Eindhoven University of Technology

MASTER

Design of an X.25 co-processor

Daanen, J.M.V.

Award date:
1987

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technolgy
Department of Electrical Engineering
Digital Systems Group (EB)

Design of an X.25 co-processor

By J.M.V. Daanen

Master thesis report by : J.M.V. Daanen

Coach : Prof. Ir. M.P.J. Stevens

Eindhoven, The Netherlands
August 1987

The department of Electrical Engineering of the Eindhoven University of technology does not accept any responsibility regarding the contents of student projects and reports.

ABSTRACT

This report describes the architectural design of a coprocessor for X.25. Unlike earlier work on this subject in the digital systems group the OSI service concept is chosen as a start. The coprocessor should implement the level 1, 2 and part of the level 3 protocol.
Interfaces between the several building blocks are described. This gives a framework that is usefull to specify a universal protocol architecture.
Within this framework earlier work on the X.25 coprocessor can be fit.

Keywords
OSI, Protocol implementation, OSI service concept, HDLC, Data communications, X.25, co-processor

Acknowledgments

At this place I would like to thank all persons who have helped me during my study and specially those who helped me during my masters thesis project at the University

Special thanks go to the people who worked on this project, for their patience, ideas and help:
Paul Nissink
Prof. Ir. M.P.J. Stevens

Eindhoven, July 1987

Hans Daanen.

Contents:

INTRODUCTION

In the digital Systems group (EB) of the departement of electrical engineering of the Eindhoven university of technology, there is a research project on a general purpose data communication architecture. One of the goals of this architecture is to minimise the hosts overhead in protocol handling and implement several layers of the ISO-Open systems interconnection reference model.
Due to speed requirements there is chosen for a solution where the lower layers are implemented in hardware and the higher layers in software. For the hardware part a single chip coprocessor, or rather several sets of building blocks for a coprocessor have to be developed. This architecture can be used in a variaty of products from a simple ISDN telephone set to a complex Datanetwork exchange, or the communications card of a computer. (See figure 1)
The first part of this project is the design of an X.25 coprocessor and the software design to implement the layers 1, 2 and 3 of the OSI-model. At this moment another graduate student, Paul Nissink, is implementing the D-channel protocol for ISDN. The first goal of the project is to design a dedicated processor on which the layers 2 and 3 of X.25 and ISDN can bee implemented. The choise of the interfaces must be according to the OSI service definitions, to make it possible to change the layerprotocols (or implement other protocols) without any consequences for the other layers.

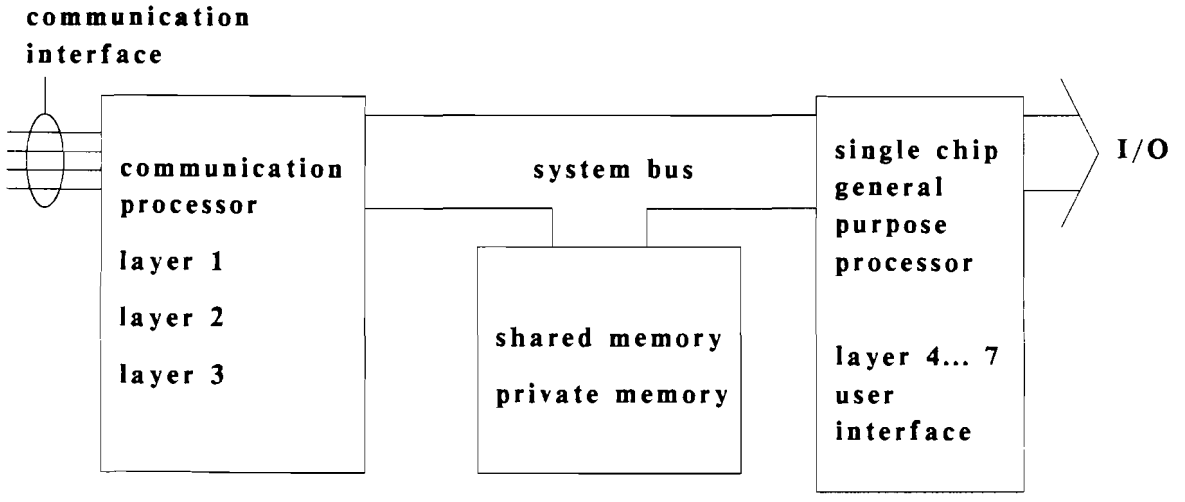


Figure 1a Minimum architecture

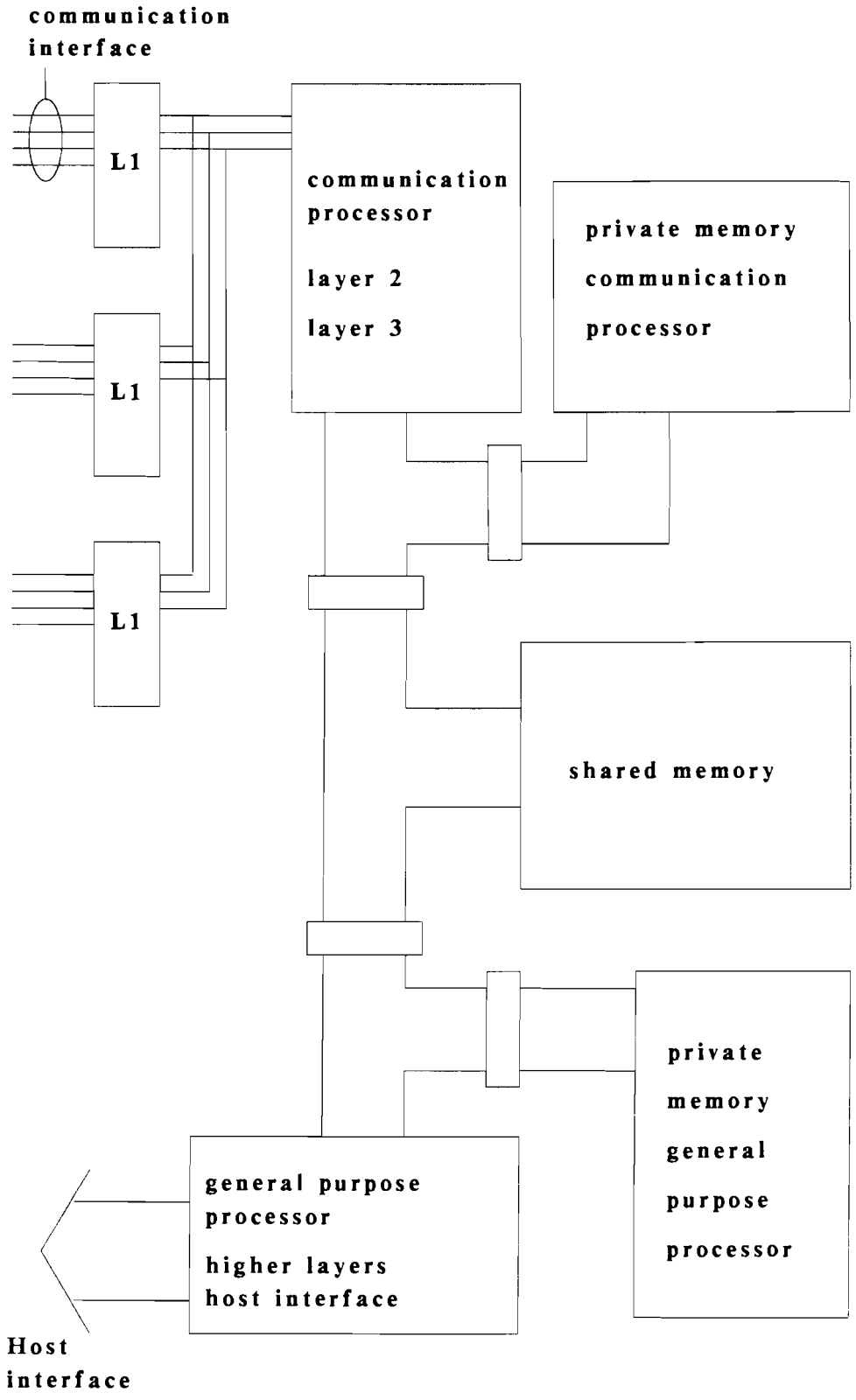


Figure 1b Maximum architecture

1. ISO-OSI seven layer model.

Computer systems maybe interconnected for various puposes, for example resource sharing and exchange of information. Datacommunication between two information systems is possible only when they use the same communication procedure. The communicating systems can have a different hardware structure, operating system or programming language, but in order to communicate they need common communication rules.

In 1978 the International Standards Organistaion (ISO) started work on the Reference Model for Open Systems Interconnection, well known as the ISO-OSI model. The objective of the OSI model is to enable systems from different manufacturers and generations, employing different hardware and software technologies, to exchange information. The reference model provides an architectural framework for the design of standards. The production of the standards is a complex and timeconsuming process.

One of the most important characteristics of the model is that it does not imply any particular systems implementation or use of technology. The model is therefore defined at a high level of abstraction. Which doesn't imply that it is useless for implementation purposes. On the contrary, because it is defined at a high level of abstraction, it can be used as a powerfull specification for a set of implementations as we will see later.

1.1 OSI concepts

To make the development of complex OSI-systems manageable a divide and conquer approach has been chosen. The model divides the communication problem into seven smaller, better manageable parts called layers. Every layer offers services to the adjacent higher layer and uses the services offered by the adjacent lower layer. Active elements within a layer are called entities. Active entities in the same layer can cooperate to offer a specific service (active entities in the same layer are called peer entities). This cooperation is accomplished by exchanging messages. The message-exchange is defined by one or more protocols.
The communication between a service-user and a service provider is realized using service-primitives. The point of interaction between a service-provider is called service-access-point (SAP).

1.2 The layers

As stated before, the model divides the communicating system into seven layers. Each layer has its own function. The OSI model consists of the following layers:

Layer
7. Application layer.
6. Presentation layer.
5. Session layer.
4. Transport layer.
3. Network layer.
2. Datalink layer.
1. Physical layer.

Layer 1-4 contain the communication oriented functions, while layer 5-7 contain the data processing oriented functions.

-Layer 1: Physical layer.

The bottom layer of the OSI model provides mechanical, electrical, functional and procedural characteristics to activate, maintain and deactivate physical connections between datalink entities. The layer provides functions for transferring bits. It takes care of the medium access procedure (CSMA/CD, control lines, tokens, etc.). It also provides errror indication to the datalink layer. The standards in this layer prescribe physical and electrical characteristics of connectors, signallevels and timing.

-Layer 2: Datalink layer.

The function of the datalink layer is to provide reliable transfer of blocks of data between adjacent systems. In order to realize this function it provides functions for connection establishment and release, error detection and recovery (including indication of non-recoverable errors), flow control and sequencing.

-Layer 3: Network layer.

The objective of the network layer is to provide a path for the transfer of information between end systems. It sets up a route by which packets travel and regulates the traffic (flow control). The network layer has been divided into 3 sublayers to allow routing between subnetworks and the levelling of the network service despite differences between subnetworks.

-Layer 4: Transport layer.

The transport service provides transparent end-to-end transfer of data between session entities, it offers a transmission service of constant quality independent from the quality of the underlying network. The transport layer is required to optimize the available communication resources to provide the performance required by each transport user at minimum cost. Transport layer functions include segmentation, error detection, error correction and multiplexing. The transport protocol specification describes five different classes of procedures to allow operation over network services of various quality.

-**Layer 5**: Session layer.

This layer provides the necessary means for cooperating presentation-entities to organize and synchronize their dialogue and manage their data exchange. It structures the dialoque by allowing full-duplex, half-duplex or simplex communication and by determining which user has the right to send. The session layer also performs the mapping from addresses to names.

-**Layer 6**: Presentation layer.

The presentation layer should allow for negotiation between application entities on a common syntax or common syntaxes to be used for the transfer of information between applications. The functions in this layer include connection establishment and release, definition and selection of transfer sysntax(es).

-**Layer 7**: Application layer.

The Application layer contains the totality of the application which is involved in OSI communication, an 'Application layer service definition' does not exist, since there is no higher layer. An entity in this layer is built using service elements which come in three types:

-User Elements
-Common Application Service Elements (CASE), which may be regarded as a toolbox of service elements for common use.
-Specific Application Service Elements (SASE), like Virtual Terminal Service (VT), File Transfer, Acces and Management (FTAM), Job Transfer and Manipulation (JTM), Database Access Languages, Command Languages (OSCRL) , Graphics standards (GKS) and Message Handling Systems (MHS like described in the CCITT X.400 recommendations).

This shows that the application layer is very 'large', each of these fuctions is defined by standard documents up to several hundreds of pages. In fact the application layer will never be completed because new user requirements will continually emerge.

1.3 Protocols and Services.

Every OSI layer is specified by two things:
1) The interactions between Peer-entities (Protocol)
2) The interaction between entities in different (adjacent) layers (Services)

The (N)-protocol, which is defined by synctactic, semantic and timing rules, bridges the gap between the service used and the service provided. Peer-entities exchange information via Protocol Data Units (PDU). These PDUs must be interpreted unambiguously by all communicating entities, that is why protocol standards define the bitcoding of the PDUs. They prescribe also the actions to be taken on receipt of a certain PDU, of course.

Since the peer entities are not physically connected they have to exchange these PDUs by using the service offered by the next lower layer. In order to communicate via the service provider (adjacent lower layer), a protocol entity utilizes so called "service primitives". A service primitive can be considered as an elementary interaction between a service user and the service provider during which certain values for the various parameters of the primitive are established to which both user and provider can refer. So a service primitive is not just a kind of message that is passed across a service boundry. In fact even bidirectional exchange of information can be possible, which allows a much richer variety of paramater value establishment than just value passing (one might think of a negotiation procedure, etc). The common boundry at which these interactions take place is called the "Service Access Point" (SAP).

In real systems these SAPs represent an internal boundry. To put as few as possible constraints on valid implementations the service primiteves are defined and expressed at a high level of abstraction. This gives the implementator the freedom to use any mechanism that he finds useful to realize the execution of the service primitives (eg. interrupt routines, procedure calls or hardware interfaces). If the concept of the primitive is implemented correctly, it should be a small problem to translate one implementation to another. If however the implementation does not follow the primitive definition it might be very difficult or maybe impossible to make an interface for two different implementations.

There are defined four subtypes of service primitives:

REQUEST is a primitive issued by a service user to invoke some service

INDICATION is a primitive issued by a service provider either to invoke some procedure or to indicate that some service has been invoked by a service user at a peer-SAP.

CONFIRM is a primitive issued by a service provider to indicate the completion of a previous issued request at that SAP. This completion may be succesfull or not, depending on the parameters exchanged.

RESPONSE is a primitive issued by a service user to indicate the completion of a previous issued indication at that SAP.

Only when the services are defined and implemented correct the major advantage of the OSI-model will appear. The services have to be independend of the protocol that is used to realize them. When this requirement is met, it means that one can implement a protocol for one layer and if it is proven correct it is useable in many applications. If there are several protocol implementations available for each layer, each having the same implementation of the corresponding services, making a specific application would be simple. Selecting the appropriate protocol implementations from a hardware or software library and put them together would do the job. Unfortunately this is not the way it is done at this moment, but with the implementation of the X.25 interface we hope to make a start.

The data flow through the OSI model can be described with so called Data Units. The relation between different data units is showed in fig. 2.

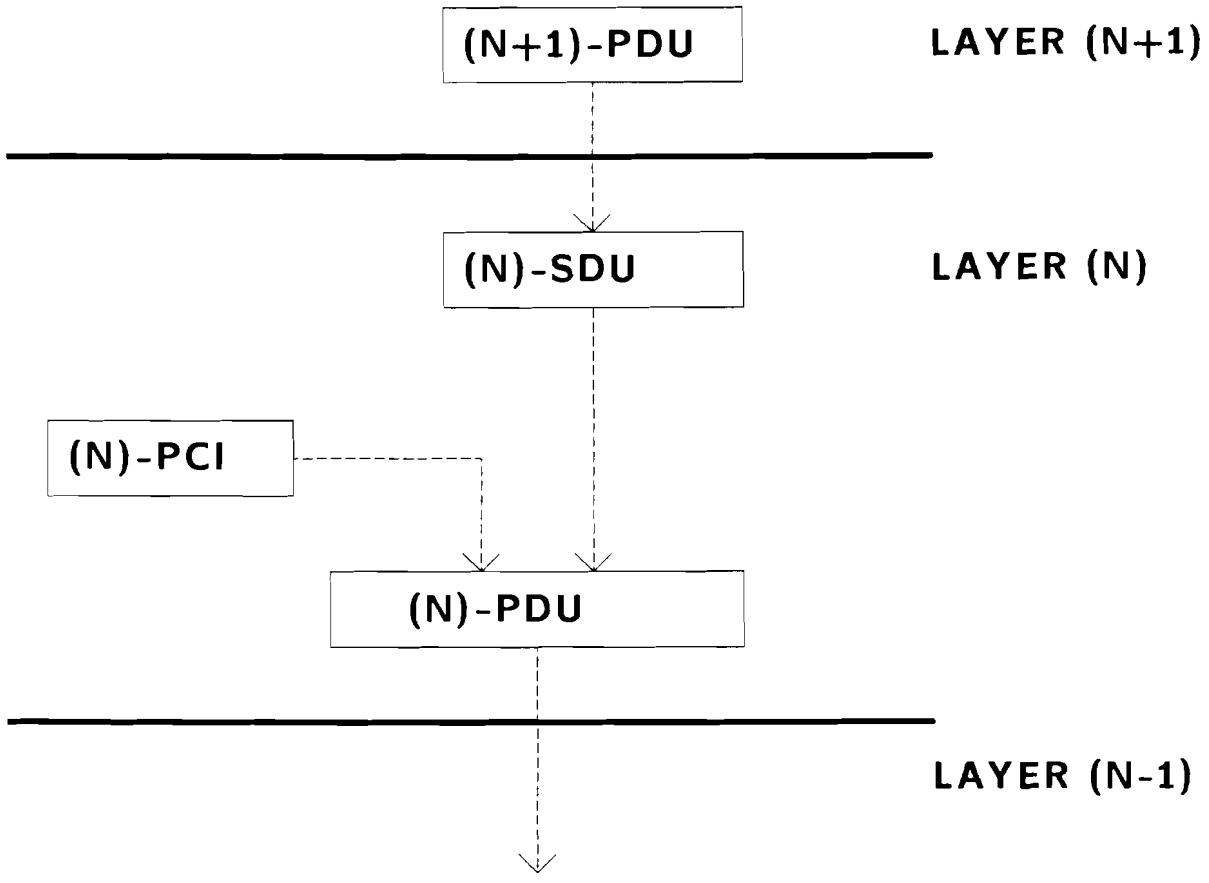


Figure 2 relation between the data units in the OSI model

In layer(N) the (N+1)-PDU is called the (N)-SDU (service data unit). Together with the protocol control information (PCI) the (N)-SDU forms the (N)-PDU. The (N)-PDU on its turn is passed to layer N-1 via the DATA-REQUEST primitive.
Apart from adding some protocol control information to an SDU there are several functions defined within the OSI model that the layers may implement, for example:

MULTIPLEXING: The sending/receiving of (N)-PDUs belonging to different (N)-connections over a single (N-1) connenction.

SPLITTING: The distribution of (N)-PDUs belonging to a single (N)-connection over multiple (N-1)-connections.

RECOMBINING: The reverse of splitting.

SEGMENTING: A function used to distribute a single (N)-SDU over multiple (N)-PDUs.

REASSEMBLING: The reverse of segmenting.

CONCATENATION: The grouping of multiple (N)-PDUs into a single (N-1)-SDU.

SEPARATION: The reverse function of concatenation.

All these functions put requirements on the way we pass data from one layer to another as we will see later.

