init(INT32 unused1,INT32 unused2,int (*ki_announce_function) ())

Description:
Is called by pSOS^{+m} during its startup phase. That means, that the interrupt level is 7, allowing no interrupts of lower levels to be processed. Additionally no library function is available. The function is expected to initialize the NTI-driver and the hardware. It also initializes the transmission protocol. The address of the message announce entry point is forwarded as a parameter and stored in a global variable.

Parameters:
INT32 unused1... this register is of no moment for this function
INT32 unused2... this register is of no moment for this function
int (*ki_announce_function)()... address of message announce entry point

Return:
int... error code, 0 if everything is OK, or
INVALID_IF_NUMBER if there is no hardware interface with the given number
NETWORK_NOT_FUNCTIONAL if a hardware error was detected

Callers:
pSOS^{+m} kernel

Calls:
init_sscmp... initializes transmit protocol
init... initializes hardware

Global Variables:
int (*ki_announce)()... W, address of the message announce entry point

ki_getpkb

Prototype:
INT32 ki_getpkb(INT32 unused,int dnn,bufx *buf)

Description:
Returns the address of contiguous buffer space up to the kernel. The size of this space is fixed to the value ki_message_size. If no buffer was available buf is NULL and an error returned.

Parameters:
INT32 unused... this register is of no moment for this function
int dnn... number of the destination node - not used here, since the buffer space in this implementation of the NTI-driver is independent of the destination node contrary to an e.g. shared memory implementation
bufx *buf... since the variables are returned directly, in the registers, this variable serves as an output parameter. In it, the address of the buffer is returned, or NULL, if none was available.

Return:
INT32... error code, is 0 if everything is OK, or
INVALID_IF_NUMBER if KI is not initialized
NO_BUF_FREE if no buffer space was available

Callers:
pSOS^{+m} kernel

Calls:
alloc_ki_tbuf... allocates required amount of buffer space

Global Variables:
int ki_inited... R, TRUE if the KI is initialized

ki_receive

Prototype:
INT32 ki_receive(INT32 unused1,INT32 unused2,bufx *buf)

Description:
Is called by pSOS^{+m} in response to a call of the message announce entry point. With it the waiting message is fetched. During startup, however, it is used by the slave nodes to poll for an incoming message that is the answer to their (previous) request to join. During this polling the interrupt level remains high, though, so that an incoming message cannot be processed (due to the lower level of the LANCE interrupt). It is therefore necessary to drop the interrupt level for a short time at each call of *ki_receive* while pSOS^{+m} is not active.
The interrupt level of the LANCE has not been set higher, since interrupts of level 7 may interrupt one another, which is not a desirable circumstance in this implementation. If no message was waiting for reception buf is NULL.

Parameters:
INT32 unused1... this register is of no moment for this function
INT32 unused2... this register is of no moment for this function
bufx *buf... again the variable serves as an output parameter, this time for the received message buffer. It is NULL if no message was waiting to be received.

Return:
INT32... is always 0

Callers:
pSOS^{+m} kernel

Calls:
mask_drop... assembler function setting the interrupt mask to 0
enable... assembler function setting the interrupt mask to a given value
ki_get_rbuf... fetches the waiting message

Global Variables:
int ki_inited... R, TRUE if the KI is initialized
INT32 PSOS_FLAG... R, 0 if pSOS^{+m} is not active, $\neq 0$ otherwise

ki_returnp

Prototype:
INT32 ki_returnp(INT32 unused1,INT32 unused2,bufx *buf)

Description:
Returns receive buffer to the pool.

Parameters:
INT32 unused1... this register is of no moment for this function
INT32 unused2... this register is of no moment for this function
bufx *buf... address of the buffer

Return:
INT32... error code, 0 if everything is OK, or
INVALID_IF_NUMBER if KI is not initialized
INVALID_BUF_ADDRESS if the given buffer does not exist

Callers:
pSOS^{+m} kernel

Calls:
ret_ki_rbuf... checks the buffer and returns it

Global Variables:
int ki_inited... R, TRUE if the KI is initialized

ki_roster

Prototype:
INT32 ki_roster(int change_type,int nno,INT32 *roster)

Description:
This function is called by the kernel to enable the driver to maintain an internal tabel of active nodes. This is not necessary in this implementation of the driver, therefore this aspect of the call is ignored. On the other hand, this function call is the last before pSOS^{+m} goes active, therefore the appropriate flag can be set to TRUE.

Parameters:
int change_type... specifies what happened with the node
INITIAL... the initial roster is handed over for initialization of the NTI-driver's equivalent of a roster

JOIN ... the node has joined the system
LEAVE ... the node has left the system
int nno... number of the concerned node
INT32 *roster... address of the initial roster, only valid in case of "INITIAL"

Return:
INT32... error code, 0 if everything is OK, or
INVALID_IF_NUMBER if the parameter was invalid

Callers:
pSOS^{+m} kernel

Calls:
nothing

Global Variables:
int ki_inited... R, TRUE if the KI is initialized confx *conf... RW, structure containing configuration parameters of the NTI-driver
INT32 PSOS_FLAG... W, 0 if pSOS^{+m} is not active, $\neq 0$ otherwise

ki_send

Prototype:
INT32 ki_send(INT32 size,int dnn,bufx *buf)

Description:
Inserts the hardware destination address instead of the number of the destination node and then hands the message to the transmission protocol. As has been said before, pSOS^{+m} expects the NTI-driver to transmit without reordering of messages. The only part of the NTI-driver where reordering **can** take place is between receiving the message from the kernel entering it into the data structure of the transmit protocol. Therefore this part had to be made atomic which is done by raising the interrupt mask. It is reset it to its previous value before the return to pSOS^{+m}.
The destination hardware address is determined by inserting the destination node number instead of the least significant byte of the individual address.

Parameters:
INT32 size... size of the data buffer
int dnn... number of the destination node
bufx *buf... address of the buffer

Return:
INT32... error code, 0 if everything is OK, or
NO_ENV_FREE if no envelope was available
NO_SPACE_IN_WINDOW if protocol has no room for additional messages
INVALID_IF_NUMBER if KI is not initialized
INVALID_ADDRESS_NUMBER if there is no active node with the given number

Callers:
pSOS^{+m} kernel

Calls:
memcopp... copies areas in memory of a given size
put... enters message into protocol data structure

Global Variables:
int ki_inited... R, 0 if KI has not been initialized yet, $\neq$ otherwise

ki_time

Prototype:
INT32 ki_time()

Description:
As soon as pSOS^{+m} is active, this function is called every tick (i.e. 10ms). It is used to implement the periodic calls to *watchdog* and to *ret_daemon*. So, these two functions are called on multiples of the pSoS tick (the multiples are specified as constants).

Parameters:
none

Return:
INT32... error code, 0 if everything is OK, or
INVALID_IF_NUMBER if KI is not initialized

Callers:
pSOS^{+m} kernel

Calls:
watchdog... counts down timers, handles retransmits, releases and error conditions.
ret_daemon... tries to return chains of RBDs, which could not be returned earlier due to conflicts with the Lance.

Global Variables:
int ki_inited... R, 0 if KI has not been initialized yet, $\neq$ otherwise

NI.C

ni_announce_up

Prototype:
void ni_announce_up(int type,bufx *buf,INT16 size,int if_num,addr *sa,addr *da)

Description:
Forwards a received message intended for pNA^{+} up to it through use of the message announce entry point. The address of it the NI part of the NTI-driver has been handed at intialization. The interface number used by pNA^{+}to designate this interface is looked up in a table filled at initialization.

Parameters:
int type... the type of the message - the possibilities are:
NI1... meaning an ARP message, or
NI2... meaning an IP message
bufx *buf... address of the data buffer
INT16 size... size of the data buffer
int if_num... interface the message came in through
addr *sa... hardware source address of the message
addr *da... hardware destination address of the message

Return:
void

Callers:
forward_rpk... shares out incoming messages between the interfaces

Calls:
ni_announce... message announce entry point of pNA$^+$, returns staus of pSOS

Global Variables:
int (*ni_announce)()... R, address of message announce entry point
INT32 PSOS_FLAG... R, 0 if pSOS^{+m} is not active, $\neq 0$ otherwise
int reifnummap[IFCNT]... R, table for mapping internal interface number to pNA$^+$'s

ni_broadcast

Prototype:
INT32 ni_broadcast(struct nibrdcast *param)

Description:
With this call pNA$^+$ transmits a message to all nodes in the network. Internally, the function just inserts the hardware broadcast address supported by the LANCE and calls ni_send. After transmission the packet buffer has to be released.

Parameters:
nibrdcast *param... parameters for broadcast function
char *buff_addr... pointer to message block list
long count... length of buffer
long type... message type, NI1 for ARP message or NI2 for IP message
long if_num... number of the interface

Return:
INT32... error code, 0 if everything is OK, or
INVALID_IF_NUMBER if the parameter was invalid
ENOBUFS... if no envelope was available (defined in error.h)

Callers:
pNA$^+$

Calls:
trans_ni... compiles a message out of the data and sends it

Global Variables:
int ifnummap[MAXIF]... R, table for mapping pNA$^+$'s interface number to internal one
int ni_inited[IF_CNT]... R, 0 if NI is not initialized, $\neq$ otherwise

ni_initial

Prototype:
INT32 ni_initial(struct niinit *param)

Description:
Sets the address of the message announce entry point and calls the initialization of the NTI-driver and the hardware. Function pointers for pNA$^+$ internal memory block allocation/deallocation functions are saved in global variables. pNA$^+$ requires that the hardware address of the interface be returned after initialization. In the case of an error the return value has to be -1 according to NI specification.

Parameters:

niinit *param... parameters for initialization function
long (*ap_address)()... address of entry point
long if_num... number of the interface
long ip_addr... IP address of this interface
struct ni_funcs *funcs... not needed

Return:

INT32... address of the Ethernet hardware address of this interface, or
NETWORK_NOT_FUNCTIONAL (-1) if initialization failed for any reason.

Callers:

pNA$^+$

Calls:

init... initializes data structures and hardware if necessary

Global Variables:

confx *conf... R, structure containing cofiguration parameters of the NTI-driver
int (*ni_announce)()... W, address of message announce entry point
INT32 Astate[IFCNT]... W, current administrative status of the NTI-driver: 1 means running, 2 down and 3 testing
INT32 Ostate[IFCNT]... W, current operative status of the NTI-driver: means running, 2 down and 3 testing
INT32 LastChange[IFCNT]... W, time of last change of administrative status
int ifnummap[MAXIF]... W, table for mapping pNA$^+$'s interface number to internal
int reifnummap[IFCNT]... W, table for mapping internal interface number to pNA$^+$'s
int usedifs... the number of interfaces already initialized

ni_ioctl

Prototype:

INT32 ni_ioctl(struct niioctl *param)

Description:

This function supports the MIB-II. The function is given a command code and the address of the structure to write the results into. Additionally, with the MIB-II command code SIOGIFSPECIFIC, the packet forwarding task can be turned on and off: the string parameter ''tenable" turns it on, while "tdisable" turns it off.

Parameters:

niioctl *param... parameters for I/O control function
long cmd... code of the command
long *arg... address for writing return information
long if_num... number of the interface

Return:

INT32... error code, is 0 if everything is OK, or
INVALID_IF_NUMBER if the parameter was invalid

Callers:

pNA$^+$

Calls:

memcopp... copies given number of bytes from one place to another

Global Variables:

INT32 PSOS_FLAG... R, 0 if pSOS^{+m} is not active, $\neq 0$ otherwise
int fwdtask_status ... W, enable/disable task for packet forwarding
addr bc_address... R, broadcast address
confx *conf... R, structure containing configuration parameters of NTI-driver
INT32 Astate[IFCNT]... RW, administrative state of NI part of the NTI-driver
INT32 Ostate[IFCNT]... R, operative state of NI part of the NTI-driver
INT32 LastChange[IFCNT]... RW, point in time of last change of administrative state
INT32 InBytes[IFCNT]... R, number of received bytes
INT32 InUcast[IFCNT]... R, number of received singlecast messages
INT32 InNucast[IFCNT]... R, number of received multi- or broadcast messages
INT32 InDiscards[IFCNT]... R, number of discarded incoming messages
INT32 InErrors[IFCNT]... R, number of faulty incoming messages
INT32 UnknownProtos[IFCNT]... R, number of messages with unknown higher layer protocol
INT32 OutBytes[IFCNT]... R, number of sent bytes
INT32 OutUcast[IFCNT]... R, number of sent singlecast messages
INT32 OutNucast[IFCNT]... R, number of sent multi- or broadcast messages
INT32 OutDiscards[IFCNT]... R, number of outbound discarded messages
INT32 OutErrors[IFCNT]... R, number of outbound faulty messages
INT32 OutQlen[IFCNT]... R, length of the outbound queue
int ifnummap[MAXIF]... R, table for mapping pNA^{+}'s interface number to internal
int ni_inited[IF_CNT]... R, 0 if NI is not initialized, $\neq$ otherwise

ni_poll

Prototype:

INT32 ni_poll(struct nipoll *param)

Description:

This function is called regularly by pNA^{+} if the interface is defined as a polling interface, i.e. as having no interrupt facilities. Ultimately the driver layer function *lance_isr* is called to check the interrupt status of the LANCE. It may be called if pNA^{+} is active while pSOS^{+m}, which does the scheduling, is not. The received messages are not returned with this function, but, as is required in the definition of the interface, via the message announce entry point (which happens automatically at the call of *lance_isr*).

Parameters:

nipoll *param... parameters for polling function
long if_num... number fo the interface

Return:

INT32... error code, is 0 if everything is OK, or
INVALID_IF_NUMBER if the parameter was invalid
EINVALID if one of the parameters was invalid (defined in error.h)

Callers:

pNA^{+}

Calls:

poll_ni... calls ISR

Global Variables:
int ifnummap[MAXIF]... R, table for mapping pNA$^+$'s interface number to internal one
int ni_inited[IF_CNT]... R, 0 if NI is not initialized, $\neq$ otherwise

ni_send

Prototype:
INT32 ni_send(struct nisend *param)

Description:
If the message is not larger than the maximum transmission unit, it is handed to the manager layer for transmission. Therefore, ni_send first allocates a transmit buffer and then copies the data from the NI internal message structure into this buffer. In NI, each message is kept scattered in a list of so-called message blocks (m_blks). Since the pNA dependent interface is used, the driver is responsible for copying the data. After sending, the packet buffer has to be released automatically.

Parameters:
nisend *param... parameters for sending a message
char *hwa_ptr... address of destination HW address
char *buff_addr... address of message buffer
long count... length of message buffer
long type... type of the message, NI1 for an ARP, NI2 for an IP message

Return:
INT32... error code, is 0 if everything is OK, or
INVALID_IF_NUMBER if the parameter was invalid
ENOBUFS if the message could not be transmitted
(defined in error.h)

Callers:
pNA$^+$

Calls:
trans_ni... compiles the message and sends it

Global Variables:
confx *conf... R, structure containing cofiguration parameters of the NTI-driver
int ifnummap[MAXIF]... R, table for mapping pNA$^+$'s interface number to internal
int ni_inited[IF_CNT]... R, 0 if NI is not initialized, $\neq$ otherwise

SSCMP.C

get_no

Prototype:
char get_no(addr ad)

Description:
Gets an Ethernet hardware address and returns the number of the node with that address. The node number is equal to the least significant byte in the Ethernet hardware address.

