Abstract

A vital feature of distributed real-time systems is the generation of a global sense of time through synchronization of the clocks. The Network Time Interface (NTI) supports very high accuracy external clock synchronization by hardware. It is designed as a piggy-back to the VME 162-512A carrier board which is equipped with Intel's i82596 LAN Coprocessor. The real-time operating system is pSOS of Integrated Systems Inc. The object of this thesis was to design and implement a driver for the network communication coprocessor in this environment, that allows the synchronization of the clocks in the system transparently to the user. This driver is called the NTI-driver. It provides the features necessary for network communications to the operating system and independently of that an interface for clock synchronization.

Kurzfassung

Ein wichtiger Aspekt von verteilten Echtzeitsystemen ist die Generierung einer für das System einheitlichen Zeit durch die Synchronisation der Uhren. Das Network Time Interface (NTI) unterstützt hochgenaue externe Uhrensynchronisation durch Hardware. Es ist als Steckkarte für das VME 162-512A carrier board entwickelt worden, das mit dem i82596 LAN Coprozessor von Intel ausgestattet ist. Das Echtzeit-betriebssytem ist pSOS, von Integrated Systems Inc. Das Thema dieser Arbeit war, einen Treiber zu entwickeln und implementieren, der die Synchronisation der Uhren eines Systems, transparent für die Benutzer, erlaubt. Dieser Treiber wird NTI-treiber gennant. Er bietet für das Betriebssystem die Funktionalität die für die Kommunikation über das Netz nötig ist und parallel dazu und unabhängig davon die einer Schnittstelle für die Uhrensynchronisation.

Contents

List of Figures

Chapter 1

Introduction

This introductory chapter provides an overview of what real-time systems are, what features are characteristic and what are the requirements of a network driver in such a system. The explanation of real-time systems will focus on the points relevant for this thesis, since it is assumed that the reader is already familiar with the basic concepts.

The second chapter describes in detail the environment of the NTI-driver. This environment consists of certain applications relying on the driver's services and, of course, the hardware it uses to perform its duties as a network interface.

Chapter three is a presentation of the NTI-driver itself. It describes its interfaces to the environment and its internal structures. The different layers and their interactions are explained in detail.

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.

Chapter five is the user's manual. Again the NTI-driver is described, but this time from the user's point of view. Installation and runtime maintenance procedures can be found there together with a description of the interface to the applications and a list of the possible error messages.

Chapter six deals with the testing of the NTI-driver, which programs exist to support the testing and how they are applied.

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

1.1 Real-time Systems

According to [RTS92] the definition of a *real-time system* reads as follows: A real-time system is a system that must satisfy explicit response-time constraints, or risk severe consequences including failure. Real-time systems therefore differ from other computer systems in that they have to meet deadlines. The requirement is not necessarily that they respond as fast as possible, but that they do so within a certain time limit. Computer systems that operate in safety critical environments (airports, aeroplanes, factory control, railway control, etc.) have to deliver their results in time or they need not do it at all. A decision that arrives too late may result in a

catastrophe even though it was the right one. For instance if the switch has already been passed by the train there is no point at all in shifting it.
Some real-time systems do not have such dramatic consequences if they fail or miss a deadline. Those are called *soft real-time systems* , in contrast to *hard real-time systems* . *Firm real-time systems* have hard deadlines, where a certain (low) probability of missing one can be tolerated. Usually the term real-time systems is only used in connection with hard real-time systems. I will follow this tradition.

Since deadlines are such an important issue in real-time systems, the prediction and measurement of response time as well as its reduction require careful consideration during design and implementation of such systems.

1.2 Time in a real-time system

Since all reactions to events have to present a result within a deadline, it is important to provide a chronoscopic sense of time to facilitate determination of the point in time when an event happened and also the time that has passed since then. This is relatively simple in a single-processor system, since there is exactly one clock that determines the flow of time, which is enough for the purposes of such a configuration. In a distributed real-time system, consisting of several processors with clocks, it is much more difficult to provide a sense of time that allows a correct ordering of all the events in the system. For physical reasons no two clocks will ever tick at exactly the same rate. Because of this, it is not enough to synchronize all the clocks in the system once and then let them run. Sooner or later they will drift apart again. Resynchronizations are therefore necessary at certain intervals. This area is already well researched together with the problem of providing a global sense of time within a distributed real-time system. For an overview of *internal clock synchronization* see [RSB90] and [SWL90]. [YM93] provides a bibliography.
Still there is an additional level of complexity to be considered. Real-time systems become more and more prevalent in daily life. It is therefore necessary that they do not only have their own sense of time, but that this time has a well-defined relation to the official time as well. This area of *external clock synchronization* is less well researched. An overview of the current state of research can be found in [Sch97a].

UTC (Universal Time Coordinated) is the only official notion of time. It is derived from *TAI* (Temps International Atomique), which is composed from various highly accurate atomic clocks placed all over the world, and provided by the US Army via the *GPS* (Global Positioning System). However, using that definition of time is not without problems, since Leap seconds are inserted from time to time, so that UTC is not chronoscopic (does not always allow to measure small intervals of time). As a consequence a chronoscopic time-scale closely related to UTC (such as GPS-time) is in general preferable. See [Sch95] for an overview.

A promising approach for external clock synchronization is the interval based paradigm, introduced in [Mar84]. Time information is represented by intervals that are known (i.e., supposed) to contain UTC. From a set of such intervals a relatively small interval that contains UTC may be calculated, even if some intervals are faulty. There is however an improvement of this approach under development that takes into account that the various nodes can keep track of time by means of their local clocks as well. A validity interval is computed from all the local clocks in the system to judge whether the information from the high accuracy timers is valid. This technique (called *clock validation*) allows the design of fault tolerant time services that relates to an external time standard with only a few nodes actually having access to it.

1.2.1 Basic Hardware Architecture

To be applicable in practice, such an algorithm has to provide certain features that are not usually considered in theoretical research. Most importantly, increasingly demanding applications require systemwide accuracies in the range of μs despite the adverse effects of clock granularity and transmission delay uncertainties, cf. [SS97]. To achieve this a certain amount of hardware support is necessary, which should be compatible to a reasonably broad class of existing systems and network architectures.

This degree of synchronization is accomplished by the Network Time Interface (*NTI*), see [HSS97]. Basically, this board is built around a newly developed ASIC, the Universal Time Coordinated Synchronization Unit (*UTCSU*). It provides the necessary features to realize event timestamping of network traffic and, in consequence, high-accuracy clock synchronization. Those features are: a state and rate adjustable clock device, locally maintained accuracy intervals, interfaces to GPS receivers and various timestamping features. Thus it is feasible to generate timestamps chronologically very close to the actual occurance of an event.

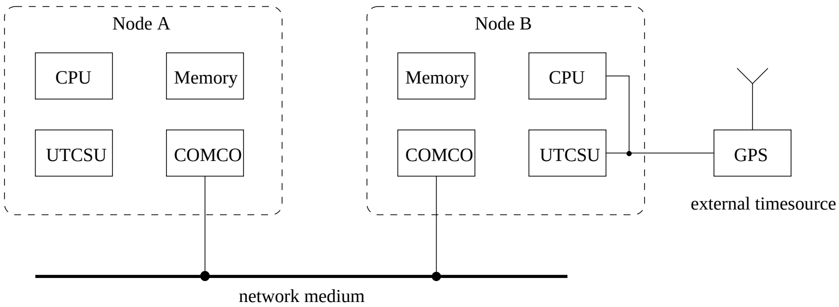

Figure 1.1: Physical structure of a node

The whole project SynUTC, see [Sch95], of which the NTI-driver is a part, is intended for distributed systems of nodes that use an ordinary packet-oriented network for communication. Each of the nodes must be equipped with a hardware clock, a CPU for running the clock synchronization (this may be the central processor of the node, or, preferably, a dedicated microprocessor or microcontroller) and a communications coprocessor (*COMCO*) which has to be able to read/write data from/to memory independently of the CPU. Some of the nodes must have access to external time sources (e.g. GPS satellite receivers). See Figure 1.1 for an overview.

1.3 Basic Software Architecture

The goal of the project is to extend pSOS^{+m} systems by providing synchronized clocks. The synchronization of those has to be transparent to the user tasks (apart from the fact that the clocks <u>are</u> synchronized).

Both, NTI and clock synchronization, are hidden from pSOS^{+m} – and hence also from the application tasks – by a device driver that has to service the NTI and the communication coprocessor on one hand and its clients on the other. See Figure 1.2 for a structural overview.
The first specification of this NTI-driver, which was the basis for this thesis, can be found in [Pu95].

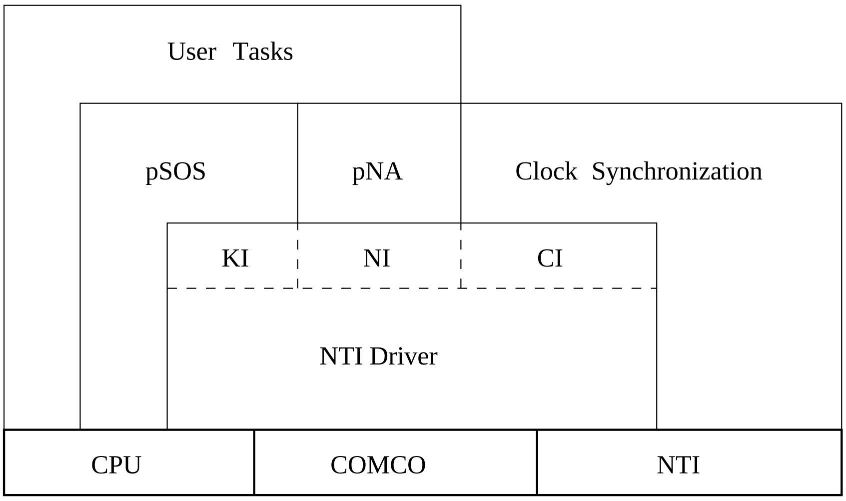

Figure 1.2: Software structure of a node

As shown in Figure 1.2 there are three *clients* using the NTI-driver, which requires the implementation of three different interfaces:

- **pSOS**$^{+m}$, the multiprocessor kernel communicating with other processors via kernel interface (KI). Basically, pSOS^{+m} supports remote objects that are managed through RPCs. The NTI-driver links pSOS^{+m} to the underlying hardware.
- **pNA**$^{+}$ communicates with other nodes through the network interface (NI). This component provides standard sockets (TCP/IP, UDP/IP,...) to user tasks. Just as pSOS^{+m} the NTI-driver links pNA^{+} to the underlying hardware.
- The **Clock Synchronization** requires nothing more of the clock synchronization interface (CI) but to provide a message passing system with one special feature: the messages received and sent by this application need to be timestamped. The clock synchronization client uses nothing but NTI-driver and NTI, and is thus transparent to the other clients apart from the fact that it contributes to the computing and network load.

In brief, the NTI-driver is a piece of software that acts as a sort of multiplexing device between three clients and the hardware. It gets messages intended for transmission, transforms them into packets containing the message and driver specific information and sends them to the destination. It also receives packets, extracts the messages and relays them to the appropriate application.

1.4 Definitions And Notational Conventions

1.4.1 Definitions

In order to increase the understandability of the following text I will introduce a few concepts used throughout the thesis.

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 general information about the packet.

When a packet is sent over the network medium it is transformed into an Ethernet *frame* , which contains all the message relevant data a packet does.

An *envelope* is the control part of a packet. It contains information about it such as the source and destination address. An envelope is a data structure added to a message by the network driver to generate a packet that can be sent by the network controller. In this thesis there are two types of envelopes, timestamped ones and normal ones. The differences will be explained in the following chapters.

There exist two logical types of *buffers*, transmit and receive buffers, though their physical representation is the same. Both are a contiguous space of memory where the message of the client is stored for transmission or after reception. Apart from the clock synchronization all clients see only the buffers and never their envelopes.

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

1.4.2 Notational Conventions

In order to facilitate finding terms that are listed in the index, such terms have been printed *italic* in the text.
<u>Variables</u> are underlined and *functions* are also printed italic.

Chapter 2

Environment

In this chapter the interested reader will find a more detailed description of the environment the NTI-driver was designed for and has to work in. This environment can be neatly parted into an upper and a lower half.

The upper half is comprised of the clock synchronization and two of the components of a package that constitutes the real-time operating system *pSOS* .
pSOS is developed and distributed by ISI (Integrated Systems Inc) and consists of the following 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
- **pRPC**$^+$ Open Network Computing and RPC libraries for pSOS systems
- **pREPC**$^+$ ANSI C Standard libraries for pSOS system
- **XRAY**$^+$ source level debugger for pSOS systems

This thesis includes a brief overview of the relevant components. Further details can be found in [pSYS96]. All pSOS components are available for many different CPUs, including the Motorola MC680x0. Apart from this it is completely hardware independent and can be installed anywhere in RAM or ROM. This hardware independency is accomplished by means of a configuration table supplied by the user and containing certain parameters, e.g., describing the hardware. The connection between the components is the so-called Software Component Interconnect Bus which is a table where pointers to all the components can be found. The address of this table is called the node anchor, because it is the one thing that combines all the software components of a node's OS.

The lower half of the environment is the CPU board with the NTI. The NTI-driver has been developed for the MVME162 VME CPU which is equipped with an MC68040 processor and an INTEL i86592 CA Network controller.

Each of the parts relevant for the driver is described in detail in one of the following sections.

2.1 Clients

According to Figure 1.2 there are three clients using the NTI-driver as a link to the network. pSOS^{+m} and pNA^{+} are the multiprocessor kernel and the TCP/IP network component of the real-time operating system pSOS, respectively. The descriptions given here are intended to provide an overview of the applications and to explain their main characteristics. See [pSYS96] for further details on both components.
The clock synchronization is part of the SynUTC project itself and has been designed and developed concurrently with the NTI and its driver.

2.1.1 pSOS^{+m}

pSOS^{+m} is a multiprocessor version of pSOS^{+}, a real time multitasking operating system kernel. 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...). The multiprocessing kernel is designed for functionally divided multiprocessing systems. Each processor 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.
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 one which is the exception to hot swap and soft fail. It must be present in the system at all times while the others may join and leave any time they please.
The kernel itself is, due to the configuration tables, entirely hardware independent with a well-defined interface to the network driver. This KI (Kernel Interface) is standard and 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 letting the driver announce incoming messages

Every node is identified by a *node number* . This number is assigned by the user, the master node is the one with number 1. Clearly the node numbers have to be unique. 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.
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 a message to the source node. There a routine, *mc_asyncerr*, 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 such a routine is not provided, however, pSOS^{+m} generates an internal fatal error. If an asynchronous RSC is called on an object that resides locally, the synchronous version is performed, 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 then proceeds with executing the application tasks.
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.
In short, agents are used to wait on behalf of the remote task that made the RSC. Just like the number of nodes, the number of available agents is limited and specified in the pSOS^{+m} configuration table.

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. 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. The user has to ensure that no node attempts to join too early.
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 calls RTE and I_RETURN. I_RETURN returns control to pSOS^{+m}, while RTE continues at the interruption address. Since the latter may cause static priority 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 resultin 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 system calls are allowed in ISRs. Generally, all those that can cause the calling task to block are forbidden.

Asynchronous Signals

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.
Asynchronous signals can be viewed as a sort of "software interrup". 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.

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 other pSOS system networking add-ons such as RPC, NFS, FTP and others. In addition, pNA^{+} supports the Management Information Base for Network Management of TCP/IP based Internet standard (MIB II). On the whole, pNA^{+} supports the following features:

- **SNMP**
- **FTP, telnet, TFTP**

- **RPCs**
- **NFS**
- **streams**

Like pSOS^{+m}, pNA^{+} is entirely hardware independent by means of a configuration table and a well-defined interface to its network driver, the *NI*. Again, the interface is standard and provides 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

Except the last function the others are called via an entry point found in the configuration table. The address of the last function, the messsage announce entry point is handed to the driver during initialization. The NI need not provide all the features of the LLC (logical link control), it is used only as a message passing system. Comparing the number and types of functions the NI and KI (as described in the previous section) are considerably different.

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 has to 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. 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 another system call.

Network Interfaces

The pNA$^+$ component accesses the network by calling a user-provided layer of software, the network driver, via the network interface (*NI*). This NI is a simple message passing interface that is standard and independent of the network's physical characteristics. From pNA$^+$s point of view, there must be one driver 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 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 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 does not have internal memory for messages and uses 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.

Up: If this is set, the initial mode of the network connection is "UP".

A network can have a *Network Subnet Mask* associated with it to support subnet addressing. The network mask is a 32-bit value with ones at all the bit positions that are to be interpreted as the network portion. For further explanations, have a look at TCP/IP addressing modes, e.g., in [TCPIP94].