1.4 Connection oriented/ connectionless services

Just the definition of all services is not sufficient to define the interface between two layers. Another aspect that has to be defined is the ordering in which the services can occur. Within OSI there are two different "schools" of communication. These differences can be noticed in all aspects of the OSI model. The two classes of communication resulting from this different view are the connectionless mode and the connection oriented mode. There are service definitions for either mode. These services are not compatible. The connection oriented mode is the ordering of the services communication oriented. That is first the call has to be established before data can be transferred. After finishing the datatransfer the call has to be cleared explicitly. In the connectionless mode the services are "user oriented". The data can be sent as it comes, the layers do not use call setup and clear procedures.
Of course it is possible to define a protocol which uses services of one mode and provides services to higher layers belonging to the other mode, but most protocols also belong to one mode. This difference in opinion dates from before the OSI model. In the very first packet switching protocols we find the datagram and virtual circuit oriented systems. The same difference occurs with the connection oriented operation and the connectionless oriented operation. The connectionless mode accords to the datagram protocols. When a layer wants to send data it simply passes this data together with addresses via a service primitive to the lower layer. No setup procedures are required. The communication flow is shown in fig 3

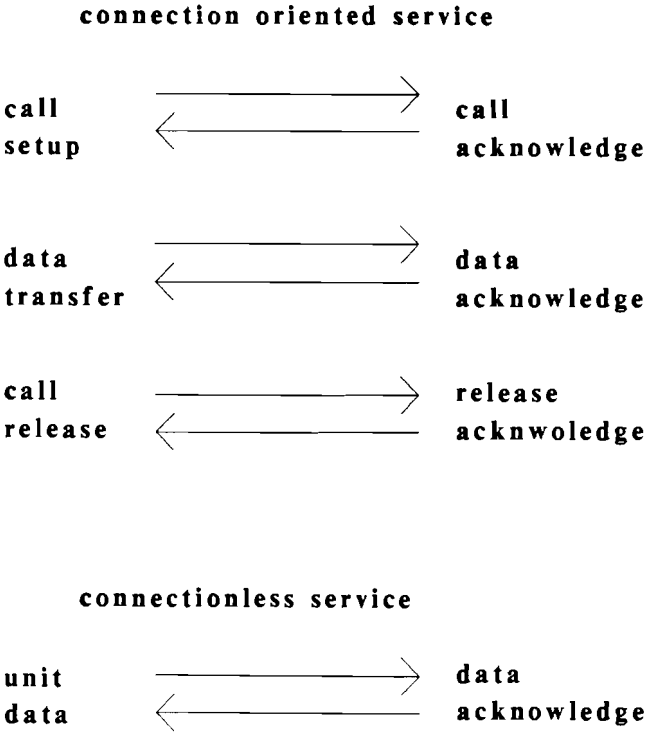
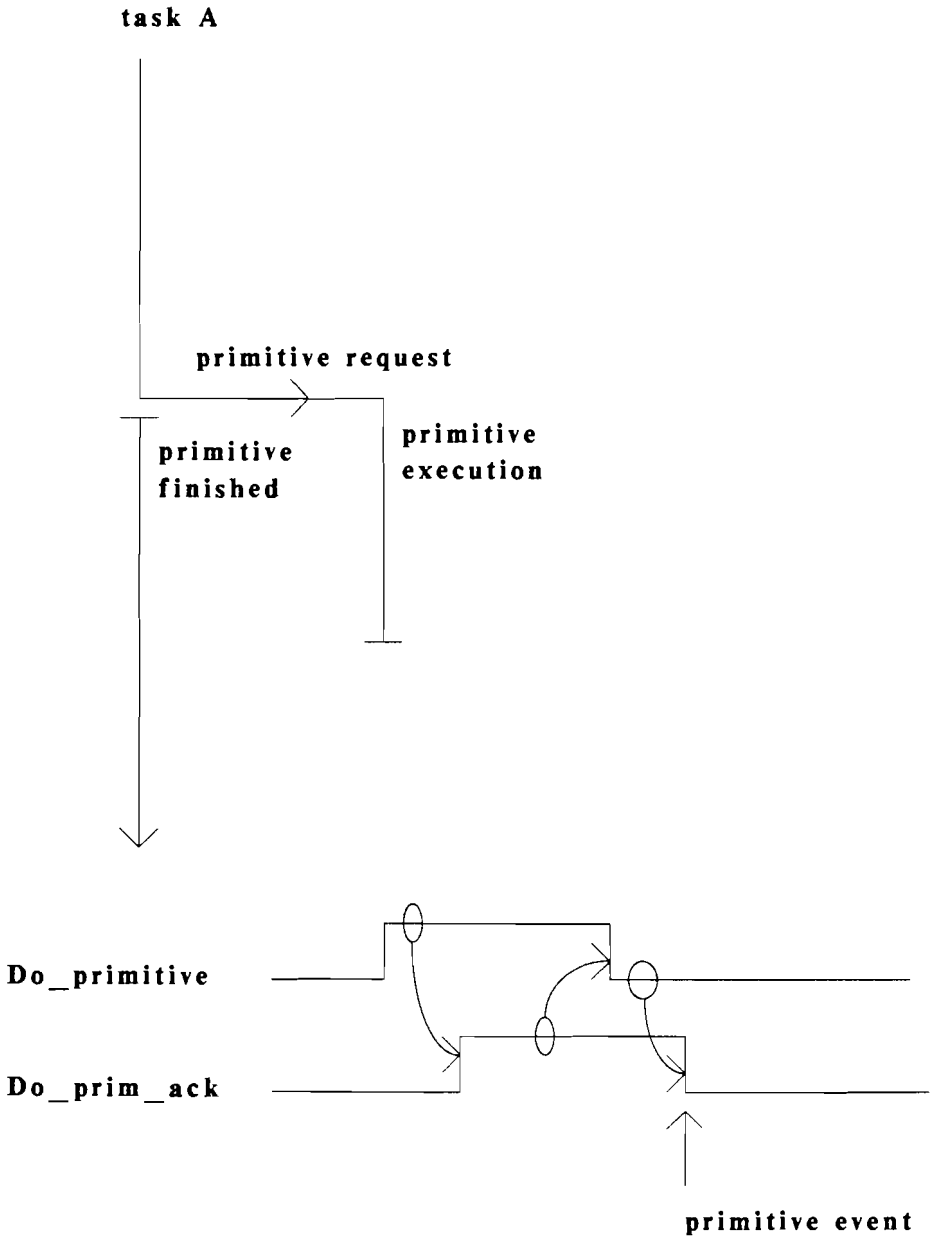


Figure 3 connection oriented/ connectionless services

1.5 Ways of implementing service primitives

When you have a look at descriptions of primitives you can distinguish two parts. One is the synchronisation and the other the parameter exchange. Either can be very simple or very complicated. The simplest primitive is one without parameters, but only synchronisation. A simple procedure call (in software) or handshake mechanisme (in hardware) could do the job. An example of this interpretation is shown in fig 4. In the upper part the programflow of a task calling a primitive is shown. In this case the synchronisation is implicite, after task A has requested the primitive it knows that the primitive will be executed. When and how the primitive is executed is not known. The execution may take place concurrent to task A but this is not necessary.
In the middle part a hardware synchronisation by means of two hardware lines is shown. The block requesting the primitive may do this when both lines are low, by activating the do_primitive line. The service provider can acknowledge this signal by activating the acknowledge line. The service user then releases the do_primitive line and next the service provider releases the do_prim_ack line indicating the completion of the primitive. Whether the completion of the primitive is indicated before or after the primitive execution depends on the service provider. The completion may be indicated before the execution, but in this case the service provider is responcible that it will be executed at all.
The lower part of figure 4 give another hardware implementation of the synchronisation is shown. The service user can set the request flipflop to indicate a primitive request. Both user and provider can sense the output line of the flipflop. When the service provider detects the request (the request pending line is active) it will acknowledge the request by reseting the flipflop.



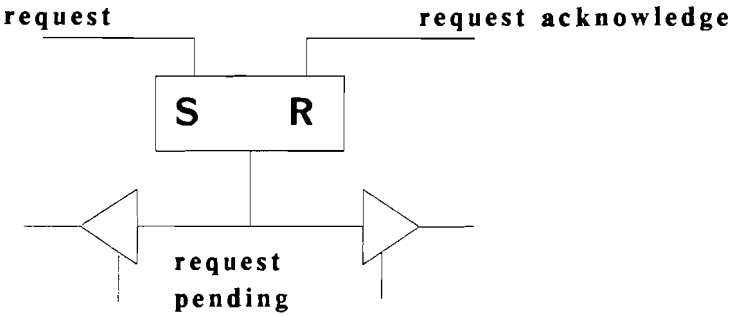


Figure 4 primitive implementations.

For primitives with parameters the basic primitive implementation is the same, ie. a primitive is considered taken place at the moment both parties agree on the parameters. The synchronisation part of the primitive can be handled in the same manner as with parameterless primitives. The parameter passing can be implemented in very many different ways. As an example of how this can be done, two different implementations of the Data request primitive will be discussed. The High level data request is less complex because the data is passed through a buffer in memory, and only pointers are passed. To implement this primitive the hardware as in fig 5 is necessary

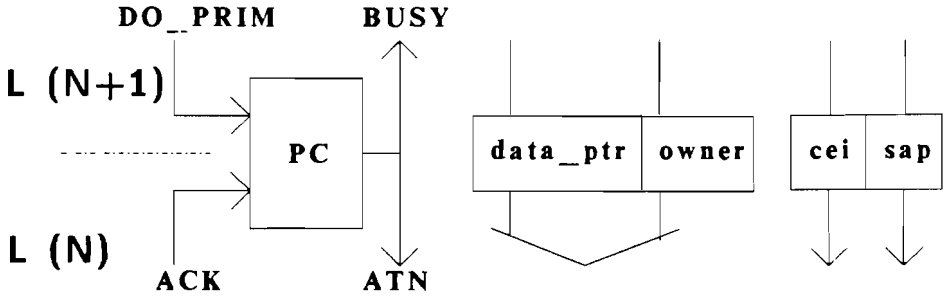


Figure 5 hardware primitive implementation highlevel datarequest.

The primitive control block is for synchronisation, whereas the other registers are used for parameter exchange. (For the implementation of the primitive control block see appendix A). In this case there is only parameter exchange in one direction via the data pointer/ownership register. The former contains a pointer to a buffer containing data. The latter is passed for memory integrity. A buffer can be passed from one block to another with or without the ownership. The block having the ownership of a buffer is responsible for its disposal. This means that after the buffer is used the owner has to pass the buffer to the buffermanager, who is the owner of all unused buffers and unused memory.
The other registers are optional, ie. when all data requests are passed via the same hardware port an explicit identification of the connection endpoint (CEI = connection endpoint identifier) and service acces point (SAP) are required. It is also possible that the datarequests are issued via different hardware ports, so the CEI and/or SAP are implicitely known. The CEI and SAP are necessary to know which entity in the layer that uses the service requested it and which entity in the service providing layer is supposed to accept the request.
The primitive takes place as follows: first layer N+1 decides that is allowed (according to the N+1 protocol) to issue a data request primitive. It checks whether there is no primitive pending, writes the parameters , the CEI and SAP, and gives a Do_data_req signal. Now the primitive control block is set and layer N notices that layer N+1 wishes to do a Data request. Layer N copies the parameters and gives a Data_req_ack signal. At this instant the data request event takes place. After this layer N will execute the data request, and the port is free for the next data request. Another possibility is that layer N cannot queue or buffer the data request and first executes the data request before giving the data_req_ack. In this case the port will be occupied for a long time.

Another implementation can be used with the physical layer primitives. The data is not passed via buffers, but explicitly via the primitive port. Since the SDU's can be very large it is obvious that this cannot be passed parallel because a very large RAM would be needed. Another possibility is to pass the data serial, but then extra synchronisation is required. An advantage of the serial interface is that the physical layer can transmit the databits as they come. In this case the implementation could be like fig 6

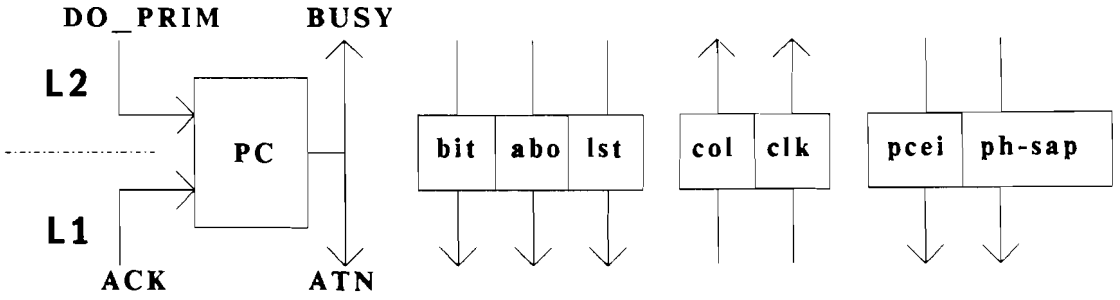


Figure 6 Hardware primitive implementation PH datarequest

In this case the primitive will be quite different from the former. Again the synchronisation part (Primitive control block) is the same, and so is the connection endpoint identifier and the sap identifier. However the parameter passing is more involved. Since the SDU passing now is serial, one register is required for the passed data bit and one for the bitclock. Since the primitive preparation takes quite a long time now, either layer can decide to abort the preparation for several reasons. A good reason to quit for layer 2 is an underrun situation. When layer 1 asks the databits quicker than layer 2 can provide them, remember the bitclock is controlled by layer 1, layer 2 decides to stop the transmission. In the layer 2 protocol might be a solution for this (eg. with HDLC an abort-pattern is available), but it could well be that layer 1 also has to indicate this according to the layer 1 protocol. (apart from this layer 1 has to know when to give the Do_data_req_ack to reset the primitive.) On the other hand layer 1 must be able to abort the Primitive preparation, for example when a collision occurs on a multisource medium. Since the data is transmitted as it is passed, layer 1 has no means to restart the transmission when collision occurs. It is up to layer 2 to solve this problem, which will probably mean layer 2 starts a data request with the same data. When nothing goes wrong and both sides decide to complete the primitive the preparation goes as follows. First layer 2 writes all parameters (ie. the first data bit, PCEI an PH-SAP) and issues the Do_data_request. Layer 1 next offers a bitclock at which layer 2 has to present the subsequent databits. Together with the last databit (last bit of the closingflag in case of HDLC) the LST-parameter is set. Layer 1 gives a Do_data_req_ack signal and the primitive is completed. The primitive-event is considered to take place at the instant the do_data_req_ack signal is given. If the preparation is canceled because of a collision or underrun the primitve is considered not happened, so the layer 2 protocol remains in the same state.
These were two examples of the Hardware implementation of primitives. For higher layers an even more complex parameter passing can be thought of. For example negotiation about parameters during the primitive preparation. Later we will describe the primitive choises made in our implementation.

2. X.25-INTERFACE

All over the world public packetswitching networks are being used. To create some uniformity and compatibility between these datanetworks the CCITT introduced a recommendation for standardisation. This recommendation is known as the X.25 interface, or with its full title: Recommendation X.25, Interface between dataterminal equipment (DTE) and Data circuitterminating equipment for terminals operating in the packet mode and connected to public data networks by dedicated circuit.
The terminals mentioned in this title are not only computer terminals but also computer systems. The number of X.25 interfaces used is growing. They are not only used with public Data networks but with many applications that use a packet switching network.
The first Question to ask is what is it that X.25 recommends? Since the X.25 recommendation dates from before the OSI-model it is not a derivate of the OSI-model (although with some effort it can be fit in the OSI-model).
As the name already implies X.25 is an interface definition, so that is what it describes. The interface is physically the cable that comes from the wall. That is exactly what the X.25 recommendation describes: All physical and electrical characteristics of connectors, signal levels, timing and all the possible bitpatterns that may occur. Specially the last subject is quite complex to describe. Therefor a three level architecture is chosen for the description and the protocols.
The three levels are:
-LEVEL 1 DTE/DCE interface characteristics (physical level)
-LEVEL 2 Link access procedures (link level)
-LEVEL 3 Packet level DTE/DCE interface (packet level)

How the bitstream is constructed is shown in fig 7. The user data is fragmented in pieces of maximal 4096 bytes. These fragments are embedded in a packet according the packet level (level 3) protocol. The packets on their turn are transmitted within the HDLC I-frames, which are inserted in the bitstream using a derivate of the HDLC protocol (ie. LAP or LAPB).

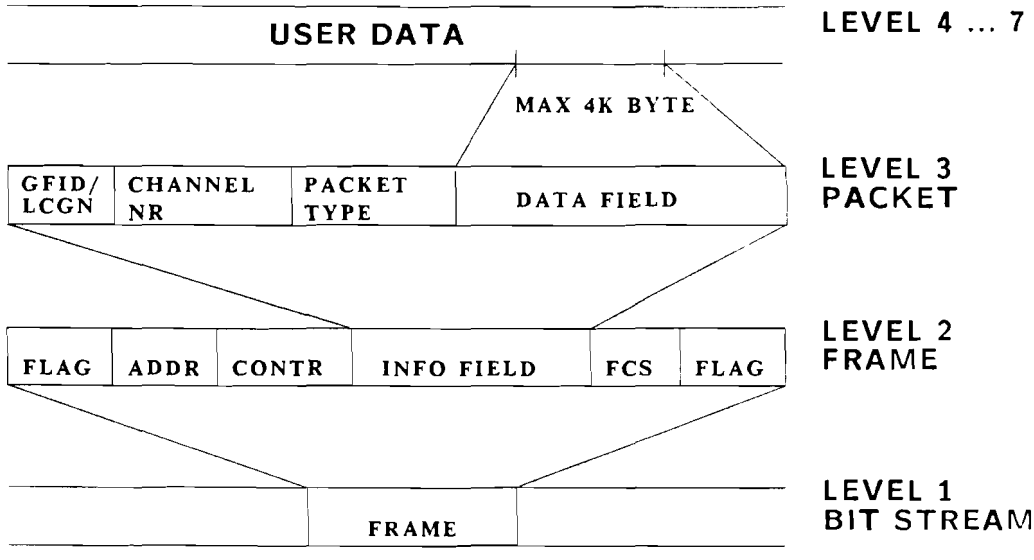


Figure 7 contents of the X.25 bitstream

We will next give a description of the levels and give an overview of all possible choises that can be made at each level.

2.1 Level 1 physical level.

This level prescribes all physical characteristics, i.e. mechanical and electrical characteristics of the connector, signals, signallevels and bittiming. On this level there are three different recommendations to be chosen from: X.21 interface, X.21bis interface, or V-series interface. These interfaces are described in other CCITT recommendations.
The X.21 interface is a true digital interface (5/6 lines) and is supposed to become the final standard. The V-series interface is hardly used these days with packetswitched networks and describes the interface with modem-equipment. As an interim measure to allow connection of synchronous DTE's which are designed for interfacing to V-series modems the X.21bis interface is defined.

2.2 Level 2 datalink level.

The function of the second level of the X.25 interface definition is to provide a single logical link for transfer of packets to level 3. This logical link must be error-free and deliver all packets in sequence. To provide the logical link there are several options. The simplest way is to use a single physical link. The error correction and sequencing is obtained by using derivates of the High-level Data Link Control (HDLC) -procedure specified by the ISO. Two procedures are used to implement the functions: LAP and LAPB.
-LAP : Link Access Procedure.
-LAPB: Link Access Procedure Balanced mode.
The LAPB single link procedure can be used in basic operation (sequence numbering modulo 8) or extended (modulo 128). The LAP single link procedure can only be used in basic (modulo 8) operation. According to the recommendation LAPB basic (modulo 8) operation must be available on all networks. But there are also network applications that operate using modulo 128 sequence numbering.
Another possible option to provide a single logical link is by using multiple physical links. Reasons to use multiple physical links are eg. higher throughput, reliability (failure on one link is not fatal) etc.
For this reason a Multilink procedure (MLP) is defined. This multilink procedure uses frames that are transmitted within frames of the single link procedure. These single link procedures (SLP) must be LAPB, but more than one link will be in use. The multilink procedure provides apart from the transmitter window, that is also used with the SLP, a receiver window to accomodate different delays in the various links.

2.3 Level 3 packet level.

The packet level of X.25 gives the possibitity to use several independent virtual circuits on one logical circuit (offered by level 2). To provide this a so called packet level protocol is used. As in the datalink level several options can be chosen. First there is the sequencenumbering, this can be chosen modulo 8 and modulo 128, both resulting in a different packet format. Next there are several options and facilities (more than 50), all leading to a different packet format or protocol behaviour. In fact all these options lead to a large number of "protocols"

all offering different features and functions, one cannot see the wood for the trees.
This level can be subdivided in several other levels to create more order and structure, as we will see later.

All this leads to an abstract X.25 representation as shown in fig 8. The virtual channel multiplexing can be recognized in the multiple level 3 channels that are connected to one level 2 logical channel. The optional multilink procedure is indicated by the dashed bridge within level 2. In this case several level 1 connections are used from the DTE to the DCE as can be seen in the figure.