Parameters:
addr ad... Ethernet address of the node in question

Return:
char... node number

Callers:
sscmp_announce... checks received message, acknowledges and announces it
put... partly compiles message, enters it into transmission protocol data structure
forward_rpk... shares oout incoming messages between the interfaces

Calls:
memcompp... returns 0 if two memory areas contain identical data

Global Variables:
none

init_sscmp

Prototype:
void init_sscmp()

Description:
Initializes the transmit protocol data structure.

Parameters:
none

Return:
void

Callers:
ki_init... initialization of KI part of the NTI-driver

Calls:
nothing

Global Variables:
awx *first_node[MAX_NODES], *last_node[MAX_NODES]... W, pointers to first (last) queue elements for storing the data of not yet acknowledged messages. These messages are organized in queues, see section 3.3.1. There are two pointers for each node.
waitx *to_node[MAX_NODES]... W, contains index of last acknowledged message from each node, the actual sequence number for messages to each node and the last sequence number received in a reply message, also for each node.

put

Prototype:
int put(bufx *buf,addr da,INT16 size)

Description:
First an envelope is allocated for the message, then the number of the node receiving it determined. Then a new queue entry is created, containing pointers to envelope and buffer, a sequence number (1 Byte) increased by *put*, a 4 Byte-timestamp updated by *ki_time* on each clock tick, a flag saying that the message is under transmission and a counter for the number of times, the message was resend (set to 0). The entry

is then inserted into the end of the appropriate queue. These queues are handled by *watchdog*, which will resend the message if the actual timestamp differs too much from the timestamp stored by *put* in the queue entry. Then the sequence number and the receiver node number are put into the SSCMP protocol header and the message is handed to *trans_ki*. *put* manipulates the transmit protocol structure and therefore could come into conflict with other instances of itself or with *watchdog*. The parts where race conditions with sscmp_announce or watchdog could arise are secured with *disable* and *enable*.

Parameters:

bufx *buf... address of the message buffer
addr da... destination hardware address
INT16 size... size of the message buffer

Return:

int... error code, is 0 if everything is OK, or
NO_SPACE_IN_WINDOW if the protocol structure is full
NO_ENV_FREE if no envelope was available
INVALID_BUF_ADDRESS if there does not exist a buffer with this address
INVALID_ENV_ADDRESS if there does not exist an envelope with this address

Callers:

ki_send... for sending a kernel message

Calls:

disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
alloc_ki_tenv... allocates an envelope
get_no... determines the number of a node with a given address
memcopp... copies specified amount of data from one memory area into another
disable... sets interrupt mask to 6
enable... sets interrupt mask to previous value
trans_ki... transmits a message
ret_ki_tbuf... releases transmit buffer
ret_ki_tenv... releases transmit envelope

Global Variables:

awx *first_node[j], *last_node[j]... RW, pointers to first (last) queue element of queue j (receiver node j) for storing the data of not yet acknowledged messages.
INT32 PSOS_FLAG... R, 0 if pSOS^{+m} is not active, $\neq 0$ otherwise

release_sscmp

Prototype:

void release_sscmp(envx *env)

Description:

Unsets the "message under transmission" flag in the queue entry for the message. If this message was already acknowledged, the entry is deleted and the queue updated.

Parameters:

envx *env... address of the envelope of the message

Return:

void

Callers:
release_cb_up... releases executed commands

Calls:
nothing

Global Variables:
awx *first_node[j]... RW, pointer to first queue element of queue j (receiver node j) for storing the data of not yet acknowledged messages.

sscmp_error_announce

Prototype:
void sscmp_error_announce(envx *env,bufx *buf)

Description:
Relays a transmit error condition to the highest layer.

Parameters:
envx *env... address of the envelope of the faulty message
bufx *buf... address of the buffer of the faulty message

Return:
void

Callers:
trans_fatal_error... transmit error handling procedure

Calls:
ki_announce_error_up... never returning error handling procedure

Global Variables:
none

sscmp_announce

Prototype:
void sscmp_announce(renvx *env,bufx *buf,INT16 size,addr *sa)

Description:
Via the source address the number of the source node is determined. If the message is an acknowledge message (determined by a field in the SSCMP header) it is handled as follows:
All messages waiting in the queue to the node with address *sa* with a lower or equal sequence number than the acknowledgment number found in the SSCMP header of the received reply message are removed from the queue if possible, or else, marked as acknowledged. They cannot be removed if the "message under transmission" flag is set, which means, that the message was not released by the transmit part of the driver yet. If it is a data message, its sequence number is checked. If it is identical to the expected number, the message is written into the queue of waiting messages and the function announcing it called. The envelope of the message is returned.
This queue of waiting messages is in reality an array. Two indizes, writ and out mark those slots where something can be written into respectively taken out of this array. A value of -1 means that there is no place to add a new buffer address respectively that no message is waiting to be received.

Parameters:

renvx *env... address of the receive envelope
bufx *buf... address of the buffer
INT16 size... size of the message buffer
addr *sa... Ethernet hardware address of the source of the message

Return:

void

Callers:

forward_rpk... distributes incoming messages between the interfaces

Calls:

disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
get_no... determine the number of a node with a given address
ret_ki_rbuf... return receive buffer space
ret_ki_renv... return a receive envelope
alloc_ki_tbuf... allocate transmit buffer
alloc_ki_tenv... allocate a transmit envelope
memcopp... copy a given number of bytes from ane memory area to another
ret_ki_tmess... return a whole transmit packet
trans_ki... transmit a message
ki_announce_up... announces a packet to the kernel

Global Variables:

awx *first_node[j]... RW, pointer to first queue element of queue j (receiver node j) for storing the data of not yet acknowledged messages.
INT32 kiq[KQ_LENGTH]... W, for storing addresses of not yet received messages
int writ... RW, index where the next address will be written to in kiq.
If it's -1 there is no space left
int out... RW, index, where the address waiting longest for receive is stored in kiq. If it's -1 there is no outstanding message.

watchdog

Prototype:

void watchdog()

Description:

This function is called at a multiple of the pSoS system tick. It compares all timestamps of waiting messages (messages not acknowledged yet). Therefore the queue entries have to be inspected starting with the oldest ones:
If the timestamps differ too much from the actual timestamp, the messages are resent. The counter of resends (i.e. the field repeats) is increased for each resent message. If it reaches MAX, *watchdog* calls *trans_fatal_error*. Since the function works on the protocol data structure, it may conflict with *put* or other instances of itself. Therefore it is secured by raising the interrupt mask.

Parameters:

none

Return:

void

Callers:
ki_time... called every tick

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
ret_ki_tmess... returns a whole message
ki_announce_error_up... reports a transmit error condition
trans_ki... transmits a message

Global Variables:
awx *first_node[MAX_NODE]... RW, pointers to first queue elements of queues 1 to MAX_NODE for storing the data of not yet acknowledged messages.

ERROR.C

trans_fatal_error

Prototype:
void trans_fatal_error(envx *env,bufx *buf,INT16 size,int if_num)

Description:
Determines the interface that tried to send this message and relays the addresses of envelope and buffer there.

Parameters:
envx *env... address of the envelope
bufx *buf... address of the buffer
INT16 size... size of the buffer
int if_num... number of the interface the messages was to be sent from

Return:
void

Callers:
release_cb... checks executed commands

Calls:
sscmp_error_announce... error handler of transmission protocol
ci_announce_error_up... error handler of CI

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
int ActiveAs[IFCNT]... tells the number of virtual nodes on an interface

receive_fatal_error

Prototype:
void receive_fatal_error(int error,int if_num)

Description:
increases the counter of the error reported by error

Parameters:
int error... encodes the error that happened:
NO_BUF_FREE if the RU ran out of buffers

NO_ENV_FREE if the RU ran out of receive frames
other values mean other errors

Return:
void

Callers:
rcv_error... relays errors to error handler

Calls:
nothing

Global Variables:
int ran_out_of_bufs[IFCNT]... W, counter of buffer errors
int ran_out_of_envs[IFCNT]... W, counter of envelope errors
int other_errors[IFCNT]... W, counter for other errors

fatal_error

This function just increases an error counter and can be entered into the pSOS^{+m} configuration table.

The function has no parameters and return value

asynchronous_error

This function just increases an error counter and can be entered into the PNA configuration table.

The function has no parameters and return value

APPLTR.C

alloc_ci_tbuf

Prototype:
int alloc_ci_tbuf(INT16 size,bufx **buf,int if_num,int area)

Description:
Allocates contiguous chain of buffer descriptors and their buffers. The address of the first buffer of this chain is written to the appointed place. If there is not enough buffer space, the number of messages discarded due to resource problems is increased and an error returned. The address of the buffer is set to NULL.

Parameters:
INT16 size... required size of the buffer
bufx **buf... address where to write the address of the buffer
int if_num... number of the interface
int area... number of the virtual node

Return:
int... error code, is 0 if everything is OK, or
NO_BUF_FREE if not enough buffer was available

Callers:
ci_get_tbuf... called by clock synchronization to allocate buffer

Calls:
get_tbd... searches contiguous chain of buffers to accomodate the required number of bytes

Global Variables:
int OutDiscards[IFCNT]... W, number of discarded outbound messages

alloc_ci_tenv

Prototype:
int alloc_ci_tenv(int mode,tenvx **tenv,int if_num,int area)

Description:
Depending on mode the function takes either a normal or a timestamped envelope from the pool and writes its address in the appointed place. If there is no envelope available, the number of messages discarded due to resource problems is increased and an error returned. The address of the envelope is set to NULL.

Parameters:
int mode... TS_PACKET means a timestamped envelope is required,
another value a normal one
tenvx **tenv... address where to write the address of the envelope
int if_num... number of the interface
int area... number of the virtual node

Return:
int... error code, is 0 if everything is OK, or
NO_ENV_FREE if no envelope was available

Callers:
ci_get_tenv... function called by clock synchronization to allocate envelope

Calls:
get_tcb... returns a normal transmit envelope
get_ts_tcb... returns a timestamped transmit command

Global Variables:
int OutDiscards[IFCNT]... W, number of discarded outbound messages

alloc_ki_tbuf

Prototype:
int alloc_ki_tbuf(INT16 size,bufx **buf)

Description:
Takes contiguous chain of buffer descriptors and their buffers from the pool and writes the address of the first buffer to the appointed place. No interface and virtual node are given, since pSOS^{+m} does not support more than one interface and especially not virtual nodes. If there is not enough buffer space available, the number of messages discarded due to resource problems is increased and an error returned. The address of the buffer is set to NULL.

Parameters:
INT16 size... required size of the buffer space
bufx **buf... address where to write the address of the buffer

Return:
int... error code, is 0 if everything is OK, or
NO_BUF_FREE if not enough buffer was available

Callers:
ki_get_pkb... called by kernel to allocate buffer
ki_broadcast... splits kernel broadcast request into singlecasts
sscmp_announce... checks received messages, acknowledges and announces them

Calls:
get_tbd... gets chain of buffers to accomodate required amount of data

Global Variables:
int OutDiscards[IFCNT]... W, number of discarded outbound messages

alloc_ki_tenv

Prototype:
int alloc_ki_tenv(envx **env)

Description:
Allocates normal transmit envelope and writes its address to the appointed place. If no envelope is available it increases the number of messages discarded due to resource problems and returns an error. The address of the envelope is set to NULL. No interface and virtual node are given, since pSOS^{+m} does not support more than one interface and especially not virtual nodes.

Parameters:
envx **env... address where to write the address of the envelope

Return:
int... error code, is 0 if everything is OK, or
NO_ENV_FREE if no envelope was available

Callers:
sscmp_announce... checks received kernel messages, acknowledges and
announces them
put... enters a message into the transmit protocol structure

Calls:
get_tcb... takes a transmit envelope from the list of free ones

Global Variables:
int OutDiscards[IFCNT]... W, number of discarded outbound messages

alloc_ni_tbuf

Prototype:
int alloc_ni_tbuf(INT16 size,bufx **buf,int if_num)

Description:
Allocates contiguous chain of buffer descriptors and their buffers to accommodate the required amount of data and writes the address of the first buffer in the appointed place. If there is not enough buffer available, the number of messages discarded due to resource problems is increased and an error returned. The address of the buffer is set to NULL.
Only the interface number is specified, since pNA^{+} does not know anything of virtual nodes.

Parameters:

INT16 size... required size of the buffer
bufx **buf... address where to write the address of the buffer
int if_num... number of the interface

Return:

int... error code, is 0 if everything is OK, or
NO_BUF_FREE if not enough buffer was available

Callers:

ni_getpkb... called by pNA$^+$ to allocate buffer space

Calls:

get_tbd... gets contiguous chain of buffer descriptors

Global Variables:

int OutDiscards[IFCNT]... W, number of discarded outbound messages

trans_ci

Prototype:

int trans_ci(int mode,tenvx *tenv,bufx *buf,INT16 size,int if_num,int area)

Description:

First it is determined, if the envelope is of timestamped mode. If this is true, the area field is set and a buffer has to be allocated. Then transm is called for compiling the message. The mode parameter defines whether the message is high priority and whether it should be returned immediately after sending. It is entered into the field packet_info of the envelope for processing of the command after it has been executed. Depending on whether the message is high priority or low, different commands for inserting it into the queue are called.

Parameters:

int mode... defines handling of the message: if the
PRIOR flag is set, it has high priority, if
RET_PKB_IMM is set, it is to be returned immediately after execution
tenvx *tenv... address of the envelope
bufx *buf... address of the buffer
INT16 size... size of the buffer
int if_num... number of the interface the message should go through
int area... number of the virtual node the message comes from

Return:

int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if tenv is not the address of an existing envelope or NULL
BUFSIZE_TO_SMALL buffer is smaller than transmit packet size given in size
INVALID_BUF_ADDRESS if buf is not the address of an existing buffer

Callers:

ci_send... called by the clock synchronization to send a message

Calls:

transm ... Compiles envelope and updates counters

put_cb... inserts a normal command into the queue
put_cb_urgent... inserts a high priority command into the queue
ucast... determines if an address is a singlecast address

Global Variables:
INT32 net_outbytes_CI[IFCNT]... W, number of bytes sent on CI
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

trans_ki

Prototype:
int trans_ki(envx *env,bufx *buf,INT16 size)

Description:
Prepares the buffer for KI transmission (additional bytes for control information) and calls transm to compile the envelope. Then the message is put to the outbound queue.

Parameters:
envx *env... address of the envelope
bufx *buf... address of the buffer
INT16 size... size of the buffer

Return:
int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if tenv is not the address of an existing envelope or NULL
BUFSIZE_TO_SMALL buffer is smaller than transmit packet size given in size
INVALID_BUF_ADDRESS if buf is not the address of an existing buffer

Callers:
watchdog... handles releases and retransmits, counts down timers
sscmp_announce... checks received kernel messages, acknowledges and announces them
put... enters a message into the transmit protocol structure

Calls:
transm ... Compiles envelope and updates counters
put_cb... inserts a normal command into the queue
ucast... determines if an address is a singlecast address

Global Variables:
int net_outbytes_KI[IFCNT]... W, number of bytes sent on KI
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

trans_ni

Prototype:
int trans_ni(addr da,int type,bufx *buf,INT16 size,int if_num)

Description:
Allocates a TCB and then calls transm to compile the envelope. Then it hands the envelope to the lower layer.