In point-to-point networks the destination address of the peer node is specified in the pNA$^+$s configuration table for the NI.

ARP

As has been said before, every NI has two addresses associated to it: an internet 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 stored in the form of messages.

The RAWMEM flag tells whether the interface uses those data structures pNA$^+$ provides or its own. If it uses its own, pNA$^+$ performs the necessary copying of data between the different structures.

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.

Startup

A power-on or reset passes control to the user-supplied boot code which performs the necessary initializations and self-tests. Additionally, it should set up the required configurations environment for the software components in the system, i.e., write the configuration tables. The *system startup sequence* is as follows:

1. Power-on or reset
2. Boot code is executed, may pass control to $pROBE^+$, if it is present
3. $pSOS^+$ is started together with the other installed components, such as pNA^+.
4. The application is started.

$pSOS^+$ can be started in one of the following ways:

- By passing control to the $pSOS^+$ startup entry
- From $pROBE^+$
- By specifying the silent startup mode in the $pROBE^+$ configuration table, so that $pROBE^+$ initializes itself and then passes control to $pSOS^+$ Startup without stopping.

Upon entry, $pSOS^+$ startup first uses the node anchor to locate the $pSOS^+$ configuration table. Afterwards it takes a segment of memory and builds there 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.

2.1.3 Clock Synchronization

This client of the NTI-driver is not imposed by the pSOS system, but by the goals of the project *SynUTC* [Sch95]. As has been explained in Chapter 1, this project deals with the integration of external time into the establishment of a common sense of time within a distributed system.
The software part of the clock synchronization algorithm (see Figure 1.2) has been placed directly above the NTI-driver and besides the pSOS^{+m} kernel and the pNA^{+} component. Putting it as close to the NTI-driver as possible is desirable since long delays, as appear when a message is handed through many protocol layers, are then avoided. Besides, the client communicates only with other instances of itself on other nodes, hence it does not require any of the services provided by either pSOS^{+m} or pNA^{+}.
The used *clock synchronization algorithm* is interval based, which means that what is transmitted, compared and computed are intervals containing (or, to be more correct, assumed to contain) real time. Those are called *accuracy intervals* . 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 clock drift, which again is characteristic for each node.

- **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 applied to compatible accuracy intervals, to compute and subsequently enforce an improved accuracy interval.

The sending and receiving timestamps have to be generated as close to the actual event as possible. To that end the NTI supplies timestamps that are generated automatically (hardwired) at transmission and reception of a packet by the network coprocessor.

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.

The clock synchronization requires a simple message passing system that provides the possibility of sending high priority messages. This feature of the NTI-driver transmits the clock synchronization messages as quickly as possible. Thus the broadcast latency can be reduced considerably.
Additionally to timestamped messages the NTI-driver has to provide the means to transmit ordinary messages as well. The messages transmitted are of variable length and must be written into a contiguous space. The received messages need not be contiguous, though the NTI-driver has to provide a function to render one contiguous, if that proves to be necessary.
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_mcon** for rendering a buffer contiguous
- **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

2.2 Communication Coprocessor

The communication coprocessor used in this version of the NTI-driver is Intel's i82596CA, from now on referred to as "*LANCE*" or "*i82596*". It is a 32-bit LAN coprocessor which, among other protocols like HDLC, supports IEEE 802.3 CSMA/CD. It can also be configured for full duplex operation or CSMA/DCR (Deterministic Collision Resolution), thus providing a bounded delay until channel access (a feature important for real-time applications). The protocol used in this implementation is CSMA/CD. Many of the LANCEs parameters can be changed by software, making it flexible and therefore usable with a number of components. 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. The CSMA/CD unit handles all OSI MAC (Medium Access Control) functions (layers 1 and 2), implementing all IEEE 802.3 MAC and channel interface functions, such as framing, preamble generation and stripping, source address generation, destination address checking, short frame detection and automatic length field handling.

2.2.1 Operational Model

Because the i82596 links the network and the host CPU it must be able to execute commands simultaneously to receiving incoming frames. In order to allow this, the chip has two independent logical units: 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 CA (Channel Attention) and returns to its other duties. The coprocessor's CU checks this area and executes the commands without further intervention from the CPU. After 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 the reservation of buffer space adequate for the rate of reception at this node. Then the RU can be activated.

2.2.2 i82596 Memory Structure

As far as the LANCE is concerned, the shared memory consists of four parts (see Figure 2.1): the initialization root, the system control block, the command block list and the receive frame area.

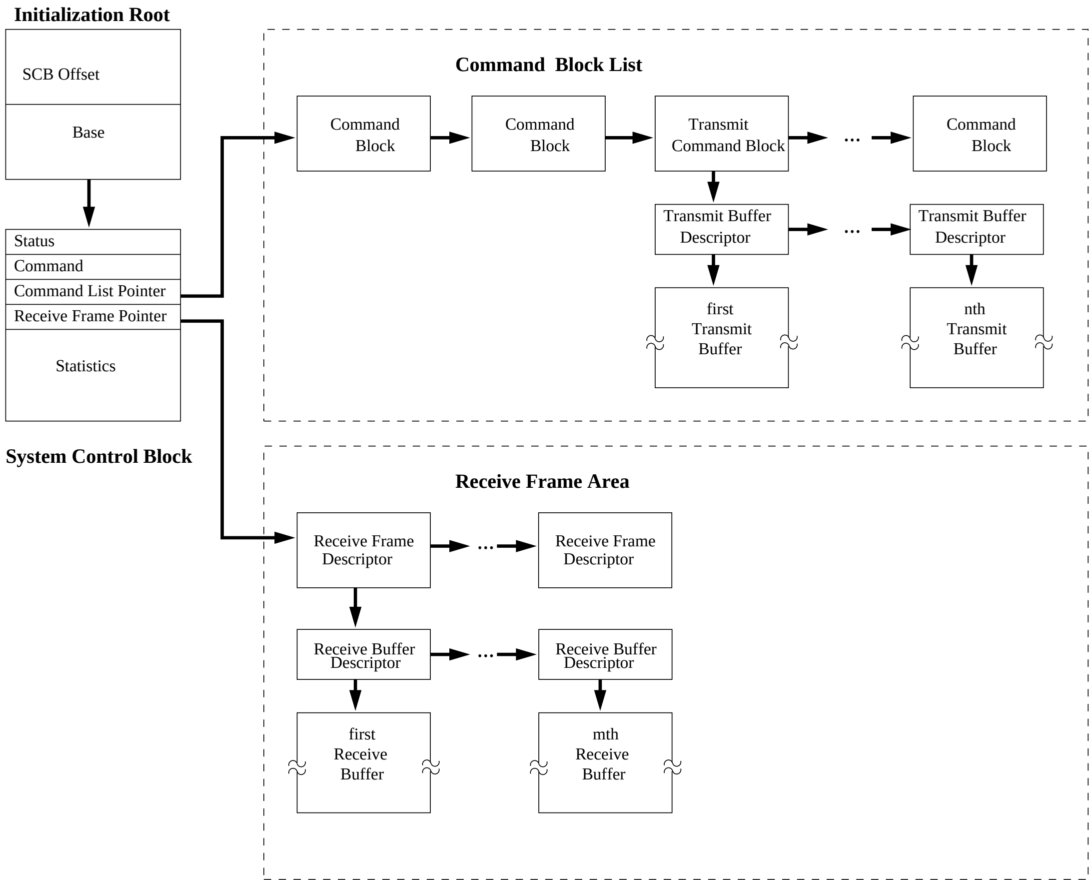

Figure 2.1: Shared memory structure of LANCE i82596

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. 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.

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 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 different possibilities of *command blocks* are:

1. **NOP** No operation
2. **IAS** Individual Address Setup, tell the LANCE its hardware address
3. **CFG** Configure, configure LANCE
4. **MCS** Multicast Setup, tell the LANCE the multicast addresses
5. **TRX** Transmit, submit the LANCE 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** writes the contents of the on-chip registers into a buffer
8. **DIAG** triggers an internal self-test procedure

Transmit

The structure containing a *transmit packet* is organized in the following way: The transmit command block)TCB) contains general information like destination address, type etc. and a pointer to the first of a list of transmit buffer descriptorsTBDs). Each of those containing address and size of a (contiguous) data buffer as well as a link to the next TBD. All transmit buffers (TBs) together contain the whole message, which need not necessarily 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 allthe TCBs in the pool have to be of the size of the largest packet that can be expected for transmission. The latter is only feasible, though, if all packets in the network are of the same size. Since the sizes of pSOS^{+m} and pNA^{+} messages may differ by a factor of up to 15, this is not practicable for this implementation. Hence the flexible memory model was chosen.
According to the previous, the following steps have to be taken to transmit a message.

1. First a TCB has to be allocated.

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

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 (by accessing CA) 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

The storage of incoming frames in the receive frame area (*RFA*) is done automatically. The receive frame area is a list of free receive frame descriptors (RFDs) and receive buffer descriptors (RBDs). Each of the latter is associated with exactly one receive buffer (RB) and contains its size, address, a link field and the number of bytes actually written into it. This way of organizing the envelopes and data buffers is called the *flexible memory model* , since it allows reception of packets of varying size. Opposed to this there is the possibility of organizing the RFA in the *simplified memory model* . Here each RFD is allocated a data area of fixed size which has to be as large as the largest message that is to be expected. It only makes sense to use this model in a network where all the packets are of approximately the same size. Since the MTU for the pNA$^+$ component is 1500 and the size of a kernel message always 100 bytes, though, the simplified memory model is not a practicable choice for our purposes.

SCB

The SCB references the first free RFD. If a bad frame is received, the RFD and buffer used for it are not taken from the RFA, but reused for the next incoming packet. After receiving a frame the LANCE automatically writes the address of the descriptor of the next free buffer into the next free RFD. The internal pointers to the next RFD and RBD are set so that the next incoming message can be written directly into them.
After reception of an entire frame an interrupt is generated (and the appropriate bits in the SCB set) to tell the host CPU that a message has come in. Still the LANCE will continue to receive as long as RFDs and RBs are available. This may cause race conditions when trying to append new RFDs or RBDs to their respective lists. Each RFD contains bits to report the following events:

- 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

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 NTI

The Network Timestamping Interface (*NTI*) has been developed as a part of our SynUTC project. The description given here is an overview, intended to describe only the points relevant for this thesis. See [HSS97] for further details of the board and its components.

The NTI has been designed to extend existing CPU boards with clock synchronization hardware support. It is built around an ASIC, called Universal Time Coordinated Synchronization Unit (*UTCSU*), which provides most of the hardware support for interval based clock synchronization.

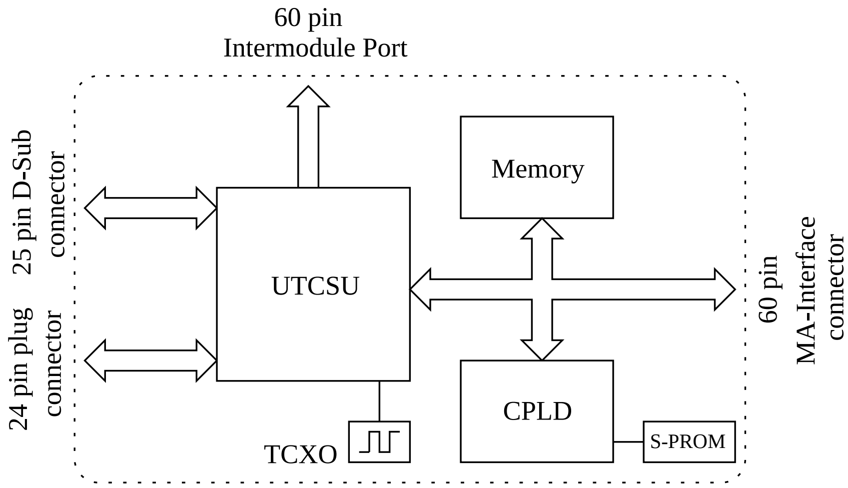

Figure 2.2: Layout of the NTI

It is realized as an MA-Module with 256k of shared memory, a Complex Programmable Logic Device (*CPLD*) and the UTCSU on board, see Figure 2.2 for an overview.

In this implementation of the NTI-driver the NTI works together with the Intel i82596 LAN Coprocessor (LANCE) situated on the MVME162.
The on-board shared memory is the communication interface between CPU and LANCE. It was placed on the NTI in order to facilitate the necessary timestamping hardware. As soon as the LANCE attempts to access a certain address, a timestamp is pulled by the UTCSU and taken as the contents of that address. In order to prevent timestamps being pulled at CPU accesses, two logical address regions are mapped to the same address space. Consequently CPU accesses use the addresses that do not trigger a timestamp. The structure of the shared memory is described in more detail in Section 3.5.1.
The CPLD is responsible for adapting the timing and the byte ordering of both, UTCSU and the shared memory to the MA-module interface. Additionally, it forwards interrupt requests from the UTCSU to the carrier board.
The UTCSU is the main component of the NTI. It is equipped with an adder-based clock and consequently very flexible concerning its pacing and modes of correcting the time information (fine grained rate-adjustability, continuous amortization, addition/deletion of leap seconds, etc.). The current time value can be read atomically and consists of in total 64 bits. The timestamp information is related to the NTP format (see [Mil91]). 32 bits of the timestamp value are the low order *timestamp* information, achieving a resolution of about 60 ns and wrapping around every 256s. The 24 bit *macrostamp* contains the high order time information and forms, together with an 8-bit checksum that protects the whole of the time information, the remaining 32 bits. The triggering occurs every time the decoding logic observes an access at certain addresses, where the data structures for the reception of messages and for the transmission of timestamped packets are stored. The information is then written into dedicated addresses, i.e. into the outgoing bitstream in the case of a transmit (see Figure 2.3).

On the receiving side the triggering works the same way. The difference is, that the timestamps are only written into a dedicated register. Afterwards an interrupt is raised. The ISR of this interrupt gets the address of the RFD “causing” the interrupt and the timestamp value. It writes

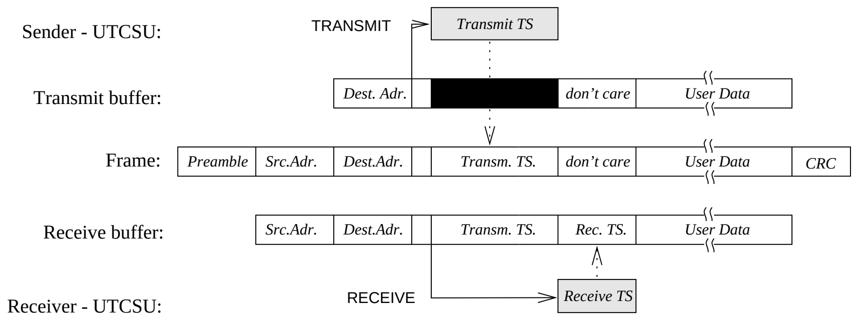

Figure 2.3: Timestamping of outgoing and incoming packets

this value at a certain offset into the RFD.
The advantage of this timestamping logic is, that the transmission delay is only affected by the uncertainties caused by the ongoing data transmission and by the bus arbitration necessary for the memory write cycles of the network coprocessor during reception of a clock synchronization packet.

Transmit timestamping is only necessary for a special kind of envelope, the *timestamped transmission envelope* s the NTI-driver has to provide for the clock synchronization. Since determining whether the incoming message is a clock synchronization message or not would take too much time, all incoming messages are timestamped.