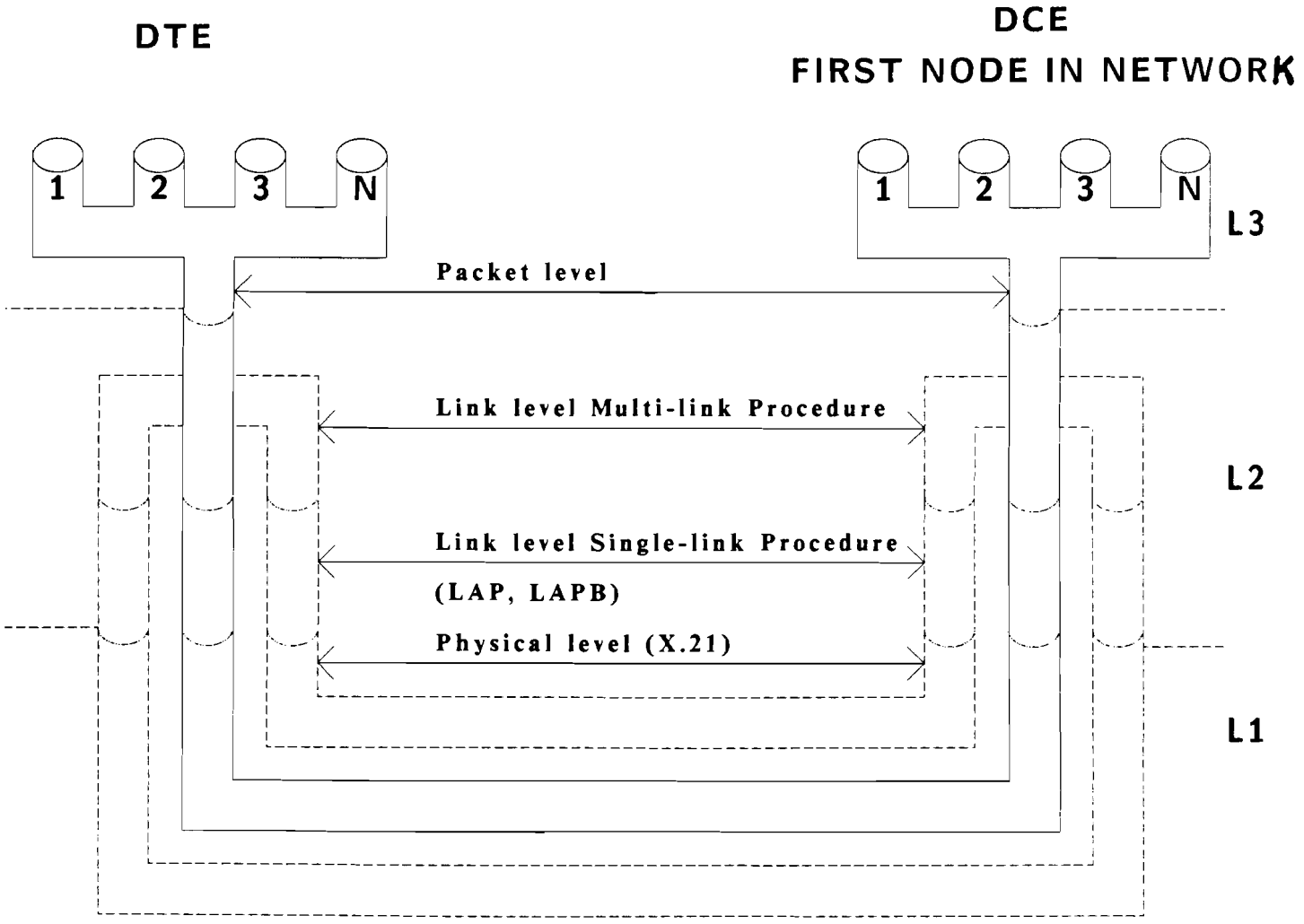


Figure 8 logical representation of X.25

2.4 X.25 versus OSI

Since X.25 was defined before the OSI model there are several differences. Apart from this X.25 was defined by the CCITT ie. the common carriers. These were not really interested how an implementation is realised, but more how the physical interface behaves. So the major difference between X.25 and the OSI model are the service definitions. Within X.25 only level-protocol definitions are presented, whereas the OSI-model also defines the interaction (services) between the layers. Exactly this property makes the OSI-model for implementors as important as it is. The layers are isolated by the service access point, because the behaviour at this point is defined regardless the protocols that implement it. Because X.25 does not prescribe how the interaction between the different levels are, one is easily tempted to use an interface fit for X.25 and the protocols used. This leads probably to an efficient, but rigid design. The several building blocks are not very well suited to use in another design, without complex glue logic, or the other design has to be adapted to the chosen interface. This interface might be not suited very well for the new situation.
Another difference between X.25 and the OSI-model is the functional division in layers and levels. The functional mapping is shown in fig 9

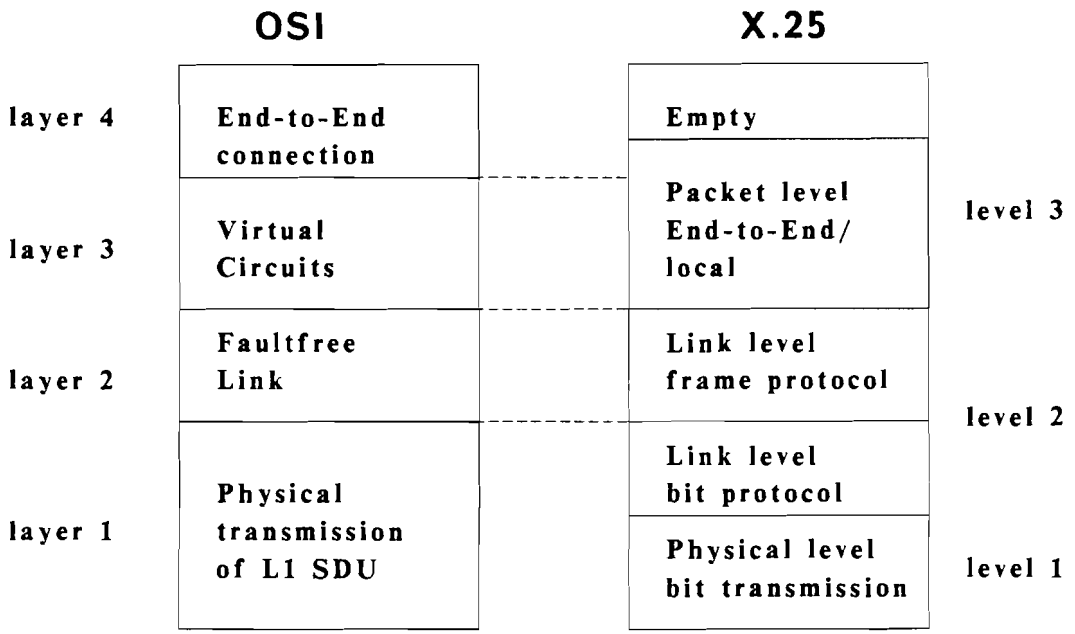


Figure 9 mapping OSI functions on X.25 functions

The shown mapping is slightly different from interpretations found in literature, specially the way level 2 is divided in a bit protocol (flags, bitstuffing etc.) and a frame protocol can be noticed.
When we look at the OSI data structure we can see that within layer N an N-SDU (service data unit) is accepted or presented via the N-service primitive. The layer adds Protocol control information (and changes maybe the data (bitstuffing,

encryption)) creating an N-PDU (protocol data unit). This N-PDU is passed to layer N-1 as an N-1 SDU (see figure 2 "realation between the data units in the OSI model). With X.25 a similar structure is found in the level 2 and level 3 protocol, but the L1 protocol suddenly only transfers bits instead of level 2 frames. It can accept a level 2 frame including flags and bitstuffing, but it has no way of discriminating the frames in the incoming datastream. Taking a closer look at level 2 you can see that exactly this function is performed by the flags and bitstuffing of level 2. So the layer 1 service primitives should partly be mapped on the interaction port between the level 2 bitprotocol and the level 2 frame protocol. On the other hand a similar mapping must be made on the interaction between level 2 and level 1, since the level 2 HDLC protocol is a complete protocol that has many applications. When the primitives are implemented on this point the level 2 block can be reused with other aplications.
A more complex problem rises with level 3. Because the X.25 definition stops with level 3 this level is open ended, whereas the OSI-layer 3 is clearly bounded by the layer 3 service definitions. The basic packet in X.25 consists only of a few fields, whereas a lot of parameters are determined by higher layers (in the OSI model), and just copied by level 3. Several of the parameter fields are only aplicable when using the extended packet format. Due to all the different facilities that can be asked for it is very difficult to map the OSI primitives on the packet level protocol, without abandoning several features. Other features that must be negotiated concern features that determine the protocol behaviour during a whole session and can only be changed at startup. Another problem occurs because some layer 4 functions, eg. seqmentation of higher layer PDU's, are defined within the level 3 protocol (in the case of segmentation the M-bit)

3. Functions and requirements

When defining an architecture one first has to realise what requirements there are, before a sensible decomposition can be proposed. With a communication chip an obvious choice for this is the OSI model. Because this design has to be done within a university where students only have limited time to get familiar with the problem the borders of the different blocks have to be clear, so every block has its own complete function. Therefore high level commands must be given at the interface. Another demand is that the blocks can be build with other blocks using their functions, or that they are sufficiently small so they can be designed by one person within reasonable time. The past has learned that blocks that have too much interaction only cause trouble. For example one block passes a difficult to implement function to another, where it is ignored because it really belongs to the former block.
When we take the OSI decomposition for granted, we have to decide which primitives have to be implemented and how the managment functions are to be implemented.
For the design we can distinguish four levels of design, that are kept in mind during the whole process of course, but require different aproaches:
-Protocol operation
-Statistics
-Conformence testing
-Hardware testing

Protocol operation.

This is the reason to design a special purpose IC, so the architecture must be fit to perform this task very well. It is described reasonable well in the several protocol and service definitions. This operation is characterized by the data-flow from the host through the chip to the communication line and vice versa. The implementation must support the tasks concerning the normal operation as efficient and fast as possible. So this is the main design item to keep in mind.

Statistics

During the protocol operation several information concerning the operation is logged. Some information because it is a protocol requirement, but also information that might be interesting for the operator to tune the communication (eg. adapt windowsizes, change packet lengths etc.). The logging of this data must be in a way that is convenient for the processes that perform the protocol. However it is not necessary that the Statistic data is easy accessible for the host or operator. Since this query is not performed very often the normal operation could be stopped for a while, to give the chip the opportunity to pass the statics information in one way or the other to the host. One way could be to use interface registers that are normally used for primitives temporarily for the transfer of data from all layers up to the host interface. An other implementation might just dump all internal status registers in shared memory for the host to collect and sort the data.

Conformance testing

This is an important point with communication devices in an public network environment. With normal X.25 networks the only conformance testing that is done takes place at the physical interface. In an OSI environment also internal boundries become important, since also the service implementation has to be according to the OSI definitions. For this reason several conformance test strategies have been proposed. One of the largest problems with this testing is that in most products the internal boundries are not accessible. To get around this problem and have the possibillity to test every layer independent from the others there has to be a solution to access the service primitives independent from other layers. Because this is even less used as the statistics query it can be implemented in a way it needs a restart of the system to get to normal operation again. One implementation could be to use start-up parameters that cause some layers to become transparant so lower layers , from the host side, or higher layers, from the data network side, become accessible. For this mode only protocol and service conformance has to be tested, so one can assume the hardware is correct.

Hardware testing

This level of "operation" is only used a few times in the lifetime of the chip. It is necessary to support since the costs of malfunction devices increase tremendously as the production proces goes on. Detection a faulty device on the wafer costs little money, but detecting hardware or software faults in an exchange or operating computers costs lots of money. So it is worth something to put a lot of effort in detecting faulty components as early in the production as possible. Because there are so many possible production faults that can occur several strategies are invented to detect them, such as scanpath design, selftests,etc. For all of these strategies extra hardware and special design rules are required. Since the hierarchical design method we used supports most of these strategies, the details of this operation are postponed. Knowing that one strategy or the other can be implemented later.

To start with the last item, not much study has been done on this subject yet. It seems natural to implement some kind of hierarchical test strategy using wellknown techniques. For example, use scanpath testing for the businterface unit and a combination of scanpath and build-in-testlogic (eg. via the businterface) for other blocks.
The difference between the conformance test mode and the operational mode, of which the statistics mode a part is, can be selected during start-up. All layers have to be configured to define the protocol parameters. This can be done in the same way several microprocessors and other coprocessors are configured, by having the configuration paramaters available for the layers. This can be accomplished by putting them on a fixed place in memory, or putting a pointer to the several configuration blocks on a known memory-address. These configuration blocks can also be used to put several layers in the conformance test mode and others in operational mode. This results in the situation that some layers are transparant or start acting as a host interface so the intern primitives come available to the outside. For these functions a different programm can be written. This means that the programmer for the operational mode does not neccesarily have to know everything of the conformance testing mode of that

layer and vise versa. Since the statics mode must be called from the operational mode, saving the current status, not the same construction can be used as described above. It would be desired the statistics mode was written as a separate subprogram, or separate part of a finite state machine, so again the clear distiction can be made with the operational mode. For this mode a special primitive must be implemented. It must suspend the current operation of the block as soon as possible, and dump the block status in a common accessible place.

To implement the various protocol algorithms several programmable micro controllers and finite state machines are used. To implement the interfaces and primitives between the blocks registers are used. Only these registers are defined as an interface and not the way they are accessed, ie bussize and address-size. This offers the possibility of a very flexible design, aspecially in an environment of connected microcontrollers. An interface could be implemented as in fig 10. In the first part an interface is shown that is accessed as 4 bit registers by one block and as an 8 bit register by the other. The second part shows a register that can be accessed as a shiftregister by one block.

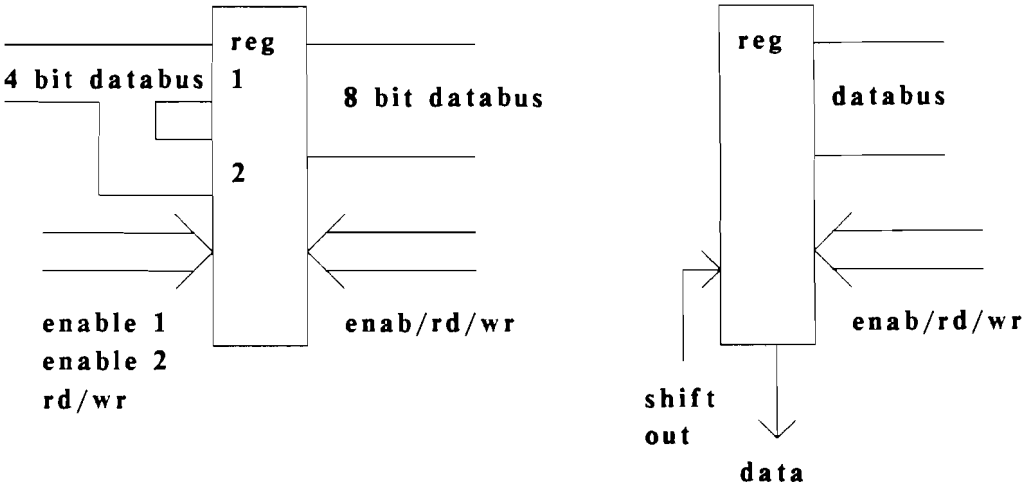


Figure 10 interface registers.

The flexibility of this interface can be seen when the blocks are viewed as processes that run on one or more microcontrollers. The processes are programms that manipulate the data according to the protocol algorithms. The state of the data variables defines the protocol state and the interface behaviour. Using the register interface the programms stay the same whether they are run on one or more microcontrollers. To illustrate this have a look at fig 11.

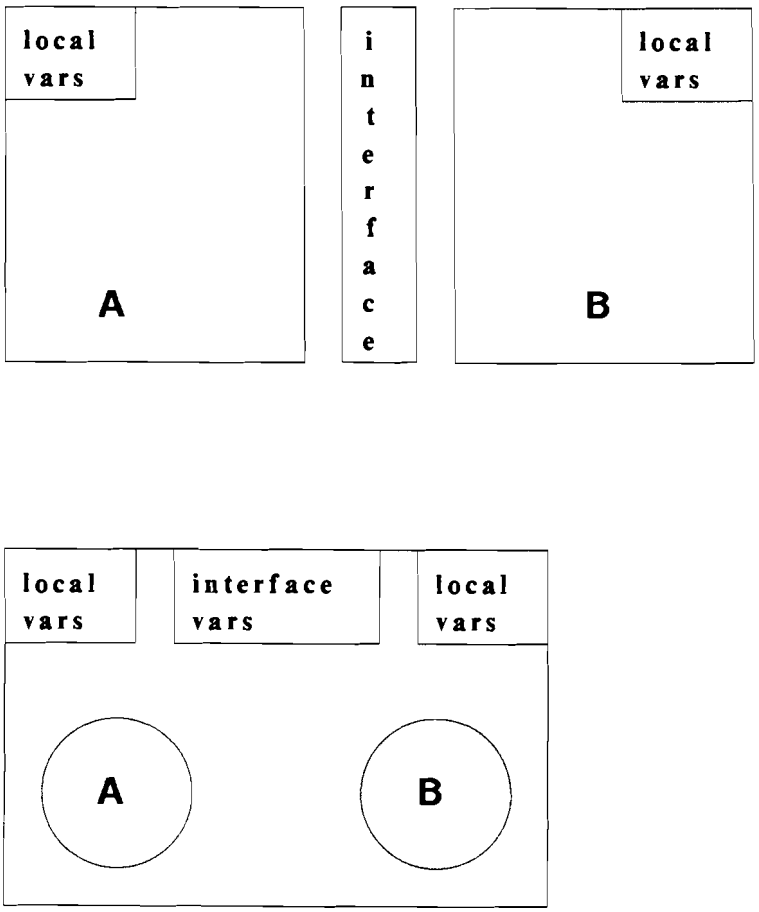


Figure 11 Interface registers in one micro controller

When process A and process B run on different controllers, they both have local variables that cannot be accessed by the other process. They also have interface registers that are accessible by both. Of course there is synchronisation needed for reading and writing the interface registers, but we will come to that point later. If the processes are idle most of the time, it would be a waste to have two controllers, one for each process. It would be more efficient to have the processes share one controller, on a timesharing basis. The only thing this requires is an additional scheduler to perform the switching, but the programms that implement the algorithms stay the same. In this case we can again recognise local variables to A and B, and the same interface registers. The difference is now that are not physically separated, but only logically. The processes could use each others variables. If the software designer uses however the same aproach as the hardware designer would be forced to, the programms can stay the same and the software designer uses only the memory model to write his programms.

Another advantage of this register model is that the complete design can be implemented in several chips if necessary, without the designers of the

communication levels noticing it. An example of this separation is shown in fig 12. The first part shows a design in one chip the right part shows the same blocks now realised in two different chips. As is shown the interface to the two communicationblocks is the same as before the only difference is the additional serial interface between the two chips (since i/o pins seem to be a scarce resource in chip design).

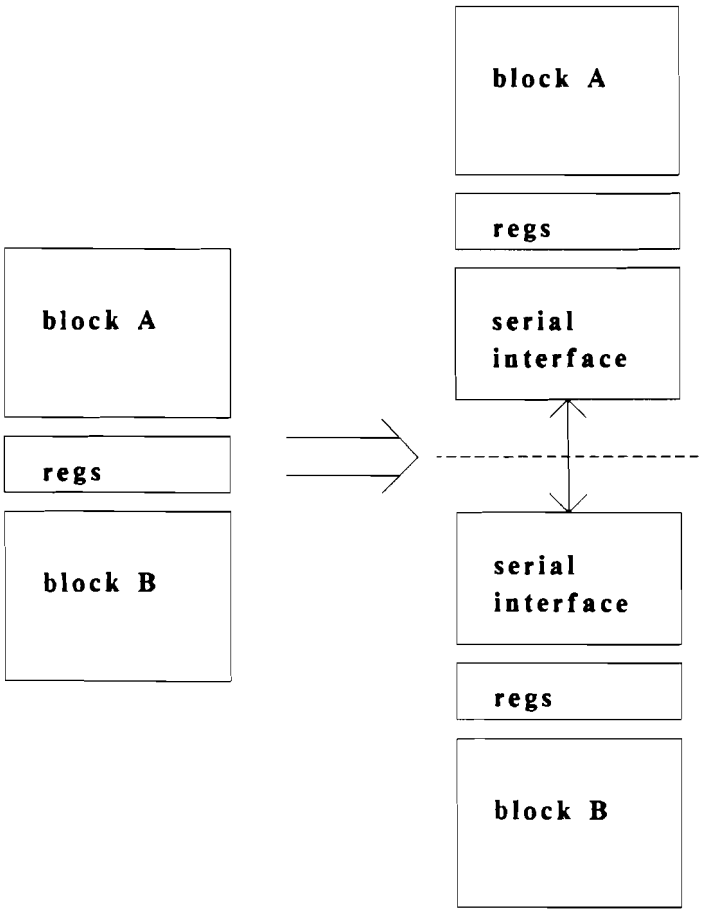


Figure 12 Interface registers in two separate Chips

3.1 Synchronisation of interface registers.

The interface registers come in various kinds. In fig 13 several examples are shown. The most common registers are the registers that one block can write and an other can read. Other registers can be read and altered by two layers. Finally there are the primitive controlblocks that are registers that can be read by both blocks , set by one and reset by the other. Even different interfaces can be thought of, eg. registers that are shift registers to one block and ordinary registers to the other. The synchronisation with these registers depends on the implementation and nothing general can be said about these.