Parameters:

addr da... destination hardware address
int type... NI1 for an ARP message or NI2 for an IP message
bufx *buf... address of buffer
INT16 size... size of data buffer
int if_num... number of the interface the message is to go out through

Return:

int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if tenv is not the address of an existing envelope or NULL
BUFSIZE_TO_SMALL buffer is smaller than transmit packet size given in size
INVALID_BUF_ADDRESS if buf is not the address of an existing buffer

Callers:

ni_send... called by pNA$^+$ to transmit a message
ni_broadcast... called by pNA$^+$ to broadcast a message

Calls:

transm ... Compiles envelope and updates counters.
get_tcb... takes a TCB from the list of free ones
put_cb... inserts a normal command into the queue
ucast... determines whether an address is a singlecast address

Global Variables:

INT32 net_outbytes_NI[IFCNT]... W, number of bytes sent on NI

transm

Prototype:

int transm(tcbx *trans,bufx *buf,INT16 size,INT32 *netbytes)

Description:

This function compiles a transmit message. Some fields in the TCB and the TBDs have to be set, especially the size fields. If any of the parameters is invalid, nothing further is done but returning an error.
If destination address is the address of a node in the system, the number of singlecast messages is increased, otherwise the number of multi- or broadcast messages is raised by 1.
If the size of the buffer is so small, that the outgoing packet would be smaller than the minimum packet size, the size field in the packet header is updated.

Parameters:

addr da... destination hardware address
int type... NI1 for an ARP message or NI2 for an IP message
bufx *buf... address of buffer
INT16 size... size of data buffer
int if_num... number of the interface the message is to go out through

Return:

int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if tenv is not the address of an existing envelope or NULL
BUFSIZE_TO_SMALL buffer is smaller than transmit packet size given in size
INVALID_BUF_ADDRESS if buf is not the address of an existing buffer

Callers:

ni_send... called by pNA$^+$ to transmit a message
ni_broadcast... called by pNA$^+$ to broadcast a message

Calls:

get_tcb... takes a TCB from the list of free ones
put_cb... inserts a normal command into the queue
ucast... determines whether an address is a singlecast address

Global Variables:

int OutDiscards[IFCNT]... W, number of discarded outbound messages
int OutUcast[IFCNT]... W, number of sent singlecast messages
int OutNucast[IFCNT]... W, number of sent multi- or broadcast messages
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

release_cb

Prototype:

void release_cb(cbx *cb,int if_num)

Description:

Receives an executed command and processes it further.The treatment of transmit commands is dependent on the sending interface, all other commands are released.
If an error occurred that made transmission impossible, the concerned TCB is handed to the error handler. On return from the error handler, packet and envelope are returned. DMA underrun errors are treated in a special way, since they sometimes occur on NTI hardware: the error handler is not called and a counter is increased.
Kernel messages are released if they are acknowledge messages of the protocol (identifiable by the value SSCMP_PACKET standing in the field packet_info of the TCB), otherwise a pointer to the envelope is handed to the protocol.
pNA$^+$ messages are released.
Clock synchronization messages are treated according to the field packet_info. If the flag RET_PKB_IMM is set, the whole message is released, otherwise nothing is done.

Parameters:

cbx *cb... pointer to the command block
int if_num... number of the interface that executed the command

Return:

void

Callers:

release_cb_up... handles the command lists, called by ISR

Calls:

memclr... clears a memory area defined by address and size

trans_fatal_error... fatal error handler of transmit commands
ret_ci_tbuf... returns CI transmit buffer
ret_ki_tmess... returns kernel transmit packet
release_sscmp... clears flag marking message as being sent
ret_rbd... returns chain of RBDs and buffers
ret_tbd... returns chain of TBDs and buffers
ret_tcb... returns TCB

Global Variables:
int OutErrors[IFCNT]... W, number of messages not sent due to errors
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_ci_tbuf

Prototype:
int ret_ci_tbuf(bufx *buf)

Description:
Determines whether the given address is a receive or a transmit buffer (since receive buffers may be reused for transmission), calculates the address of the descriptor to that buffer and calls the apropriate function for returning it. If the buffer address does not belong to any of the pools, an error is returned.

Parameters:
bufx *buf... address of the buffer

Return:
int... error code, is 0 if everything is OK, or
INVALID_BUF_ADDRESS if the buffer does not belong to any of the pools

Callers:
ci_ret_rbuf... called by clock synchronization to return receive buffers
release_cb... processes executed commands

Calls:
ret_rbd... returns chain of RBDs and buffers
ret_tbd... returns chain of TBDs and buffers

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
int ActiveAs[IFCNT]... tells the number of virtual nodes on an interface

ret_ci_tenv

Prototype:
int ret_ci_tenv(envx *env)

Description:
Determines the type of the envelope by checking into which shared memory area it belongs. If it does not belong to any an error is returned. Otherwise the appropriate function is called to release the envelope.

Parameters:
envx *env... address of the envelope

Return:
int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if the envelope does not belong to any of the pools

Callers:
ci_ret_tenv... called by clock synchronization to free transmit envelope
release_cb... processes executed commands

Calls:
ret_tcb... returns TCB
ret_ts_tcb... returns TSTCB

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
int ActiveAs[IFCNT]... tells the number of virtual nodes on an interface

ret_ki_tbuf

Prototype:
int ret_ki_tbuf(bufx *buf)

Description:
Calculates the address of the buffer descriptor belonging to the given buffer (after checking it, of course) and calls the appropriate function to release it. If buf does not belong to any pool, an error is returned.

Parameters:
bufx *buf... address of the buffer

Return:
int... error code, is 0 if everything is OK, or
INVALID_BUF_ADDRESS if the buffer does not belong to any of the pools

Callers:
put... enters message into protocol data structure

Calls:
ret_rbd... returns a chain of RBDs and buffers
ret_tbd... returns a chain of TBDs and buffers

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_ki_tenv

Prototype:
int ret_ki_tenv(envx *env)

Description:
Checks if the envelope belongs to the pool. If not, an error is returned, otherwise the envelope is released.

Parameters:
envx *env... address of the envelope

Return:
int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if the envelope does not belong to any of the pools

Callers:
put... enters a message into the protocol data structure

Calls:
ret_tcb... returns a TCB by appending it to the list of free ones

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_ki_tmess

Prototype:
int ret_ki_tmess(envx *env)

Description:
Checks the validity of the address of the BD entered in the TCB. If it is valid, its type (receive or transmit) is determined and the appropriate function for releasing it called. If the address of the envelope is valid, it is released as well. If either is invalid, an error is returned.

Parameters:
envx *env... address of the envelope

Return:
int... error code, is 0 if everything is OK, or
INVALID_BUF_ADDRESS if the buffer does not belong to any of the pools
INVALID_ENV_ADDRESS if the envelope does not belong to any of the pools

Callers:
watchdog... handles releases, retransmits, counts down timers
sscmp_announce... checks received kernel messages, acknowledges and announces them
put... enters a message into the protocol data structure
release_cb... processes executed commands

Calls:
ret_rbd... returns chain of RBDs and buffers
ret_tbd... returns chain of TBDs and buffers
ret_tcb... returns TCB by appending it to the list of free ones

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

APPLREC.C

forward_rpk

Prototype:
void forward_rpk(rfdx *rfd,INT16 bsize,int if_num,INT16 norbds,INT16 contigous)

Description:
Gets the recently filled RFDs taken from the list of free RFDs and shares them out between the application interfaces. On each call, one RFD is handed up together with the interface number, its size, the number of RBDs used and a flag, which is set if the frame is not contigous. If pSoS is up and the option was set, *forward_rpk* is called by *t_forward_rpk* and therefore runs in its own task context, not in the ISR context. This is not true for NI messages, which are always forwarded directly (within the ISR). For all KI- and CI- frames, which are not contigous, *copy_rbd* is called to render them

contigous.
NI messages always have to be copied into the pNA$^+$ internal message structure: Therefore, a message block is allocated from the pNA$^+$, the data is copied and the message block internal pointers are updated. Instead of a buffer pointer, the message block pointer is forwarded up to the pNA$^+$.
Messages are only forwarded to the intended application interface if that has been initialized and is therefore active, otherwise they are discarded.

Parameters:
rfdx *rfd... address of the frame of the received message
INT16 bsize... size of the data buffer
int if_num... number of the network interface the message came in through INT16 norbds... number of RBDs used by frame INT16 contigous... flag, which is set if frame is not contigous

Return:
void

Callers:
forward_rpk_up... handles receive data structure lists t_forward_rpk... own task, decoupled from ISR

Calls:
memcompp... returns 0 if the contents of two memory areas are identical
ret_rbd... releases RBD and buffer
ret_rfd... releases RFD
copy_rbd... renders buffer contiguous
sscmp_announce... checks received kernel messages, acknowledges and announces them
ci_announce_up... forwards incoming messages to the clock synchronization
ni_announce_up... forwards incoming messages to pNA$^+$
copy_block ... copies a number of bytes from the shared memory RBs into a destination buffer
get_no... determines the number of a node with a given address

Global Variables:
int InUcast[IFCNT]... W, number of received singlecast messages
int inNucast[IFCNT]... W, numbe of received multi- or broadcast messages
int InDiscards[IFCNT]... W, number of inbound messages discarded due to resource problems
int UnknownProtos[IFCNT]... W, number of received messages with unknown higher protocol (neither CI nor KI nor NI)
int ki_inited... R, 0 if KI part of the driver has not been initialized yet
int ni_inited[IFCNT]... R, 0 if NI part of the driver has not been initialized yet
int ci_inited[IFCNT]... R, 0 if CI part of the driver has not been initialized yet

t_forward_rpk

Prototype:
void t_forward_rpk()

Description:
The only purpose of this function is it, to split the sometimes time-consuming task

of handing messages up into two parts: The lower part (including *lance_isr0* and *forward_rpk_up*), which runs in the ISR context, and the upper part, which starts with *forward_rpk* and can include many functions like *copy_rbd*, *sscmp_announce*, ni_announce_up and others. This part runs in the supervisory task context of *t_forward_rpk*. The decoupling is done via a message queue: *forward_rpk_up* puts the parameters for *forward_rpk* into a queue and sends a message to *t_forward_rpk*, which calls *forward_rpk* with the message values. This is only done if pSoS is active and the option is set, otherwise task switching and message passing is not available and *forward_rpk_up* calls *forward_rpk* directly.

Parameters:
None

Return:
void

Callers:
None

Calls:
forward_rpk... dispatches contigous messages between the interfaces

ki_get_rbuf

Prototype:
int ki_get_rbuf(bufx **buf)

Description:
If there is a message waiting for reception, it is taken out of the array and its address written to the appointed place. If no message is waiting, NULL is written to that place.

Parameters:
bufx **buf... place to write the address of the buffer to

Return:
int... always 0

Callers:
ki_receive... called by pSOS^{+m} to fetch an announced message

Calls:
disable... sets interrupt mask to 6
enable... sets interrupt mask to previous value

Global Variables:
INT32 kiq[KQ_LENGTH]... RW, stores addresses of not yet received messages
int writ... RW, index where the next address will be written to in kiq.
If it's -1 there is no space left
int out... RW, index, where the address waiting longest for receive is stored in kiq. If it's -1 there is no outstanding message.

poll_ni

Prototype:
int poll_ni(int if_num)

Description:
Calls *forward_rpk_up* to simulate the receivement of packets. This function is called by NI if pSOS^{+m} is not active and therefore no interrupts can be handled. In order to make the call atomic the interrupt mask is raised prior to calling it and lowered afterwards.

Parameters:
int if_num... number of the network interface to be polled

Return:
int... always 0

Callers:
ni_poll... called by pNA^{+} to poll for messages

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
lance_isr... handles the interrupt conditions

Global Variables:
none

rcv_error

Prototype:
void rcv_error(int error,int if_num)

Description:
relays a receive error report to the error handler

Parameters:
int error... encoded error
int if_num... number of the network interface with the error

Return:
int... always 0

Callers:
forward_rpk_up... handles lists of receive data structures

Calls:
rcv_fatal_error_handler... logs the error

Global Variables:
none

ret_ci_rbuf

Prototype:
int ci_ret_rbuf(bufx *buf)

Description:
Checks the validity of the buffer address, calculates the address of the corresponding RBD and releases it.

Parameters:
bufx *buf... address of the buffer

Return:
int... error code, is 0 if everything is OK, or
INVALID_BUF_ADDRESS if buffer does not belong to the pool

Callers:
ci_ret_rbuf... called by clock synchronization to return receive buffer
ci_mcon... called by clock synchronization to make buffer contiguous
ci_announce_up... forwards received messages to clock synchronization

Calls:
ret_rbd... releases chain of RBDs and buffers

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_ci_renv

Prototype:
int ret_ci_renv(renvx *renv)

Description:
Checks validity of a receive frame address and releases the RFD if it has a correct address.

Parameters:
renvx *renv... address of receive frame

Return:
int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if envelope does not belong to the pool

Callers:
ci_ret_renv... called by clock synchronization to release a receive envelope
ci_announce_up... forwards received messages to clock synchronization

Calls:
ret_rfd... releases RFD by appending it to the list of free ones

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_ki_rbuf

Prototype:
int ret_ki_rbuf(bufx *buf)

Description:
Checks the validity of the buffer address. If it is correct, it calculates the address of the corresponding RBD and releases it.

Parameters:
bufx *buf... address of the receive buffer

Return:
int... error code, is 0 if everything is OK, or
INVALID_BUF_ADDRESS if buffer does not belong to the pool

Callers:
ki_returnp... called by pSOS^{+m} to release receive buffer
sscmp_announce... checks received kernel message, acknowledges and announces it

Calls:
ret_rbd... releases chain of RBDs and buffers

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_ki_renv

Prototype:
int ret_ki_renv (renvx *renv)

Description:
Checks the validity of the receive frame and releases it if the address is correct.

Parameters:
renvx *renv... address of the receive frame

Return:
int... error code, is 0 if everything is OK, or
INVALID_ENV_ADDRESS if envelope does not belong to the pool

Callers:
sscmp_announce... checks received kernel message, acknowledges and announces it

Calls:
ret_rfd... releases RFD

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

LANCETR.C

do_something_else

Prototype:
void do_something_else(int turns)

Description:
This function does nothing but count from turns down to zero. It serves for passing time via a busy wait loop, since *tm_wkafter()* is a library function and may not be called from the NTI-driver for reasons mentioned in section 1 of this chapter. If turn is negative, the function still counts to zero.

Parameters:
int turns... the number to count down from

Return:
void

Callers:
init_lance... initializes hardware
lance_isr... handles interrupt conditions
put_cb... appends a command to the command block list
put_cb_urgent... inserts a command as far to the front of the CBL as possible
copy_rbd... renders an incontiguous chain of RBDs contiguous
ret_rbd... releases a chain of RBDs and buffers

Calls:
nothing

Global Variables:
none

tbd_index

Prototype:
INT32 tbdindex(tbdx *tbd,int area,int if_num)

Description:
This function calculates the index for a TBD (transmit buffer descriptor) by its address in the shared memory. The index is the number of this TBD in the array of all TBDs in the shared memory.

Parameters:
tbdx *tbd... the shared memory address of the TBD
int area... the shared memory area (virtual node)
int if_num... the network interface number

Return:
INT32... the index

Callers:
ret_tbd... returns TBD chains to the common pool.

Calls:
None

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

tb_index

Prototype:
INT32 tbindex(bufx *buf,int area,int if_num)

Description:
This function calculates the index for a TB (transmit buffer) by its address in the shared memory. The index is the number of this TB in the array of all TBs in the shared memory.