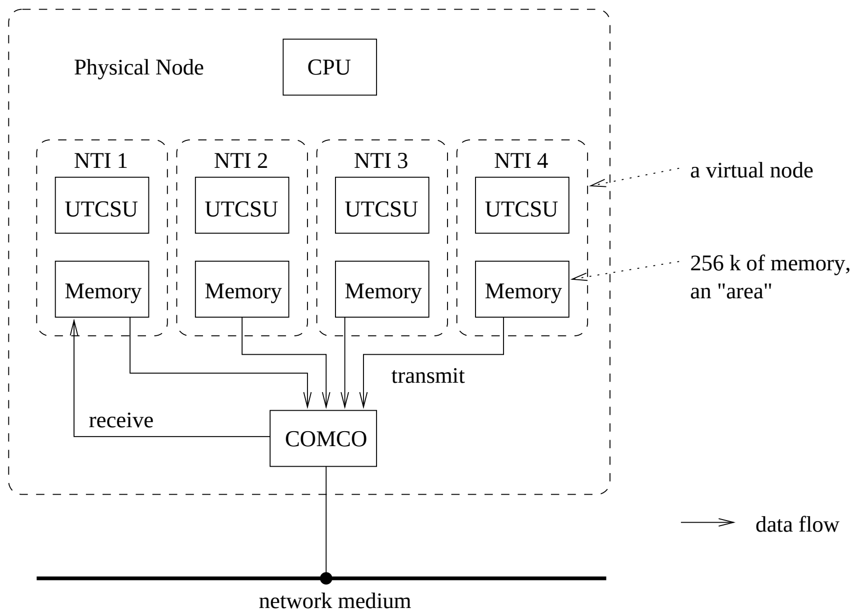

Figure 2.4: Structure of a physical node with multiple NTIs

For cost reduction purposes the NTI-driver must be able to handle more than one NTI per physical node. With multiple NTIs it is possible to emulate multiple "*virtual nodes* " with a single physical node. See Figure 2.4 for the structure of such a node. Thus several timestamps are triggered upon reception of a message, though the only area affected by a receive is the one of NTI 1. All timestamps have to be stored in the RFD, so that the number of NTIs per node is limited to four (as that is the maximum number of timestamps that fit into the frame or RFD). Outgoing timestamp messages may "originate" from one of the different NTIs, each of which has its own shared memory area. Thus up to four different sets of transmission buffers and timestamped transmit envelopes have to be managed per interface. Since each NTI has its own shared memory, this is relatively easy to manage. Each of those shared memories constitutes an *area* , which consequently has to be specified at a transmission or buffer allocation request.

For the evaluation of the transmission delay of packets another feature has been added. From each node's NTI a wire goes to the one of every other node in the LAN. Since during the first phase of the project the nodes will be very close together physically, the implementation of this feature is no difficulty. Each node knows where a specific wire ends, i.e. which other node it belongs to. Whenever a packet is sent, a signal (e.g. a rising edge) is raised on the wire to the receiving node. This causes a timestamp to be pulled there. It is stored in a register dedicated to incoming messages from the node just sending. If the incoming message is one for clock synchronisation, the ISR transferring the receive timestamps into the envelope will also read the register, compute the transmission delay from the receive timestamp (that is already in the frame at this point) and the remotely triggered timestamp and write it into the frame as well. This feature is necessary to determine certain parameters useful for clock synchronization.

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.

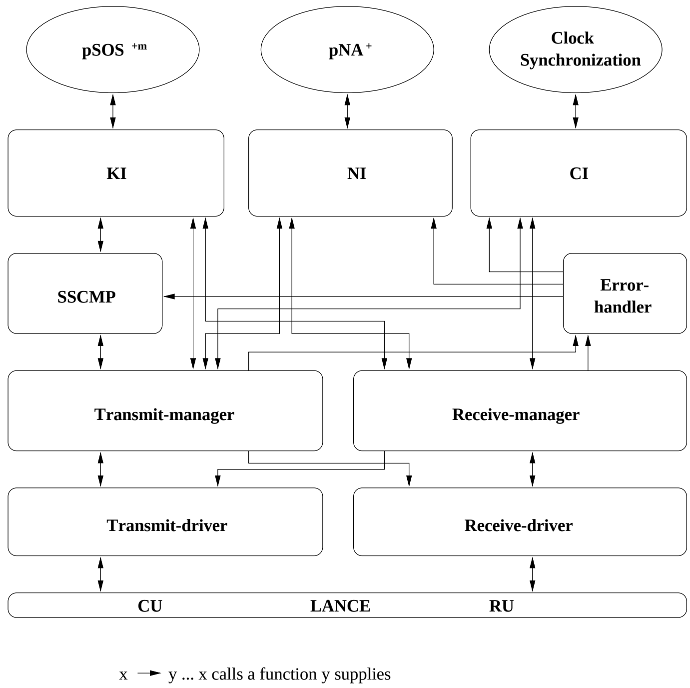

Figure 3.1: Structure of NTI-driver

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. Of the parameters the destination node number and the packet size can be ignored, since the former is not needed until transmission and the latter is 100 for all KI buffers regardless of the parameter value. 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. This parameter is not really necessary, since all KI messages have a size of exactly 100 (see [pSOS96, 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,
INVALID_IF_NUMBER if the KI is 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
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 if the KI is not initialized.

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
INVALID_IF_NUMBER if the KI is 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
INVALID_IF_NUMBER if the KI is 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,
NO_SPACE_IN_WINDOW if too many messages are yet unacknowledged,
INVALID_IF_NUMBER if the KI is not initialized.

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.

The input parameters are:
D0 = 7

The return value is 0 if everything went OK, or
INVALID_IF_NUMBER if the KI is not initialized.

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_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

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.

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 in a global variable.

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 functions, not needed here

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

ni_getpkb

is called to allocate a packet buffer. The size of this packet buffer must not be greater than the maximum transmission unit specified for the interface.

The parameter block has the following structure:
int count... requested size of the buffer in bytes. This number should not exceed the value for the maximum transmission unit entered in the pNA^+ configuration table.
hwa_ptr... pointer to destination HW address, not needed
int if_num... number of the interface.

The return value is a pointer to the allocated buffer, or
NO_BUFS_FREE if there were no buffers available.

ni_retpkb

is called to return a receive packet buffer to the NTI-driver

The parameter block has the following structure:
char *buff_addr... pointer to the buffer
int if_num... number of the interface

The return value is 0 if everything is OK, or
EINVALID if an argument was invalid,
INVALID_IF_NUMBER if the parameter was invalid.

ni_send

With this call pNA^+ sends a packet. 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 buffer used to store the message in may be a receive buffer. This must be taken into consideration when releasing the packet.

The parameter block has the following structure:
addr *hwa_ptr... pointer to destination HW address
char *buff_add... pointer to buffer
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
ENOBUFS if there was no envelope available,
INVALID_IF_NUMBER if the parameter was invalid.

ni_broadcast

With this call pNA^+ transmits a message to all nodes in the network. After transmission the packet buffer has to be released. Just as in normal sending it is possible that the data buffer is a reused receive buffer.

The parameter block has the following structure:
char *buff_addr . . . pointer to data buffer
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
ENOBUFS if there was no envelope available,
INVALID_IF_NUMBER if the parameter was invalid.

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 0 if everything was OK, or
INVALID_IF_NUMBER if that parameter was invalid.

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 ... get specific object
SIOCSIFADMINSTATUS ... set administrative status

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

This function returns 0 if everything was OK, or
INVALID_IF_NUMBER if that parameter was invalid.

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 <u>da</u>. <u>Length_field</u> 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 starts with 16 bytes (four longwords) containing the *transmit timestamp information* in the following sequence:

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

ts ... timestamp information

ms ... macrostamp

acc ... accuracy

The receive envelope contains the *receive timestamp information* , regardless of the transmission mode of the sender:

```
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 rts;
  INT32 rms;
} timex;
```

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

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
- **ci_mcon** for rendering a packet contiguous

The clock synchronization calls the CI services as subroutines with a JSR instruction.

Entry The clock syncronization calls the NI with a JSR instruction. The 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 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 (not necessarily 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.
It may be that the packet buffer is not contiguous but rather a linked list of internal buffers. The clock synchronization may call *ci_mcon* in *ci_announce* to copy it into contiguous buffer space.
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 announce 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 - only relevant during test)

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.

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 - only relevant during test)

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
NO_ENV_FREE if no envelope was available.

ci_mcon

This function renders a buffer contiguous. It calls the manager level function to do htis and returns its return code to the user. In either case the NTI driver takes care of the original buffer, it must not be released by the clock synchronization. Neither must the clock synchronization return IMMEDIATE_RELEASE if it has called *ci_mcon* directly from the ISR, since then an already released and maybe rewritten buffer is released again. In both cases the NTI-driver takes care of the original buffer, it must not be released by the clock synchronization. Neither must the clock synchronization return IMMEDIATE_RELEASE if it has called *ci_mcon* directly from the ISR, since then an already released and maybe rewritten buffer is released again.

The function requires the following parameters:
bufx *buf... address of the original buffer
bufx **nbuf... address to write the address of the new buffer
char *mem... address of memory to copy buffer into
int if_num... number of the interface

It returns 0 if the data buffer is now contiguous or
COULD_NOT_BE_MADE_CONTIGUOUS if it is not,
INVALID_IF_NUMBER if the parameter is not valid.

ci_ret_tbuf

Returns transmit 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 - only relevant for test)

The returnvalue is 0 if everything went OK or
INVALID_IF_NUMBER if the given interface or area doesn't exist or
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 currently being developed at the Department for Automation, the Sequenced Synchronized Clock Message Protocol(*SSCMP*). See [SP95]. Ultimately the NTI-driver will be equipped with it, but since the project is yet at an early stage, a relatively simple sort of sliding window protocol has been employed in the meantime.

Each node maintains an array (the window) - organized as a circular buffer - where certain important data of messages that have been sent are stored. The window size (and consequently the size of the array), which is the maximum number of messages for which the acknowledgement from the receiving node may still be outstanding, is 10. The protocol itself is realized with three functions: *put*, *watchdog* and *sscmp_announce*, having the following duties:

- **put** inserts messages into the window.
- **watchdog** transmits messages, controls the timers and removes acknowledged messages
- **sscmp_announce** receives messages from other nodes, marks acknowledged messages so that *watchdog* can remove them.

When pSOS^{+m} requests a transmission, *put* looks in the appropriate window (specified by the number of the node the message should go to) whether the slot directly behind the one containing the message that was sent last, is free. If it is, *put* writes the index of the slot into the protocol data field of the buffer. This index is the sequence number of the message. The data of the message are entered into the free slot, and the counters set such that the next occurence of *watchdog* will "re"transmit the message.
Handling the messages in the window one has to bear in mind, that to remove messages that are somewhere in the process of being retransmitted may result in the LANCE trying to transmit from data structures that already have been reinitialized or similar effects. Therefore the messages being handed to the LANCE are marked by a flag that is cleared after transmission.
Watchdog checks every entry in the window. If it is marked as acknowledged and not currently being transmitted, the message in the slot is released. If a not acknowledged message is encountered, the counter until retransmission is decreased. If it has reached 0, and the number of retransmission has reached the maximum value, the error handler is called. If the number of retransmission is not that high yet, the message is sent again and the counter keeping track of the retransmissions is increased.
At the receiving node *sscmp_announce* gets messages from the lower layer. It determines the type of the packet (acknowledge or data) from its length. 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.
The acknowledge message is received by *sscmp_announce* on the source node, recognizeable as an acknowledge message by its length. If the number specified in that packet identifies a slot that contains a message for which no acknowledgement has been received yet, the acknowledge flags of all slots from the last acknowledged one to the indicated one are set.

Since this protocol is nothing but a substitute for SSCMP, all the functions and the module itself have been given names with respect to "SSCMP" rather than "transmission protocol".

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 looks in the window indicated by the number of the destination node at the slot following the one where the last message was entered. If it is free, the data of the message is written into the slot, the counter of retransmissions is set to 1 (to mark the slot as full) and the counter to the next retransmission is set to 1, so that the next occurence of *watchdog* will transmit the message.
If the slot in question is full, *put* returns an errorcode.

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 0 if everything was OK or
NO_SPACE_IN_WINDOW if there was no slot available.

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.

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 their length it distinguishes between acknowledge and data messages. Acknowledgement messages are 16 bytes long and contain nothing but the sequence number of a message that has been received correctly by the other node. The acknowledge flags of all messages in the window from the last acknowledged one to this explicitly acknowledged one are set, so that they are released at the next occurrence of *watchdog*.
If the incoming message is of the data type, the protocol field in the buffer contains its sequence number. If that number is equal to the expected one, the message is entered into the queue containing messages that were announced to but not yet fetched by pSOS^{+m}.
Finally an acknowledge message to the source node is generated and sent off and the packet announced to pSOS^{+m}.

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 tick (i.e. 10ms) by pSOS^{+m} through the *ki_time* function.
First it releases all messages that have their acknowledged flags set and which are not currently transmitted. Afterwards it decreases the retransmission timer counter of the remaining entries. If the countdown to the next retransmission is 0 and the number of retries has already reached a maximum value, the error handler is called. If there have not been that many retries yet, the message is transmitted again, and the counter of retransmissions increased.

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

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 <u>mode</u> 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 <u>if_num</u> nor <u>area</u> 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 <u>area</u> 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.

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 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. Finally it hands it 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.
If the buffer is so small, that the resulting packet would stay under the minimum length, it is padded with additional zeros until the minimum size is reached.

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
NO_ENVELOPE_ATTACHED if the pointer to the envelope was NULL

trans_ki

gets a nearly complete packet in driver layer format from the SSCMP transmission protocol. All it has to do is write the pointer to the first buffer descriptor into the TCB and the correct data into the buffer descriptors.
If the buffer is so small, that the resulting packet would stay under the minimum length, it is padded with additional zeros until the minimum size is reached.

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

trans_ni

the function allocates an envelope for transmission at the required network interface. It then writes the necessary data into the envelope and the buffer descriptors and calls the NTI-driver function for delivering the command to the LANCE.
If the buffer is so small, that the resulting packet would stay under the minimum length, it is padded with additional zeros until the minimum size is reached.

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 does not belong to any buffer pool

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.

ci_make_con

this function is called from the CI in order to copy a possibly incontiguous linked list of receive buffers into a contiguous area. If the function has been given an address of where to copy the data, it does so and releases the original buffer. Otherwise the address of the RBD belonging to the first buffer is computed and then *copy_rbd* of the driver layer called to perform the copying. If that is successful, it writes the new buffer address (which may be the same as the old) at the address nbuf. If the try

failed, it writes NULL there and returns an error code.
If the original buffer is not needed anymore (the message has been copied, or copying is not possible) it is released.

the function has the following parameters:
bufx *buf... address of the original buffer
bufx **nbuf... pointer to the pointer to the buffer. Points to NULL if the buffer could not be copied or was already empty in the first place.
char *mem... address of memory to copy buffer into
int if_num ... number of the network interface

the return value is 0 if everything was OK or
COULD_NOT_BE_MADE_CONTIGUOUS if there was no possibility of copying the data into a contiguous memory area.

forward_rpk

this function is called from the driver layer. It gets a pointer to a received message. If this message is intended for the KI it is made contiguous and then handed to the SSCMP. The same procedure is performed for an NI message. CI messages are simply handed up.

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

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

ret_ni_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
int if_num ... number of the interface

the return value is 0 if everything was OK or
INVALID_BUF_ADDRESS if the address does not belong to any buffer 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 65) 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 <u>cl</u>.

  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 <u>rfa</u>.