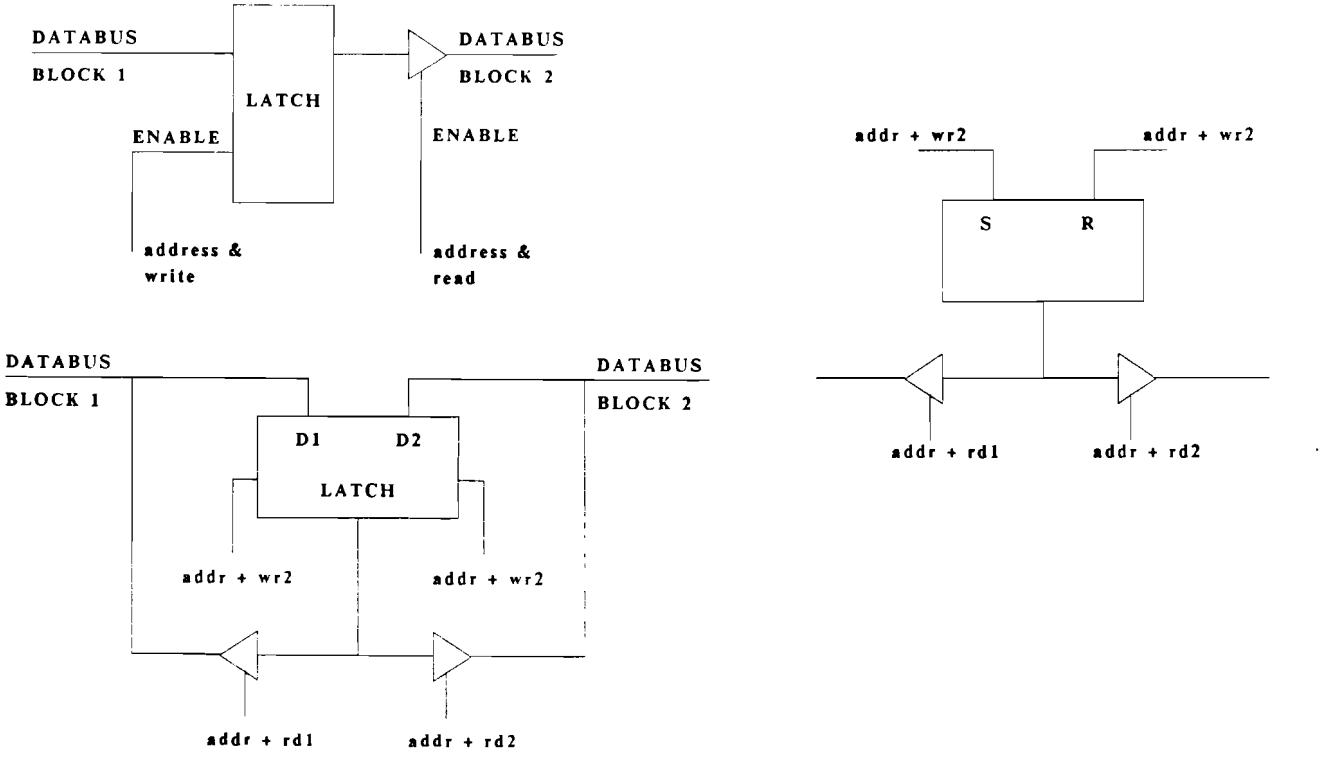


Figure 13 Implementation of interface registers

For the other registers three kinds of synchronisation are neccessary:
-Read/Read synchronisation
-Read/Write synchronisation
-Write/Write synchronisation

1. Read/Read synchronisation

This synchronisation is necessary when two blocks connected to the same bus want to read in two different registers at the same time. In this case data can be changed or even worse there may be a short-circuit. So obviously some measures to prevent this have to be taken. In our design this is no problem since both

blocks are physically separated. They do not have a common bus which they have to use during a read cycle. The read cycles don't have to be synchronised because no data can be lost or destroyed.

2. Read/Write synchronisation

For data integrity it is neccessary that the data in a register is stable when it is read by a block or evaluated by a finite state machine. When one block is writing in a register and another is reading the data at the same time there might be a problem. To solve this problem a two phase clockcycle is defined for the controllers. One cycle is the read cycle in which the data is copied in the master flipflop of a register, the second cycle is the write cycle in which the data is copied in the slave flipflop. A microcontroller that functions according to this rule is already developed by other students in the same project ([KLIP,OOIJ]).

3. Write/Write synchronisation

When two blocks want to write different data in a register at the same time both data might be lost because it is not known in advance which data bits will be stored. In the worst case some bits of both values will be copied. This problem comes with dualported registers in which two blocks can both read and write. If the two blocks want to write in one register at the same time, data of one or both blocks will be lost. To prevent this a semaphore mechanism can be used. One bit indicates which block has write permission for a register. If the bit is set to one the first block may write it, and subsequently reset the bit. If the bit is set to zero the other block may write it, and also set the bit again. For this synchronisation the primitive control block can be used, but it is also possible that another bit is required when several read/write actions have to take place within a service primitive.

3.2 Data flow: Data buffers

The most important task of a communication processor is the manipulation of data. Since very large quantities of data must be transported it is crucial that the data is not transported from one place in memory to the other every time it is passed from one layer to another. To prevent this an obvious solution is chosen: put the data in databuffers in memory, and pass only the pointers to these buffers from one layer to the other.
Before a data structure can be found for these buffers we must first decide what properties the buffers should have and what operations can be performed on the data in the buffers. Since the data passes several layers (maybe from layer 7 to layer 1) as it is processed by the communication application (normally a general purpose processor together with a communication processor) all operations defined by the OSI model must be possible. As stated before within the OSI context there are four operations that can be performed on PDU's:
-segmenting
-reassembling
-concatenation
-separation

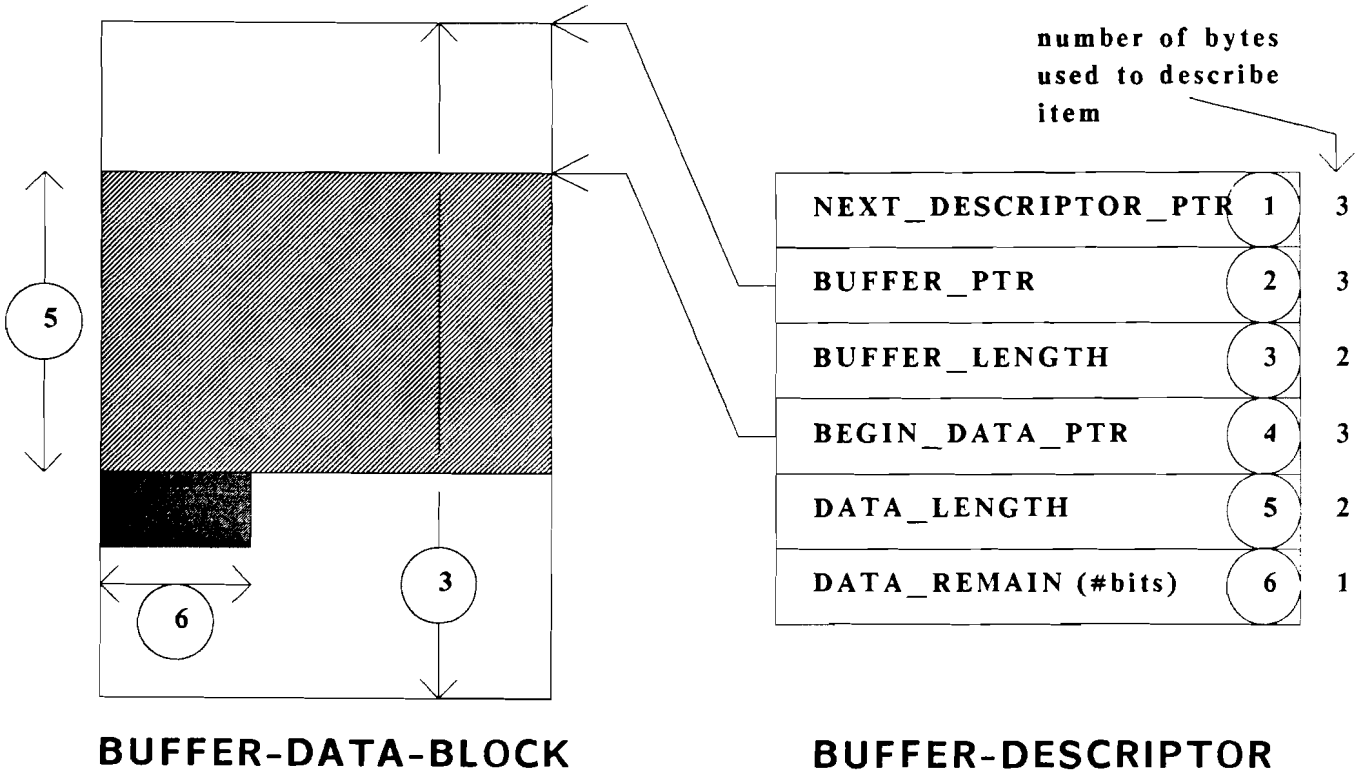
Apart from these four OSI operations the protocol adds information to a higher layer PDU (now called SDU) this means that a header and/or a trailer must be added to the SDU. When a received SDU from a lower layer is processed the reverse operation must be performed, the header and/or trailer must be removed and the stripped PDU is passed upwards.

From all these operations we can learn that buffers must be connected together (for concatenation and reassembling), and divided into several new buffers (for seqmenting and separation). Also adding and removal of headers must be possible.

Another very important property that must be chosen is the length of the buffer. Since the data that is received in packets always comes in a certain size it is wise to adapt the bufferlength to this size.

Buffer implementation.

To enable all operations previously described we defined a bufferstructure as shown in fig 14. The buffer consists of a buffer data block in which the contents (data) is stored and a separate buffer descriptor in which the administration of the buffer is stored. The buffer descriptor and the data block are linked together via pointers.



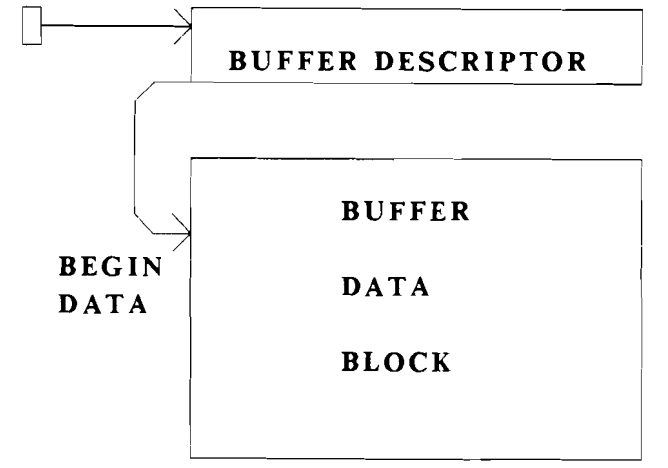


Figure 14 Databuffer structure

The memory space the buffer is stored in is assumed to be byte oriented and addressed with an address of maximal 24 bits (3 bytes), so a maximum of 16 Mbytes of memory can be addressed. This is considered fair enough for the current state of the art. (Since the data is byte oriented, the next lower number of address bits was 16, which allows 64 Kbyte of memory, was considered to little). The size of the pointers is now also fixed to 3 bytes since we use absolute addresses with pointers.
The buffer descriptor consists of the following fields:

1. Next_descriptor_pointer.
This pointer is used to link several buffers together to form a SDU or PDU that is too large to be contained in one buffer. Thanks to this pointer it is possible that the representation of SDU's in all layers is the same. If the buffer is the last buffer in a chain an no other buffers are linked to it, this pointer contains the value NILL (== 0) to indicate it points to nothing.

2. Buffer_pointer.
This pointers indicates the first byte of the buffer data block. It is required because the start of the data block must be known, because it is not always the same location as the data starts. This pointer only changes with segmentation, separation or by the buffer manager.

3. buffer_length.
The maximum buffer length is fixed on 64 Kbyte, which should be sufficient since buffers can be linked together if they are too small to contain a whole PDU or SDU. Again this is an arbitrary choise based on the byte oriented buffers. The minum size of this field is 1 byte allowing a maximum buffersize of 256 bytes. This was too small since PDU's can be much larger, eg the X.25 packet level protocol allows packets upto 4 Kbytes data. If the buffers are used in such an environment it is clear that one byte buffer_length is not enough. This would require linking of 16 buffers for every packet. Since higher levels allow even larger SDU's ,because they are segmented before transmission, the choise is made for a 2 bytes buffer_length field. Together with the buffer_pointer this field describes the buffer_data_block.

4. begin_data_pointer.
To describe the data contained in the buffer_data_block several other fields are defined. These are required to allow the operations that have been defined, examples of these operations will be described later. The first thing one wants to know about the data is where it starts. Basically there are two solutions to indicate the start of the data. The first one is to give the offset from the start of the data_block. This would require a two bytes field, since the maximum offset is 64 Kb. To get the actual address this field and the data_block pointer should be added.
The other solution is to use the absolute address as a pointer. This solution requires a three bytes field, so one byte more memory access is needed. On the other hand no addition has to be performed to get the actual address. The choise is made for the last solution because the address calculation would require probably the same time as the extra memory access, but for the address computation extra hardware is needed.
So the start of the data is indicated by a three byte absolute address.

5. Data_length
This is the second field that describes the data. Once you know where the data starts it can be usefull to know where it stops. To indicate this there is chosen for an offset. A two byte field indicating the length of the data field in bytes is reserved. This value can be copied and used in a counter to detect when the data is finished. When new data is written to the buffer simply counting the number of transferred bytes would give the new value of this field.

6. Data_remain
Since the data does not necessary consist of an integer number of bytes, there is a need to indicate the number of restbits. It is for example perfectly legal for two 13 bit oriented dedicated control computers to communicate using an HDLC protocol transmitting frames with a datafield that is a multiple of 13 bits. Once this is stored in a buffer it is unlikely to result in an integer number of bytes. A HDLC layer 2 protocol however should be able to transmit this layer 3 PDU. To allow this kind of weird computers to use the same buffer structure an extra field Data_remain is introduced. This field concatenated with the Data_length field, which gives the number of databytes, gives the total number of bits contained in a data_block. In most cases however this field will contain zero, because the supported data units consist of an integer number of bytes.

In the next part we will describe the operations on PDU's and SDU's, and give the corresponding operation on the buffers.

Segmentation/separation

Imagine a layer N SDU that is passed and has to be segmented into several layer N PDU's, as shown in fig 15. Asume the SDU is contained in four buffers and has to be segmented or separated into three PDU's. There are two possible situations that can occur and are both indicated in the figure. The first possibility is that the segmentation has to be performed at the end of a buffer, as between BD1 and BD2. In this case new buffers are filled with the Layer N header and linked to the newly separated buffers. The second possibility is when the segmentation has to be performed in the middle of a buffer. In this case a new buffer_descriptor must be generated. This new descriptor forms a new buffer together with the last part of the old data block. The old buffer descriptor needs to be updated according to the new situation. The buffer length and the data length must be updated to the first part of the old data block.

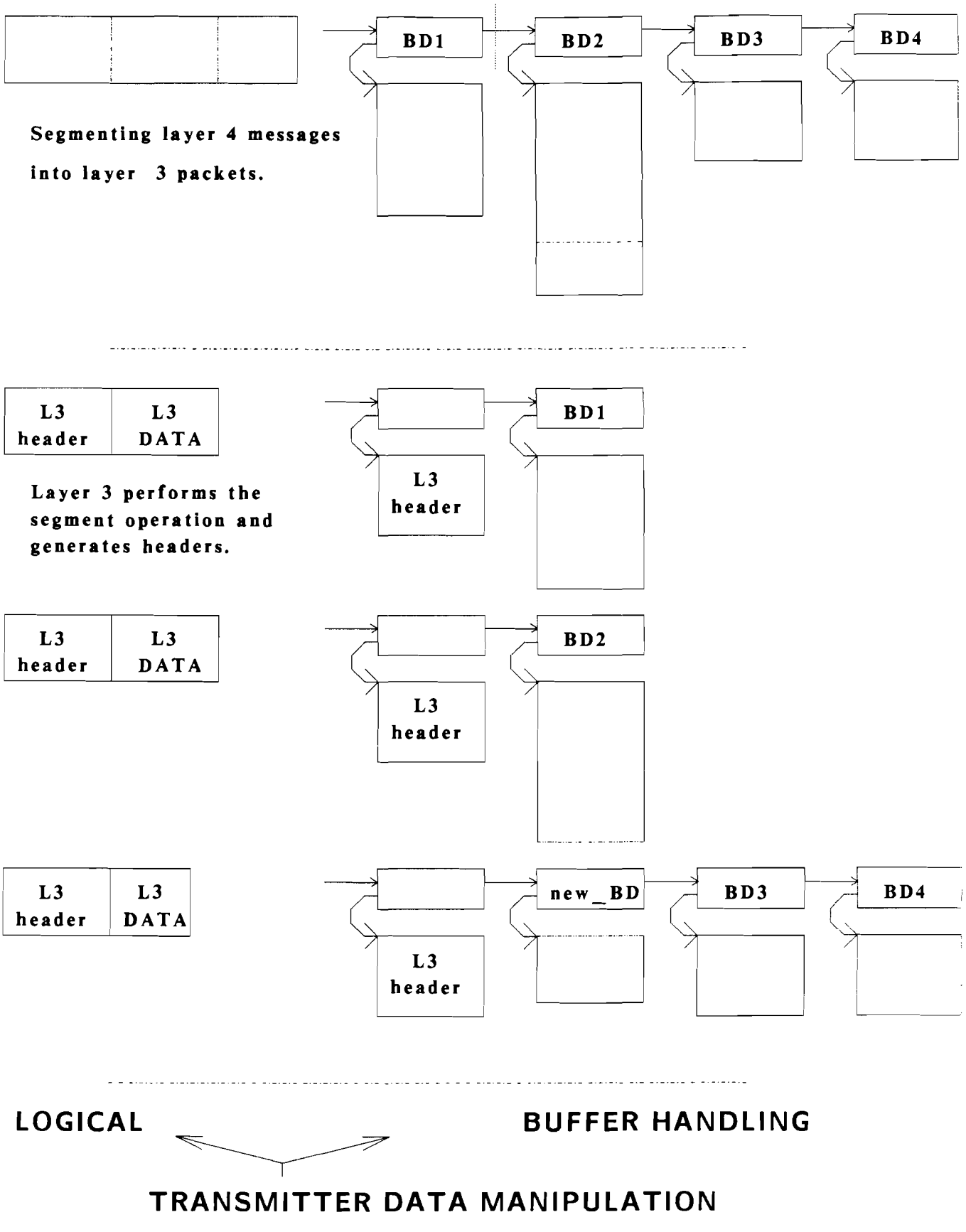


Figure 15 segmenting a PDU and adding a new header.

Reassembling/concatenation

This is the reverse operation of the previously described segmentation and separation. These operations are much easier to perform since no buffers have to be altered. The only thing that has to be done is linking the buffers together and put a new header in front as in the previous example. This is shown in fig 16 below.