Parameters:
bufx *buf... the shared memory address of the TB
int area... the shared memory area (virtual node)
int if_num... the network interface number

Return:
INT32... the index

Callers:
ret_tbd... returns TBD chains to the common pool.

Calls:
None

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

get_tbd

Prototype:
tbdx *get_tbd(INT16 size,int if_num,int area)

Description:
Returns the address of the first TBD of a contiguous chain of free ones (large enough to accommodate the requested number of bytes) which is done by searching a free chain large enough in the free list for the required interface and then updating the free list, see section 3.5.2. for a detailed description.
There may arise race conditions between different instances of this function. It is possible that two *get_tbd* allocate overlapping chains of TBDs. To avoid that, the function is secured by raising the interrupt mask, so that it is partially atomic.

Parameters:
INT16 size... the requested buffer size
int if_num ... number of the network interface
int area... number of the virtual node

Return:
tbdx *... address of the first TBD in the chain or NULL if none were available

Callers:
alloc_ki_tbuf... allocates buffer for kernel message
alloc_ni_tbuf... allocates buffer for pNA$^+$ message
alloc_ci_tbuf... allocates buffer for clock synchronisation message

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
t_tfreelist *tfree[MAXASICS][IFCNT]... free list for managing TBD pool.

get_tcb

Prototype:
tcbx *get_tcb(int if_num)

Description:
Returns a pointer to a free TBD taken from the TCBL.
Race conditions may arise between multiple instances of *get_tcb* and also with the function *ret_tcb*. The former may result in doubly reserved TCBs, the latter in lost ones that have not been reinserted into the TCBL. Therefore the function is secured by raising the interrupt mask.

Parameters:
int if_num... number of the network interface

Return:
tcbx *... address of TCB; or NULL, if none was available.

Callers:
alloc_ki_tenv... allocates envelope for kernel message
alloc_ci_tenv... allocates envelope for clock synchronization message
trans_ni... transmits pNA$^+$ message

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value

Global Variables:
tcbx *tcbl_begin[IFCNT]... RW, address of TCBL
tcbx *tcbl_end[IFCNT]... RW, last entry in TCBL

get_ts_tcb

Prototype:
ts_tcbx *get_ts_tcb(int in,int area)

Description:
Returns the address of a free TSTCB taken from the TSTCBL.
Race conditions may arise between multiple instances of *get_ts_tcb* and also between *get_ts_tcb* and *ret_ts_tcb*, the former resulting in possible double allocations of TSTCBs and the latter the loss of TSTCBs. Therefore the function is secured by raising the interrupt mask.

Parameters:
int area... number of the virtual node
int if_num... number of the network interface

Return:
tcbx *... address of the TSTCB or NULL, if none is available

Callers:
alloc_ci_tenv... allocates envelope for clock synchronization message

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value

Global Variables:
ts_tcbx *ts_tcbl_begin[MAXASICS][IFCNT]... RW, address of TSTCBL
ts_tcbx *ts_tcbl_end[MAXASICS][IFCNT]... RW, address of last entry in TSTCBL

init_cfg

Prototype:
void init_cfg(INT8 cfga[CFG_CNT],int if_num)

Description:
Writes the given data into the CFG command block of network interface if_num.

Parameters:
INT8 cfga[CFG_CNT]... bytes for configuration of LANCE
int if_num... the required network interface

Return:
void

Callers:
init_lance... initializes hardware

Calls:
nothing

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

init_ias

Prototype:
void init_ias(addr ich,int if_num)

Description:
Initializes the specified IAS command block using the given individual address.

Parameters:
addr ich... Ethernet hardware address of the interface
int if_num... number of the network interface

Return:
void

Callers:
init_lance... initializes hardware

Calls:
nothing

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

init_lance

Prototype:
int init_lance(int if_num)

Description:
Initializes the initialization root, the hardware and the command blocks remaining after the intialization of the transmit and the receive data structures. It tells the LANCE the address where the initialization root can be found and configures it according to the data in the configuration structure. The RU is given the address of the receive frame area and started.
Apart from configuring the LANCE this function has to write the interrupt control register and the interrupt base register for the LANCE on the MVME162. The vector base register is set thus, that the interrupt triggered by the LANCE has the offset 0x07 from the value presumably already written into the interrupt base vector register. The interrupt is enabled in the interrupt control register for the LANCE, set to high polarity and edge sensitive. Due to the unknown state of the wires the bit for clearing the interrupt is set as well before resetting and initializing the LANCE.
If the memory used as communication between LANCE and CPU is situated in RAM, the snoop bit in the bus control register must be set to circumvent problems arising out of the use of cache. It must not be set if the memory is situated on the NTI and thus accessed via VME-Bus.

Parameters:
int if_num... number of the network interface

Return:
int... error code, is 0 if everything is OK, or
NETWORK_NOT_FUNCTIONAL if a hardware error was detected

Callers:
init... called by an application interface to initialize the hardware

Calls:
do_something_else... passes time
init_cfg... initializes CFG command block
init_ias... initializes IAS command block
init_mcs... initializes MCS command block
init_scb... initializes initialization root and remaining command blocks

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
INT32 *NTITS[MAXASICS]... R, address of timestamp register
INT32 *NTIRFD[MAXASICS]... R, address of RFD-address register
INT32 *NTICR[MAXASICS]... R, address of NTI interrupt control register
INT32 *licr[IFCNT]... RW, address of LANCE interrupt control registers on board MVME162
confx *conf[IFCNT]... R, address of configuration record
INT16 *port[IFCNT]... W, address of register mapped to PORT
INT32 *ca[IFCNT]... W, address of register mapped to CA pin

init_mcs

Prototype:
void init_mcs(int adrc,valx *adrl,int if_num)

Description:
Initializes the MCS command block specified by if_num with the addresses contained in adrl.

Parameters:
int adrc... number of addresses in adrl
valx *adrl... list of multicast addresses
int if_num... identifies the network interface

Return:
void

Callers:
init_lance... initializes hardware

Calls:
nothing

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

init_tbd

Prototype:
void init_tbd(tbdx *tbd)

Description:
Initializes TBD and its buffer. The pointers to the next TBD and to the buffer are cleared.

Parameters:
tbdx *tbd... address of the TBD

Return:
void

Callers:
init_trans... initializes transmit part of the shared memory
ret_tbd... releases chain of TBDs and buffers

Calls:
memclr... clears memory area specified by address and size

Global Variables:
none

init_tcb

Prototype:
void init_tcb(tcbx *tcb)

Description:
Initializes TCB. In the command word the flags are set that identify the TCB as a transmit command, tell the LANCE to raise an interrupt after executing this block and that it uses the flexible memory model. Since there is no list of TBDs attached to the TCB yet, tbd_add is set to EMPTY. Eof_tcbcount is set to TADDSIZE, so that length_field and da will be taken from the TCB upon transmission. In short, as many values as possible are set during initialization, what cannot be set are the values dependent on the individual message.

Parameters:
tcbx *tcb... address of the TCB

Return:
void

Callers:
init_trans... initializes transmit part of the shared memory
ret_tcb... returns TCB

Calls:
memclr... clears memory area specified by address and size

Global Variables:
none

init_ts_tcb

Prototype:
void init_ts_tcb(ts_tcbx *ts_tcb)

Description:
Initializes TSTCB. In the command word the flags are set that identify the block as a transmit command, tell the LANCE to raise an interrupt after executing it and that it uses the flexible memory model.Since there is no list of TBDs attached to the TSTCB yet, tbd_add is set to EMPTY. Eof_tcbcount is set to TSADDSIZE, so that length_field, da area and tst will be taken from the TCB upon transmission. In short, as many values as possible are set during initialization, what cannot be set are the values dependent on the individual message.

Parameters:
ts_tcbx *ts_tcb... address of the TSTCB

Return:
void

Callers:
init_trans... initializes transmit part of shared memory
ret_ts_tcb... returns TSTCB

Calls:
memclr... clears memory area specified by address and size

Global Variables:
none

init_scb

Prototype:
void init_scb(int if_num)

Description:
Initializes the initialization root and the remaining command blocks (TDR,NOP,DIAG,DUMP).

Parameters:
int if_num... number of the network interface

Return:
void

Callers:
init_lance... initializes the hardware

Calls:
nothing

Global Variables:
sh_memx smem[MAXASICS][IFCNT]...R, addresses of shared memory areas

init_trans

Prototype:
void init_trans(int if_num)

Description:
Initializes transmit part of the shared memory. The TCBL and the TS_TCBL are constructed and the pointers to their beginnings and ends set. The beginnings and ends of the command list, the urgent command list and the alternative command list are set to NULL.

Parameters:
int if_num... number of the network interface

Return:
int... always 0

Callers:
init... called by an application interface to initialize the hardware

Calls:

init_tbd... initializes TBD and its buffer
init_ts_tcb... initializes TSTCB
init_tcb... initializes TCB

Global Variables:

sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
INT8 alloc_t[TB_CNT][MAXASICS][IFCNT]... W, allocation table
int begin_alloc[MAXASICS][IFCNT]... W, starting point for search in the allocation table
ts_tcbx *ts_tcbl_begin[IFCNT]... W, address of first entry in the TSTCBL
ts_tcbx *ts_tcbl_end[IFCNT]... W, address of last entry in the TSTCBL
tcbx *tcbl_begin[MAXASICS][IFCNT]... W, address of first entry in the TCBL
tcbx *tcbl_end[MAXASICS][IFCNT]... W, address of last entry in the TCBL
cbx *cl_begin[IFCNT]... W, address of first entry in the command block list
cbx *cl_end[IFCNT]... W, address of last entry in the command block list
cbx *cl_alt_begin[IFCNT]... W, address of the first entry in the alternative command block list
cbx *cl_alt_end[IFCNT]... W, address of the last entry in the alternative command block list
cbx *cl_urg_begin[IFCNT]... W, address of the first entry in the urgent command block list
cbx *cl_urg_end[IFCNT]... W, address of the last entry in the urgent command block list
int ActiveAs[IFCNT]... tells the number of virtual nodes on an interface

put_cb

Prototype:

int put_cb(cbx *cb)

Description:

Gets a command block to hand to the LANCE. It tries to append it to the current CBL. If that is not possible, because the LANCE has already reached the end of that CBL, it appends it to the alternative CBL.

The function handles and changes the pointers to beginnings and ends of the command block list and the alternative command block list. Conflicts may arise with the functions *put_cb_urgent* and *forward_rpk_up*, which also manipulate those pointers. To avoid misdirections, etc, the function has been secured by raising the interrupt mask.

Parameters:

cbx *cb... address of the command block
int if_num... number of the network interface

Return:

int... always 0

Callers:

trans_ki... transmits a kernel message
trans_ni... transmits a pNA$^+$ message
trans_ci... transmits a clock synchronization message

Calls:

do_something_else... passes time

disable... sets interrupt level to 5
enable... sets interrupt level to previous value

Global Variables:

cbx *cl_begin[IFCNT]... RW, address of first entry in the command block list
cbx *cl_end[IFCNT]... RW, address of last entry in the command block list
cbx *cl_alt_begin[IFCNT]... RW, address of first entry in the alternative command block list
cbx *cl_alt_end[IFCNT]... RW, address of last entry in the alternative command block list
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
int OutQlen[IFCNT]... RW, number of messages waiting for transmission

put_cb_urgent

Prototype:

int put_cb_urgent(cbx *cb,int if_num)

Description:

Gets an urgent command block to hand to the LANCE. It tries to insert it as far to the front of the current CBL as possible. If the CB cannot be inserted, because the LANCE has progressed to far in its execution of the CBL, it is appended to the urgent command list (UCBL).
The function handles and changes the pointers to beginnings and ends of the CBL and the UCBL. There can arise conflicts with the functions *put_cb* and *forward_rpk_up*. To avoid them the function has been secured by raising the interrupt mask.

Parameters:

cbx *cb... address of the command block
int if_num... number of the network interface

Return:

int... always 0

Callers:

trans_ci... transmits a clock synchronization message

Calls:

do_something_else... passes time
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value

Global Variables:

cbx *cl_begin[IFCNT]... RW, address of first entry in the command block list
cbx *cl_end[IFCNT]... RW, address of last entry in the command block list
cbx *cl_urg_begin[IFCNT]... RW, address of first entry in the urgent command block list
cbx *cl_urg_end[IFCNT]... RW, address of last entry in the urgent command block list
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
int OutQlen[IFCNT]... RW, number of messages waiting for transmission

release_cb_up

Prototype:
void release_cb_up(int if_num,INT16 state)

Description:
Is called by the ISR to process the CBL. Depending on the state of the CU it either just hands the completed commands up after taking them from the list, or it additionally tries to generate a new list out of the rest of the old one, the alternative and the urgent list and restarts the CU with it.
The function handles and changes the pointers to beginnings and ends of the command block list the alternative command block list and the urgent command block list. There can arise conflicts with the functions *put_cb_urgent* and *put_cb*. To avoid them, the interrupt level of the ISR has been fixed at 6.

Parameters:
int if_num... number of the network interface

Return:
void

Callers:
lance_isr... ISR of LANCE interrupt

Calls:
release_cb... processes the executed commands

Global Variables:
cbx *cl_begin[IFCNT]... RW, address of the first entry in the command block list
cbx *cl_end[IFCNT]... RW, address of the last entry in the command block list
cbx *cl_alt_begin[IFCNT]... RW, address of the first entry in the alternative command block list
cbx *cl_alt_end[IFCNT]... RW, address of the last entry in the alternative command block list
cbx *cl_urg_begin[IFCNT]... RW, address of the first entry in the urgent command block list
cbx *cl_urg_end[IFCNT]... RW, address of the last entry in the urgent command block list
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
int OutQlen[IFCNT]... RW, number of messages waiting for transmission

ret_tcb

Prototype:
int ret_tcb(tcbx *tcb,int if_num)

Description:
Initializes a TCB and returns it to the TCBL. Race conditions may arise between multiple instances of *ret_tcb* and also with the function *get_tcb*, resulting in lost TCBs. Therefore the function is secured by raising the interrupt mask.

Parameters:
tcbx *tcb... pointer to the TCB
int if_num... number of the network interface

Return:
int... always 0

Callers:
release_cb... processes executed commands
ret_ki_tmess... returns whole kernel message
ret_ki_tenv... returns envelope of kernel message
ret_ci_tenv... returns envelope of clock synchronization message

Calls:
init_tcb... initializes TCB
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value

Global Variables:
tcbx *tcbl_begin[MAXASICS][IFCNT]... RW, address of first entry in TCBL
tcbx *tcbl_end[MAXASICS][IFCNT]... RW, address of last entry in TCBL

ret_tbd

Prototype:
int ret_tbd(tbdx *tbd,int if_num,int area)

Description:
Tbd is the address for a contigous chain of TBDs. Each of them, together with its buffer, is initialized and returned to the pool by updating the free list for the required interface, see section 3.5.2 for more detail. Since some instances of *ret_rbd* may be called concurrently, the global data structures have to be secured by raising the interrupt mask.

Parameters:
tbdx *tbd... address of the chain of TBDs
int area... number of the virtual node
int if_num... number of the network interface

Return:
int... always 0

Callers:
release_cb... processes executed commands
ret_ki_tmess... returns whole kernel messages
ret_ki_tbuf... returns kernel transmit buffers
ret_ci_tbuf... returns clock synchronization messages

Calls:
init_tbd... initializes TBD and its buffer

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
t_tfreelist *tfree[MAXASICS][IFCNT]... free list for managing TBD pool.

ret_ts_tcb

Prototype:
int ret_ts_tcb(ts_tcbx *ts_tcb,int if_num,int area)

Description:
Initializes the TSTCB and returns it to the TSTCBL.
Race conditions may arise between multiple instances of *ret_ts_tcb* and also with the function *get_ts_tcb* resulting in lost TSTCBs. Therefore the function is secured by raising the interrupt mask.