  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. 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 2.1 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 [LANCE92, 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
{
  INT16 sw;
  INT16 cw;
  INT32 link;
  INT32 tbd_add;
  INT16 eof_tcbcount;
  INT16 zero;
  addr  da;
  INT16 length_field;
  INT32 area;
  ti_acx tst;
  INT32 packet_info;
  INT16 unused[INFOCNT];
} ts_tcbx;


typedef volatile packed struct
{
  volatile INT16 sw;
           INT16 cw;
           INT32 link;
           INT32 tbd_add;
           INT16 eof_tcbcount;
           INT16 zero;
           addr  da;
           INT16 length_field;
           INT32 packet_info;
} 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 not transmitted with either type of message. It contains information about what is to be done with the frame after transmission.
The 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 written directly 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 [LANCE92, 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 2.1 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:

```
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;
  timex utcsu[MAXASICS];
  INT16 free;
} rfdx;
```

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 rts;
  INT32 diff;
} timex;
```

where rts is the *receive timestamp* of the NTI (written into the RFD by *nti_isr*, see Section 2.3) and diff the transmission delay from source to destination (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 68k are decoded in such a way that a timestamp is being triggered when a certain offset within a 64 byte block is reador written by the LANCE, though not at a CPU access. 4k of those are reserved for timestamped transmit commands (TSTCBs) and 64k for receive frames (RFDs), see Figure 3.2.

Given that both, TCBs and TSTCBs are 64 bytes long, there are 64 timestamped envelopes and 1024 receive frames available. Proceeding from this precondition the 184k of Header and Buffer space have been distributed among the nun-timestamped TCBs, TBDs, RBDs and buffers. The initial idea was to assign about double as many TBDs respectively RBDs (and the according buffers, of course) as there were TCBs + TSTCBS respectively RFDs. However, since 2048 RBDs plus buffers require rather more than the available space, the requirements were reduced to about as much RBDs as RFDS. The remaining space was distributed between TCBs and TBDs with their buffers according to the 2:1 principle. The result were 501 TCBs, 1046 TBDs

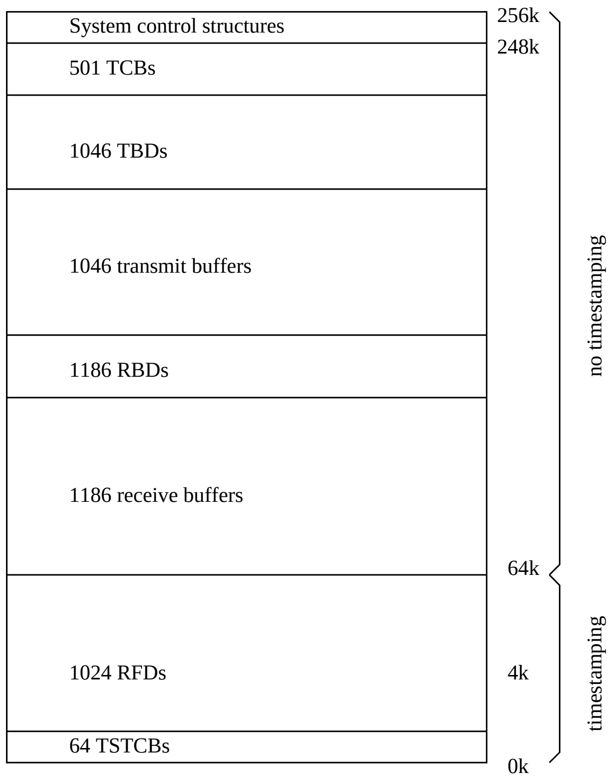

Figure 3.2: Partition of the shared memory area on the NTI

and transmit buffers and 1186 RBDs and receive buffers. (If you care to recalculate this you will find that 12 bytes remain unassigned.)
The remaining 8k 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 probably be requested rather frequently and hence have to be provided in multiple instances to avoid unnecessary contention.

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 contiguous. Therefore a chain of TBDs has to be taken from the pool. To facilitate this, the TBDs are organized in an array. An *allocation table* of the appropriate size shows which of the TBDs are in use and which are free. If a TBD is free, the corresponding entry in the allocation table is 1, otherwise it is 0. When allocating a buffer the NTI-driver looks in the allocation table for a chain of 1s of sufficient length, sets those entries to 0 and returns the address of the TBD that corresponds to the address of the found chain to the caller.

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 1s is sought. When it is found, those 1s are set to 0 and the address of the first TBD (having the same offset in its array as the first 1 of the found chain in the allocation table) returned to the user.

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 sets the corresponding slots in the allocation table to 0.

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.

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. Since the LANCE works asynchronously to the NTI-driver it may happen, that, when appending an RFD to the list of free ones, the last one in that list is being filled by the LANCE. In that case, the ISR processing received packets will have to restart the RU with the newly appended (list of) free RFDs.

The organization of the RBDs is a bit more complex. The LANCE itself does not require the entire space it writes the data into to be contiguous. It is handed a list of free RBDs (*RBDL*) 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. Therefore it is not practical to simply append released RBDs to the list in arbitrary order. They have to be reinserted at their appropriate (successive) places to make tha buffer space available to the LANCE as contiguous as possible, which reduces the need to copy data from incontiguous chains of buffers into contiguous ones. To facilitate this all RBDs are organized in a circular buffer, i.e., an array, where the next address after the highest one is wrapped around to the lowest one. This circular buffer can be parted into three zones:

- *Zone 1* This zone contains RBDs that are free (in the RBDL) and (perhaps) some that are still in use of higher layers or clients. Its limits are the point in the RBDL where the LANCE has prefetched the next RBD for reception and the end of the RBDL, pointed to by *RBDL_END*.

- *Zone 2* This zone contains RBDs that are all in use of higher layers or clients. It is limited by beginning and end of the RBDL, but it covers that part of the ringbuffer that lies outside the RBDL and is hence unavailable for the LANCE. The beginning of the RBDL is pointed to by *RBDL_BEGIN*.

- *Zone 3* This zone contains RBDs that have been newly filled by the LANCE and those still under use of higher layers or clients. It stretches from the beginning of the RBDL to the point where the LANCE has marked an RBD as prefetched.

The limit between zones 1 and 3 cannot be reliably determined by the NTI-driver since it depends on the (asynchronous) execution of the LANCE, so that they will be taken together as zone 1/3. It thus covers the same area in the circular buffer as the RBDL, reaching from its beginning to its end.

As has been said above, the buffers of a (possibly incontiguous) list returned to the RBDL have to be inserted one by one into the list at their appropriate positions. First they are all initialized. Afterwards the area of the circular buffer covered by the chain is checked RBD by RBD, all free RBDs encountered this way are linked together, thus inserting them into the list.
Since the LANCE works asynchronously to the NTI-driver and cannot be stopped, inspecting the circular buffer in the same direction as the LANCE writes its data can cause race conditions, as pointers in the RBDL have to be redirected. Therefore the NTI-driver proceeds in the other

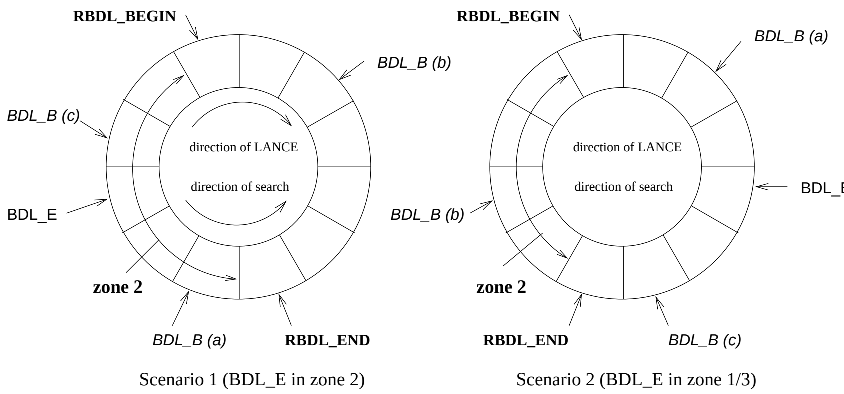

Figure 3.3: The scenarios possible when inserting a list of RBDs into the RBDL

direction when returning RBDs. It starts with the last RBD of the (reinitialized) list (*BDL_E*) and proceeds towards its first one (*BDL_B*).

First the zone where the insertion starts has to be determined (see Figure 3.3 for comparison). If BDL_E lies within zone 2 all free RBDs are collected, starting with BDL_E, until either RBDL_BEGIN or BDL_B is reached. If the former happens, the scenario relates to 1(c). Proceeding from the current position the first free RBD in Zone 2 is sought and marked as the new RBDL_END.
In the other case the situation conforms to scenario 1(a) or 1(b). 1(b) requires no further action but appending the collected list to the first free RBD that is encountered, 1(a) requires that all RBDs up to RBDL_END are collected and thus appended to the RBDL. In all cases, RBDL_END has to be moved.

Should BDL_E lie within zone1/3 the search for free RBDs is started with RBDL_END and goes to RBDL_BEGIN. Situation 2(a) is thus taken care of. If a check determines that the current scenario relates to 2(b) or 2(c), however, the collection has to go on. Starting with the RBD lying directly "before" RBDL_BEGIN the search is continued to the first free RBD or RBDL_END. If RBDL_END has been reached, no further action needs to be taken (since the collection started with RBDL_END and thus the whole list has been inserted already). In the other case, though, the newly found free RBD is marked as the designated RBDL_END and the collection continued until RBDL_END is encountered. Then the designated RBD becomes the new RBDL_END.

Despite the efforts put into the management of the RBDL it could of course happen that a received packet is spread over non contiguous buffers. Since all clients usually require their messages to be contiguous, though, the NTI-driver has to provide a possibility to render a buffer so. This is done by allocating another, contiguous chain of RBDs long enough copying the data into it and releasing the old chain after completion. The search for a contiguous chain starts at RBDL_END and proceeds in the opposite direction of the LANCE to avoid race conditions, as usual. A pointer (*END*) is set to the first free RBD found. From there on a counter is increased with each free RBD that lies directly in front of the last one. If a prefetched RBD is encountered, the search is aborted since the LANCE has already filled all free RBDs. If a full one is encountered, the next free RBD is looked for and the search starts anew. If the counter has reached the appropriate number, the NTI-driver has to "secure" the found area. First it sets a pointer (*MARK*) to the last RBD found to mark the beginning of the whole chain. Then it looks for the previous RBD (in front of MARK in the RBDL) and checks the position of END. If it is identical to RBDL_END, the "previous" RBD is marked as RBDL_END. Otherwise the next free RBD behind END is located and its address entered into the link field of the "previous" RBD. If the "previous" RBD is prefetched or filled after having set its link field, the procedure is aborted, since the LANCE has already reached this area. Otherwise the data can be copied from the old list into the new one without any problems. The old list is released.

copy_rbd

This function copies the data from an incontiguous buffer into a contiguous one. First it checks whether copying is necessary. If yes, it uses the algorithm described above. The address of the new chain of RBDs is returned to the caller, the original chain is released. If copying was not necessary, the return value is the address of the original chain of RBDs.

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

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.

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. Clearly, RFDL_BEGIN points to the first received message. RBDL_BEGIN is moved to the first RBD in the RBDL behind the last RBD belonging to the RFD pointed to by RFDL_BEGIN. The packet is handed up to the manager layer. 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 RBDs to return to the RBDL. After reinitialization it inserts them into the RBDL 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 buffer belonging to the TBD is cleared.

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 TBDs are kept in an array. An allocation table of the same size shows which TBDs are free and which are currently in use. A "1" in the slot with the same index as the TBD indicates that the TBD as free, a "0", that it is in use. The pointers to beginnings and ends of the lists are set to their appropriate values and the index for starting search of free TBDs in the allocation table is set to 0.

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 <u>cw</u> where the command TRANSMIT and the flags indicating flexible mode and interrupt generation after command execution are set, <u>tbd_add</u> which is set to EMPTY to indicate a non valid link to the LANCE, <u>eof_tcbcount</u> which is set to 24 so that the LANCE takes the <u>destination address</u>, <u>length_field</u>, <u>area</u> and the timestamp from the TSTCB during transmission and <u>link</u> 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 (see Section 2.3)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 <u>rfa</u> 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.

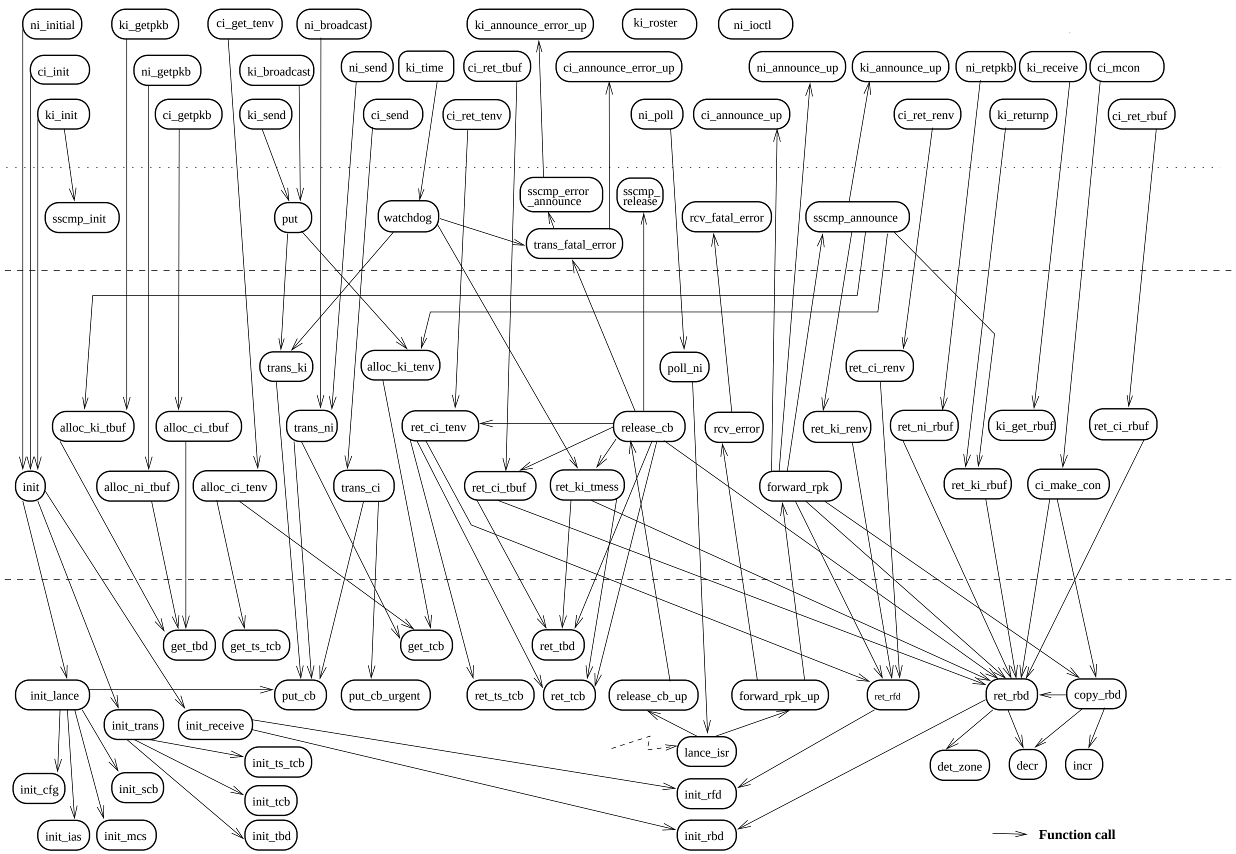

Figure 4.1: Structure of NTI-driver Functions

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
- own_v.c declaration of all the global variables of the NTI-driver

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

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 necessary for all function files and some other include files as well.
- **appl.h** definitions of middle layer data structures and constants
- **ci.h** definitions and prototypes for the clock synchronization

- **error.h** definitions of the error codes
- **ki.h** KI specific definitions
- **lance.h** all data structures and constants describing the hardware
- **sscmp.h** data structures and constants for transmission protocol

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

NODE_ANCHOR

Value: 0x44

Description: This is the address of the main configuration table of pSOS^{+m}, where the addresses of the configuration tables of all components are entered. The address has been taken from the manual.

MAX_NNO

Value: 3

Description: The maximum possible number of active nodes in the system, i.e, the number of nodes in the system. To be entered into the multiprocessor configuration table.

MAXIF

Value: 32

Description: The maximum number pNA^{+} may use as its interface number.

IFCNT

Value: 2

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.

MAX_ADDR_NO

Value: 6

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

CFG_CNT

Value: 14

Description: The number of bytes needed for configuring the LANCE.

SSCMPDS

Value: 16

Description: Number of bytes of a kernel buffer that contain protocol data.

CI

Value: 0x1000

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.

real_setintvec(nr,vec)

Value: (*((void(**)())(nr*4))=vec)

Description: This macro enters the address given in vec into the interrupt vector table, as number nr.

UP

Value: 1

Description: Signifies the NTI-driver is up and running. Needed for MIB-II support.

DOWN

Value: 2

Description: Signifies the NTI-driver is down. Needed for MIB-II support.

TESTING

Value: 3

Description: Signifies the NTI-driver is still testing. Needed for MIB-II support.

APPL.H

KQ_LENGTH

Value: 200

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

SSCMP_PACKET

Value: 0x93

Description: This value identifies an acknowledge message of the transmit protocol.

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.

MAXASICS

Value: 4

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

DEFA

Value: 0

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

ADDRCNT

Value: 6

Description: Number of bytes in a hardware address.

BROADCAST_ADDRESS

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

Description: The hardware address for broadcast.

B_SIZE

Value: 64

Description: Number of bytes in a buffer.

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.

CI_MCON

Value: 8

Description: Code for CI function ci_mcon.

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: -4

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

INVALID_ADDRESS_NUMBER

Value: -5

Description: The node identified by a given number is either not active or not existent.

NO_SPACE_IN_WINDOW

Value: -6

Description: There was no room in the transmit protocol structture to insert this message. Too many messages are already waiting for their acknowledges.

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.

KI.H

INIT_FC

Value: 1

Description: Function code for initialization.

GETPKB_FC

Value: 2

Description: Function code for allocating buffer space.

RETPKB_FC

Value: 3

Description: Function code for returning receive buffer space.

SEND_FC

Value: 4

Description: Function code for transmitting a message.

BROADCAST_FC

Value: 5

Description: Function code for broadcasting a message.

RECEIVE_FC

Value: 6

Description: Function code for receiveing an announced message.

TIME_FC

Value: 7

Description: Function code for timer called every tick.

DUMMY_FC

Value: 8

Description: This number is left free in the specification.

ROSTER_FC

Value: 9

Description: Function code for maintaining NTI-driver's roster.

INITIAL

Value: 0

Description: If this value is passed to the function maintaining the roster, the other value handed over is the initial roster table.

JOIN

Value: 1

Description: If this value is passed to the function maintaining the roster along with a node number, that node has joined the system.

LEAVE

Value: 2

Description: If this value is passed to the function maintaining the roster along with a node number, that node has left the system.

ki_standard_size

Value: 100

Description: Size of a KI message buffer.

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.

MIRROR_D

Value: 0x40000

Description: The distance in address space between the two parts of the shared memory area, where the lower address triggers a timestamp and the other does not.

DUMP_AREA_SIZE

Value: 322

Description: Number of bytes the LANCE writes when performing a DUMP.

SELF_TEST_AREA_SIZE

Value: 24

Description: Number of bytes the LANCE writes when performing a DIAG.

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.

TSFILLCNT

Value: 20

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

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.

RU_STATES

Value: 0x00F0

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

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_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.

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.

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.

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.

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).

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.

SYSBUS_VALUE

Value: 0x00000054

Description: This is the value that is written into the ISCP to preconfigure the LANCE. This configuration means interrupt is active high, lock is disabled (otherwise there arise problems with accesses via VME-Bus), triggering is internal and the LANCE should use the linear addressing mode.

EMPTY

Value: 0xFFFFFFFF

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

DESCR

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.

TYPE

Value: 17

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

MTU

Value: 1500

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

SPEED

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.

HW1

Value: 0x08

Description: The first byte of the ethernet hardware address.

HW2

Value: 0x00

Description: The second byte of the ethernet hardware address.

HW3

Value: 0x3E

Description: The third byte of the ethernet hardware address.

HW4

Value: 0xFF

Description: The fourth byte of the ethernet hardware address.

HW5

Value: 0x00

Description: The fifth byte of the ethernet hardware address.

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.

CA_ADDRESS_1

Value: 0xFFF46004

Description: Address of the register on the MVME162 mapped to the CA pin of the LANCE.

PORT_ADDRESS_1

Value: 0xFFF46000

Description: Address of the register on the MVME162 mapped to the PORT of the LANCE.

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.

LANCE_ICR_ADDRESS_1

Value: 0xFFF42028

Description: Address of the interrupt control register of the LANCE on the MVME162.

LANCE_IVBR_ADDRESS_1

Value: 0xFFF42000

Description: Address of the interrupt vector base register on the MVME162.

IC_REG_MASK

Value: 0xFFFF00FF

Description: Mask covering all those entries in the interrupt control register not of interest to the LANCE.

INT_LEVEL

Value: 0x06

Description: Level of the LANCE interrupt.

EDGE_SENSITIVE

Value: 0x40

Description: Interrupt happened upon occurance of an edge.

INT_EN

Value: 0x10

Description: Enables interrupt.

HIGH_PLRTY

Value: 0x00

Description: High polarity.

I_CLR

Value: 0x08

Description: Clears the pending interrupt bit.

SNOOP

Value: 0x40

Description: enables bus snooping, so that caching cannot work around you.

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.

ZWEI

Value: 2

Description: ID of zone 2 in the ringbuffer of RBDs.

EINSDREI

Value: 3

Description: ID of zone 1/3 in the ringbuffer of RBDs.

NONE

Value: 10

Description: Meaning that no zone is needed.

FREECOUNT

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)%16)

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.

SSCMP.H

ARRAY_SIZE

Value: 0x10

Description: The number of messages that may wait for their acknowledgements, i.e., the size of the window in the sliding window 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.

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

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;
  int  madrc;
  valx *madrl;
  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), the number of multicast addresses in the table (madrc), the address of the table itself(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.

APPL.H

envx