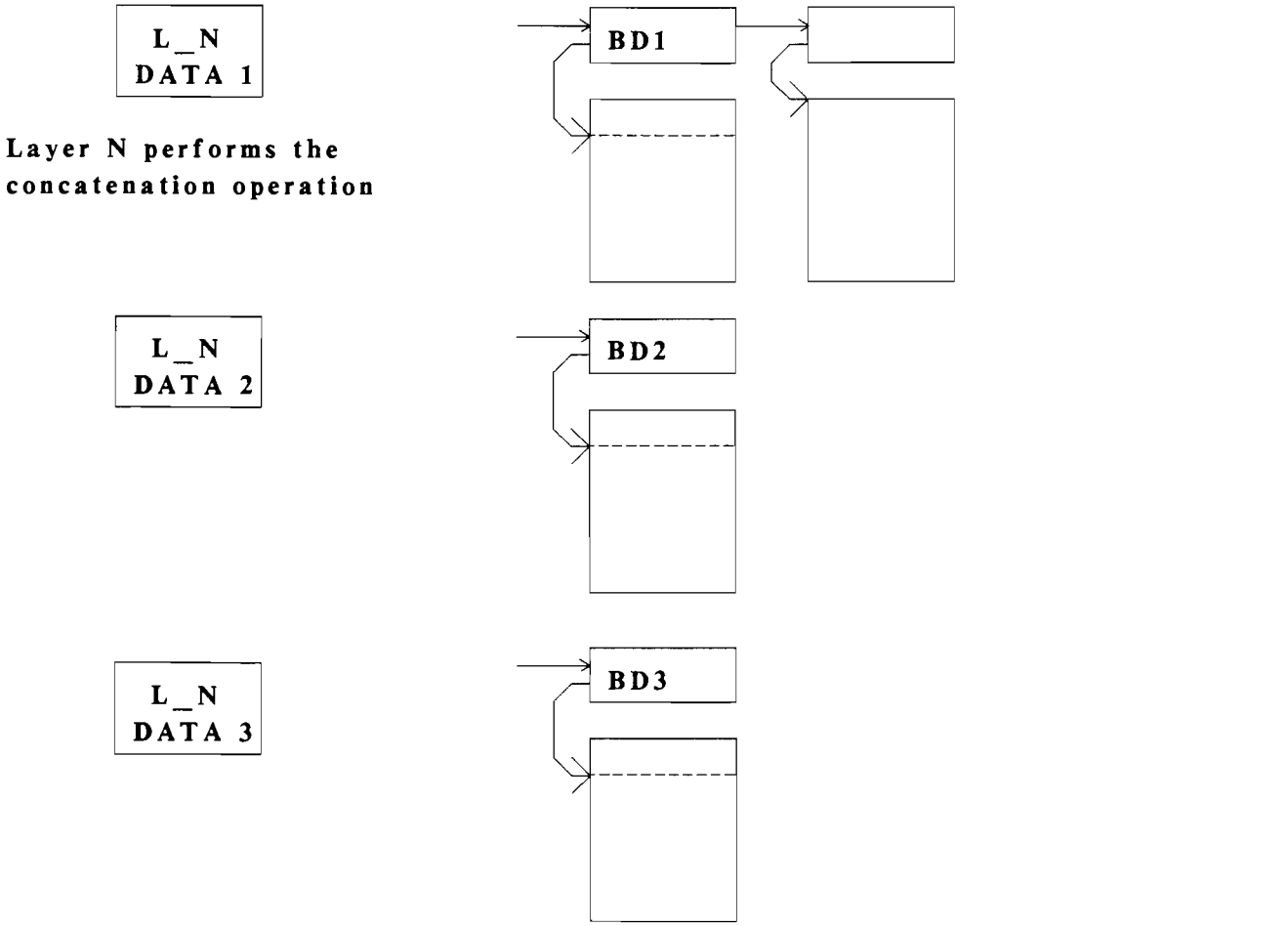


Figure 16 reassembling/concatenation of an SDU

Protocol processing.

When none of the above operations have to be performed, it is still necessary to process the buffers according to the protocol definition. This means for the receiving data: process the header and if the data is correct remove it and pass the remaining SDU upwards. For the transmitting data a header (or PCI) has to be made and added to the SDU.
The processing is shown in fig 17. The removal of the PCI is done by changing the begin_data_pointer and updating the data_length and data_remain field. If the first buffer gets empty after this operation it can be removed and returned to the buffermanager. The addition of the header to the SDU can be done in two different ways. One has been described allready above. It is done by requesting a new buffer, fill it with the header and link it to the SDU. Another strategy needs anticipation of the higher layers. In this case a higher layer leaves as many bytes free at the begin of the buffer as is necessary for all lower layer headers. This can be obtained by a priory knowledge of the higher layer but also by this knowledge in the buffermanager. When the buffermanager passes the buffer with the begin_data_pointer at the right place or the higher layer leaves the begin of the buffer empty adding the header could go as follows: the layer fills the last free bytes before the data and adjusts the begin_data_pointer and the length fields as shown in fig 17

Layer N removes header
after processing.

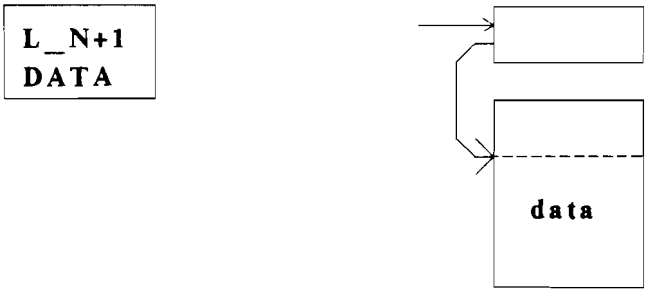
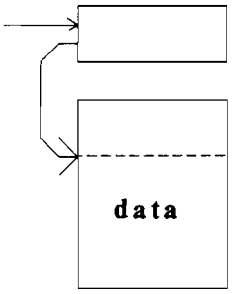
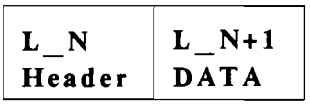


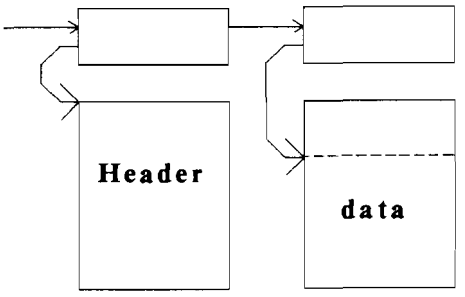
Figure 17a Removal of a PCI (header)



Layer N adds a header:



1)



2)

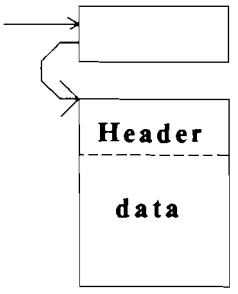


Figure 17b Adding a PCI to a PDU

3.3 Buffermanager and buffersizes

One of the most important things to get a good performance is the size of the buffers. If they are too large either memory is waisted because most of the buffers are not filled completely or throughput is restricted because there is too litlle memory. If the buffersize is too small the time required to process the buffers is too long since the buffers always have to be linked, wich takes processing power that could be used for other purposes like protocol handling. The buffermanager decides the initial size of the buffers. The buffermanager is divided in two parts: one in software on the hostprocessor and one on a microcontroller in the coprocessor. The hostpart is the master of the two and the coprocessorpart is the slave. This does not mean that the coprocessorpart cannot do anything without permission of the host, but it means that the slave does not create buffers, does garbage collection in the buffer part of memory or whatsoever. All these tasks can be performed by the host buffermanager. The coprocessor buffermanager only keeps enough buffers to allow the coprocessor to maintain the communication. When the processes in the coprocessor need a buffer they request it to the coprocessor buffermanager. When they want to get rid of a buffer they pass it to the coprocessor buffermanager. The buffermanager asks for buffers with the host buffermanager if he has got too little, or gives them back when he has too much. The host buffermanager can vary from very simple to extremely sofistocated. The most simple host buffermanager can be implemented when no segmenting or concatenation is performed. In this case all buffers can be installed during initialisation and the only thing the buffermanager has to do is maintain a linked list of free buffers. The most sofistocated buffermanager could try to adapt the bufferlengths to optimize the performance, do garbage collection to prevent the memory from getting scattered. The buffermanager would perform the same tasks as a memorymanager does in an operating system. The master buffermanager is implemented on the hostprocessor because this processor has in many cases powerfull addressing facilities and good hardware to do all the address calculation that are necessary. In this way the slave on the coprocessor can be a simple process that doesn't have to do much calculations but merely load, store and compare operations, thus saving hardware on the coprocessor.

Memory integrity

When a system is using the buffers in various processes, passing them from one process to the other, buffers may get lost, because one process thinks the other takes care of the buffer while the other thinks the same for the first. Both processes will forget the bufferpointers and the buffer is lost forever, or more likely untill the next systemreset. To prevent such a thing happen the property ownership is introduced. The owner of a buffer is responsible for the buffer. If

he passes pointers to other processes it must maintain the buffer until it is sure that the others don't need the buffer anymore. This can be obtained by explicite acknowledgements between the processes, but also implicitely by protocol algorithms. It is for example impossible for layer 3 to receive an acknowledgement on a packet, when layer 2 has not received an acknowledgement on the frame that contained the packet. The worst case would be that the return frame containing the acknowledgement contains also the ack-packet, but there is no way that a layer 3 packet can arrive correct while the layer 2 frame containing it didnot arrive. The buffers can be passed with or without ownership. When they are passed without ownership the bufferpointers can be forgotten after use, but if the buffers are passed with ownership they have to be returned to the buffermanager who is owner of all idle buffers and unused memory.

4. General architecture

After we have made an inventory of all requirements a choice can be made for the architecture. As mentioned before we want to implement three layers of the OSI model so an obvious decomposition is to define three functional blocks that implement the several layers. The interfacing between these blocks should be according the OSI service definitions. Apart from these communication-tasks there are several general functions that have to be performed. The first one is the buffer managment on the chip. As we have seen in preceeding chapters this means taking care of the empty buffers on the chip. For this function a separate functional block is defined. Another function that has to be performed is the interfacing with the (shared) memory. On the chip several tasks want to access memory independent of each other. A priority rule for all these tasks has to be implemented, since some tasks can wait without penalty whereas other tasks may loose data if they are not serviced fast enough. The memory interface has to decide whose turn it is. Another function of the memory interface is to hide the busacces mechanism for the blocks. This means that changing the memory interface would be the only modification for use with another processor family. To the other blocks the memory is a byte oriented lineair memory in which they can read/write bytes or words. The third general block has not been discussed before but more or less derived from the communication protocols, it is the Timer Managment Unit. All data protocols use timers for several reasons. These timers can be very many or very few. We decided to chose for one general timer facility over the alternative, timer facilities in every layer. Because the timer process can be constructed very efficiently, using event-driven techniques, and all layers need timers, this solution is probably the one with the least hardware overhead. The last block which is neither general nor communitcation specific is the host interface unit. The host interface unit takes care of the interconnection of the coprocessor hardware and the communication software implemented on the hostprocessor, in cooperation with the coprocessor interface program implemented on the host.
Summarizing this gives lead to the coprocessor architecture shown in fig 18

The blocks in this figure do not represent hardware units, but merely functional units. One functional unit (eg the layer blocks) can be implemented in several concurrent operating hardware blocks. It is also possible that several functional units are implemented as several processes in one microcontroller using a multitasking scheduler (eg the buffermanager and the host interface).

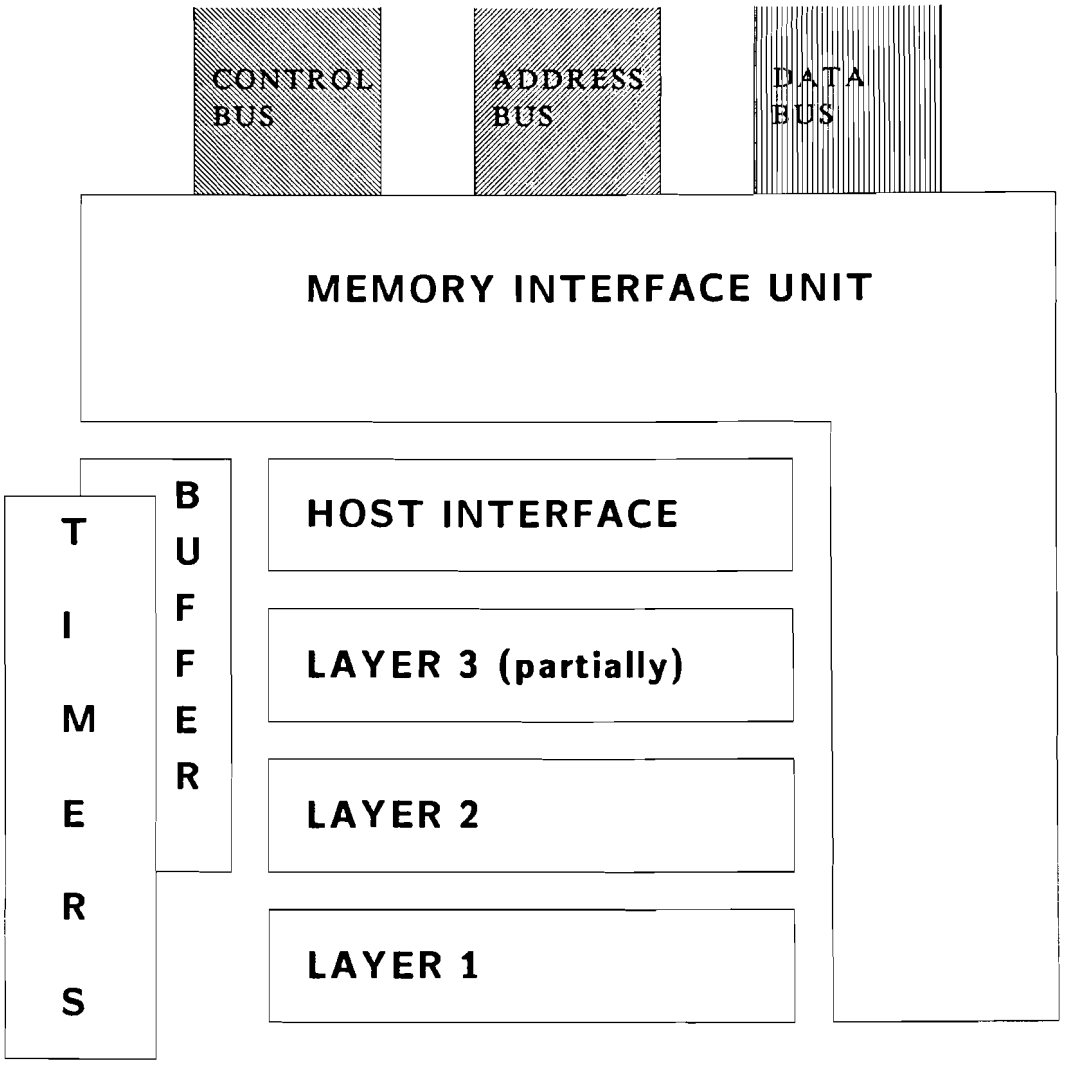


Figure 18 Decomposition coprocessor

The general system architecture is shown in fig 19. As was to be expected this architecture does not differ very much with the proposed architectures in the introduction. In this case one coprocessor implementing the layers 1 till 3 is used. It can have private memory for the coprocessor administration, but this is optional. The host processor private memory also is optional. Both local administrations (of host and coprocessor) can be located in the shared memory. The rest of the shared memory is used for data transfer via buffers and for interprocessor communication. Since both host and coprocessor administration might represtent a considerable amount of data, it can save a lot of shared memory to use private memory. Because one private memory is in the same memory space as the shared memory there is a clear tradeoff between the two.
In the next part we will describe the function of the several blocks and their hardware interface. The administration data structure and memorymap will be discussed in the next chapter.

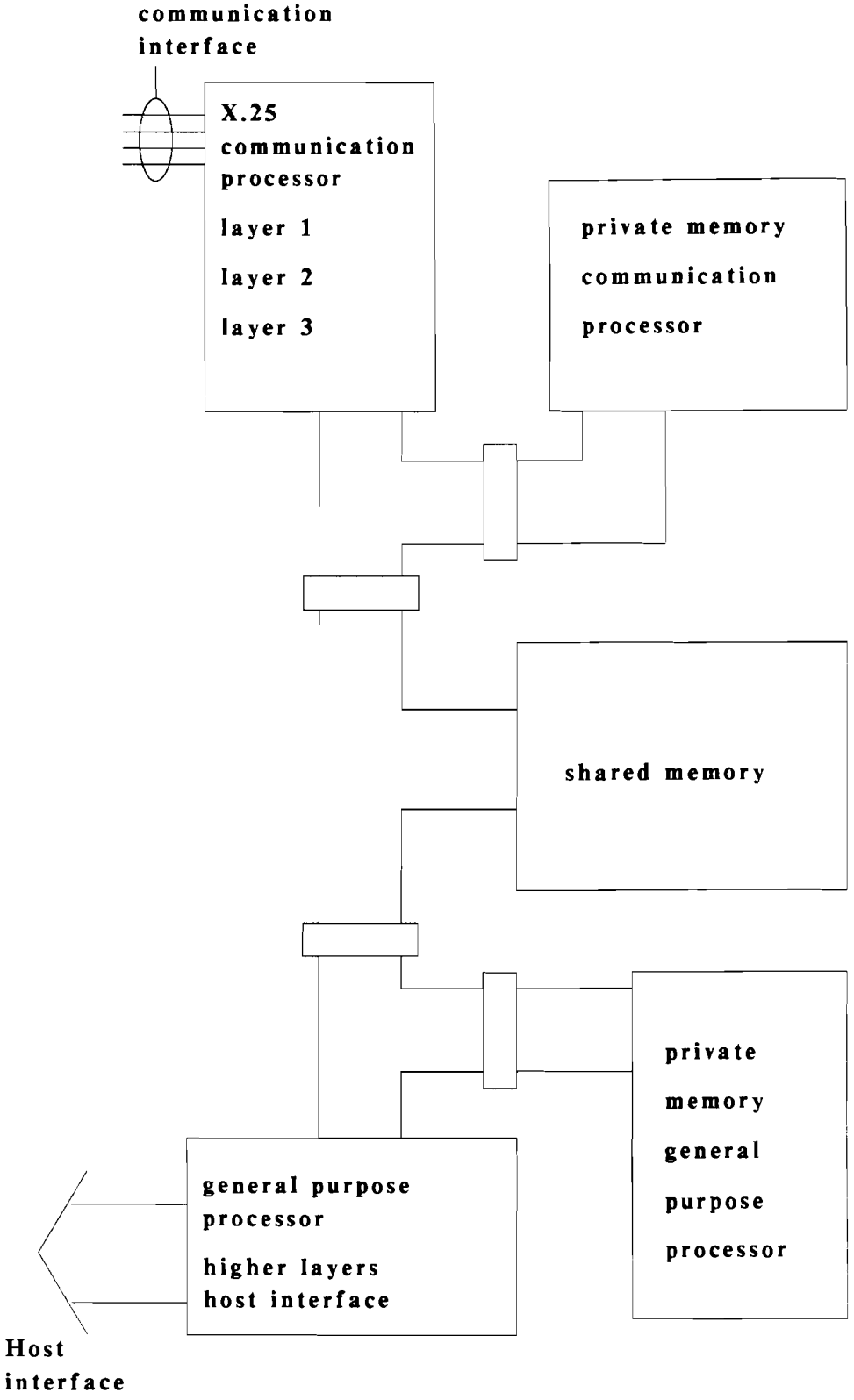


Figure 19 System architecture

4.1 Memory interface Unit

First we will describe the several blocks that do not have a communication function, to begin with the memory interface unit. This unit must interface between the physical memory organistion and the logical memory organisation that is used by the blocks intern. The interfaces to this block are shown in fig 20. The blocks have a register interface where they can request a read, write or exchange operation on a memory address. This can be a word or byte operation. The memory model used by the blocks is a lineair byteoriented memory with 24 address bits, so maximal 16 Mbyte datamemory can be addressed. This is the total of private memory and shared memory, since they form one address space.