Parameters:
int area... number of the virtual node
ts_tcbx *ts_tcb... address of the TSTCB
int if_num... number of the network interface

Return:
int... always 0

Callers:
ret_ci_tenv... external, returns envelope of clock synchronisation message

Calls:
init_ts_tcb... initializes TSTCB disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value

Global Variables:
ts_tcbx *ts_tcbl_begin[MAXASICS][IFCNT]... RW, address of first entry in TSTCBL
ts_tcbx *ts_tcbl_end[MAXASICS][IFCNT]... RW, address of last entry in TSTCBL

ucast

Prototype:
int ucast(addr ad,int if_num)

Description:
The given address is compared one by one with all addresses in the net. If it is identical with one TRUE is returned, FALSE else.

Parameters:
addr *ad... the address in question
int if_num... number of the network interface

Return:
int... TRUE if the address is a unicast address, FALSE else

Callers:
trans_ki... transmits a kernel message
trans_ni... transmits a pNA$^+$ message
trans_ci... transmits a clock synchronization message

Calls:
memcompp... compares the contents of two memory areas

Global Variables:
confx *conf..., R, address of the configuration structure

LANCEREC.C

rbd_index

Prototype:
INT32 rbdindex(rbdx *rbd,int area,int if_num)

Description:
This function calculates the index for a RBD (receive buffer descriptor) by its address in the shared memory. The index is the number of this RBD in the array of all RBDs in the shared memory.

Parameters:
rbdx *rbd... the shared memory address of the RBD
int area... the shared memory area (virtual node)
int if_num... the network interface number

Return:
INT32... the index

Callers:
ret_rbd_nc ... partitions not contigous chains of RBDs into contigous ones and calls *ret_rbd* ret_rbd ... returns contigous chains of RBDs to the common pool copy_rbd ... copies not contigous chains of RBDs into contigous ones forward_rpk_up ... does some receive packet processing and forwards them

Calls:
None

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

rb_index

Prototype:
INT32 rbindex(bufx *buf,int area,int if_num)

Description:
This function calculates the index for a receive buffer by its address in the shared memory. The index is the number of this RB in the array of all RBs in the shared memory.

Parameters:
bufx *buf... the shared memory address of the RB
int area... the shared memory area (virtual node)
int if_num... the network interface number

Return:
INT32... the index

Callers:
ret_rbd ... returns contigous chains of RBDs to the common pool

Calls:
None

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

copy_block

Prototype:
void copy_block(rbdx *rbd,void *buf,int if_num, int no_rbs);

This function follows a chain of RBDs (contigous or incontigous) and copies the associated RBs into a contigous destination buffer.

Parameters:
rbdx *rbd... address of the first RBD of the chain (contigous/incontigous)
int if_num... number of the network interface
void *buf ... pointer to the destination buffer
int no_rbs ... number of RBDs in chain to be copied

Callers:
forward_rpk... processes the received messages

Calls:
memcopp ... copies a number of bytes in a fast way

copy_rbd

Prototype:
rbdx *copy_rbd(rbdx *rbd,int if_num,int no_rbs);

This function copies the data from an incontiguous buffer into a contiguous one. It is only called by forward_rpk() and only, if the buffer is not contigous. The address of the new chain of RBDs is returned to the caller, the original chain is released. Since *copy_rbd* can take some time (up to 6 ms), the need of disabling interrupts during complete function execution time is avoided: Firstly a space large enough to hold the whole frame is seeked. Then the fields in this space are marked as filled and the pointers of the list containing the free buffers are redirected (look at the found space and the redirected pointers in the picture). Concurrency problems with the LANCE have to be considered here. Such problems may arise, if the LANCE is writing to the free space found, before the pointers can be redirected. Then the interrupts are turned on again and the buffers are copied into the new space, which is reserved now and cannot be touched by the LANCE any more. This part takes most of the function's execution time.

Parameters:
rbdx *rbd... address of the first RBD of the incontiguous chain
int if_num... number of the network interface
int no_rbs ... number of RBDs in chain to be copied

Return:
rbdx *... address of the contiguous chain, or NULL if no copying was possible

Callers:
forward_rpk... processes the received messages

Calls:
ret_rbd... returns RBDs
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value

Global Variables:

rbdx *rbdl_begin[IFCNT]... R, address of first enrty in RBDL
rbdx *rbdl_end[IFCNT]... RW, address of last entry in RBDL
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
INT8 free_slot[NORBS] ... RW, array of flags indicating free/used RBDs

forward_rpk_up

Prototype:

void forward_rpk_up(int if_num,INT16 state)

Description:

this function is the ISR for the LANCE receive unit. It takes the received packets, i.e., the RFDs and the associated RBDs from the list and hands them up to *forward_rpk*, which is responsible for processing and distributing them. The packets are handed up directly (by calling *forward_rpk*) or by sending a message to the task FRPK, implemented in *forward_rpk_stub*, which itself calls *forward_rpk*. This is done for decoupling the forwarding process, which can contain calls to *copy_rbd*, but also protocol management and complete resends of messages for the Kernel communication, from the ISR. Therefore the ISR execution time is reduced significantly. Clearly, RFDL_BEGIN points to the first received message. The FREE_SLOTS array is updated: FREE_SLOTS contains a flag for each RBD and the RBDs used for the received frames are marked as "full". Furthermore the total number of RBDs, the size of the frame and if it is contigous are determined. Then the packet is handed up to the manager layer together with this information in the way described above. RFDL_BEGIN is set to the next frame in the list and the whole procedure started again, until either the end of the RFDL is reached or a frame that is not (yet) full. Finally the status of the receive unit is checked. If it has stopped due to resource problems, it is provided with a new RFA, or, should this prove to be impossible, an error message is generated.

Parameters:

int if_num... number of the network interface
INT16 state... the status word of the SCB

Return:

void

Callers:

lance_isr0 ... handles interrupt condidtion

Calls:

rcv_error... error handler
forward_rpk... shares out received messages between the application interfaces

Global Variables:

rbdx *rbdl_begin[IFCNT]... RW, address of first entry in RBDL
rbdx *rbdl_end[IFCNT]... RW, address of last entry in RBDL
rfdx *rfdl_begin[IFCNT]... RW, address of first entry in RFDL
rfdx *rfdl_end[IFCNT]... RW, address of last entry in RFDL
INT8 free_slot[NORBS]... W, array of flags indicating free/used RBDs int InBytes[IFCNT]... W, number of received bytes
int InErrors[IFCNT]... W, number of errors during receive
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

init_rbd

Prototype:
void init_rbd(rbdx *rbd)

Description:
Initializes the RBD by setting everything to 0 but the addresses of the next RBD and of the buffer, which are left alone, and el_pref_size, where the size of a buffer (in bytes) is entered. Thus the RBD is prepared in such a way, that it and its buffer will be filled correctly by the LANCE upon reception of a message.

Parameters:
rbdx *rbd... address of the RBD

Return:
void

Callers:
init_receive... initializes receive part of shared memory
ret_rbd... returns chain of RBDs and buffers

Calls:
memclr... clears a memory area specified by address and size

Global Variables:
none

init_receive

Prototype:
int init_receive(int if_num)

Description:
Initializes receive data structures and sets the pointers to the first entries of the RBDL and the RFDL. Everything is prepared, so that the receive area is ready to be handed over to the LANCE.

Parameters:
int if_num... number of the network interface

Return:
int... always 0

Callers:
init... called for initialization of hardware

Calls:
init_rbd... initializes RBD and its buffer
init_rfd... initializes RFD

Global Variables:
rbdx *rbdl_begin[IFCNT]... RW, address of first entry in RBDL
rbdx *rbdl_end[IFCNT]... W, address of last entry in RBDL
rfdx *rfdl_begin[IFCNT]... W, address of first entry in RFDL
rfdx *rfdl_end[IFCNT]... W, address of last entry in RFDL
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
INT8 free_slot[NORBS]... W, array of flags indicating free/used RBDs

init_rfd

Prototype:
void init_rfd(rfdx *rfd)

Description:
In the command word the flag is set that tells the LANCE that it should use the flexible memory model. The pointer to the RBDs is set to EMPTY and the field size is set to RADDSIZE so that the length field, the source and the destination address are taken from the buffer and entered into the frame upon reception. All other fields are set to zero. The RFD is thus prepared so that it will be interpreted and filled correctly by the LANCE upon reception of a message.

Parameters:
rfdx *rfd... address of the RFD

Return:
void... always 0

Callers:
init_receive... initializes receive part of shared memory
ret_rfd... returns RFD

Calls:
memclr... clears memory area defined by address and size

Global Variables:
none

ret_rbd_nc

Prototype:
int ret_rbd_nc(rbdx *rbd,int if_num)

Description:
This function takes a not contigous list of RBDs, portions this list into contigous blocks of RBDs and hands these blocks to ret_rbd one after another.

Parameters:
rbdx *rbd ... address of the RBD starting the not contiguous list.
int if_num ... number of the requested network interface

Return:
None

Callers:
copy_rbd ... copies not contigous chains of RBDs into contigous ones

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
ret_rbd ... returns a contigous chain of RBD to a common pool

Global Variables:
INT8 free_slot[NORBS] ... R, array of flags indicating free/used RBDs

ret_rbd

Prototype:
int ret_rbd(rbdx *rbd,int if_num)

Description:
this function receives a chain of contigous RBDs to return to the list of free RBDs. After reinitialization it inserts them into the list according to the algorithm described in section 3.5.3.
The function manipulates the ringbuffer containing the RBDs and the global array free_slot[NORBS]. It is possible, that *ret_rbd* is called from different contexts concurrently and also an interrupt could occur during execution. Therefore it has to be secured by raising the interrupt mask.

Parameters:
int if_num... number of the network interface

Return:
int... always 0

Callers:
ret_ki_tmess... returns a whole pSOS^{+m} transmit message
ret_ki_tbuf... returns a pSOS^{+m} transmit buffer
ret_ci_tbuf... returns a clock synchronization transmit buffer
release_cb... processes executed commands
forward_rpk... shares out received messages between the applications
ret_ki_rbuf... called to return a pSOS^{+m} receive buffer
ret_ci_rbuf... called to return a clock synchronization receive buffer
ret_ni_rbuf... called to return a pNA^{+} receive buffer

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
init_rbd... initializes RBD

Global Variables:
INT8 free_slot[NORBS] ... RW, array of flags indicating free/used RBDs sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_daemon

Prototype:
void ret_daemon()

Description:
This function is called periodically by *ki_timer* and tries to return chains of RBDs associated with frames, that could not be returned by calling *ret_rbd* due to conflicts with the LANCE.
These RBD chains are kept in lists until it is possible to return them.

Parameters:
None

Return:
None

Callers:
ki_timer ... activated on each pSoS clock tick

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
mfree... returns a fixed size block to the memory management
ret_rbd ... returns contigous chains of RBDs to a common pool

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
do_init[IFCNT]... R, do_init[i] set, if network interface i is up
t_tfreelist rbd_rest[IFCNT]... First list element pointers. The list elements contain pointers to not yet returned chains of RBDs

ret_rfd

Prototype:
int ret_rfd(rfdx *rfd,int if_num)

Description:
Returns an RFD to the RFDL by appending it at the end. If the last RFD is already prefetched or filled by the LANCE (as can be determined by the status word of the RFD) there is no point in appending the RFD to the RFDL. It is instead appended to the ARFDL. If the RFDL is empty and the ARFDL is not, the ARFDL is tranferred to the RFDL and the RU is started again with the new list. The function manipulates the RFDL. Therefore it has to be secured against other instances of itself, which is accomplished by raising the interrupt mask.

Parameters:
rfdx *rfd... address of RFD
int if_num... number of the network interface

Return:
int... always 0

Callers:
forward_rpk... shares out received messages between the applications
ret_ki_renv... returns a pSOS^{+m} receive envelope
ret_ci_renv... returns a clock synchronization receive envelope

Calls:
disable... sets interrupt mask to 5
enable... sets interrupt mask to previous value
init_rfd... initializes an RFD

Global Variables:
rfdx *rfdl_begin[IFCNT]... RW, address of first entry in RFDL
rfdx *rfdl_end[IFCNT]... RW, address of last entry in RFDL
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

INTERF.C

lance_isr0

Prototype:
void lance_isr0()

Description:
This function is the actual ISR servicing the LANCE interrupts. First the interrupt bit on the board is cleared, so that no interrupt is lost if they come at high frequency. Next the interrupt is acknowledged to the LANCE. Then it is dispatched: If the interrupt was raised by the CU, *release_cb_up* is called, if it was raised by the RU, *forward_rpk_up*. They handle the data structures and, if necessary, restart the LANCE.
Mind that the IRQ levels of multiple LANCEs must be the same.

Parameters:
none

Return:
INT32... PSOS_FLAG, which is set if pSoS is up

Callers:
lance_drv... assembler function serving as ISR wrapper

Calls:
do_something_else... passes time
release_cb_up... handles command block lists
forward_rpk_up... handles RFDL and RBDL

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

init

Prototype:
int init(int ifc,int if_num)

Description:
If the hardware and the data structures of the required network interface have not been initialized yet, the following steps are taken. First a function is called which returns the addresses of the active memory areas (i.e., virtual nodes) on this interface. Then those areas and the LANCE are initialized.
The array where the waiting kernel messages are kept is initialized if the calling interface is the KI. The initialization of the hardware must not take place more than once. Therefore the reading and writing of do_init has been secured by raising the interrupt mask. Other instances of *init* have to wait.

Parameters:
int ifc... ID of the application interface requiring initialization
int if_num... number of the hardware interface that is to be initialized

Return:
int... error code, 0 if everything is OK, or
INVALID_IF_NUMBER if there is no hardware interface with the given number

NETWORK_NOT_FUNCTIONAL if a hardware error was detected INVALID_PROTO if ifc is none of the constants KI, NI or CI. error value from task or queue creation system calls, if they were not successful

Callers:

ci_init... initializing function called by clock synchronization
ni_init... initializing function called by pNA$^+$
ki_init... initializing function called by pSOS^{+m}

Calls:

init_trans... initializes transmit data structures
init_receive... initializes receive data structures
init_lance... initializes hardware
disable... sets interrupt mask to 6
enable... sets interrupt mask to previous value
SysSetVector... sets interrupt vector for LANCE on MVME 162
nti_init... returns addresses for shared memory areas
q_create... creates queue
t_create... creates task
t_start... starts task
getEtherAddress0... returns ethernet address from MVME 162
intlanceon0... enables LANCE interrupts on MVME-162

Global Variables:

char do_init[IFCNT]... RW, 0 if this network interface has not been intialized, $\neq 0$ else
int ki_inited... RW, 0 if KI part of the driver has not been initialized yet
int ni_inited[IFCNT]... RW, 0 if NI part of the driver has not been initialized yet
int ci_inited[IFCNT]... RW, 0 if CI part of the driver has not been initialized yet
INT32 kiq[KQ_LENGTH]... W, for storing addresses of not yet received messages
int ActiveAs[IFCNT]... W, number of active memory areas on this interface
ifnummap[IFCNT]... W, maps NI interface numbers to the driver's interface numbers conf... RW, complete configuration structure for LANCE

LanStop

Prototype:

void LanStop()

Description:

The function disables LANCE reception on all interfaces. It is required, if the LANCE controlled by the NTI driver acts as a network controller for the pRobe connection between host and target. Then it can be necessary to stop the LANCE and start another network driver.