```
typedef packed struct
{
  INT8  reserved[16];
  addr  da;
  INT16 length_field;
  INT32 packet_info;
} envx;
```

This structure is, strictly seen, the one of a TCB with all the fields that are of no interest to the higher layers "taken out". Da is where the destination hardware address is written to, length_field is for the type of the message (KI, CI, ARP or IP) and packet_info contains information necessary for the handling of the envelope, such as should it be returned immediately after transmission, or not.

CI.H

ti_acx

```
typedef struct ti_acx
{
  INT32 ts;
  INT32 ms;
  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 rts;
  INT32 diff;
} timex;
```

The receive timestamp information is differently structured. Only the timestamp (rts) is inserted into the receive frame. The reason for this is that the clock synchronization can easily get the associated macrostamp from the local NTI afterwards, should it be necessary. Diff is only filled during the testing phase where multiple virtual nodes are situated on one node and a transmit timestamp is being pulled on the receiving node (see Section 2.3). It then contains the difference between the transmit timestamp and the receive timestamp as measured on the receiver's clock.

bufx

```
typedef INT8 bufx[B_SIZE];
```

This is the structure of a receive or transmit buffer. Nothing but B_SIZE bytes space for data.

addr

```
typedef unsigned char addr[ADDRCNT];
```

The structure of an Ethernet hardware address.

tenvx

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

The reduced structure of both, TCB and TSTCB. It is reduced to the fields that are needed in the upper layers and are common to both, envx and ts_envx. <u>da</u> contains the destination address and <u>length_field</u> the type of the message.

ts_envx

```
typedef packed struct
{
  INT8  reserved[16];
  addr  da;
  INT16 length_field;
  INT32 tstamp[4];
  INT32 packet_info;
} ts_envx;
```

Reduced structure of a TSTCB for use of the clock synchronization. <u>da</u> contains the destination address and <u>length_field</u> the type of the message. <u>tstamp</u> is reserved for the area (4 bytes) and the timestamp information (12 bytes, see ti_acx). (Area denotes the virtual node the message "came from"). <u>Packet_info</u> contains internal information regarding the handling of the command block.

renvx

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

Here the original structure was an RFD. Sa is the source address of the message, length_field its type. Utcsu contains the receive timestamps of the different areas, each on its virtual node.

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 incoming message to the clock synchronization.

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.

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 [LANCE92, p. 4-153ff] for details on the various fields.

nopx

```
typedef packed struct
{
  INT16 cw;
  INT16 sw;
  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 cw;
  INT16 sw;
  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 cw;
  INT16 sw;
  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 [LANCE92, 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 cw;
  INT16 sw;
  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 packed struct
{
  INT16  cw;
  INT16  sw;
  INT32  link;
  INT32  tbd_add;
  INT16  eof_tcbcount;
  INT16  zero;
  addr   da;
  INT16  length_field;
  ti_acx tst;
  INT32  area;
```

```
  INT32  packet_info;
  INT8   unused[TSFILLCNT];
} ts_tcbx;
```

Ts_tcbx 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 is always 0. Da is the destination address, lenght_field the type of the message. 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 next field, packet_info is not transmitted and only used for storing information of what to do with the TSTCB after transmission. 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.
There are multiple copies of this command block in shared memory, since it is used fairly often.

tcbx

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

Tcbx 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. The next field is always 0. Da is the destination address, lenght_field the type of the message. The next 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.

tdrx

```
typedef packed struct
{
  INT16 cw;
```

```
  INT16 sw;
  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 [LANCE92, 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 cw;
  INT16 sw;
  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 cw;
  INT16 sw;
  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.

messx

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

```
  INT16 eof_tcbcount;
  INT16 zero;
  addr  da;
  INT16 length_field;
} messx;
```

This data structure is used when the information contained in a cbx is not sufficient to determine the exact command block structure. That is, if it is a transmit command, where there is still the possibility of the block being a TCB or a TSTCB. As can be seen, the structure contains all fields of a transmit command but packet_info. The first three fields are the same 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 is always 0. Da is the destination address, lenght_field the type of the message.

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.

rfdx

```
typedef volatile packed struct
{
  INT16 cw;
  INT16 sw;
  INT32 link;
  INT32 rb_add;
  INT16 eof_full_actc;
  INT16 size;
```

```
  addr  da;
  addr  sa;
  INT16 length_field;
  timex utcsu[AREACNT];
  INT16 free;
} rfdx;
```

The RFD can be described with this structure. The first three fields are analoguous to the first three of a command block. The fourth, rb_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 last field has been added only to make the RFD exactly 64 bytes in size, which is necessary due to NTI hardware requirements.

scbx

```
typedef volatile packed struct
{
  INT16 cw;
  INT16 sw;
  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 [LANCE92, 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.

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).

SSCMP.H

awx

```
typedef struct
{
  char  repeats_til_now;
  INT16 time_to_next_repeat;
  envx  *env;
  bufx  *buf;
  INT16 size;
  char  flag;
  char  acked;
} awx;
```

This structure contains the data necessary to store a message waiting for its acknowledgement. That includes first the count of retransmits in repeats_til_now. Then the counter until the next retransmission in time_to_next_repeat. 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. If acked is $\neq 0$, the message has been acknowledged by the destination node, otherwise it is still waiting.

waitx

```
typedef struct
{
  char ack_index;
```

```
  char last_sent;
  char rec_index;
  awx  ack[ARRAY_SIZE];
} 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. Ack is the actual array where the messages are stored.

senvx

```
typedef packed struct
{
  INT8  reserved[16];
  addr  da;
  INT16 length_field;
  INT32 packet_info;
} tenvx;
```

The protocol needs the field packet_info, which is not available to the other functions of this layer, for storing information. da contains the destination address and length_field the type of the message.

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_broadcast
  - 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_getpkb
  - ni_init
  - ni_ioctl
  - ni_poll
  - ni_retpkb
  - 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_mcon
  - 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
- ERROR.C is very short since it contains nothing but the two error handling functions.
  Functions
  - rcv_fatal_error
  - trans_fatal_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
  - 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
  - init
  - ki_get_rbuf
  - poll_ni
  - rcv_error
  - ret_ki_rbuf

- ret_ki_renv
- ret_ni_rbuf
- ret_ci_rbuf
- ret_ci_renv
- ci_make_con

- LANCETR.C contains the functions responsible for handling data structures of the transmit part and serving the CU.
  Functions
  - get_tbd
  - get_tcb
  - get_ts_tcb
  - init_cfg
  - init_ias
  - init_lance
  - init_mcs
  - 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
  - copy_rbd
  - decr
  - det_zone
  - forward_rpk_up
  - incr
  - init_rbd
  - init_receive
  - init_rfd

  - lance_isr
  - ret_rbd
  - 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

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 buffer the message is contained in may be incontiguous. In order to render it contiguous, the clock synchronization has to call *ci_mcon*. If *ci_mcon* is not able to render the buffer contiguous, it returns an error message and releases the buffer.
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
- CI_MCON... make a buffer contiguous

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_mcon

Prototype:

int ci_mcon(bufx *buf,bufx **nbuf,char *memory,int if,int size)

Description:

Calls a manager layer function to render a buffer contiguous. Mind, that the original buffer is returned by the NTI-driver(if it is not needed any more, i.e., if it was incontiguous, or the clock synchronization specified a memory address where to write the buffer). It must not be released (again) by the clock synchronization! Therefore, if *ci_mcon* is called from *ci_announce*, that function must not return the value IMMEDIATE_RELEASE.

Parameters:

bufx *buf... address of the original buffer
bufx **nbuf... address to write the address of the new buffer
char *memory... address of memory to copy buffer into
int if... number of the interface
int size... size of buffer

Return:

int... error code, is 0 if everything is OK, or
INVALID_IF_NUMBER if the parameter was invalid
COULD_NOT_BE_MADE_CONTIGUOUS if rendering contiguous was not possible

Callers:

ci_dispatcher ... entry point for commands from clock synchronization

Calls:

ci_make_con... renders a buffer contiguous

Global Variables:

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_broadcast

Prototype:
INT32 ki_broadcast(INT32 size,INT32 unused,bufx *buf)

Description:
This function is a relic of older pSOS^{+m} versions, the current one does not need it anymore It has been kept since the code for this function is not (yet) for another one. Usually broadcasting is realized by using the special broadcast address (in Ethernet an address with all 1s). Since the transmit protocol cannot handle such messages, the broadcast is split and a copy of the message sent explicitly to each node in the net.

Parameters:
INT32 size... size of the data buffer
INT32 unused... this register is of no moment for this function
bufx *buf... address of the data buffer

Return:
INT32... error code, is 0 if everything is OK, or
NO_ENV_FREE if a message could not be sent due to no more envelopes
NO_BUF_FREE if a message could not be sent due to no more buffers
INVALID_IF_NUMBER if KI is not initialized
NO_SPACE_IN_WINDOW if a message could not be sent due to protocol problems

Callers:
pSOS^{+m} kernel

Calls:
alloc_ki_tbuf... allocates buffer and descriptor, returns buffer address
memcopp... copies specific amount of bytes from one data area to another
put... enters message into protocol data structure

Global Variables:
int ki_inited... R, TRUE if KI is initialized

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, ≠ 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 transmission timeout counters.

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

Global Variables:
int ki_inited... R, 0 if KI has not been initialized yet, ≠ 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:
Calls function for sending a message with the broadcast address as the destination address.

Parameters:
nibrdcast *param... parameters for broadcast function
char *buff_addr... address of buffer
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. 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:
1 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_getpkb

Prototype:
INT32 ni_getpkb(struct nigetpkb *param)

Description:
Takes a buffer large enough to write the given number of bytes into from the pool and hands its address up to the pNA$^+$. If no buffer was availablle the returned address is NULL and the return value an error code. This error code has to be -1 according to the NI specification.

Parameters:
nigetpkb *param... parameters for allocating a buffer
long count... required size of the buffer
long *hwa_ptr... pointer to destination hardware address
long if_num... number of the interface

Return:
INT32... error code, is 0 if everything is OK, or
NO_BUF_FREE (-1) if no buffer was available

Callers:
pNA$^+$

Calls:
alloc_ni_tbuf... takes the buffer from the pool

Global Variables:
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_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.

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
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_retpkb

Prototype:

INT32 ni_retpkb(struct niretpkb *param)

Description:

Returns receive buffer to the pool.

Parameters:

niretpkb *param... parameters for returning buffers
char *buff_addr... address of the buffer
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
EINVALID if one of the parameters was invalid
(defined in error.h)

Callers:

pNA$^+$

Calls:

ret_ni_rbuf... returns a receive buffer to the pool

Global Variables:

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_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.

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:
waitx *waiters... W, structure for storing the data of not yet acknowledged messages
confx *conf... R, structure containing cofiguration parameters of the NTI-driver

memcopp

Prototype:
int memcopp(void *a,void *b,int no)

Description:
This is one of the functions that have been implemented to replace the needed library functions since they are not available while pSOS^{+m} is inactive. The NTI-driver may be active when pSOS^{+m} is not and consequently those library functions may not be used in the NTI-driver. Byte by byte a specific amount of data is copied from the source area to the destination area.

Parameters:
void *a... pointer to the destination area
void *b... pointer to the source area
int no... number of bytes that are to be copied

Return:
int... is always 0

Callers:
ni_ioctl... function for supporting MIB-II
ki_broadcast... splits broadcast into singlecasts
sscmp_announce... checks received messages, acknowledges and announces them
put... partly compiles a message, enters it into transmit protocol structure
ki_send... called py pSOS^{+m} to send a message

Calls:
nothing

Global Variables:
none

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. Afterwards the first slot in the transmit protocol structure behind the one containing the message that was last sent is checked. If it is free, the message is entered and the counters of this slot set thus, that the next occurence of *watchdog* will transmit it (time_to_next_repeat and repeats_til_now are both set to 1). *Watchdog* won't be called, though, while pSOS^{+m} is not active. Therefore the message has to be sent explicitly by *put* itself in that case. Such a message can only be the join request of a slave node. If it is lost, it won't be retransmitted (since *watchdog*, which handles the timers and retransmits) is not called until the kernel is fully active) and the node remains locked in the startup procedure.
put manipulates the transmit protocol structure and therefore could come into conflict with other instances of itself or with *watchdog*. It is assumed, that pSOS^{+m} does not call *ki_send* again when one call is in the process of being executed, so that no two instances are ever active at the same time. The parts where race conditions with sscmp_announce or watchdog could arise have to be 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
ki_broadcast... for broadcasting a kernel message

Calls:
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:
waitx *waiters... RW, structure containing 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 flag in the transmit protocol structure that marks a message as being currently sent. The place in the control structure is encoded in the field packet_info of the envelope.

Parameters:
envx *env... address of the envelope of the message

Return:
void

Callers:
release_cb_up... releases executed commands

Calls:
nothing

Global Variables:
waitx *waiters... RW, structure containing 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 (easily identifiable by its length, or rather lack of it) the corresponding slots in the protocol data structure are marked as acknowledged. The message buffer and envelope are returned.