The function of the MIU (memory interface unit) is to implement some priority scheme for the several requests that can occur. This block also generates the appropiate control- and bussignals. How this information is given is subject for further study. One can think of the intel construction used for example with the 8089 I/O processor where the busconfiguration is stored in memory in a way that it is accessible in all busconfigurations. Another possibility is to use hardwired inputsignals on certain pins that indicate the used bus, or to design several MIU's one for each busconfiguration.
The commands that can be given by the blocks are:

1/2. read word/byte
3/4. write word/byte
5/6. exchange word/byte

The according parameters are a 24 bit address and in case of an exchange or write command the new data. If a word operation is requested there have to be 2 bytes data (higher order and lower order byte), when a byte operation is requested only the lower order byte of the data paramater has to be offered. Maybe a two byte operation is desirable in the future to provide better bus usage in 16 bit systems. This operation is different to the word operation since with some processors the high order and low order byte are store in another sequence as two bytes would be. With this mode the lower order byte is byte 1 and the higher order byte 2.
The exchange command is necessary to implement semaphores or other synchronisation constructions. Because several processes will operate on the same variables in memory there has to be a strict organisation to maintain dataintegrity. The variables have to be protected somehow. This can be done by using well known software synchronisation constructions.

MIU: Memory Interface Unit

BI: Bus Interface

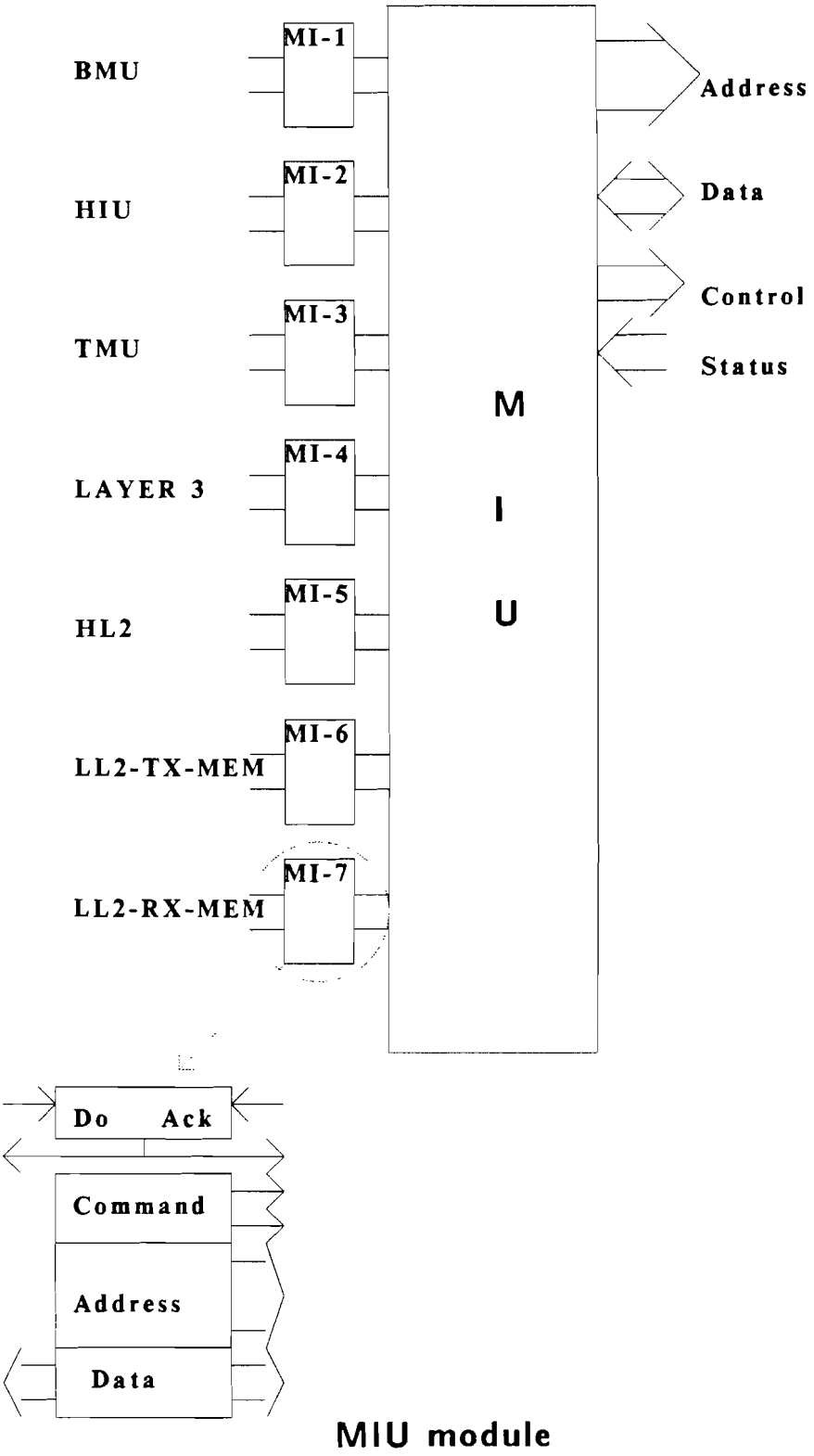


Figure 20 MIU interface definition

4.2 Buffer manager

The on-chip buffermanager is ment to be a reasonable simple process or hardware block. It has the function to keep enough buffers in store to garantee a smooth operation of the chip. To do this it can communicate with the host buffermanager via the host interface unit. It is connected via interface registers to all communication blocks, the host interface and the memory interface unit. It can request buffers to the host buffer manager if there are buffers short and return buffers if there are too many, via the host interface. The buffer manager maintains linked lists with free buffers for every interface port, since each of these lists may contain buffers of a different length. The architecture of this block is shown in fig 21

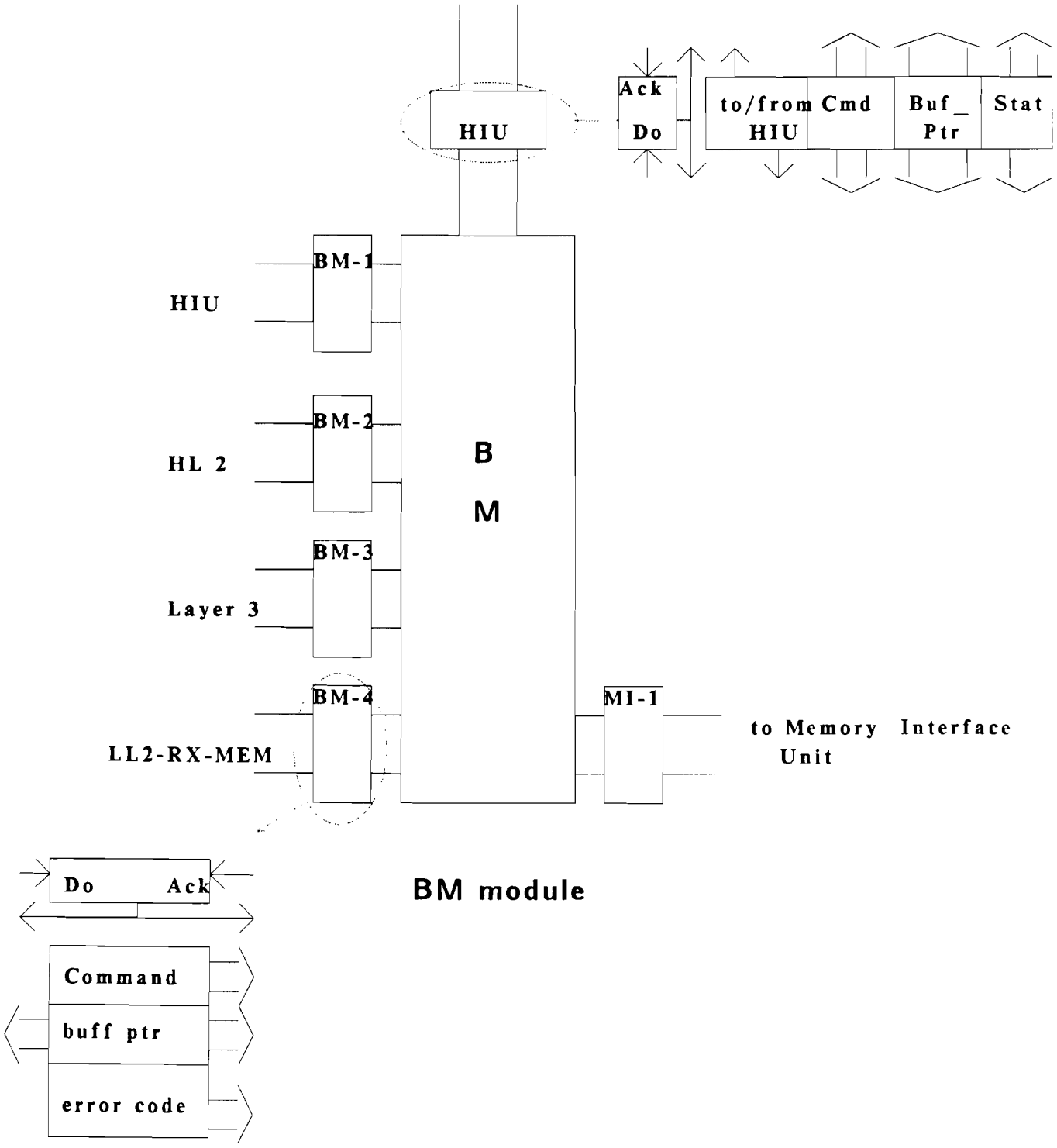


Figure 21 Interface definition of the buffer manager.

4.3 Timer Unit

The timer unit maintains all timers for all layers. To do this it contains a real time clock which ticks in at arbitrary but fixed periods of time. For this reason it gets startup parameter at initialisation time to generate the clocktick from the system clock. Since the system clock is different in each application this value also differs.
All blocks that use timers are connected to the timer unit. Via the interface port only three commands can be passed:

1. to the timer unit, Start timer
2. to the timer unit, Stop timer
3. from the timer unit, Timer expired

With these three messages all timer operations can be implemented. Together with the start command a timer identification and a timer value (in reference ticks) have to be passed. This identification has to be unique to the port the start command was issued. The timer unit has to add a port identification to this timer ID. The timer value is added to the current clock value modulo the maximum clock value (That has to be larger than the largest timervalue). After this the timer is stored in a running timer list. This list can be maintained in the coprocessor, but in that case only a few timers can be supported. A more likely candite for storing the list is the private or shared memory. The first timer that will expire is at the head of the list, and the expiration time is also stored in the timerunit. When the current clock value is the same as the expire time of the head of the list, all timer-records with this time are transfered to a list containing all expired timers. Finally the timers from this list are removed one by one, a timer expired response is generated at the interface port and the timerrecord is added to the free timerrecord list. The timer unit has to maintain three linked lists with timerrecords: the running list, the expired list and the free list. At initialisation a list of timerrecords is passed to the timer unit. For applications that use a restricted (small) number of timers this will be sufficient. If an application can use very many timers at high traffic but normally very little, an other strategy might be applied. In this case the timer unit could be made so it generates its own timerrecords from data buffers it can request from the buffermanager. In this case normally it would need just a few timerrecords, but if suddenly much more timers are needed it can request for another buffer to generate records. When the hausse is over and that many timers are not needed anymore the timer unit can request for another new buffer, convert it to timer records, replace the current running and expired timers by the new records and return all buffers that are in use but the last new one. There has to be requested a new buffer because all timers that are running or expired will probably be scattered through all old data buffers in use; so the only solution is to do some garbage collection to a new buffer and return all old ones.
Apart from these operational commands some error messages might be necessary to implement. Possible errors can be:
With the start command: No records left, Time too long.
With the stop command: Timer not found.
In [NISS] an extensive description is given on different implementations of the timer unit.

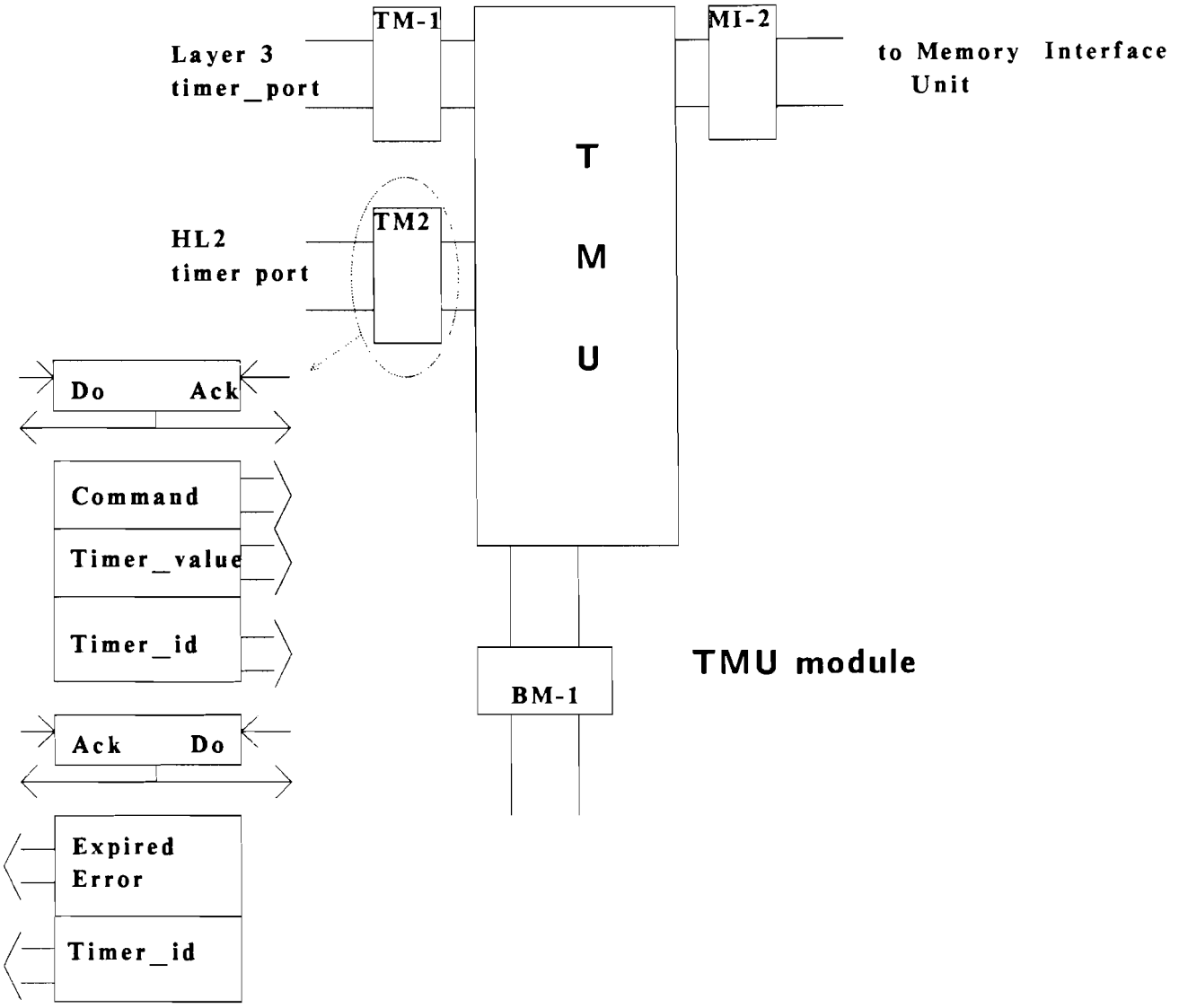


Figure 22 Interface definition of the timer management unit

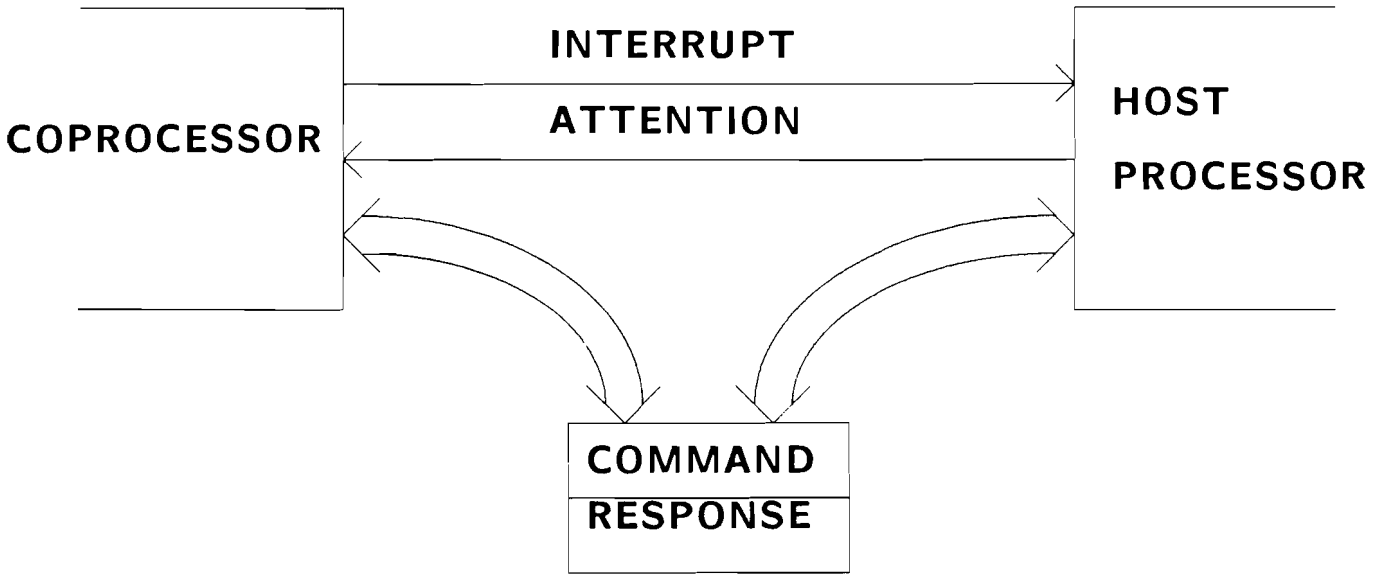
4.4 Host interface

The host interface is the block that handles the communication with the host. Together with the communication interrupt handler program on the host it links the various programs that run on the host to the according blocks in the coprocessor. The exact command's that are implemented depend on the place the host interface is situated. There must be command's for the onchip buffermanager to communicate with the host buffermanager. If the host interface is situated between two layers the commands must be according the service primitives.

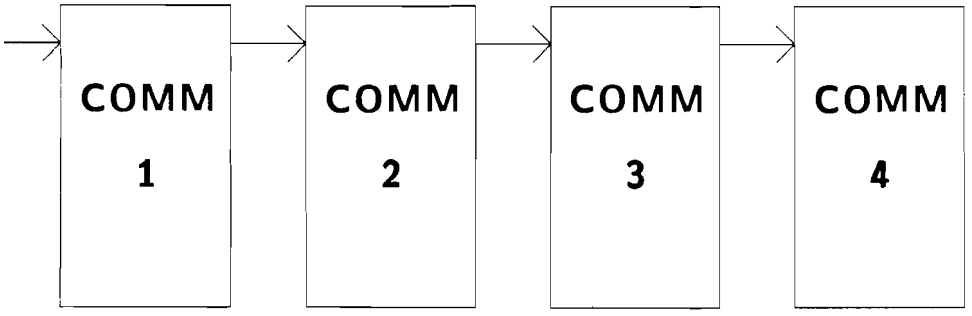
As mentioned above the communication between the coprocessor and the host is done via an interrupt mechanisme. The mechanism is shown in fig 23.
In shared memory two fields are reserved for the host-coprocessor communication. In these fields pointers to command lists can be stored. When the host wants to give a command to the coprocessor it stores the command in a buffer. The host can give more than one command at a time by linking several command buffers together. A pointer to this command is stored in the command area. When this is done the host issues an attention to the coprocessor, which knows now that there is a command available to process. After the command is accepted the command area can be cleared by writing the nill-pointer. When the command area is cleared the host can write the next (series) of commands. The host has to check the command area to make sure it is free, before it can give new commands. In this way an implicit synchronisation is obtained with respect to shared command area. When it contains a pointer to a command, only the coprocessor is allowed to clear it. The host can check this by polling the command area before giving a command. When there is no command pending the host is allowed to write the pointer for the next command(s), and notifies the coprocessor by means of an attention signal. The response area is used in a similar way. The only difference is that the information flow is now the other way round, and the host is notified by means of an interrupt.

As already mentioned before the exact commands and responses that can be exchanged between the two processors depends on the place of the HIU in the design. For communication purposes the commands and responses must be some sort of primitives, but since the layer 3 implementation is partly in hardware and partly in software in our case these must be chosen by the implementor of layer 3. There are however other blocks that are connected to programs on the host by the HIU. For these blocks some proposals can be made.