Parameters:

None

Return:

None

Callers:

???

Calls:
None

Global Variables:
do_init[IFCNT] ... flag, do_init[i] set, if network interface i is up

getEtherAddress0

Prototype:
void GetEtherAddr0(addr *ich)

Description:
Copies the ethernet address permanently stored on the MVME-162 board to the RAM address, ich is pointing to.

Parameters:
addr *ich... a reference for keeping the ethernet address

Return:
None

Callers:
init... initializes LANCE itself and calls other init-functions

Calls:
None

Global Variables:
None

intlanceon0

Prototype:
void intlanceon0()

Description:
This function enables all required Lance-Interrupts on MVME-162.

Parameters:
None

Return:
None

Callers:
init... initializes LANCE itself and calls other init-functions

Calls:
None

Global Variables:
None

LIBFUNCTIONS.C

memcompp

Prototype:
int memcompp(void *a,void *b,int size)

Description:
Compares size bytes beginning from the address a with size bytes beginning from the address b. If all are identical, 0 is returned. At the first different byte the comparison is stopped and -1 returned. The function is optimated for 32 bit memory accesses.

Parameters:
void *a... address in memory
void *b... address in memory
int size... number of bytes to compare

Return:
int... 0 if the contents of the area are identical, -1 otherwise

Callers:
forward_rpk... processes executed commands
ucast... determines whether a given address is singlecast

Calls:
nothing

Global Variables:
none

memclr

Prototype:
void memclr(char *ptr,int size)

Description:
Clears the specified number of bytes starting at the given address. The function is optimated for 32 bit memory accesses.

Parameters:
char *ptr... address of the to be cleared area
int size... number of bytes to clear

Return:
void

Callers:
release_cb... processes the executed commands
init_tcb... initializes a TCB
init_ts_tcb... initializes a TSTCB
init_mem... initializes memory for memory managment functions
init_rfd... initializes an RFD

Calls:
nothing

Global Variables:
none

memcopp

Prototype:
int memcopp(void *a,void *b,int no)

Description:
Copies no bytes from the source to the destination in memory. This function is optimized for 32 bit memory accesses.

Parameters:
void *a... points to the destination void *b... points to the source int no... number of bytes to copy

Return:
None

Callers:
getEtherAddress0... gets ethernet address from MVME-162 ki_send... calls put for sending messages to a destination node ni_ioctl... handles ioctl calls by the network interface (NI) sscmp_announce... handles received messages for the KI and acknowledgments put... sends messages via the reliable sliding window protocol

Calls:
None

Global Variables:
None

mmalloc

Prototype:
void * malloc(int caller)

Description:
This function searches for a memory block from the common pool (array), marks it as used and returns it to the caller.

Parameters:
int caller... for debugging reasons, each caller has to identify himself by an index

Return:
void *... a pointer to a fixed size block is returned to the caller

Callers:
ret_rbd... returns a contigous chain of RBDs to a common pool ret_tbd... returns a contigous chain of TBDs to a common pool put... put... sends messages via the reliable sliding window protocol init_trans... initializes transmit part of LANCE

Calls:
None

Global Variables:
entry[MAX_BLOCKS][MAX_SIZE]... contains an array of fixed size memory blocks (pool for memory management) full... counts the number of blocks in use

mfree

Prototype:
void mfree(void * ret)

Description:
This function returns a block to the memory management pool (an array of fixed size blocks).

Parameters:
void *ret... a pointer to the fixed size block, which is returned

Return:
None

Callers:
get_tbd... allocates a contigous chain of TBDs from a common pool ret_tbd... returns a contigous chain of TBDs to a common pool ret_daemon... periodically tries to return RBD chains sscmp_announce... handles KI receive messages and acknowledgments

Calls:
None

Global Variables:
entry[MAX_BLOCKS][MAX_SIZE]... contains an array of fixed size memory blocks (pool for memory management) full... counts the number of blocks in use

mem_init

Prototype:
void mem_init()

Description:
This functions initializes the whole space reserved for the memory mangement so that it can be used by mmalloc and mfree.

Parameters:
None

Return:
None

Callers:
init_trans... initialzes transmit part of LANCE

Calls:
memclr... clears a memory portion to zero

Global Variables:
entry[MAX_BLOCKS][MAX_SIZE]... contains an array of fixed size memory blocks (pool for memory management)

DISPATCHER.C

ni_dispatcher

Prototype:
INT32 ni_dispatcher(INT32 function_code, INT32 param_pointer)

Description:
The address of this function has been entered into the table of interfaces pNA$^+$ keeps. pNA$^+$ therefore calls it whenever it wants something of the interface with a function code and the pointer to a paramter block as a parameter (on the stack. All the functions supported by the NTI-driver for pNA$^+$ are listed in a table according to their function codes (e.g. *ni_initial* with function code 1 stands in the first place). The dispatcher calls the functions by calling the function in the slot the code number

specifies. Again the pointer to the parameter block is given as a parameter. The return value of the called function is returned to pNA$^+$.

Parameters:
INT32 function_code... the number of the required function
INT32 param_pointer... the address of the parameter block, the format of which is different for each function

Return:
INT32... error code, which is faithfully handed up to pNA$^+$

Callers:
pNA$^+$

Calls:
ni_initial... initializes NTI-driver
ni_getpkb... allocates transmit message buffer
ni_retpkb... releases receive buffer
ni_send... transmits a message
ni_broadcast... broadcasts a message
ni_poll... polls for incoming messages
ni_ioctl... support MIB-II features

Global Variables:
none

ki_dispatcher

Prototype:
cannot be called directly, since the parameters have to be in registers

Description:
The address of this function is entered into the pSOS^{+m} multiprocessor configuration table as the entry point for the NTI-driver. It is therefore called, whenever pSOS^{+m} wants something of the NTI-driver. The registers D1, D2 and A0 contain the parameters for the different functions, D0 the function code. The semantics of the parameters differ, dependent on the requested service (or function). Again, a table has been bueilt where the functions are entered in the sequence of their code number. The function indicated by D0 is called. Its return value, standing in register D0, is left there as the return value of the dispatcher.

Parameters:
D1 register value, usually integer number or meaningless
D2 register value, usually integer number or meaningless
A0 register value, usually pointer or meaningless

Return:
INT32... in D0, error code

Callers:
pSOS^{+m}

Calls:
ki_init... initializes NTI-driver
ki_getpkb... allocates transmit message buffer
ki_returnp... releases receive buffer

ki_send... transmits message
ki_broadcast... broadcasts message, unused
ki_receive... fetches announced message
ki_dummy... this slot would have styed empty
ki_roster... for relaying roster information to NTI-driver

Global Variables:
none

LANCE.S

disable

Prototype:
INT32 disable()

Description:
This function sets the interrupt mask in the CPU status register to 5. This value has been chosen, since it is sufficiently high that clock interrupts cannot interrupt a program running with this level. Thus the functions are virtually atomic as long as the interrupt level is high. Level 6 was not acceptable, since the timestamping hardware triggers interrupts at level 6 and has absolute priority. The old value of the status register is returned to the caller, so that it may be reinstalled at the end of the critical section.

Parameters:
none

Return:
INT32... the old status register

Callers:
watchdog... handles releases and retransmits, counts down timers
ki_get_rbuf... fetches buffer from waiting area
poll_ni... calls ISR for polling
init... initializes hardware
get_ts_tcb... gest TSTCB
ret_ts_tcb... releases TSTCB
put_cb... appends command to queue
put_cb_urgent... inserts commands into queue
ret_tcb... releases TCB
get_tcb... gets TCB
get_tbd... gets transmit buffer
ret_rfd... releases RFD
ret_rbd... releases chain of RBDs and buffer
copy_rbd... renders chain of RBDs contiguous

Calls:
nothing

Global Variables:
none

enable

Prototype:
void enable(INT32 merken)

Description:
Sets the interrupt mask in the status register to the old value, taken from the copy of the status register handed down by the caller.

Parameters:
INT32 merken... old status register value

Return:
void

Callers:
watchdog... handles releases and retransmits, counts down timers
ki_get_rbuf... fetches buffer from waiting area
poll_ni... calls ISR for polling
init... initializes hardware
get_ts_tcb... gest TSTCB
ret_ts_tcb... releases TSTCB
put_cb... appends command to queue
put_cb_urgent... inserts commands into queue
ret_tcb... releases TCB
get_tcb... gets TCB
get_tbd... gets transmit buffer
ret_rfd... releases RFD
ret_rbd... releases chain of RBDs and buffer
copy_rbd... renders chain of RBDs contiguous
ki_receive... fetches announced message for the kernel

Calls:
nothing

Global Variables:
none

lancedrv

Prototype:
void lance_drv()

Description:
Is nothing but a wrapper for the actual ISR. It does the things pSOS^{+m} usually does when a ISR is called. The wrapper had to be rewritten since the NTI-driver may be active when pSOS^{+m} is not, so that for instance, a different return command is necessary. After return of the ISR, PSOS_FLAG is checked. If it is 0 (no pSOS^{+m}) RTE is used, otherwise trap #13, returning control to the scheduler.

Parameters:
none

Return:
void

Callers:
LANCE interrupt

Calls:
lance_isr... the actual ISR, handling the interrupt conditions

Global Variables:
INT32 PSOS_FLAG... R, 0 if pSOS^{+m} is not active, $\neq 0$ otherwise

mask_drop

Prototype:
INT32 mask_drop()

Description:
This function is sort of the complement to *disable*. It does not raise the interrupt mask, but drop it to zero, of course returning the old value of the status register to the caller. The function is needed, since otherwise the LANCE interrupts when receiving a message could not be handled during pSOS^{+m} startup, when the interrupt mask remains high at 7.

Parameters:
none

Return:
INT32... old value of the status register

Callers:
ki_receive... fetches announced message for kernel

Calls:
nothing

Global Variables:
none

Chapter 5

Usage Considerations

This final chapter discusses several issues that are important when using the NTI-driver in an application: Configuration, memory requirements and error messages are described in some detail and complemented by an overview of a sample testing application. A brief overview of how to debug NTI-driver applications and a list of shortcomings, pitfalls and related issues is appended for future enhancements.

5.1 Installation, Configuration and Usage

Running an application incorporating the NTI-driver assumes a properly installed and preconfigured hardware. Each node must be equipped with

- one MVME-162 CPU running the application (target CPU),
- one MEN A203 resp. AcQ i6360+EXT M-Module carrier board with at least one NTI plugged in slot #0 resp. #2,
- optionally: one gateway CPU acting as a TCP/IP-gateway from Ethernet to VMEbus shared memory

that are all plugged into a VMEbus rack. System, hardware and software must be configured as in usual pSOS^{+m} environments and under consideration of the specific requirements explained in the following subsections.

5.1.1 pSOS Configuration

As mentioned in Chapter 2, pSOS^{+m} and pNA^{+} are configured by setting up the respective configuration tables. The standard configuration setup of the pSOS system usually enters the values given in the application's *sys_conf.h* into the configuration tables. In the following subsections, we will discuss the setting of the entries that are relevant for the NTI-driver. Note that we will not discuss general constraints and limitations that apply when setting configuration table parameters.

Multiprocessor Configuration Table

void (*mc_kicode)() Address of the KI's entry point (*ki_dispatcher*)

unsigned long mc_nnode Number of nodes in the system. Must notbe larger than 16383 and the same at all nodes! Note that the NTI-driver uses its own #define for the maximum number of nodes supported, which should be chosen consistently.

unsigned long mc_nglbobj The maximum number of global objects. Must be the same at all nodes!

unsigned long mc_flags There are two flags that can be set: KIROSTER: If it is set, the NTI-drivers *ki_roster* service is called on changes of roster data. Since KI does not have to support broadcasting, i.e., does not need the information about the nodes present in the system, the state of this flag is of no moment to the NTI-driver. SEQWRAP: If it is 1 and the sequence number of a node trying to start up has the maximum allowable value, the number is wrapped around to 1. If it is 0, the node fails to start up. The value of this flag is of no moment to the NTI-driver.

unsigned long mc_kimaxbuf maximum length of a KI buffer. This parameter must be the same at all nodes. It must be set large enough to accomodate special requirements of certain pSOS^{+m} services like variable message queues, which may exceed the standard size of 100.

void (*mc_asyncerr)() asynchronous error callout. The NTI-driver provides a simple handler *asynchronous_error()* that may optionally be entered here.

pNA$^+$ Configuration Table

An network interface can be registered with pNA$^+$ either statically via the configuration table, or dynamically via the *add_ni()* service call. The capabilities of the NI are specified in a certain struct that will be described below under the *nc_ini* entry.

void *nc_data Specifies address of pNA$^+$'s data memory; if it is 0, a suitably sized segment is allocated from pSOS region 0.

long nc_datasize If pNA$^+$'s memory is not allocated from region 0, its size must be given here. It can be calculated according to formulas given in [IS96a, p.4-11]; note that the buffer configuration given in *nc_bcfg* below has to be considered here!

long nc_nni Maximum number of network interfaces even registered with pNA$^+$used for allocating the internal network interface table.

struct ni_init *nc_ini Initial network interface table for static registration of certain NIs, terminated by 0. For the NTI-driver, the *ni_init* looks as follows:

int (*entry)() the address of the entry point. The address entered here is that of the *ni_dispatcher*.

int ipadd The IP address of the interface. It should conform to the others in the network the interface belongs to. This IP address is individual for each network interface and therefore totally independent of IP addresses of other interfaces on this node.

int mtu The maximum transmission unit. This value tells pNA$^+$ how long a message it wants to send over this interface may at the most be. Since this number is specified as 1500 in an Ethernet, that value must be entered.

hwalen The length of a hardware address. In Ethernet an address is an array of 6 bytes, therefore 6 is the value to enter here.

flags The possible flags to enter are:
IFF_BROADCAST 0x01... the interface supports broadcasting
IFF_ARP 0x02... the interface does ARP by itself
IFF_POINTTOPOINT 0x10... the interface is a point-to-point interface
IFF_MULTICAST 0x800... the interface supports multicasting
IFF_UNNUMBERED 0x1000... the interface is unnumbered
IFF_RAWMEM 0x2000... the interface understands pNA$^+$ data structures
IFF_EXTLOOPBACK 0x4000... the interface loops back outgoing messages
IFF_POLL 0x8000... the interface has to be polled for incoming messages.
The flags that should be set in conjunction with the NTI-driver are IFF_BROADCAST, IFF_RAWMEM, and IFF_EXTLOOPBACK. Note that the current version does not support IFF_MULTICAST, since the required *ni_ioctrl()*-calls have not been implemented yet.

int subnetaddr Here the subnetmask can be entered.

int dstipaddr This field is for the destination IP address in a point-to-point connection. Since this NTI-driver is not of that type, the field can remain 0.

int reserved[1] is reserved for future use and set to 0 in the meantime.

struct pna_bufcfg *nc_bcfg Pointer to buffer configuration table, terminated by 0.