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:
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:
waitx *waiters... RW, structure for storing data of messages not yet acknowledged
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 each tick. It first releases the acknowledged messages that are not currently being resent (a message that is being sent is identified by a flag). Afterwards it decreases the field time_to next_repeat of each slot, (i.e. the timer until retransmission) by 1. If it has reached 0, the counter of resends (i.e. the field repeats_til_now) is increased. If it has not reached the maximum value MAX, the message is simply resent. Otherwise a transmission error is reported. 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 6
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:
waitx *waiters...RW, structure for storing data of messages not yet acknowledged
confx *conf...R, structure containing cofiguration parameters of the NTI-driver

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

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:

Compiles a transmit command from the given data (the destination address has already been written into the envelope by the clock synchronization) and then appends it to the queue of outgoing messages. 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. If the destination address is the address of a node, the counter of outgoing singlecast messages is increased, otherwise the counter of outgoing multi- or broadcast messages is raised by 1.
If any of the parameters is invalid nothing further is done but returning an error.
If the size of the buffer is so small, that the outgoing packet would be smaller than the minimum packet size, that packet is padded with zeros until it reaches the required size.

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
NO_ENVELOPE_ATTACHED if tenv is NULL
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:

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:
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
int OutBytes[IFCNT]... W, number of bytes sent
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

trans_ki

Prototype:
int trans_ki(envx *env,bufx *buf,INT16 size)

Description:
Compiles a transmit command from the given data and inserts it into the queue of outgoing messages. If the destination address is the address of a node, the number of sent singlecast messages is increased, if not, the number of sent multi- or broadcast messages is raised by 1.
If any of the parameters is invalid, nothing further is done but returning an error.
If the size of the buffer is so small, that the outgoing packet would be smaller than the minimum packet size, that packet is padded with zeros until it reaches the required size.

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 <u>tenv</u> is not the address of an existing envelope
INVALID_BUF_ADDRESS if <u>buf</u> 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:
put_cb... inserts a normal command into the queue
ucast... determines if 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
int OutBytes[IFCNT]... W, number of bytes sent
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, fills it with the given data and inserts it into the queue of outbound messages. If no envelope is available or one of the parameters is invalid, the number of discarded messages is increased and an error returned. If the 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, that packet is padded with zeros until it reaches the required size.

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
NO_ENV_FREE if no envelope was available
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
int OutBytes[IFCNT]... W, number of bytes sent
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.

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

ci_make_con

Prototype:

int ci_make_con(bufx *buf,bufx **nbuf,char *mem,int if_num)

Description:

First it checks whether <u>mem</u> is NULL. If it is not, the data is copied into it from the original buffer, which is released afterwards. If it is NULL, *ci_make_con* calls a function to render the buffer indicated by <u>buf</u> contiguous. The address of the contiguous buffer is then written at <u>nbuf</u> (note that the value may be different from the address of the original buffer) and the return value is 0. If the try failed, the buffer is released and an error message returned to the clock synchronization. The address at <u>nbuf</u> is then NULL. In all cases the incontiguous buffer is released.

Parameters:

bufx *buf... address of the original buffer
bufx **nbuf... address to write the address of the new buffer
char *mem... address of memory to copy buffer into
int if_num... number of the network interface

Return:

int... error code is 0 if everything is OK, or
COULD_NOT_BE_MADE_CONTIGUOUS in case of an error

Callers:

ci_mcon... called by clock synchronization to make buffer contiguous

Calls:

copy_rbd... checks contiguity of a chain of RBDs and tries to copy it into a contiguous one, if necessary
ret_rbd... releases RBD and buffers

Global Variables:

int InDiscards[IFCNT]... W, number of inbound messages discarded due to resource problems
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

forward_rpk

Prototype:

void forward_rpk(rfdx *rfd,INT16 bsize,int if_num)

Description:

Gets the recently filled RFDs taken from the list of free RFDs and shares them out between the application interfaces. In the case of kernel or pNA$^+$ message it tries to render them contiguous before forwarding them. If that was successful, the message is discarded. Messages are only forwarded to the intended application interface if that has been initialized and is therefore active.

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

Return:
void

Callers:
forward_rpk_up... handles receive data structure lists

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$^+$
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

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. Due to prudence more than a real need, that part of the function has been secured by raising the interrupt mask.
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 until the variable init_done is set and then return the value the initializing function has returned and entered into init_val.

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

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
hw_init... initializes UTCSU and VME-Bus, has to be provided

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 init_done[IFCNT]... RW, TRUE if hardware initialization is done
int init_val[IFCNT]... RW, contains return value of hardware initialization if that is done
int ActiveAs[IFCNT]... W, number of active memory areas on this interface

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.

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.

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:
get_no...returns the number of a node with a given address
forward_rpk...processes executed commands
ucast...determines whether a given address is singlecast

Calls:
nothing

Global Variables:
none

poll_ni

Prototype:
int poll_ni(int if_num)

Description:
Calls the ISR of the LANCE to simulate the handling of an interrupt. This function is called by NI if pSOS^{+m} is not active and therefore no interrupts can be handled. In order to make the ISR 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 6
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

ret_ni_rbuf

Prototype:
int ret_ni_rbuf(bufx *buf,int if_num)

Description:
Checks the validity of the address of the receive buffer. If it is correct, the function 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:
ni_retpkb... called by pNA^+ to release receive buffer

Calls:
ret_rbd... release chain of RBDs and buffers

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, 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. Due to wraparound it will surely reach it, eventually.

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

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 are now marked as used in the allocation table.
There may arise race conditions between different instances of this function. It is possible that two *get_tbd* allocate overlapping chains of TCBs. 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 chain of TBDs 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 6
enable... sets interrupt mask to previous value

Global Variables:
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas
int begin_alloc[MAXASICS][IFCNT]... RW, starting point for search in the allocation table
INT8 alloc_t[TB_COUNT][MAXASICS][IFCNT]... RW, allocation table

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 6
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 6
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. Essentially the TBD is cleared, apart from the pointers to the next TBD and to the buffer.

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

memclr

Prototype:

void memclr(char *ptr,int size)

Description:

Clears the specified number of bytes starting at the given address.

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_tbd. . . initializes a TBD and its buffer
init_rfd. . . initializes an RFD
init_rbd. . . initializes an RBD

Calls:
nothing

Global Variables:
none

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 6
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 6
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 6
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 fo a chain of TBDs. Each of them, together with its buffer, is initialized and returned to the pool. Its slot in the allocation table is set to "1". The TBDs are initialized before they are marked as free in the allocation table, so that no race conditions arise.

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
INT8 alloc_t[TB_CNT][MAXASICS][IFCNT]… W, allocation table

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 6
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

copy_rbd

Prototype:

rbdx *copy_rbd(rbdx *rbd,int if_num)

Description:

Tests if a chain of RBDs beginning at rbd forms a contiguous buffer space. If it does not it tries to copy the data into such a space, beginning with its search for it at rbdl_end and working towards rbdl_begin, in the direction contrary to the LANCE. The function manipulates the ringbuffer containing the RBDs and the RBDL. Therefore it has to be secured against other instances of itself, *ret_rbd* and *forward_rpk_up*. This is done with the help of raising the interrupt mask. If the chain need not be copied, since it is already contiguous, rbd is returned.

Parameters:

rbdx *rbd... address of the first RBD of the incontiguous chain
int if_num... number of the network interface

Return:

rbdx *... address of the contiguous chain, or NULL if no copying was possible

Callers:

forward_rpk... processes the received messages
ci_make_con... called to render clock synchronization messages contiguous

Calls:
incr. . . returns address of the next RBD in front
decr. . . returns address of the next RBD behind
ret_rbd. . . returns RBDs disable. . . sets interrupt mask to 6
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
scbx *scb[IFCNT]. . . R, address of SCB

decr

Prototype:
rbdx *decr(rbdx *rbd,int if_num)

Description:
Calculates the address of the next RBD in the other direction than the one in which the LANCE proceeds.

Parameters:
rbdx *rbd. . . address of RBD
int if_num. . . number of the network interface

Return:
rbdx *. . . address of next RBD behind rbd

Callers:
ret_rbd. . . releases chain of RBDs and buffers
copy_rbd. . . renders a chain of RBDs contiguous

Calls:
void

Global Variables:
sh_memx smem[MAXASICS][IFCNT]. . . R, addresses of shared memory areas

det_zone

Prototype:
int det_zone(rbdx *rbd,int if_num)

Description:
Returns the number of the zone in the ringbuffer the given RBD lies in. Zone 1/3 comprises the whole RBDL, i.e free buffers and those only recently filled by the LANCE, and also taken ones. zone 2 holds only RBDs currently being processed by higher layers or by applications. The borders between Zone 2 and the others are defined by rbdl_begin and rbdl_end. The border between zone 1 and zone 3 is defined by the RBD the LANCE is waiting at, i.e, the one it has just marked as prefetched. Since that cannot be determined exactly the border between zone 1 and zone 3 is an indeterminate affair.

Parameters:
int if_num. . . the requested network interface

Return:

int... EINSDREI if the RBD lies in zone 1/3, ZWEI else

Callers:

ret_rbd... releases a chain of RBDs and buffers

Calls:

nothing

Global Variables:

rbdx *rbdl_begin[IFCNT]... RW, address of first entry in the RBDL
rbdx *rbdl_end[IFCNT]... RW, address of last entry in the RBDL

forward_rpk_up

Prototype:

void forward_rpk_up(int if_num,INT16 state)

Description:

Takes the RFDs filled by the LANCE from the RFDL and the chain of RBDs belonging to this RFD from the RBDL and hands them up to *forward_rpk*, which shares them out between the application interfaces. Further action depends on the state of the LANCEs RU. IF it is active, nothing else is done. If it is inactive due to resource errors, an attempt is made to provide RFDs and RBDs. If that is not possible, the error handler is called. The function manipulates the ringbuffer containing the RBDs and the RBDL. Therefore it has to be secured against other instances of itself, *ret_rbd* and *forward_rpk_up*. This is done by raising the interrupt mask.

Parameters:

int if_num... number of the network interface
INT16 state... the status word of the SCB

Return:

void

Callers:

lance_isr... handles interrupt condidtion

Calls:

rcv_error... error handler
forward_rpk... shares out rceived messages between the application interfaces disable... sets interrupt mask to 6
enable... sets interrupt mask to previous value

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
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

incr

Prototype:

rbdx *incr(rbdx *rbd,int if_num)

Description:
Returns the address of the RBD lying next to the given one in the direction the LANCE proceeds.

Parameters:
rbdx *rbd... address of the RBD
int if_num... number of the network interface

Return:
rbdx *... address of the next RBD in the ringbuffer

Callers:
copy_rbd... renders a chain of RBDs contiguous

Calls:
nothing

Global Variables:
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

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

lance_isr

Prototype:
void lance_isr()

Description:
This function is the actual ISR servicing the LANCE interrupts. It checks each active network interface for interrupt conditions. (Whether an interface is active is determined by the global variable do_init.) 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. 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:
void

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:
int do_init[IFCNT]... R, 0 if the network interface is not initialized, $\neq 0$ otherwise
INT32 *licr[IFCNT]... R, address of LANCE interrupt control register
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

ret_rbd

Prototype:
int ret_rbd(rbdx *rbd,int if_num)

Description:
The chain of buffers is initialized and then inserted RBD by RBD into the RBDL. The function manipulates the ringbuffer containing the RBDs and the RBDL. Therefore it has to be secured against other instances of itself, *ret_rbd* and *forward_rpk_up*. This is done by raising the interrupt mask.

Parameters:
int if_num... number of the network interface

Return:
int... always 0

Callers:
copy_rbd... renders a chain of RBDs contiguous
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
ci_make_con... called to render a message received by the clock synchronization contiguous
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 6
enable... sets interrupt mask to previous value

det_zone... determines the zone an RBD lies in
decr... returns the address of the next RBD to the fore
do_something_else... passes time
init_rbd... initializes RBD

Global Variables:
rbdx *rbdl_begin[IFCNT]... RW, address of first entry in RBDL
rbdx *rbdl_end[IFCNT]... RW, address of last entry in RBDL
sh_memx smem[MAXASICS][IFCNT]... R, addresses of shared memory areas

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 6
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

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 6. 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. 7, the highest level, was not acceptable, since a level 7 interrupt may interrupt another of the same level, so that tha whole idea of atomicity would have been lost again. 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

User's Handbook

5.1 Introduction

The NTI-driver has been specifically designed for working with pSOS^{+m}, pNA^{+} and the clock synchronization. What has to be done to link all together is described in this section. The functions of the driver (described the previous chapters) are not directly accessible for the user. They are invoked by pSOS^{+m} when the user calls functions for accessing a remote global object (see Section 2.1.1 for details on RSCs) or by pNA^{+} when the application wants to set up communication with another node via sockets. The third client, the clock synchronization is transparent to the user and therefore not accessible for him.
Each interface returns the encoded error status of a function call. pSOS^{+m} and pNA^{+} treat these messages depending on context and circumstances. It may be relayed directly to the user, or cloaked in another error code, or may cause pSOS^{+m} to stop altogether.
Since pNA^{+} sometimes relays the returned error to the user, its error codes have been used in the NI part of the NTI-driver. The defintions can be found in the library file error.h. See Section 4.3 for a list of all possible error messages and Section 4.5 for details which function may return which error codes.

Memory Requirements

The NTI-driver requires memory for the transmit protocol waiters structure, its allocation table for the TBDs, the queue of messages to be received by pSOS^{+m}, a bunch of global variables and the data structures for reception and transmission of packets. Of those, the latter two are located on the NTI (see Section 2.1) and hence do not use system memory.

The requirements of the remaining variables are:

- **Waiters** needs 16 bytes per message entry and 4 additional bytes per possible node, 16*ARRAY_SIZE+4 per node.
  In a system where 20 messages may be waiting for their acknowledges and there are up to 3 active nodes, this table has a size of approximately 1k.

- **Allocation_table** is a table of bytes. Its size is dependent on the number of interfaces, maximum virtual nodes and TBDs. The formula for calculating the memory rquirements is: TB_COUNT*MAXASICS*IFCNT.

A node with 200 TBDs, 2 network interfaces in a system where there may be up to 4 virtual nodes per physical node therefore needs 1600 bytes of space for the allocation table.

- **Kiq** serves for keeping the addresses of waiting messages. One entry needs, since it is an address, 4 bytes of space. The size of the whole array depends on the maximum number of pending messages. For up to 100 pending messages, the size of the array would be 400 bytes.
- Additionally the NTI-driver needs about 1.1 kB per hardware interface for its global variables such as pointers, counters etc.

5.2 Installation

The NTI-driver has been designed for and implemented and tested in the environment described in Section 1.2.1 and Section 1.3. That means it runs on an MVME162-512A board, equipped with a MC68040 CPU and an i82596CA LAN coprocessor under pSOS^{+m} and pNA^{+}, in conjunction with the custom clock synchronization hard- and software developed at the Department for Automation at the Technical University in Vienna.

5.2.1 Software Configuration

As has been described in Chapter 2, pSOS^{+m} and pNA^{+} are configured by setting up the respective configuration tables. This is done at startup of the board. The respective configuration structures can be found in the files PSOSCFG.C and PNACFG.C of the board support package. pNA^{+} initializes at least partly at board startup, pSOS^{+m} at a later point in time, depending on the user. Therefore the NI is in all probability the first of the interfaces to be initialized.

pSOS^{+m}

The NTI-driver is connected to pSOS^{+m} by setting the fields of the multiprocessor configuration table and entering its address into the main configuration table. The *multiprocessor configuration table* (defined in PSOSCFG.C) contains the following fields, see [pSYS96Pr, p.3-5] for details:

unsigned long mc_nodenum Node number of this node; 1 is the master node, there is no node 0. The maximum number is limited by <u>mc_nnode</u>.

void (*mc_kicode)() Address of the KI's entry point.

unsigned long mc_nnode Number of nodes in the system. Must not be larger than 16383.

unsigned long mc_nglbobj The maximum number of global objects.

unsigned long mc_nagent The maximum number of agents per node.

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, 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.

void (*mc_roster) Address of roster change callout – is NULL here.

void *mc_dprext external starting address of dual port RAM - is NULL here.

void *mc_dprint internal starting address of dual port RAM - is NULL here.

unsigned long mc_dprlen length in bytes of dual port RAM

unsigned long mc_kimaxbuf maximum length of a KI buffer - is set to 100.

void (*mc_asyncerr)() asynchronous error callout - is NULL now, but an asynchronous error handler may be entered here. It is called when an error is reported as the result of an asynchronous RSC.

unsigned long mc_reserved[6] is unused and set to 0

pNA$^+$

Submitting a network interface to pNA$^+$ works similar to submitting one to pSOS^{+m}. In its configuration table, defined in PNACFG.C, there is one entry pointing to the connected interfaces. During initialization of pNA$^+$ an initial table with interfaces is loaded and entered into this array belonging to the pNA$^+$ configuration table. The structure of an entry in the table of interfaces is as follows, see [pSYS96Pr, p.3-32] for details:

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 should 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_NOARP 0x02…the interface has no ARP
IFF_POINTTOPOINT 0x10…the interface is a point-to-point interface
IFF_UP 0x80…the initial mode of the interface is up
IFF_MULTICAST 0x800…the interface supports multicasting
IFF_UNNUMBERED 0x1000…the interface is unnumbered
IFF_RAWMEM 0x2000…the interface uses pNA$^+$ data structure
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_NOARP, IFF_UP, IFF_MULTICAST and IFF_EXTLOOPBACK. IFF_POLL should be disabled, since otherwise difficulties may arise concerning the ISR servicing the LANCE.