Host Interface Unit



COMMANDS / RESPONSES



-WRITE COMMANDS / RESPONSES

-GIVE ATTENTION / INTERRUPT

Figure 23 Host interface mechanisme

For the communication between the buffer managers there must be several commands to exchange buffers. The exact form is subject for further study but we can already make a proposal.
To request or release buffers from/to the host manager the following responses are proposed:
Buffer_request (# blocks, # bytes per block)
Buffer_release (pointer to first buffer)
Header_request (# headers)

The buffer_request response is given when the on chip buffermanager runs out of buffers. To put no constraints on buffer sizes or on the hostmanager blocks of a specific size are requested instead of buffers. It is up to the host buffermanager how these blocks are composed, as one buffer of the requested size or as a linked list of smaller buffers.

The buffer_release response is used to get rid of buffers on the chip. The buffer manager in the coprocessor can eg have two thresholds, one for buffer request and one for buffer release. If the number of buffers or the number of bytes in free buffers gets lower than the request threshold it has to request for buffers. If the number is higher than the release threshold there are too many unused buffers available in the coprocessor, which is bad for the performance, so the exessive buffers have to be released.

The Header_request is necessary if the protocols in the coprocessor can perform segmentation or separation. In this case buffers might be split half so new headers have to be added. To release header that are not needed anymore the buffer_release response can be used.

To give buffers or request buffers to/from the coprocessor buffer manager the next commands are available:
Header_disposal (pointer to list of headers)
Buffer_disposal (pointer to block, # bytes of block)
Buffer/header_release_request

In answer to buffer or header requests of the coprocessor the host can give them to the coprocessor by means of the disposal commands. Only one block can be given through the buffer_disposal because otherwise the it would be implossible for the coprocessor to determine to which request this is an answer (this can be determined via the # bytes field). That this is no constraint is clear when you realise that commands can be linked. If the host wants to pass more blocks at once to the coprocessor it just links several buffer_disposal commands.

The buffer/header_release_request can be given when the host buffermanager runs out of buffers or wants to perform an operation for which it needs as many free buffers as possible (eg. garbage collection). It gives the buffer/header_-release_request to which the coprocessor has to respond with a buffer_release response containing all superfluous buffers and headers (eg. all buffers and headers it has over the request threshold).

If processes on the host also want to use the Timer management unit the commands and response have to be extended with commands to start, stop and cancel timers. For this reason the same commands that can be issued at an interface port to the TMU have to be added to the commands and responses.

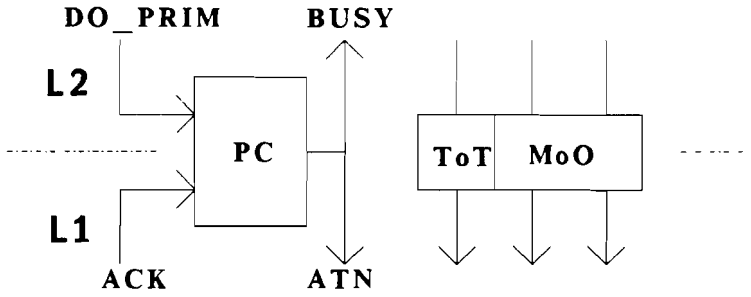
4.5 Layer 1 interface.

The functions of layer 1 are not very complicated and can be implemented through one or more finite state machines. That does not mean that it is easy to implement this layer. Because the standards that have been defined for this layer are not clear at all (see eg. X.21). Several characteristics are not or ambiguish defined and without information about the local implementation this can not be implemented.
The primitives that have to be implemented according the standards are:
-PH_activate_request/indication(mode of operation, type of transfer)
-PH_deactivate_request
-PH_deactivate_indication (originator)
-PH_data_request_indication(PH-SDU)

The activate and deactivate primitives are for the layer management unit to prepare for data transmission. The parameters are:
Type of transmission (Asynchronous/ Synchronous)
Mode of operation (Duplex /Half duplex/ Simplex)
In case of the X.21 interface these are always standard Synchronous, Duplex.
The implentation of the primitives is shown in fig 24

The implementation of the data primitive is in this case the most interesting one. The data is transferred serially on initiative of layer 1. This is done to keep the implementation of layer 1 as simple as possible.
Lets first have a look at the PH_data_request primitive. This primitive is used to transfer data from layer 2 to layer 1. Again we can recognize the synchronisation with a primitive control block. Apart from the synchronisation there are three parameters: PH_SDU (the data), CEI (connection endpoint identifier and SAP (service access point). The latter two are implemented as a register, but since the data is transferred serially the PH_SDU is more involved. For the data transfer there are three bits from layer 2 to layer 1 and twe bits the other way around. The first bit from layer 2 is of course the data, which is presented on request of layer 1. This brings us at the first bit from layer 1, the clock, at which layer 2 has to generate the subsequent data bits. To indicate the end of an SDU layer 2 has a bit indicating the last databit (LST). Together with the last databit this bit is activated. Since the data is transferred serially there might be reasons for either layer to abort the primitive preparation. For this reason an abort bit is provided for layer 2 to indicate the abort to layer 1. For layer 1 a collision indication is provided to abort the preparation, since a collision is the most obvious reason to abort the primitive preparation for layer 1.
The primitive preparation works as follows. Layer 2 writes the first data bit, the appropriate Service acces point (SAP) and Physical connection endpoint identifier (PH_CEI) to the according registers. The PH_CEI is used in case more than one layer 1 are connected to the same layer 2. This field is the identification used to distinguish the several layers. After the parameters and the first databit are written layer 2 starts the primitive preparation via the primitive control block. After the primitive is started layer 1 provides a bitclock at which layer 2 has to react by supplying the subsequent databits. It is up to layer 2 to be fast enough with the next bit or to solve the error situation in case of an underrun. Layer 2 also has to solve the error in case of a collision, which is told via the col bit. In case of X.25 this can not happen since there is only one source and one destination. To solve underrun errors or for other reasons layer 2 could decide to

cancel the transmission. This can be done by setting the abort bit. In normal situations the primitive is ended by setting the last paramater together with the last data bit. Layer 1 acknowledges the primitive by resetting the PCB. The abort bit is introduced because some layer 1 protocols might have means to indicate this situation to their peer-entities.
With the PH_data_indication we can recognize the same parameters as with the request: PH_SDU (data), PCEI and PH_SAP. The direction of the parameter transfer is now the other way around. Since layer 1 receives the data and transfers this to layer 2 the PCEI and PH_SAP have to be presented by layer 1 to layer 2. Also the data bits come from layer 1, together with the clock and the bit indicating the last data bit of the transferred SDU. Again we find the col(lision) bit and the abort bit to end the primitive preparation previously.
The primitive preparation now starts on initiative of layer 1. After layer 1 has written the PCEI , PH_SAP and the first data bit it starts the primitive preparation via the primitive control block. Layer 2 is now obliged to store the subsequent data bits as they are presented on the clock (that is provided by layer 1). The preparation can be canceled by layer 1 through the collision bit and by layer 2 via the abort bit (eg. in case of an overrun, when the data bits are presented to fast after another). After layer 1 transferred the last bit of the SDU, indicated through the LST bit, layer 2 acknowledges the PH_data_indication by reseting the primitive control block. At this moment the primitive is considered to be taken place, and both protocols can proceed to their next protocol state.

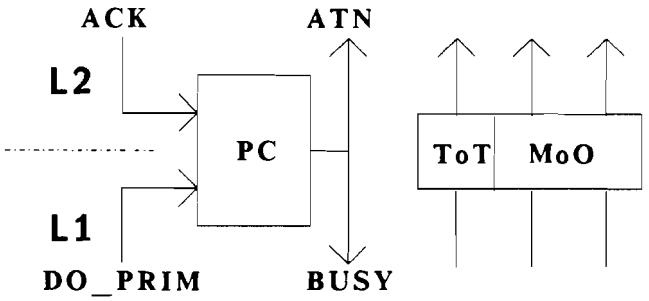


PC : Primitve Control Block

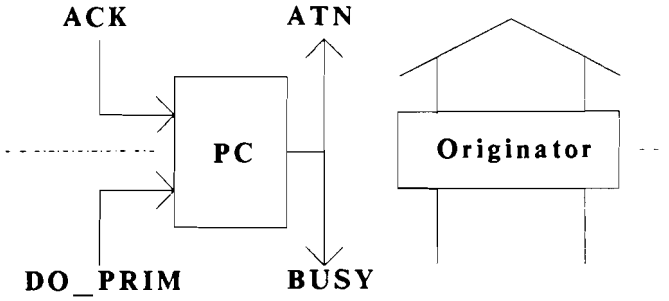
MoO : Mode of Operation

ToT : Type of Transfer

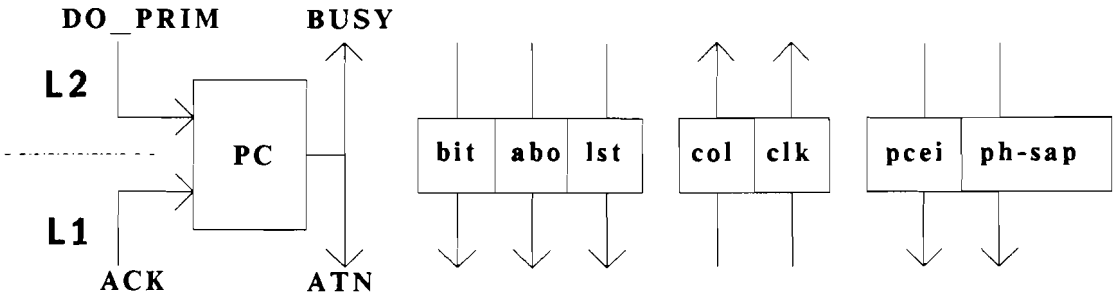
PH-ACTIVATE-REQUEST



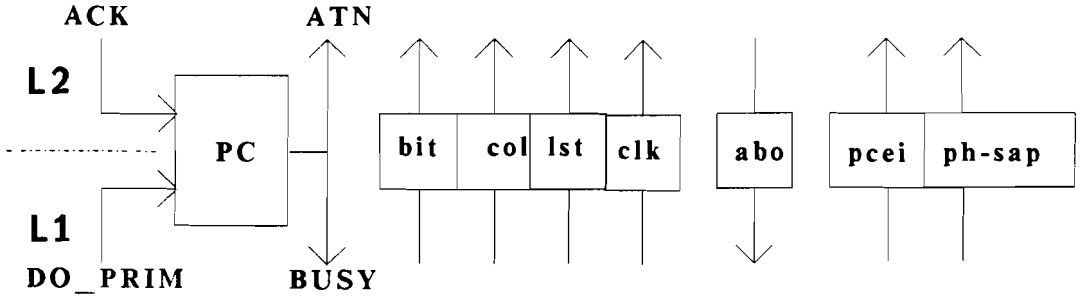
PH-ACTIVATE-INDICATION



PH-DEACTIVATE-INDICATION



PH-DATA-REQUEST



PH-DATA-INDICATION

Figure 24 Interface definition Layer 1 primitives

4.6 Layer 2 interfaces

One of the functions of layer 2 is, beside the protocol processing, the transformation of the data from serial data to buffers and the other way around. In this way the pressure of the bitclock is hidden to higher layers, and response times are changed from bittime to frame time. To perform this transformation two memory modules are defined that must take care of the transparent (with respect to buffer implemention) conversion from serial data to buffers and vice versa.
To understand the decomposition chosen for the protocol processing block one must realise that the layer 2 protocol (HDLC) consist of two subprotocols:
-the bit protocol.
-the frame protocol.

The bit protocol (flags, bitstuffing, idle- and abort pattern) takes care of the transparent transmission of frames. Exactly this protocol makes it difficult to fit the X.25 level 2 into the OSI model, since these functions are really layer 1 functions. In the OSI model layer 2 provides layer 1 SDU's which are transmitted to the peer entity and offered as an layer 1 SDU to the peer entitie in layer 2. Because of the bitprotocol in X.25 this function can not be performed by layer 1, since layer 2 always transmits data and takes care of the frame delimiting itself.
The frame protocol is a true layer 2 protocol since it has clear layer 2 PDU's and performes the functions that are prescribed for layer 2.

From these functions an architecture as shown in fig 25 is derived. The High level 2 module performs the frame protocol , the low level 2 modules perform the bit protocol (and calculate and process the FCS) for the receiving bitstream (RX) and the transmitted bitstream (TX) respectively. The two memory modules take care of the conversion from and to buffers.

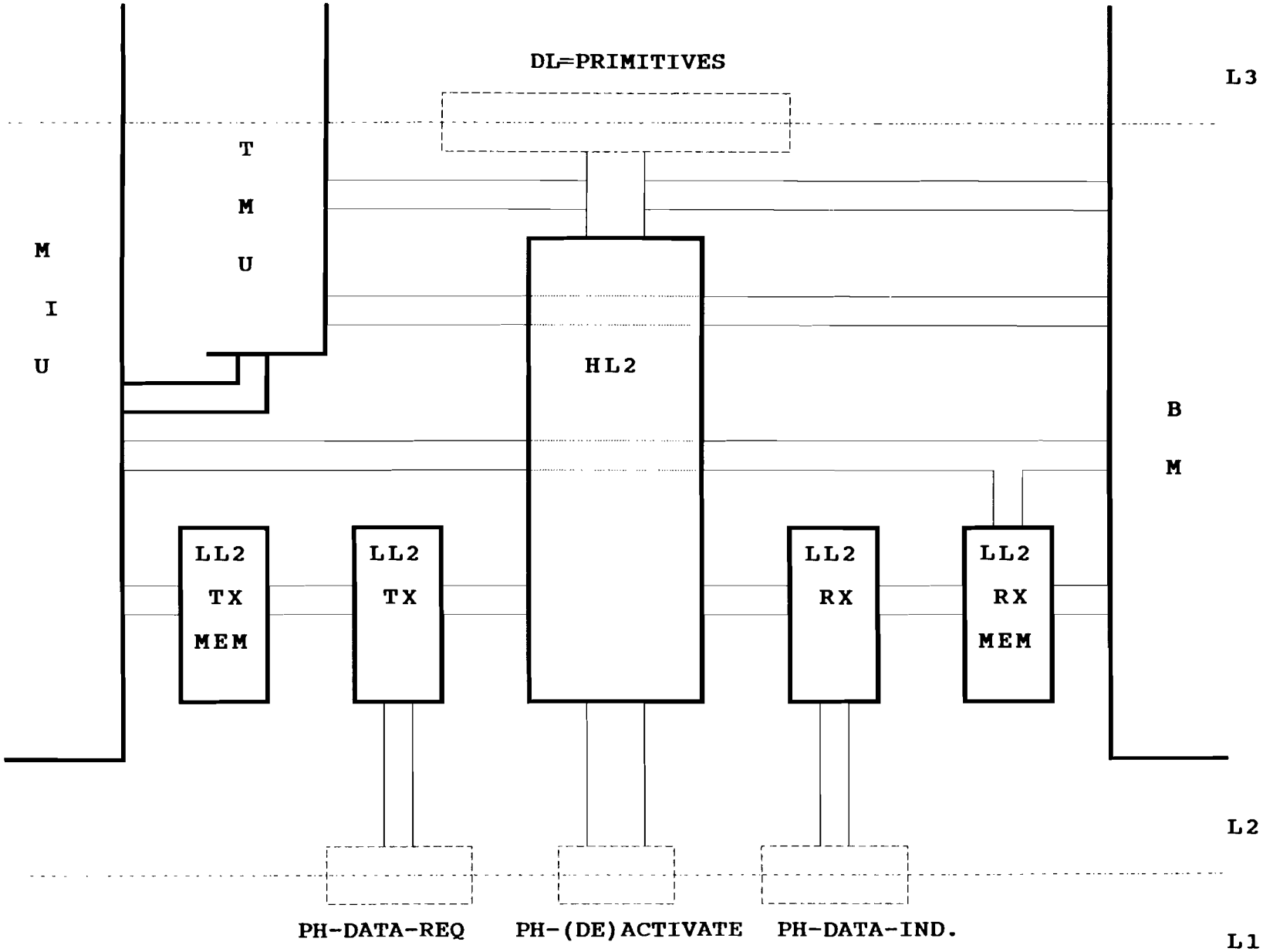


Figure 25 layer 2 architecture.

Low level 2 transmitter and memory interface

These two units perform part of the layer 1 function. A layer 1 SDU is passed via the HL2 interface. This interface is according the PH_data_request definition. Together with the SDU a SAP identifier and a CEI are passed to indicate the layer 1 entity that has to be used for transmitting the data. These last two values are passed to the layer 1 interface. The data can be passed via the HL2 interface (with X.25 in case of supervisory and control frames), via a buffer in memory or partly via the interface registers and partly via memory (with X.25 in case of an I-frame). For this reason several fields are reserved in the interface. One bit is used to indicate whether the SDU is passed via the interface only or also via buffers. If the SDU is passed via the interface only, the following field is used to pass it. Another field is used to indicate the number of bits of that is passed via the interface. This complex interface is chosen because all frames in X.25, exept I-frames are very short (max 8 bytes, but more often 2 or 3 bytes). In this case it is easier to pass these explicitly via the interface, since the overhead to store it in buffers is much to large. The header of an I-frame can also be passed via this interface. Since in other cases it might be desirable to pass the SDU via buffers only, this is possible by choosing the number of bits to zero. A pointer to the first buffer is also passed if the SDU is partly in buffers. For the synchronisation a Primitive control block is used.
To stop the transmission in case of a serious error a reset signal is available to HL2 .
To get data from a linked list of buffers in memory an interface to the memory interface is available. LL2 gives a start command together with the pointer to the first buffer. The memory interface stops whatever it is doing and reads the first data byte/word from the buffer. The data is offered to LL2 via another interface bit by bit. The memory interface takes care of linking the buffers.