As the NTI-driver understands pNA$^+$'s message block triplets (IFF_RAWMEM), sufficiently many buffers of the most common lengths must be provided here for reasonable performance. The NTI-driver does not scatter received packets over linked message blocks, so contigous buffers of sufficient size must be available for successful reception. This means that at least MTU-sized buffers (as well as 0 and 128 byte one) must be present here!

pROBE$^+$ Configuration Table

If the NTI-driver's NI is used for debugging via XRAY$^+$(see Section 5.3), one has to ensure that pROBE$^+$ does not raise the interrupt mask above the level of the i82596 interrupt. This can be secured via a configuration table entry.

unsigned long rc_ilevel interrupt level while pROBE$^+$ has control. It has to be set to something less than 5 (the current NTI-driver's level of the i82596 interrupt).

5.1.2 NTI Configuration

The jumpers and configuration switches on the carrier board and the NTIs must be setup properly. At least one NTI ("area #0", recall Section 2.3) must be plugged into M-Module slot #0 at the MEN A203 resp. into slot #2 at the i6360/EXT. If additional NTIs are present in higher slots, their RxTS-input must be connected by some wiring according to Figure 2.5.

For the application software, a suitable *pcb_conf.h* describing the NTI setup for the NTI-Handler [SM99] must be provided, and the appropriate handler object module (*pcb_install_i6360.o* resp. *pcb_install_a203.o*) must be linked with the application.

5.1.3 Miscellaneous Configuration Issues

An important point for getting the NTI-driver to work is proper selection of the MVME-162's Ethernet address, which is programmed into the MVME's non-volatile RAM via the M162-BUG debugger's CNFG-command. The NTI-driver assumes that all nodes have an identical Ethernet address except of its last byte, which must be equal to the pSOS node number.

It must be secured that the pSOS tick interrupt cannot interrupt the i82596 ISR, i.e., has level at most 5. This must be done in the pSOS board support package of the MVME-162 (*m162_5xx*).

5.1.4 Usage

There are not many things to consider upon usage of the NTI-driver, since the only interface likely to be of interest to a programmer is the CI.

Clock Synchronization Algorithm

The source file containing the clock synchronization algorithm just has to include *ci.h* and call the CI-functions via the *ci_dispatcher*. The buffer management of the CI allows to re-use the buffers of a previously received packet as transmit buffers in a *ci_send* command.

Error Messages

All functions of all interfaces return error conditions via a non-zero return code. pSOS^{+m} and pNA^{+} treat errors differently, depending on context and circumstances: The error code may be relayed directly to the user, cloaked in another error code, or may cause pSOS^{+m} to stop altogether. Section 4.3 contains a list of all possible error codes, Section 4.5 provides details which function may return which error code.

5.1.5 Memory Requirements

There are some basic requirements for global variables, which depend on some constants and accumulate to 3 KB with the standard setting. 1719 Bytes of the 3 KB depend on the constant IFCNT, which is set to one. If the constant is increased, this amount has to be multiplicated. The the NTI-driver further requires memory for its memory management function *mmalloc*. This function is called to allocate memory for the reliabletransmission protocol queues (below KI), for free list managementof transmit buffers, and for the lists handled by*ret_daemon* (see Section 3.5.3). The total size is the product of *BLOCK_SIZE* and *MAX_BLOCKS*, both constants defined in *libfunction.h*. With the standard values for these constants, about 7.5 KB are used. Therefore, with the standard header files, exactly 10737 Bytes of global memory are required. Note that the detailed memory requirements can be found in the linker's .map-file (*vars* and *zerovars* section).

Note that if one of the error counters no_mem[MAXASICS] should ever go up during operation of the NTI-driver (and errors being generated), the MAX_BLOCKS value should be increased.

5.2 Sample Application

The NTI-driver has been tested and evaluated by means of a 4-node application that uses all of its interfaces (though for a single area only). It allows to measure and statistically evaluate the transmission delay uncertainty arising in the transmission of timestamped messages, subject to various NI and KI load conditions; see [SN99] and [SKM+99] for the results.

5.2.1 System Overview

Fig. 5.1 outlines the basic hardware architecture.

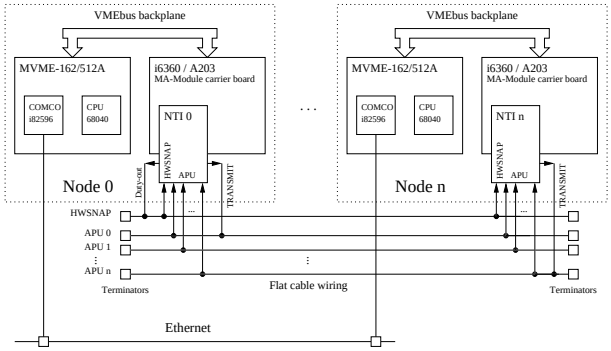


Figure 5.1: Hardware architecture of the evaluation system

Dedicated flat cable wiring is used for measuring the transmission delay between any two nodes in the system. More specifically, the TRANSMIT-signal (recall Section 2.3) triggering the transmit timestamp on node i's NTI is exported and fed to the application timestamp input A_i on each NTI in the system. This way, any node that receives a CSP from node i just has to compute the difference of the —locally available— timestamps RxTS$-A_i$TS to obtain the actual transmission delay. In addition, the HWSNAP-timestamp inputs of all NTIs are also tied together and driven by a duty-timer output of NTI 0. This enables the generation of simultaneous interrupts at all nodes, a feature required for initiating coordinated activity.

The application software consists of a number of concurrently executing tasks (the same at each node), which can be parameterized online by means of a custom configuration dialogue. Fig. 5.2 outlines its task map.

- The *CI-Client* is the major task of the evaluation system. It periodically broadcasts CSPs to the *CI-server* at any other node and collects the reply messages. After a specific number of rounds has been reached, the CI-client computes the transmission delay statistics of the messages collected so far and prints them on the network terminal. Among the configurable parameters are e.g. the number of rounds, broadcast period, CSP size, TxTS and RxTS trigger offsets, etc.

- Any pair of *NI-Send* and *NI-Receive* tasks at a remote node exchange data via a TCP/IP stream socket, thereby generating network traffic of configurable load and type.

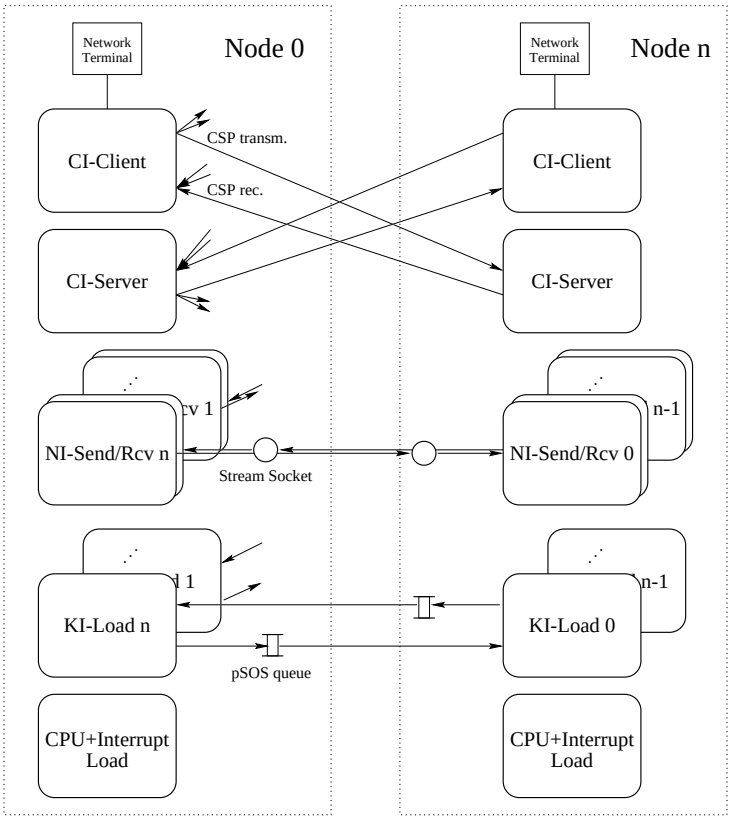


Figure 5.2: Software architecture of the evaluation system

- Similarily, any pair of *KI-Load* tasks at different nodes use two global pSOS^{+m}-queues to exchange data over this IPC mechanism.
- Finally, there is a task responsible for generating a parameterizeable CPU and/or interrupt load with optional accesses to NTI memory.

The testing application is provided in the subdirectory TESTAPPL of the NTI-driver and consists of the following files:

- *emu.h*: global definitions
- *new_appl.c*: application startup (*main()*) and configuration dialogue + NI-Send and NI-Receive task
- *new_pisos.c*: KI-Load task
- *ci_client.c*, *ci_server.c*: CI-client and CI-server task
- *ci_wrapper_eval.c*: special NTI-Handler for one-way transmission delay measurement
- *ci_tsdiff.s*: utility function for 64-bit subtraction of timestamps

In addition, there are 4 subdirectories (*node1* ... *node4*) that contain the following node-specific files:

- *sys_conf.h*: pSOS system configuration
- *sys_config.h*: netterm setup
- *drv_conf.c*: Device driver and network interface setup
- *root.c*: pSOS^{+m} ROOT-task
- *new_appl.h*: Application node-specific setup
- *makefile*: Building the node's executable

Some advices how to start and debug this application can be found in the README-files in the appropriate directories.

5.2.2 Performance Evaluation

A powerful tool was used for performance evaluation and execution tracing of the NTI-driver using the sample application, namely, Applied Microsystem's *CodeTEST*. It consists of a VME-bus data acquisition board linked via Ethernet to a Sun workstation running a visualization software. At compile time, the C-code is first instrumented with tags by means of a precompiler. For example, tags can be inserted upon any function's entry and exit points. All tags perform memory accesses to a certain address on the VME bus, which is monitored by CodeTEST; the tag events are timestamped and sent to the host for further processing.

The host software can work in trace mode (program traces at different levels: function entry/exit, control flow or even single statements can be listed) or in continous mode. In continous mode,

the max/min/average/total time spent in all instrumented functions, the total time spent in the different tasks, the function coverage and also statistics on memory allocation/deallocation can be viewed. During the tests, the tool was mainly used to get an overview over function performance (max time and total time) and function coverage. The maximum time spent in functions was mainly interesting for one reason: Many functions disable interrupts and if this happens too long, inacceptable delays for interrupt handling (for example, for the pSOS timer interrupt or the LANCE interrupt) can occur. The total (accumulated) time spent in functions is clearly the most important factor w.r.t. throughput.

As increasing throughput was one of the major goals of the NTI-driver redesign, CodeTEST was just the right tool to spot the "bad guys" and to eliminate them in the new version. The worst of them were:

- **do_something_else** This function implements a busy-wait loop. It was sometimes called with the loop index much too high and consumed 38% of CPU time.
- **memclr** Huge amounts of buffer memory were unnecessarily cleared.
- **memclr, memcopp, memcompp** These functions performed very badly due to byte-wise memory accesses instead of 32 bit accesses.
- **ret_rbd** Due to the complicated ring-buffer algorithm, this function was very slow.
- **incr, decr** These tiny functions were called so often that the function entry/exit consumed a lot of time.

5.3 Debugging

Debugging the NTI-driver and/or an application based upon it can be done in several ways. First of all, there are two different startup scenarios that can be considered:

- *Warmstart*

  In this case, the application is built seperately (nti_app.x) and uses the pSOS components in ROM. The application is downloaded via XRAY$^+$ and started via the RESTART-command, which passes control to a user-supplied procedure *boot()* that modifies (part of) the configuration tables of the ROMed OS according to the application's *sys_conf.h*. After that, pSOS^{+m} is started.

  Note that the usual board and component configuration code is not executed here. This implies that reconfiguration of certain pNA$^+$ configuration table entries, like the buffer configuration, does not work here since pROBE$^+$ and some basic functionality of pNA$^+$ is continously required for the communication with XRAY$^+$.

  The primary advantage of this method is faster download, since only the application has to be loaded. In conjunction with the NTI-driver, however, it has several deficits:

  – It can only be used if the XRAY$^+$ connects to the target CPU via shared memory, i.e., if a second gateway CPU is available. This is due to the fact that the NTI-driver would take over the target CPUs i82596, which is, however, needed for the debugging connection.

- Reconfiguring the pSOS components via *boot()* is limited.

- *Coldstart*

  In this case, application and pSOS are built together (nti_ram.x) and downloaded via XRAY$^+$. Execution of the downloaded system is started by the OSBOOT-command, which passes control to the usual board and component configuration code and finally to pROBE$^+$. The XRAY$^+$ on the host is waiting for pROBE$^+$ to come up again to reconnect with it, which eventually completes booting. When the RESTART-command is finally given here, it does not call *boot()* but immediately starts pSOS^{+m}as before.

  It is important to note that, if using the NTI-driver in the downloaded system, this cold startup mode implies operation of the NI before pSOS^{+m} is up. This rules out any pSOS^{+m} system call in the NI-related program paths and requires some care w.r.t. returning from ISRs.

  As far as debugging in this scenario is concerned, there are two possibilities:

  - In case of a single target CPU, there are only limited debugging capabilities for the NTI-driver itself. More specifically, a breakpoint in a thread related to the NI interface part (e.g., in the i82596 ISR) also breaks the debugging connection. However, debugging other (lower-priority!) ISRs or pSOS tasks does not cause problems.
  - If a second CPU acting as a gateway to shared memory is available, XRAY$^+$can connect to the target CPU via shared memory. This allows unrestricted debugging as in the warmstart case, with the additional advantage of full component customization. However, downloading takes much more time here.

  It should be noted, though, that the RESTART-commands does not cause the (initialized) variables of the application to be reinitialized. The *initvars*-function is in fact only called in *boot()*. Note that it might also be called in the ROOT-task, if no driver etc. running at the time of its invocation had variables in the *vars* and *zerovars* section. Unfortunately, this is not true for the NTI-driver. Hence, this method of re-initializing variables is not allowed in the single target CPU case above.

5.4 Shortcomings and Pitfalls

1. The current version of the NTI-driver does not support multicasting to the extent required for switching on IFF_MULTICAST (see Section 5.1.1). What is missing is the implementation of the *ni_ioctrl*-functions

   - SIOCADDMCAST
   - SIOCDELMCAST
   - SIOCMAPMCAST

   for multicast address management. Note that the internal data strucure required for storing multicast addresses resp. initial LANCE setup are already present (in the conficuration record confx conf[IFNUM] resp. the LANCE's *mcs*-command), but not dynamic insertion/deletion and LANCE setup.

2. The NTI-driver provides an *ni_ioctrl*-function that activates receive packet handling via a dedicated pSOS task. Normally, this is done in the context of the LANCE ISR. Packet forwarding, however, can be a time-consuming operation: The KI utilizes a reliable transmission protocol that may require various activities upon message reception, and reception of a large packet into non-contiguous buffers demands copying it into a contigous buffer area. Thus, decoupling by means of a task is an appealing alternative, but one has to keep in mind that switching this feature on makes operation of the NTI-driver dependent upon a running pSOS^{+} kernel. This is only acceptable for KI- and CI-messages, therefore NI packets are always forwarded within the ISR context and are not touched by the task, regardless if it is turned on or off. Although NI packet forwarding can cause latency due to packet copying, the ISR execution time only can get huge for KI packets.

3. During its initialization sequence, pSOS^{+m} enforces an interrupt mask of 7, thereby disabling all (including the LANCE's) interrupts. This does not cause any problem on the master node, where pSOS^{+m} calls the NTI-driver to initialize itself, completes its internal startup activities and goes active, lowering the interrupt mask thereafter. On the slave node, though, pSOS^{+m} transmits a join request message to the master node immediatly after initializing the driver and then polls for a reply by calling *ki_receive* until a reply comes in. Since the interrupt mask remains at 7, such a message may come in, but cannot be taken from the RFDL and handed up to the client since this is done in the LANCE ISR.