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.

Software Configuration for Clock Synchronization

The last of the interfaces of the NTI-driver is the one to the clock synchronization. Upon compilation it has to include the file CI.H. On the other hand it has to be given the address of the entry point, i.e., the address of *ci_dispatcher*.

5.2.2 Hardware Configuration

The NTI-driver contains the necessary information to work with the LANCE on board the MVME162. At system installation the structure containing the configuration parameters of the NTI-driver and the LANCE has to be adapted to the current requirements in the file ROOT.C:

INT8 cfga[CFG_CNT] The configuration bytes for the LANCE. It is currently configured as an Ethernet network controller. Those bytes that deviate from the default value or are otherwise of importance are listed below. See [LANCE92, p.4-126ff] for exact bit and byte positions and meanings.

- **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 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 stay turned off due to difficulties that otherwise could arise 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 needed for testing the clock synchronization where one node simulates up to four ones. Messages have to be looped back to let the other virtual nodes receive messages originating from virtual nodes on the same 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 considered correct. By default the value is set to 64. This value should not be lowered, since 64 is the minimum frame length in an Ethernet.

char *descr Is the pointer to a string containing a short description of the system. This address is handed back up when *ni_ioctl* with the command SIOCGIFDESCR is called.

long type Is a number between 1 and 32 that signifies the type of the network. It is handed back up when *ni_ioctl* with the command SIOCGIFTYPE is called.

long mtu Is the maximum transmission unit. This is another value that is handed up when *ni_ioctl* is called with the appropriate command.

long speed Is the speed of the network. This number is handed up to pNA$^+$ as well. These last four values are solely for support of the MIB-II and of no consequence whatsoever for the NTI-driver.

addr ich This array contains the 5 high significant bytes of an Ethernet hardware address. Each node is given the least significant byte at initialization of NI or KI. This has been done to facilitate easier installation, since the addresses of other nodes can be calculated and need not be kept in a table.

int madrc The number of multicast addresses in the following table.

valx *madrl The address of a table of madrc multicast addresses. Those addresses are read once during intialization, when they are given to the LANCE.

int maxnodes The maximum number of nodes in the system.

Usually all the copies of the NTI-driver, the multiprocessor and the pNA$^+$ configuration tables will be basically the same on all nodes, apart from slight differences. In the multiprocessor configuration table the entry specifying the node number will be different.

5.3 Adding a Hardware Interface or an NTI

Adding a hardware interface to the configuration requires changes in the include files ALL.H and LANCE.H. In the following text, x is the number of the new interface decreased by 1.

In ALL.H the constant IFCNT has to be increased by 1.

In LANCE.H the addresses for the signals Channel Attention, Port Access and for the configuration registers of the interrupt on the MVME have to be defined. Those are CA_ADDRESS_x, PORT_ADDRESS_x, LANCE_ICR_ADDRESS_x and LANCE_IVBR_ADDRESS_x.
In the file LANCETR.C the global variables containing those values have to be expanded. Those variables are: ca for CA_ADDRESS_x, port for PORT_ADDRESS_x, licr for LANCE_ICR_ADDRESS_x and ivbr for LANCE_IVBR_ADDRESS_x.

5.4 Design Difficulties

During its initialization sequence pSOS^{+m} enforces an interrupt mask of 7, thereby prohibiting other interrupts, such as the LANCE's to be processed. 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 at this point.

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 that is done in an ISR which cannot become active. Obviously this design was intended for environments where no special action of the ISR is necessary to prepare a message that has been received (e.g. shared memory). Since this is definitely not the case with the NTI-driver, the solution to the problem was to lower the interrupt mask to 0 at each call of *ki_receive* as long as pSOS^{+m} is not active (as can be determined by the variable PSOS_FLAG). This allows the LANCE ISR to become active and deliver received messages.

Another problem during startup 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 may be initiated by *put* in case of pSOS^{+m} not being active, but there can be no retransmissions 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.

Chapter 6

Testing

6.1 Introduction

Testing of the NTI-driver was carried out in three phases. First the general workings were checked in a simulation of the final environment. The LANCE and the clients were replaced by tasks that had roughly the same functionality as their physical counterparts.

In the second phase the lower layers were tested by taking away the lower half of the simulation and using the NTI-driver with the actual hardware, retaining only the simulated clients. This was the phase were testing expanded to two seperate nodes.

The third step was to take away all simulations and link the NTI-driver with pSOS^{+m} and pNA^{+}. The NTI-driver was tested in a local area network consisting of three nodes, all situated on the same rack, with a testing application that is described in Section 6.4.

Due to those phases the challenges for devising the testing environment were as follows:

- The environment should be as transparent as possible for the NTI-driver. Ideally one should not have to change anything of its code when transferring it from one testing environment to another.
- The environment should be modular, so as to provide the possibility to disassemble it easily, using only a part of it for testing in phase two.
- Apart from that modularity also facilitates easy replacement of modules in case of changing requirements of hardware or clients.

According to these directives the testing environment was built of a number of tasks communicating by queues or global variables. Each task represents a module of the environment.

6.2 Full Simulation Testing

In the first phase of testing [1] the whole environment of the NTI-driver, as described in Chapter 3, was simulated by a number of tasks. Into these the NTI-driver was embedded as can be seen in

[1]Note that this environment supports only earlier versions of KI and NI.

Figure 6.1.
The lower ("hardware") half of the environment is represented by the tasks RU, CU, MSGL and a part of MAIN. The upper ("client") half is comprised of the other part of MAIN and of MSGH. The division of MAIN into two parts is easily explained. This task serves as the user interface and consists of mainly two functions, *control_high* and *control_low*, where the former serves for controlling the simulation of the clients and the latter for controlling the simulation of the LANCE and the network. The whole environment consists of the following files:

- **HAUPT.C** the initialization of the system and the main menu
- **CONTRH.C** *control_high* and the functions it needs
- **CONTRL.C** *control_low* and the functions it needs
- **MSGH.C** MSGH
- **MSG_GEN.C** MSGL
- **CU_EM.C** CU
- **RU_EM.C** RU

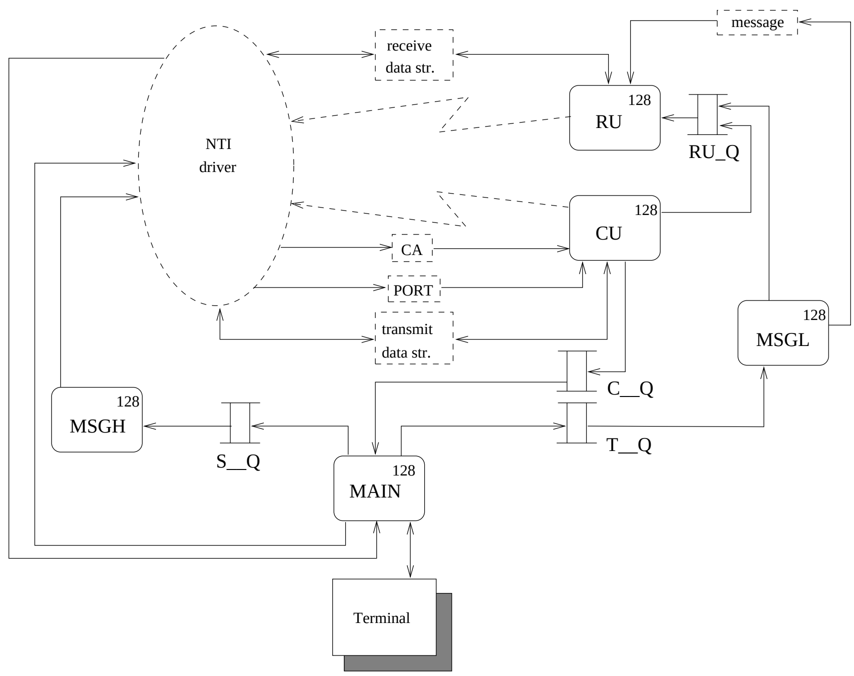

Figure 6.1: Task map of the testing environment including NTI-driver in phase 1

.

MAIN is the task started first. It creates and starts the remaining tasks and provides the necessary queues for them. It also calls the initialization procedures of each of the NTI-driver interfaces, KI, NI and CI. Having completed this startup procedure, MAIN offers the user the choice to perform functions checking the command part of the NTI-driver or those for checking the receive part. The former have been united under the name *control_high*, the latter under *control_low*.

control_high is the sum of functions simulating the clients of the NTI-driver. It uses MSGH for sending messages via the NTI-driver, but provides all other features, such as message announce entry points by itself. Additionally it offers means of viewing driver internal variables as a help for debugging. The user can chose from the following:

1. **Sending a message**
   There is the possibility of sending an automatically generated message. If the user wants to assemble it, he is asked for the relevant data, such as the type of the message (KI, NI, CI or Ethernet CI - this is a CI message with a non-timestamped envelope), the data to send (a simple string), the destination address (has no effect, but is entered in the envelope) respectively node number, dependent on the message type. If the message is to be a NI message, the type is asked as well. In the case of a CI message the user has to specify whether the message is to be of high priority and whether it should be realeased immediately after transmission.
   Having entered those data, or, in the case of an automatically generated message, after preparation of a KI message with destination node number 1 and a fixed data field, the user is asked how many times he wants that message to be sent and with how many ticks passing between two sends. If -1 is specified for the number of repeats, the message is sent by MSGH until it receives a STOP or a new message send request from *control_high*.
   The request is then sent to MSGH via the queue "S__Q". This queue takes all orders of MAIN to MSGH.
   MSGH sends the message using the functions the specified application interface (KI, NI or CI) provides. The NTI-driver puts them into the appropriate list and the whole thing is done for it. The entity processing the command blocks is the task CU that simulates the LANCE's command unit. It simulates transmission by putting copies of the transmit command blocks into the queue "C__Q" to *control_low* and the acknowledge numbers of KI messages into a global variable for use of *control_low* when generating an acknowledgement packet.

2. **Send STOP**
   The STOP signal is sent to MSGH which then aborts sending and waits for new orders.

3. **Listing of received messages**
   Messages that come in from the various interfaces are, in the case of CI and NI kept in the queue "R__Q", or, in the case of KI messages, not fetched until this feature is called. Then all the data of the messages that have come in are listed. After each message, the user is asked if he wants to forward it. If he does, he is requested to enter the destination address and then the message is handed to *MSGH* to send it (once).
   Having been printed on the screen and not resent by the user, a message is released and its frame and buffer returned. The user may abort viewing the messages any time.

4. **Listing error messages**
   The testing environment keeps counters for the different errors that may happen. The current number of those counters can be viewed by the user. The logged errors are the transmission errors of KI and CI and the errors of the RU, running out of buffers or running out of frames.

5. **Listing of data structures**
   The user can view various of the NTI-driver's global variables, to get an idea of its internal status. The variables are the RBDL, the <u>allocation_table</u>, the addresses of the different

shared memory areas, beginnings and ends of the CBL, the ACBL, the UCBL, the RFDL, the TCBL, the TSTCBL, the individual address of the node, the transmit protocol data structure, ...

6. **Polling for messages**
   The NI function *ni_poll* is called to poll for incoming messages. This is only practical to do if no interrupt was activated.

7. **MIB-II support**
   First the user has to specify which of a list of commands supported according to the MIB-II specification he wants to have performed. Then *ni_ioctl* is called and the result of that call printed to the screen.

control_low does not have as many features as its counterpart, since it simulates the LANCE by generating "received" messages, etc:

1. **Receive messages**
   This feature works very similar to the one for transmitting messages of *control_high*. There is another task for generating messages, however, (MSGL) and it sends the messages according to the specifications (time between receives, number of repeats) directly via the queue "RU_Q" to the task RU, that simulates the LANCE's receive unit. RU copies the messages into RFD and RBDs and generates a software interrupt after completion. Processing of the "received" frame is duty of the NTI-driver, which will ultimately put the message into "R__Q" or the structure for storing pending KI messages.

2. **Listing of transmitted messages**
   This feature is analogous to "listing received messages" in *control_high*.

3. **STOP sending**
   The signal STOP in the queue "T__Q" between *control_low* and MSGL causes MSGL to abort transmission.

4. **Sending acknowledgements**
   Here an acknowledgement message is generated to release the waiting messages from the transmit protocol structure. The appropriate number is taken from the global variable, provided by CU. The acknowledgement message is sent and received as a normal message.

The remaining tasks, MSGH, MSGL, RU and CU of the testing environment of this phase, perform the following functions.

MSGH is the "high level" message generator. It gets the specification of a message from MAIN via "S__Q" and then calls the appropriate functions of the interfaces to transmit it. It waits the specified time and calls the interface again for subsequent transmissions. It does this as often as has been specified or until a new message or a STOP comes in.

MSGL is the "low level" message generator. It performs similar to MSGH, but puts the message into "RU_Q" instead of calling the NTI-driver's interface to send it.

CU serves for simulating the command unit of the LANCE. It periodically checks the global variable <u>ca</u>. If it is set, it reads the SCB and performs the commands encoded there according to [LANCE92]. If the command concerns the RU, the command word of the SCB is sent to RU via "RU_Q".

Working through a CBL it copies the data from MCS, IA and CFG commands and ignores the remaining ones except transmit command blocks. For those it allocates memory to copy them along with their chains of TBDs and buffers. The address of that memory space is then put into "T_Q". If the transmit command block carries a kernel message the acknowledge number is entered into a global variable where it is read by the function generating an acknowledgement message.
After having processed a CB that had its interupt flag set or after having finished command list processing, the task sets the appropriate flags in the status word of the SCB and generates a software interrupt.

RU simulates the receive unit of the LANCE. It waits at "RU_Q" for a message to come in. If it is a command relayed from CU it acts according to [LANCE92]. If it is a message and the RU is active, it copies the data of the message into the RFD pointed to by its internal pointer and the next RBDs and buffers pointed to. Having finished this, it sets its internal pointers to the next RFD respectively RBD, sets the appropriate flags in the status word of the SCB and generates a software interrupt.

6.3 Hybrid Testing

Having completed the full simulation testing of the NTI-driver, the lower half (responsible for the simulation of the hardware) was removed and replaced by an actual LANCE [2]. This allowed us to check whether the procedures controlling the hardware were correct. Due to the modularity of our testing environment, this was relatively easy to accomplish. The structure of the environment was now as depicted in Figure 6.2.
The simulated environment was reduced to the two tasks ROOT and MSGH, contained in:

- **HAUPT.C** MAIN
- **MSGH.C** MSGH

All the others had become superfluous because of the hardware. The following files contain:

MAIN is the task started first. It creates and starts MSGH and provides the necessary queues for it and the incoming messages. It also calls the initialization procedures of each of the NTI-driver interfaces, KI, NI and CI. Having completed this startup procedure, MAIN offers the user a choice of commands to verify the behaviour of the NTI-driver in conjunction with the actual hardware. This choice of commands is exactly the same as the one of *control_high* in phase 1.

MAIN is the sum of functions simulating the clients of the NTI-driver. It uses MSGH for sending messages via the NTI-driver and provides all other features, such as message announce entry points by itself. Additionally it offers means of viewing NTI-driver internal variables as a help for debugging. The user can chose from the following:

1. **Sending a message**
   There is the possibility of sending an automatically generated message. If the user wants to assemble it, he is asked for the relevant data, such as the type of the message (KI, NI, CI or Ethernet CI - this is a CI message with a non-timestamped envelope), the data to send (a simple string), the destination address (has no effect, but is entered in the

[2] Note that this environment supports only earlier versions of KI and NI.