All interfaces and their configuration are shown in fig 26

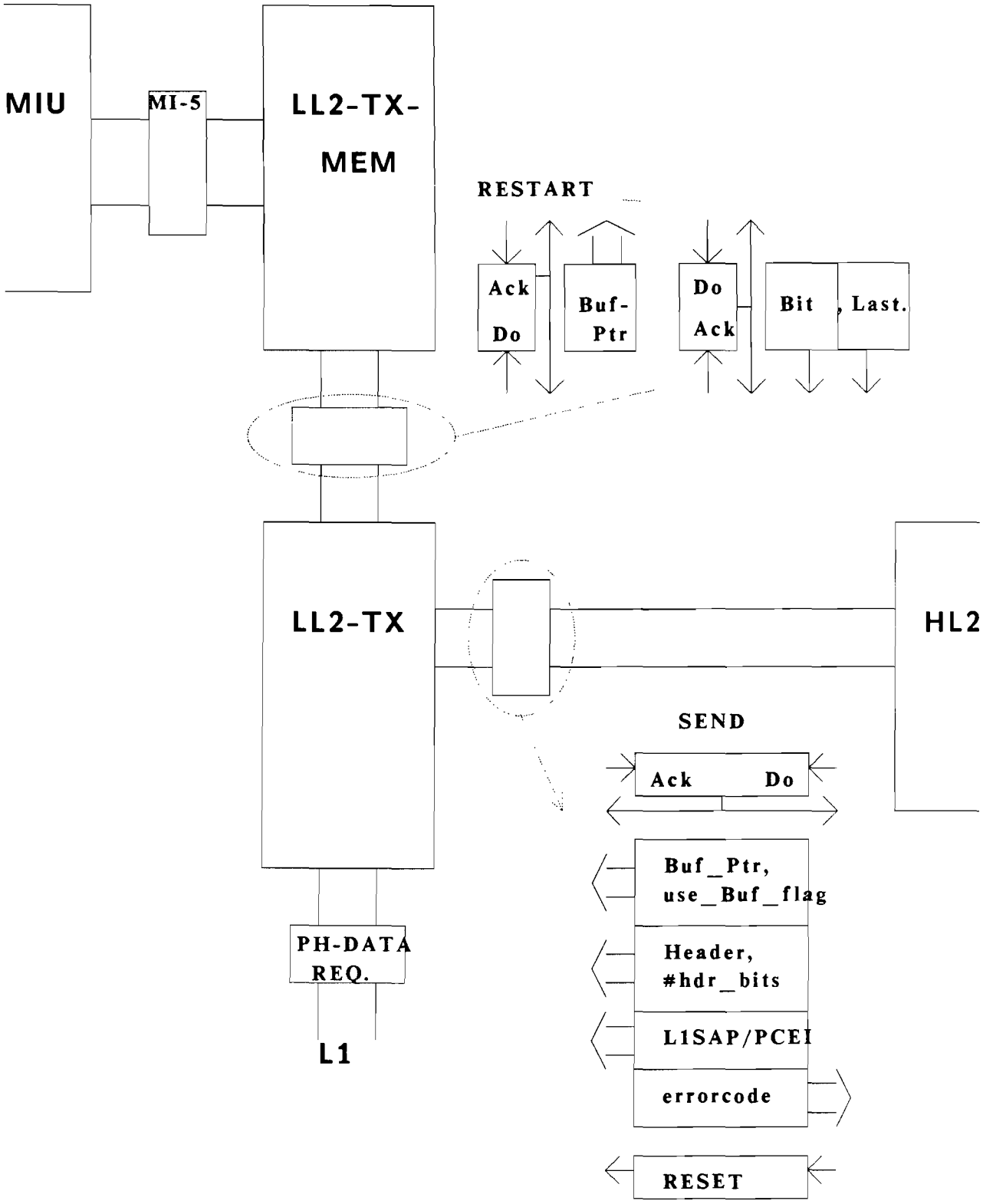
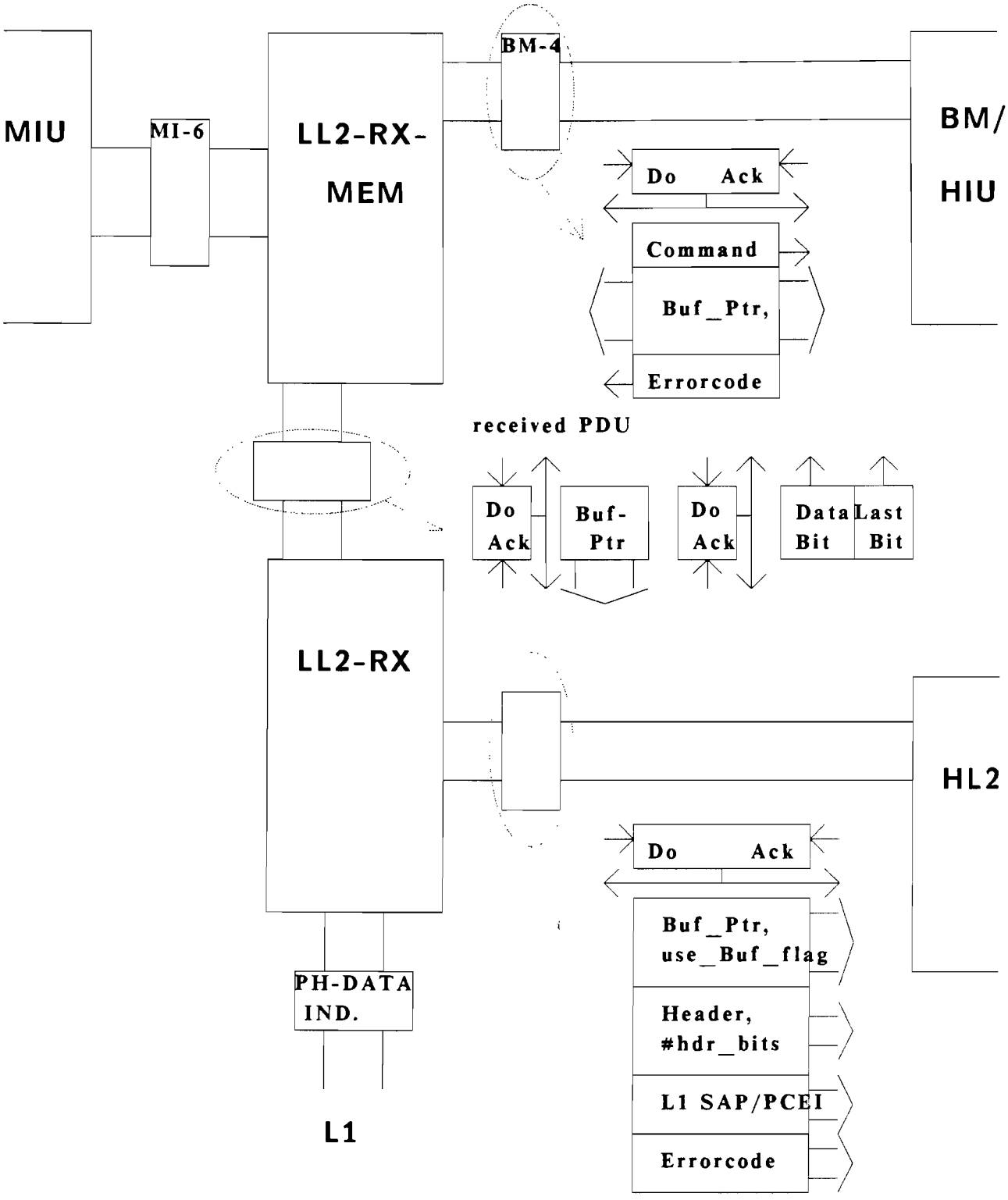


Figure 26 Interface definition LL2 TX en memory manager.

Low level 2 receiver and memory interface

The LL2 receiver and the memory interface perform the complement function of the transmitter part. The LL2 RX receives incoming data bit by bit via the L1 interface. If a starting flag is recognized the data is passed to the memory interface and stored in memory. Parallel the first 3 or 4 bytes are copied to be passed to HL2. The memory interface takes care of storing the incomming bits in buffers, linking and requesting the buffers. After the reception is stopped (because of an overrun, correct or incorrect end of a frame) the buffer is updated and the pointer to the first buffer is passed to the LL2 RX. After receiving a correct or incorrect frame the data buffers are passed to HL2. If the reception was incorrect an error code is added, otherwise the header and the L1 SAP and PCEI are passed.
All interfaces and their configuration are shown in fig 27.



LAYER 2 RECEIVER MODULES

Figure 27 Interface definition LL2 RX en memory interface

High level 2

This module contains a micro controller that runs (with X.25) two programs. The first program, the receiver, does the protocol processing. The second program, the transmitter, is merely a multiplexer that handles the transmission of control frames and data frames. The transmitter eases the programming of the protocol processing and makes it possible to start the transmission of a new frame as soon as the transmission line becomes idle, even when the receiver is doing other protocol calculations. This releaves the receiver from fast response time requirements and makes an efficient use of the transmission part possible. The only thing the transmitter program does is reading the queues (data and control), composing the frames and starting the low level transmitter. The receiver decides when data and control frames have to be written to the queue according the protocol.
The interfaces this module has are to:
-Memory interface unit
-LL2 RX/TX
-Buffer manager
-Layer 3
-Layer 1

The interface to all modules exept layer 3 are described in the previous chapters. The interface to layer 3 consist of same primitive implementation as in layer 1. There are primitives for connection establishment , release and data transfer. The parameters belonging to the primitives (as indicated in CCITT recommendation X.212) can be passed explicitly via the interface or via memory. The exact choise must be made at implementation time since at this moment is not clear which choise is best.

5. Memory map coprocessor

The memory of the coprocessor is (for the modules in the coprocessor) a linear byte oriented memory of 16 Mbyte. Part of it is used for administration of the modules, a few bytes are used for communication between the host and the coprocessor and the rest is used for buffers. The buffers can be used to store data, but also for dynamically allocated variables. In the case of X.25 there is not much necessaty for this since most administration sizes are known because they have to be determined at subscription time. The total memory map is shown in figuur 28

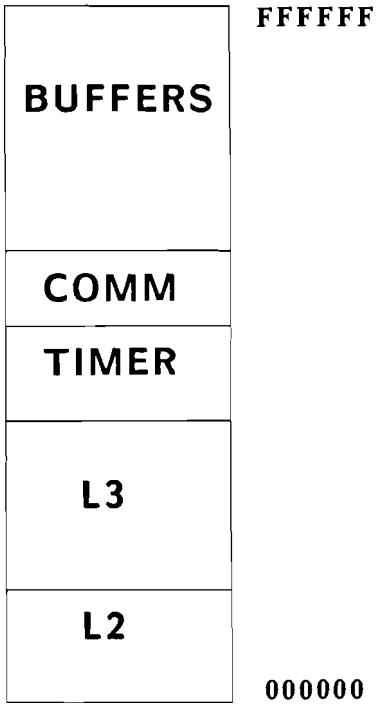


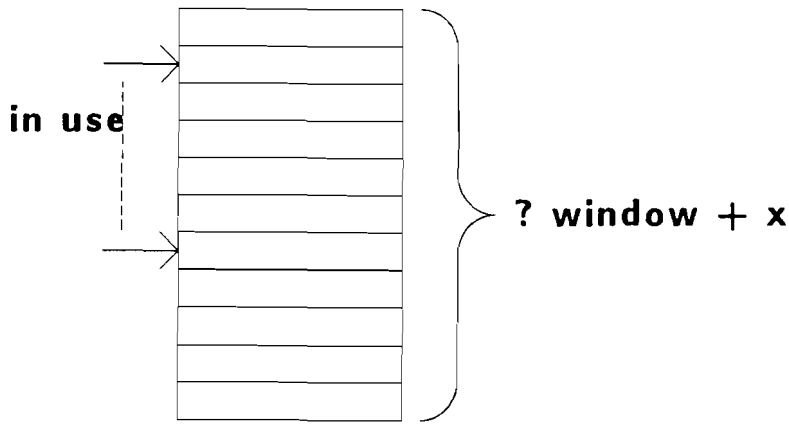
Figure 28 memory map X.25 coprocessor

The layer 2 administration is small enough to be stored on the chip except for the transmit queue (see also [KLIP]). The transmit queue must be at least the window size and this can be maximal 128. To make it possible for the transmitter to resume immediatly after acknowledgements it is sensible to accept a few more packets from layer 3. So the queue length should be the window size plus a few places. Since this window size is known at initialisation time the queue space can be reserved. The queue can be implemented as a circular buffer that is as large as the window and a bit more.
The layer 3 administration consists of context blocks for each channel. The number of channels is known at initialisation so enough memory space can be reserved. The queues can also be implemented as circular buffers, but now a problem occurs. Since some networks allow the windowsize to be negotiated about on a per call basis this cannot be reserved on beforehand. A solution would be to

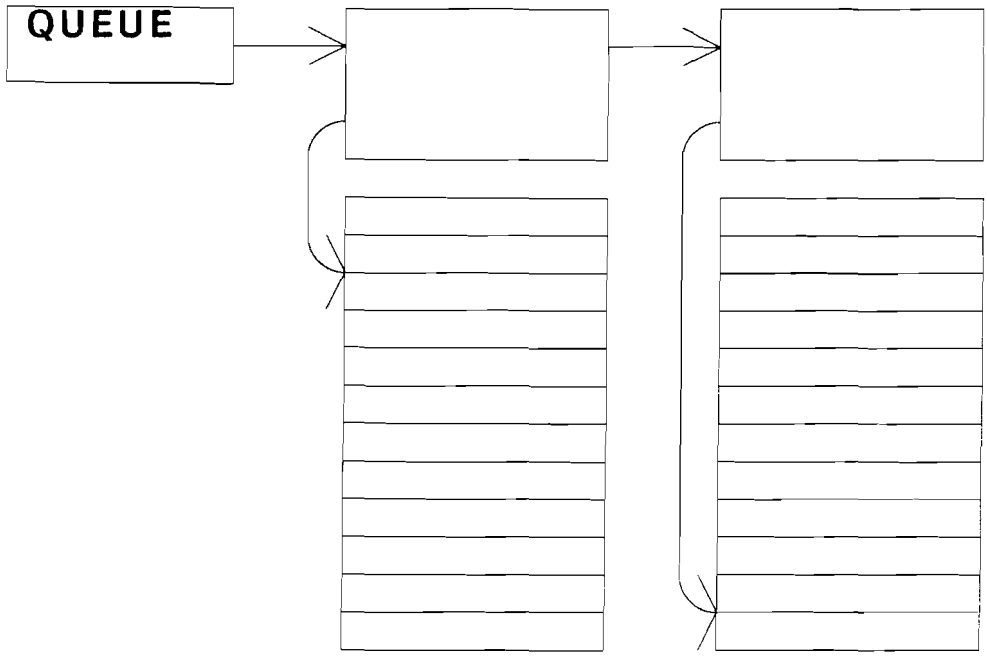
reserve a space large enough to contain the maximum queue length. Since the maximum window is 128, but more often smaller windows are used, this is clearly inefficient. A more sensible solution is to create the queue during the call establishment phase. The queue can be implemented in data buffers, so only a pointer to the queue has to be stored in the context block. If the buffer is to small to contain the complete queue, several buffers can be linked together. The circular buffer is now implemented through all buffers. If the last byte of the last buffer is filled the first buffer is used again. These queue implementations are shown in fig 29

QUEUES:

LEVEL 2



LEVEL 3



Figuur 29 Level 2, 3 Queues

6. Conclusions

The design that is presented forms a good basis for further study. The decomposition that is chosen makes it possible to derive separate projects for the implementation of the several modules. These project are feasable in a university environment where students only have limited time do their research. The protocol implementation is so involved that it is implossible to study, understand and implement a protocol within this time. Because of the clear boundries it is possible to implement on module without knowing anything but the interface definition. The OSI service definitions lead to a protocol independant interface. The services are not fixed yet (for the Physical services in 1992 a definite OSI standard is expected), and more primitives will be needed for management and statistics functions. In prelimenary publications these are already defined but no implementation or coding is given. So a lot of work still has to be done.

The earlier designs can be adapted to fit in the proposed framework. In this way a library of layer implementations could be available. A designer could choose from this library the layer implementation suits his requirements.

More research needs to be done on how to implement the initialisation. Also the conformance testing and managment operation have to be defined further.
An interesting subject of study can also be the conformance validation and implementation validation. The clear boundries and high level services that have to be offered are well suited to be tackled using formal description techniques.

Resuming we can say this study has proven that service definitions are not just an abstract way to define protocols, but are a sound basis for protocol implementation. This design is the first step toward a universal protocol architecture and much research and work is waiting for new students to be done.

LITERATURE

[CCITT-3]
CCITT Red book
Volume VIII fascicle VIII.3
Recommendations X.20-X.32
Geneva 1984
ISBN 92-61-02321-5

[CCITT-5]
CCITT Red book
Volume VIII fascicle VIII.5
Recommendations X.200-X.250
Geneva 1984
ISBN 92-61-02341-X

[Deasing]
R.J. Deasington
X.25 Explained
Ellis Horwood 1985
ISBN 0-85312-626-7

[DNL]
Dr.Neher Lab. (DNL) Dutch PTT
Integrated Services Digital Network colloquium
Dec. 86

[HALS1]
F. Halsall
Seminar: OSI Protocol and Implementation (10/12-3-87)
ORSYS Institute 17, Shirley Gardens, London W7 U.K.

[HALS2]
F. Halsall
Introduction data communications and computer networks
Addison Wesly 1985
ISBN 0-201-14540-5

[JANS]
P.A. Janson
Operating systems structures and mechanisms
Academic Press 1985
ISBN 0-12-380230-x

[NIJH]
J.A.M. Nijhof
Syllabus symposium "Informatie over Communicatie"
KIVI-ASI-NGI, 11 march 1987
ir. J.A.M. Nijhof, Archictuur en Implementatie

[NGI-SIC]
Proceedings van de telematica-3 conferentie
20 Nov 1986 NGI-SIC
ch. 3 Basis (architectuur concepten van het OSI ref. model
C.A. Vissers
ISBN 90-5005-010-7

[PHIL]
Philips
X.25 DTE/DCE interface
Pilips data systems Apeldoorn nov. 1983

[STAT1]
State of the Art report 14:3
Communication standards
Pergamon Infotech Ltd 1986
ISBN 0-08-034-092x

[STAT2]
State of the Art report 11:8ii
Local Area Networks
Pergamon Infotech Ltd 1983
ISBN 0-08-028-5813

[TANEN]
A.S. Tanenbaum
Computer networks
Prentice Hall 1981
ISBN 0-13-164699-0

[Viss]
Vissers/Logrippo
Protocol specification, testing and verification V
M. Diaz (editor)
IFIP, 1986 Elsevier Science Publ. (North-Holland)

Related Master thesis reports

Eindhoven University of Technology
Department of Electrical Eningeering
Digital Systems group (EB)
PObox 513, 5600 MB Eindhoven

[BAKK]
E.P.M. Bakker
X.25 Coprocessor Design
Level 3 Software Interface
1986

[KLIP]
A. Klip
X.25 coprocessor, Functional design level 1&2
1985

[Luyt]
R. Luyten
X.25 Functional Design
1984

[NNIS]
N.T. Nissink
Conformance Testing for OSI, Theory and practice
1986

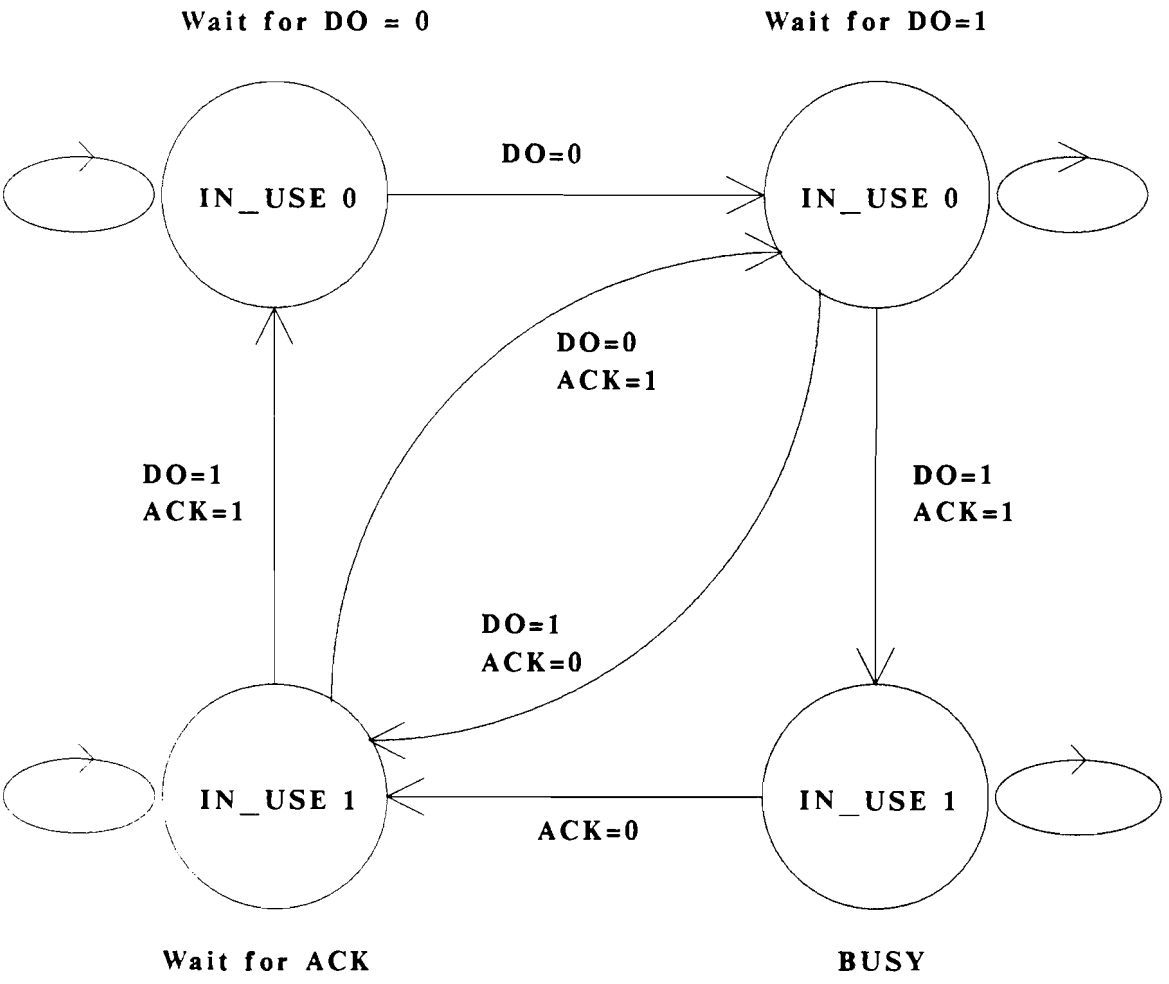
[NISS]
P.L.H.M. Nissink
Design of an ISDN coprocessor
1987

[OOIJ]
H. van Ooijen
Design level 2, X.25 coprocessor
1986

APPENDIX A: Primitive Control Block definition

DO : DO_PRIMITIVE

ACK: PRIMITIVE ACKNOWLEDGE



State diagram for primitive-control-block