   Obviously, this design of the pSOS^{+m} startup was implemented with environments in mind where no interrrupt activity is required for handing over received messages (e.g. shared memory). Since this is definitely not true for the NTI-driver, our workaround is to lower the interrupt mask at each call of *ki_receive* when pSOS^{+m} is not yet active (which can be determined by the variable PSOS_FLAG). This allows the LANCE ISR to become active and deliver received messages.

4. A related problem during startup of a slave is the fact that pSOS^{+m} does not call the *ki_time* function while it is polling for a join reply message. Thus, *watchdog* cannot become active, disabling the whole transmission protocol. The first transmission is initiated by *put*, but there will not be any retransmission due to timeouts. In fact, pSOS^{+m} remains waiting for a reply to the join request for eternity if it should have been lost due to an error in the network or on the master node.

Appendix A

List of Abbreviations

ACBL Alternative Command Block List
ARP Addresss Resolution Protocol

CA Channel Attention
CBL Command Block List
CD Collision Detection
CI Clock synchronisation Interface
CPU Central Processing Unit
CSMA Carrier Sense Medium Access
CU Command Unit

DCR Dynamic Collision Resolution

FTP File Transfer Protocol

GPS Global Positioning System

ICMP Internet Control Message Protocol
IFS InterFrame Spacing
IGMP Internet Group Management Protocol
IP Internet Protocol
ISR Interrupt Service Routine

KI Kernel Interface

MAC Medium Access Control
MIB Management Information Base
MTU Maximum Transmission Unit

NI Network Interface
NFS Network File System

NTI Network Time Interface

OS Operating System OSI Open Systems Interface

PNAD pNA$^+$ Daemon

RB Receive Buffer
RBD Receive Buffer Descriptor
RFA Receive Frame Area
RFC Request For Comment
RFD REceive Frame Descriptor
RPC Remote Procedure Call
RSC Remote System Call
RU Receive Unit

SCB System Control Block
SNMP Simple Network Management Protocol
SSCMP Sequenced Synchronized Clock Message Protocol

TB Transmit Buffer
TBD Transmit Buffer Descriptor
TCB Transmit Command Block
TCP Transmission Control Protocol
TP Transmission Protocol
TSTCB TimeStamped Transmit Command Block

UCBL Urgent Command Block List
UDP User Datagram Protocol
UTC Universal Time Coordinated

Bibliography

[Cor92] Intel Corp. *32-bit LAN Component User's Manual*, 1992.

[HSS98] Martin Horauer, Ulrich Schmid, and Klaus Schossmaier. NTI: A Network Time Interface M-Module for high-accuracy clock synchronization. In *Proc. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'98)*, pages 1067–1076, Orlando, Florida, March-April 1998.

[IS96a] Inc. Integrated Systems. *pSOS System*, 1996.

[IS96b] Inc. Integrated Systems. *pSOS System Programmer's Reference*, 1996.

[Loy96] Dietmar Loy. *GPS-Linked High Accuracy NTP Time Processor for Distributed Fault-Tolerant Real-Time Systems.* Dissertation, Technische Universität Wien, Faculty of Electrical Engineering, April 1996.

[MNS99] Thomas Mandl, Herbert Nachtnebel, and Ulrich Schmid. Network time interface user manual. Technical Report 183/1-87, Department of Automation, TU Vienna, January 1999.

[NKC+98] Herbert Nachtnebel, Nikolaus Kerö, Gerhard R. Cadek, Thomas Mandl, and Ulrich Schmid. Rapid Prototyping mit programmierbarer Logik: Ein Fallbeispiel. In *Proc. Austrochip 1998*, pages 99–104, Wiener Neustadt, Austria, October 1998. (ISBN 3-901578-03-X).

[Pus95] Alfred Pusterhofer. *NTSU Network Timestamp Unit: Ein Modul für Eventtimestamping und Uhrensynchronisation via LAN.* Diplomarbeit, Technische Universität Wien, Department of Automation, 1995.

[Ric97] Gerda Richter. *Driver for Real-Time Communications Coprocessor.* Diplomarbeit, Technische Universität Wien, Department of Automation, June 1997.

[SKM+99] Ulrich Schmid, Johann Klasek, Thomas Mandl, Herbert Nachtnebel, Gerhard R. Cadek, and Nikolaus Kerö. A Network Time Interface M-Module for distributing GPS-time over LANs. Technical report, Technische Universität Wien, Department of Automation, 1999. (forthcoming).

[SM99] Ulrich Schmid and Thomas Mandl. Implementation of the NTI device-handler. Technical Report 183/1-86, Department of Automation, TU Vienna, January 1999.

[SN99] Ulrich Schmid and Herbert Nachtnebel. Distributing GPS time over LANs using COTS components. In *Proc. IFAC Workshop on Real-Time Programming (WRTP'99)*, 1999.

[SP95] Ulrich Schmid and Alfred Pusterhofer. SSCMP: The sequenced synchronized clock message protocol. *Computer Networks and ISDN Systems*, 27:1615–1632, 1995.

[SSHL97] Klaus Schossmaier, Ulrich Schmid, Martin Horauer, and Dietmar Loy. Specification and implementation of the Universal Time Coordinated Synchronization Unit (UTCSU). *J. Real-Time Systems*, 12(3):295–327, 1997.

Index

accuracy intervals, 17
ACK_REC, 92
ACK_TR, 92
ADDRCNT, 82
address, 85
agent, 11
alloc_ci_tbuf, 46, 138
alloc_ci_tenv, 46, 139
alloc_ki_tbuf, 46, 139
alloc_ki_tenv, 47, 140
alloc_ni_tbuf, 47
application layer, 28
area, 25
ARP, 16
ARRAY_SIZE, 92
asynchronous signals, 12
asynchronous_error, 45, 138
awx, 108

B_SIZE, 82
BLOCK_SIZE, 93
BROADCAST_ADDRESS, 82
buffer, 6
BUSY, 90
BUSYCOMPL, 90

C1, 94
C10, 95
C11, 95
C12, 95
C13, 95
C14, 95
C2, 94
C3, 94
C4, 94
C5, 94
C6, 94
C7, 94
C8, 95
C9, 95
CA, 19, 52, 86
CBASE(x), 97
CBL, 20
cbx, 57, 101
CFG, 58, 86
CFG_CNT, 86
cfgx, 104
Channel Attention, 19
CI, 36, 82
ci announce packet entry, 38
CI services, 37
ci_announce_error_up, 116
ci_announce_up, 116
ci_dispatcher, 117
ci_efn, 101
ci_fn, 101
ci_get_tbuf, 39, 118
ci_get_tenv, 39, 119
CI_GTB, 83
CI_GTE, 83
ci_init, 38, 120
CI_INIT, 83
ci_ret_rbuf, 39, 121
ci_ret_renv, 40, 121
ci_ret_tbuf, 39, 120
ci_ret_tenv, 39, 121
CI_RRB, 84
CI_RRE, 84
CI_RTB, 83
CI_RTE, 83
ci_send, 40, 122
CI_SEND, 84
ciparam, 100
CNA, 89
CodeTEST, 189
COMCO memory area, 64
commands, 57
COMMANDS, 86
COMPL, 90
configuration tables, 8
confx, 110
copy_block, 165
copy_rbd, 71, 166
COULD_NOT_BE_MADE_CONTIGUOUS, 84
COUNTDOWN, 92
CU, 19
CU_ACTIVE, 87
CU_IDLE, 87
CU_INTS, 89
CU_START, 90
CU_STATES, 87

CU_SUSPENDED, 87
CUC, 89
CX, 89

DEFA, 81
DEFI, 81
DESCR0, 95
DIAG, 62, 87
diagx, 106
disable, 180
DMA, 19
DMA_UNDERRUN, 91
do_something_else, 152
driver layer, 52
DUMP, 61, 87
dump_areax, 102
dumpx, 106

EL, 91
EMPTY, 92
enable, 181
envelope, 6, 21, 22
EOFR, 91
ERROR, 91

fatal_error, 44, 138
FLEX_MODE, 91
flexible memory model, 21, 22
forward_rpk, 50, 147
forward_rpk_up, 72, 166
FR, 89
frame, 6
FREECNT, 93
FULL, 91
FWDPRIO, 96

G_INT, 90
get_no, 132
get_tbd, 67, 154
get_tcb, 67, 154
get_ts_tcb, 68, 155
getEtherAddress0, 174
global object, 9

hardware address, 15

i82596, 18
I_RETURN, 12
IAS, 58, 86
iasx, 104
IFCNT, 81
IMMEDIATE_RELEASE, 83
init, 45, 172
init_cfg, 74, 155
init_ias, 74, 156
init_lance, 76, 156
init_mcs, 75, 157
init_rbd, 73, 167
init_receive, 74, 168
init_rfd, 74, 168
init_scb, 75, 159
init_sscmp, 42, 133
init_tbd, 75, 157
init_tcb, 75, 158
init_trans, 76, 159
init_ts_tcb, 76, 158
initialization of the LANCE, 56
initialization root, 19
INT_LEVEL0, 96
Intermediate System Configuration Pointer, 55
interval-based clock synchronization, 17
intlanceon0, 174
INVALID_BUF_ADDRESS, 84
INVALID_COMMAND, 85
INVALID_ENV_ADDRESS, 85
INVALID_IF_NUMBER, 85
INVALID_POINTER, 85
INVALID_PROTO, 85
ISCP, 56
iscpx, 107
ISR, 12

Kernel Interface, 9, 28
KI, 9, 82
KI announce message entry point, 29
KI services, 28
ki_announce_error_up, 122
ki_announce_up, 123
ki_dispatcher, 179
ki_dummy, 124
ki_get_rbuf, 51, 149
ki_getpkb, 29, 125
ki_init, 30, 124
ki_receive, 30, 125
ki_returnp, 30, 126
ki_roster, 31, 126
ki_send, 31, 127
ki_time, 32, 128
KQ_LENGTH, 82

LANCE, 18
LANCE commands, 20
LANCE ISR, 62
LANCE_INT_VEC_OFFSET, 96
lance_isr, 72
lance_isr0, 171
lancedrv, 181
LanStop, 174
LBASE(x), 97
LOAD_BT, 90

macrostamp, 64
manager layer, 45
mask_drop, 182
master node, 9
MAX, 92
MAX_ADDR_CNT, 86
MAX_BLOCKS, 93
MAXASICS, 82
MAXpNAIF, 97
MCS, 59, 86
mcsx, 104
mem_init, 178
memclr, 175
memcompp, 175
memcopp, 176
memory requirements, 186
message, 5
mfree, 178
MIB II, 12
MIB-II, 17
MIB-II commands, 35
mmalloc, 176
MTU, 15
MTU0, 96
MULT_TICK_RETDAEMON, 96
MULT_TICK_SSCMP, 92

network interface, 14
NETWORK_NOT_FUNCTIONAL, 84
NI, 14, 32
NI announce packet entry point, 33
NI control flags, 15
NI services, 32
NI1, 82
NI2, 82
ni_announce_up, 128
ni_broadcast, 34, 129
ni_dispatcher, 178
ni_initial, 33, 129
ni_ioctl, 35, 130
NI_IOCTRL_PSOSDOWN, 97
ni_poll, 34, 131
ni_send, 34, 132
NO_BUF_FREE, 84
NO_ENV_FREE, 84
NO_ENVELOPE_ATTACHED, 84
node anchor, 8
node configuration table, 8
node number, 9
NOP, 57, 86
nopx, 103
NTI, 23
NTI error handler, 44
NTI-Handler, 26

OK, 90

packet, 6
packet dependent, 33
packet independent, 33
pNA, 12
poll_ni, 51, 149
PORT, 53
PORT_DUMP, 88
PORT_RESET, 88
PORT_SCP, 88
PORT_SELF_TEST, 88
PREF, 91
PRIOR, 83
programming of the LANCE, 62
pSOS, 8
put, 42, 133
put_cb, 68, 160
put_cb_urgent, 68, 161

RADDSIZE, 85
RB, 22
RB_COUNT, 93
rb_index, 165
RBD, 64
rbd_index, 164
RBDs, 22
rbdx, 106
rcv_error, 51, 150
rcv_fatal_error, 44
receive buffer, 22
receive buffer descriptors, 22
receive frame area, 22
receive frame descriptors, 22
receive timestamp, 64
receive timestamp information, 37
receive_fatal_error, 137
release_cb, 47, 144
release_cb_up, 69, 161
release_sscmp, 43, 134
remote service call, 9
renvx, 99
RESET, 88
ret_ci_rbuf, 51, 150
ret_ci_renv, 51, 151
ret_ci_tbuf, 48, 145
ret_ci_tenv, 48, 145
ret_daemon, 72, 170
ret_ki_rbuf, 52, 151
ret_ki_renv, 52, 152
ret_ki_tbuf, 48, 146
ret_ki_tenv, 48, 146
ret_ki_tmess, 48, 147
RET_PKB_IMM, 83
ret_rbd, 73, 169

ret_rbd_nc, 71, 169
ret_rfd, 73, 171
ret_tbd, 69, 163
ret_tcb, 69, 162
ret_ts_tcb, 69, 163
RFA, 22, 63
RFD, 63
RFD_COUNT, 93
RFDs, 22
rfdx, 63, 105
RNR, 89
RSC, 9
RU, 19
RU_IDLE, 88
RU_INTS, 89
RU_NORBDS, 88
RU_NORSOURCES, 88
RU_R_WITHOUT_RBDS, 88
RU_READY, 87
RU_RESUME, 89
RU_START, 90
RU_STATES, 87
RU_SUSPEND, 90
RU_SUSPENDED, 88
RUC, 89

SCB, 20, 53
scbx, 107
scpx, 108
sh_memx, 110
shared memory, 23
shared memory layout, 64
simplified memory model, 21
SIZE, 91
slave node, 11
socket, 13
SPEED0, 96
sscmp_announce, 43, 135
sscmp_error_announce, 43, 135
SSCMP_PACKET, 92
SSCMPDS, 92
STATES, 89
swapping, 57, 86
System Configuration Pointer, 55
System Control Block, 53

t_forward_rpk, 148
T_OFF, 94
T_ON, 94
t_tfreelist, 108
TADDSIZE, 85
TASK_STAT_INVALID, 85
TB_COUNT, 93
tb_index, 153
tbd_index, 153
tbdx, 60, 105
TC_COUNT, 93
TCB, 21
tcbx, 105
TCTS_COUNT, 93
TDR, 61, 87
tdrx, 105
tenvx, 99
ti_acx, 60, 97
timestamped transmit envelope, 24
timex, 64, 98
trans_ci, 49, 141
trans_fatal_error, 44, 137
trans_ki, 49, 142
trans_ni, 49, 142
transm, 50, 143
transmit command block, 21
transmit packet, 21
transmit timestamp, 60
transmit timestamp information, 36
TRX, 59, 87
TS_PACKET, 83
ts_tcbx, 105
ts_tenvx, 98
TSADDSIZE, 86
TSFILLCNT, 82
TX_FAILED, 85
TYPE0, 96
TYPES, 91

ucast, 164
UTCSU-ASIC, 23

valx, 109
virtual nodes, 25

waitx, 109
watchdog, 43, 136