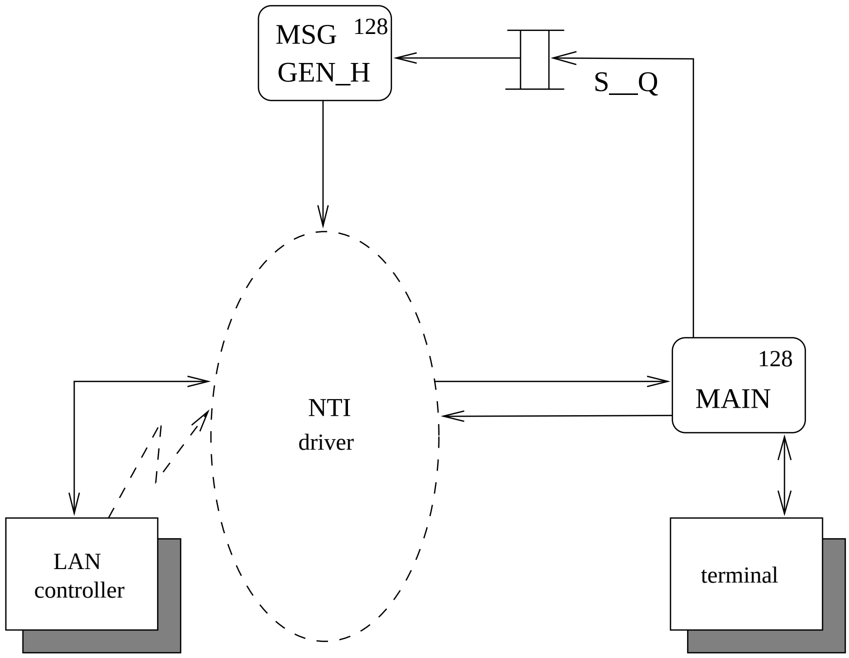

Figure 6.2: Task map of the testing environment including NTI-driver in phase 2

.

envelope) respectively node number, dependent on the message type. If the message is to be a NI message, the type is asked as well. In the case of a CI message the user has to specify whether the message is to be of high priority and whether it should be realeased immediately after transmission.
Having entered those data, or, in the case of an automatically generated message, after preparation of a KI message with destination node number 1 and a fixed data field, the user is asked how many times he wants that message to be sent and with how many ticks passing between two sends. If -1 is specified as the number of repeats, the message is sent by MSGH until it receives a STOP or a new message send request from *control_high*.
The request is then sent to *MSGH* via the queue "S__Q". This queue takes all orders of *MAIN* to *MSGH*.
MSGH sends the message a first time using the functions the specified application interface (KI, NI or CI) provides. The NTI-driver puts them into the appropriate list and the LANCE performs further processing as described in Chapters 2 and 3. Through use of the loopback feature outgoing message are immediately received and handed up to MAIN.

2. **Send STOP**
   The STOP signal is sent to MSGH which then aborts sending and waits for new orders.

3. **Listing of received messages**
   Messages that come in from the various interfaces are, in the case of CI and NI kept in the queue "R__Q", or, in the case of KI messages, not fetched until this feature is called. Then all the data of the messages that have come in are listed. After each message, the user is asked if he wants to forward it. If he does, he is requested to enter the destination address and then the message is handed to MSGH to send it (once).
   Having been printed on the screen and not resent by the user, a message is released and

its frame and buffer returned. The user may abort viewing the messages any time.

4. **Listing error messages**
The testing environment keeps counters for the different errors that may happen. The current number of those counters can be viewed by the user. The logged errors are the transmission errors of KI and CI and the errors of the RU, running out of buffers or running out of frames.

5. **Listing of data structures**
The user can view various of the NTI-driver's global variables, to get an idea of its internal status. The variables are the RBDL, the allocation_table, the addresses of the different shared memory areas, beginnings and ends of the CBL, the ACBL, the UCBL, the RFDL, the TCBL, the TSTCBL, the transmit protocol data structure, ...

6. **Polling for messages**
The NI function *ni_poll* is called to poll for incoming messages. This is only practical to do if no interrupt was activated.

7. **MIB-II support**
First the user has to specify which of a list of commands that are supported according to the MIB-II specification he wants to have performed. Then *ni_ioctl* is called and the result of that call printed to the screen.

MSGH is the message generator. It gets the specification of a message from MAIN via "S__Q" and then calls the appropriate functions of the interfaces to transmit it. It waits the specified time and calls the interface again for susequent transmissions. It does this as often as has been specified or until a new message or a STOP comes in.

Note finally, that this testing environment is not limited to one node. It can also be started on multiple nodes.

6.4 Full Hardware/Software Testing

In the final step of the testing procedure the NTI-driver was connected to its clients as well as to the hardware. This time the interfaces as well as the overall operation and performance were checked. For the latter purposes, the load put on the LANCE and thus on the NTI-driver was to be made as high as possible.
Since the NTI hardware as well as the clock synchronization client are still under development, they could not be linked to the NTI-driver yet. Another application using the supported CI features was put in its stead, and the timestamping feature simulated by using a counter/timer of the MVME162. The counter is started in *ci_init*. When a TSTCB is put into the outbound queue for transmission, the current value of the timer is written into the ts field of the timestamp structure of the TSTCB. The receive timestamp is read from the timer by *forward_rpk_up* and written into rts of the receive timestamp structure of the RFD.
Having linked the NTI-driver to pNA$^+$ and pSOS^{+m} the old testing environment used in phases one and two became entirely superfluous for testing, since it accesses the NTI-driver directly.
Another program for testing was developed for this phase, which was completely different in structure and functionality. The NTI-driver was tested running on three nodes, with one node the server and two other its clients. The task structure of the server can be seen in Figure 6.3

and the one of a client in Figure 6.4.
The file structure is the same on all nodes.

- **HAUPT.C** The initialization of the system and then the clock synchronization
- **PISOS.C** The tasks using the pSOS^{+m} features.
- **PENA.C** The tasks using the pNA^{+} features.

In brief, a sort of ping-pong was established for the tasks making use of pSOS^{+m}'s RSC features and the ones employing pNA^{+}'s sockets. In addition, the client parts of the stand-in for the clock synchronization request a number of timestamped replies at certain intervals from the server. Following you will find a description of the server node, followed by the description of a client node (the structure of the two clients is identical).

6.4.1 Master Node

As there are three different clients interfacing with the NTI-driver independently of each other, the testing environment has been split into three parts. One for testing the NTI-driver through the RSC features of pSOS^{+m}, one for testing it through the sockets of pNA^{+} and one as a stand-in for the clock synchronization.

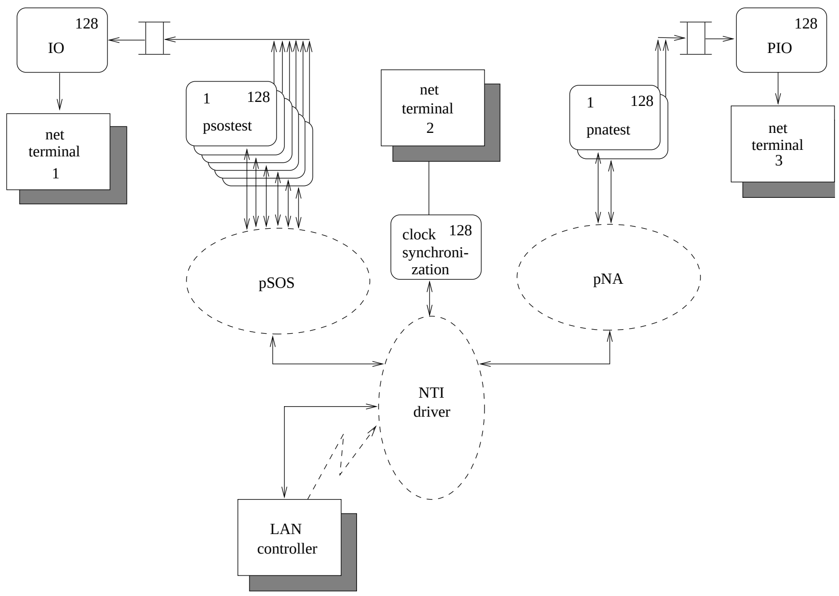

Figure 6.3: Task map of the server node of the testing environment including NTI-driver in phase 3.

The part that uses RSC features to test the functionality of the KI part of the NTI-driver consists of 6 tasks, each of them creating a global queue, making an RSC call to get the ID of its counterpart on the client node (the complements of the three queues being on one client node, the other three on the other) and then alternately waiting at the remote queue for a message and writing the message into the local queue to be receive by the complementary task. Each value received from the remote queue should be one higher than the last. If that is not the case, an error happened and the appropriate counter (there is one for each task) is incremented.

All the six tasks have been created and started from one template since what they have to do is exactly the same for all of them. Having exchanged 100 messages (i.e. one round) the number of the next round is printed to the screen. Since troubles may arise with so many tasks trying to write independently a single output task has been created. It gets the numbers from a queue where they were put by the application tasks and prints a message with those numbers to the screen.

The principle of the two tasks that communicate via sockets to test the NI part of the NTI-driver is the same as that of the pSOS^{+m} application tasks. Each of them communicates with a counterpart (on one of the two client nodes) using a pair of stream sockets. The server task creates two sockets, binds them to their addresses and then tries to connect the first one to its counterpart on the client node. It then waits for connection requests from the complementary task at the second socket. If both connections have been established, data exchange begins. The client node sends 1000 bytes of data to the server and then waits at the other socket for them to be sent back. The master node receives them, puts the number of bytes it has received into the queue for the output task that handles the printings of the applications using pNA^{+} (different from the one responsible for the output of the pSOS^{+m} applications) and then sends the whole packet back to the client via the second socket.
Since TCP/IP splits and reassembles packets, the size of the data received on the destination node may vary from the one of the data sent on the source node, though ultimately all the data should come through. This is checked the following way: Each of the 1000 byte blocks is filled with 4byte numbers, starting at 1 and incremented for the next block. On the receiving side the number of bytes received is logged, both are printed on the screen at each transfer. By comparing those two numbers one can observe the correctness of the connection.

The server part of the clock synchronization consists of a set of functions. Namely those, the clock synchronization has to provide for the NTI-driver - one for handling transmit errors and one for handling received messages. The former one increases an error counter and releases message envelope and buffer. The latter sends a reply to the source node of the message. That reply is sent with a timestamped envelope and with high priority. It contains the transmit and receive timestamps of the received message in its message buffer.
A task prints the number of messages received til now to the screen every five seconds.

6.4.2 Client Node

The partition into three independent testing applications has been continued on the client nodes. Each of those has an identical structure.

There are only three applications using pSOS^{+m}s RSC features. They follow nearly the same algorithm as their counterparts on the server node. They first create their local queues (as global objects). Then they get the IDs of the remote queues of their complementary tasks. Beginning with 1 they put the current number into the local queue and then wait at the remote queue for a message. If the received value is the same as the sent one, everything is OK, otherwise an error counter is increased. Having sent and received 100 times the next round is started and the number of rounds put into the queue where the values are stored for the output task to write to the screen.

On a client node there is only one instance of the task complementing the server node's socket application. It creates and binds two sockets and then waits for a connection request for the

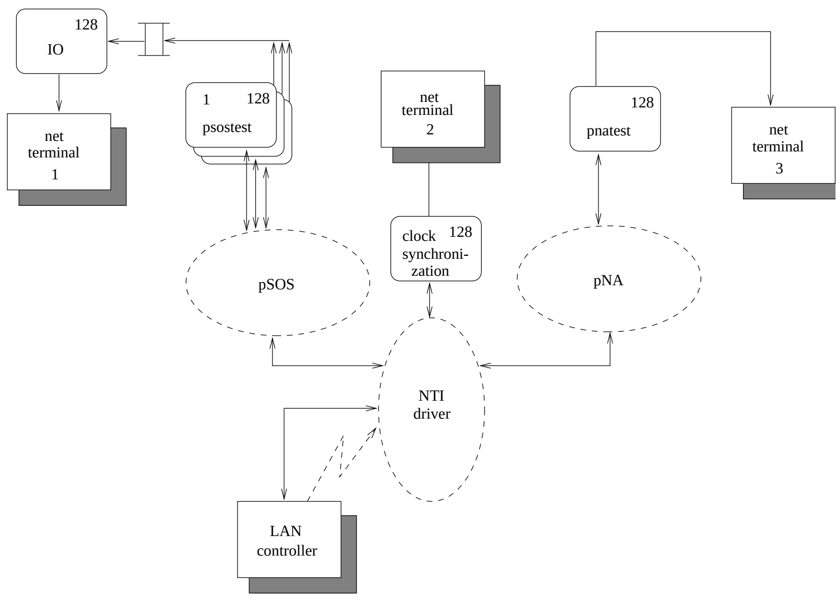

Figure 6.4: Task map of the client node of the testing environment including NTI-driver in phase 3.

first one (which comes from the complementary task on the server node). Afterwards it tries to connect its second socket to the second one on the server. As soon as both connections are up, it sends the first 1000 bytes. How many bytes are actually transmitted in the first packet depends on the network traffic and how TCP/IP split the data to be able to send them. Having sent the data the client waits at the second socket for them to be sent back. The number of bytes received and the number of send/resend cycles up to now is printed each time (directly) to the screen.

The client side of the clock synchronization is the one requesting the timestamped replies from the server. Every 50 ticks (or every half second) its sets off 100 requests and then waits a short time to see how many reply packages came back. The four timestamps collected in such a roundtrip (transmit sender, receive destination, transmit destination, receive sender) are stored in an array. After waiting for all messages the client calculates statistical data and writes them to the screen. Those are the average values, the minima and the maxima of roundtrip delay, of delay between reception on the server and transmission from it and of the sum of both transmission delays (calculated by substracting the delay on the server from the roundtrip delay) and the actual transmission delay provided by the feature described in Section 2.3.
Having printed the statistical data (directly) to the screen, the task waits half a second and then sets off the next bunch of requests.

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

[HSS97] M.Horauer, U. Schmid, K. Schossmaier. *NTI: A Network Time Interface M-Module for High-Accuracy Clock Synchronization*, manuscript submitted, 1997.

[LANCE92] Intel Corporation. *32-bit LAN Component User's Manual*, 1992.

[Mar84] K. Marzullo. *Maintaining the time in a distributed system. An example of a loosely-coupled distributed service*, PhD thesis, Department of Electrical Engineering, Stanford University, February 1984.

[Mil91] D.L. Mills. *Internet Time Synchronization: The Network Time Protocol*, IEEE Transactions on Communications, 39(10), p. 1482–1493, October 1991.

[pSOS96] Integrated Systems Inc. *pSOS System Programmer's Reference*, 1996.

[Pu95] A. Pusterhofer. *NTSU Network Timestamp Unit Ein Modul für Eventtimestamping und Uhrensynchronisation via LAN*, Diploma thesis, Department of Automation, Technische Universität Wien, 1995.

[pSYS96] Integrated Systems Inc. *pSOS System*, 1996.

[pSYS96Pr] Integrated Systems Inc. *pSOS System Programmer's reference*, 1996.

[RSB90] P. Ramanathan, K. G. Shin, R. W. Butler. *Fault-Tolerant Clock Synchronization in Distributed Systems*, IEEE Computer, 23(10), p. 33–42, October 1990.

[RTS92] P. A. Laplante. *Real-Time Systems Design and Analysis* IEEE Computer Socieaty Press 1992.

[Sch94] U. Schmid. *Synchronized UTC for Distributed Real-Time Systems*, Proc. IFAC Workshop on Real-Time Programming WRTP'94, Lake Reichenau/Germany, 1994,

[Sch95] U. Schmid. *Synchronized Universal Time Coordinated for Distributed Real-Time Systems*, Control Engineering Practice 3(6), p. 877–884, 1995. (Reprint of [Sch94]). URL: http://www.auto.tuwien.ac.at/~kmschoss/synutc.html

[Sch97a] U. Schmid (ed.). Special Issue J. Real-Time Systems on *The Challenge of Global Time in Large-Scale Distributed Real-Time Systems*, Journal of Real-Time 12(1–3), 1997.

[SP95] U. Schmid, A.Pusterhofer. *SSCMP: The sequenced synchronized clock message protocol*, Computer Networks and ISDN Systems 27(95), 1995.

[SS97] U. Schmid, K. Schossmaier. *Interval-based Clock Synchronization*, J. Real-Time Systems 12(2), March 1997.

[SWL90] B. Simons, J. Lundelius-Welch, N. Lynch. *An Overview of Clock Synchronization*, in B. Simons, A. Spector (eds.): Fault-Tolerant Distributed Computing, Springer Lecture Notes on Computer Science 448, p. 84–96, 1990.

[TCPIP94] W. R. Stephens. *TCP/IP Illustrated, Volume 1 The Protocols* Addison-Wesley 1994.

[YM93] Z. Yang, T. A. Marsland. *Annotated Bibliography on Global States and Times in Distributed Systems*, ACM SIGOPS Operating Systems Review, p. 55–72, June 1993.

Index