Functional design of a chip for implementation of the CCITT X.25 protocol

Luijten, R.P.

Award date:
1984

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

FUNCTIONAL DESIGN OF A CHIP FOR IMPLEMENTATION OF THE CCITT X.25 PROTOCOL

by R.P. Luijten

Report of gradation project by Ronald P. Luijten
Supervisor: Prof.ir. A. Heetman
Coach: Ir. M.P.J. Stevens

Eindhoven, The Netherlands
October 18, 1984

SUMMARY

This is a report from my final work of my study at the Eindhoven University of Technology, at the department of electrical engineering. In this project a functional design has been made of a chip that implements the complete CCITT X.25 data communication protocol, which is used on worldwide packet switched networks.

A study has been made of X.25, and it was researched which X.25 functions should be implemented in the chip. Also studied was where this chip should be used, and what kind of systems it should be possible for the chip to be used with. This chip can be used to implement a simple as well as a sophisticated DTE, with a large range of host microprocessors, including modern 16 bit versions.

One of the goals was to minimise the hosts overhead in X.25 protocol handling, which leads to the incorporation of a three channel DMA controller. This makes it possible for the chip to access shared memory autonomously. This implies that a memory management system must be used. The memory management system was also designed, keeping in mind that on chip data storage had to be kept to a minimum. Special data structures are held in shared memory which are designed to fit X.25, but also to keep hardware within the chip as simple as possible and also to maintain efficiency.

A functional interface also has to be designed, forming the command primitives for the chip.

The design method followed in this project, is based on techniques as found in high level programming languages such a Pascal. A decomposition of X.25 into a number of subtasks was performed, which each are implemented by means of sequencers. A hierarchy of such sequencers implements the complete protocol.

The layout of this VLSI chip has not been designed in this phase of the project, but use was made of regular structures which simplifies the layout generation, where wild logic merely complicates it.

Also testability has been considered, let it be a functional one. As no integration technology has yet been chosen, no technology dependent testing has been studied.

This report contains the design of an X.25 chip at datapath level, including machine definitions to be found in appendices of this report. One of the conclusions is that it seems very well possible to implement the X.25 protocol on a single chip.

List of abbreviations.

ALE	Address Latch Enable
ALU	Arithmatic Logic Unit
BHE	Bus High Enable
CAD	Computer Aided Design
CCB	Communication Control Block
CCITT	International Telegraph and Telephone Consultative Committee
CMDR	Command Reject
D-bit	Delivery bit
DCE	Data Circuit terminating Equipment
DMA	Direct Memory Access
DTE	Data Terminal Equipment
F-bit	Final Bit
FBT	Free Block Table
FCS	Frame Check Sequence
FIFO	First In First Out
FRMR	Frame Reject
FSM	Finite State Machine
GFID	General Format Identifier
HDLC	High level Data Link Controller
HIC	Highest Incoming Channel
HOC	Highest Outgoing Channel
HTC	Highest Twoway Channel
LAN	Local Area Network
LAP(B)	Link Access Protocol (Balanced)
LIC	Lowest Incoming Channel
LOC	Lowest Outgoing Channel
LSB	Least Significant Bit
LTC	Lowest Twoway Channel
MSB	Most Significant Bit
OSI	Open Systems Interconnection
PLA	Programmable Logic Array
PVC	Permanent Virtual Circuit
Q-bit	Qualifier bit
R	Reset
RNR	Receive Not Ready
RR	Receive Ready
S	Set
S-bit	Sequence bit
SEC	Second
VLSI	Very Large Scale Integration

CONTENTS

1. INTRODUCTION.

This report describes the functional specification of a chip that implements the X.25 protocol, of which the design is performed as the final work of my study at Eindhoven University of Technology.

This design started from the CCITT Yellow Book Fascicle VIII.2, which describes amongst others the X.25 protocol. The design does not include a layout of the chip, it defines the system to be integrated rather than how to integrate it. Furthermore, no technology has been chosen, which is to be used for integration.

It is advised for the reader to have some knowledge about the X.25 protocol, and also to be familiar with the basics of switching theory (5).

2. Why X.25 ?

At present, there are no chips commercially available that implement the full X.25 protocol. This mainly is because of complexity problems. A number of chips are available that implement parts of X.25 only. An example of a very simple chip would be an HDLC-controller, which only performs the lowest level functions of X.25 such as flag hunting, zero-insertion, zero-deletion and FCS generation and checking. These chips need a great deal of additional software to complete the X.25 protocol. A more sophisticated chip would be the Western Digital WD2511 Packet Network Interface (Link Level Controller), which implements the second level of the three levels of X.25 This chip implements about half of the X.25 protocol, since level 1 is only very small.

As a result, present X.25 implementations need large printed circuit boards, together with expensive software.
The chip described in this report implements the full X.25 protocol.

3. Goals of this project.

One of the goals of this project is to get insight in complex architectures of chips, and methods for designing such architectures. An approach to be used, is the Pascal-like division into procedures of a complex task. These procedures will be implemented in hardware by use of sequencers or combinatorial logic.

X.25 is divided into three separate layers. It is important to maintain this separation throughout the design. As few concessions as possible must be made to this predifined decomposition, which of course is only a global one.

A functional specification of the chip is also to be desired. No integration technology will be chosen and no layout will be generated in this phase of the project. One of the intentions is to make use of regular structures, of which the layout can be easily generated.

4. Design Methods.

The object of this project is to design this chip using a structured way of thinking, as found in Pascal. This should lead to a way of 'structured programming with hardware', where procedures are not realized in software, but in hardware.

Another thing that has to be avoided, is the use of 'wild logic'. This usually means that the design is not structured, and more important: it is very difficult to generate the layout of 'wild logic'. We want to use regular structures instead, such as PLA's. (6), (7), (8).

The complete X.25 problem will have to be decomposed into a number of subproblems, and that has to be repeated until very simple tasks appear, which can easily be realized by means of finite state machines, or combinatorial logic. A number of low-level machines will be coordinated by a higher level machine, which in turn will be coordinated by again a higher level machine.
This should lead to a hierarchy of synchronised sequencers, which is analogous to the Pascal division into procedures.
See fig. 1.

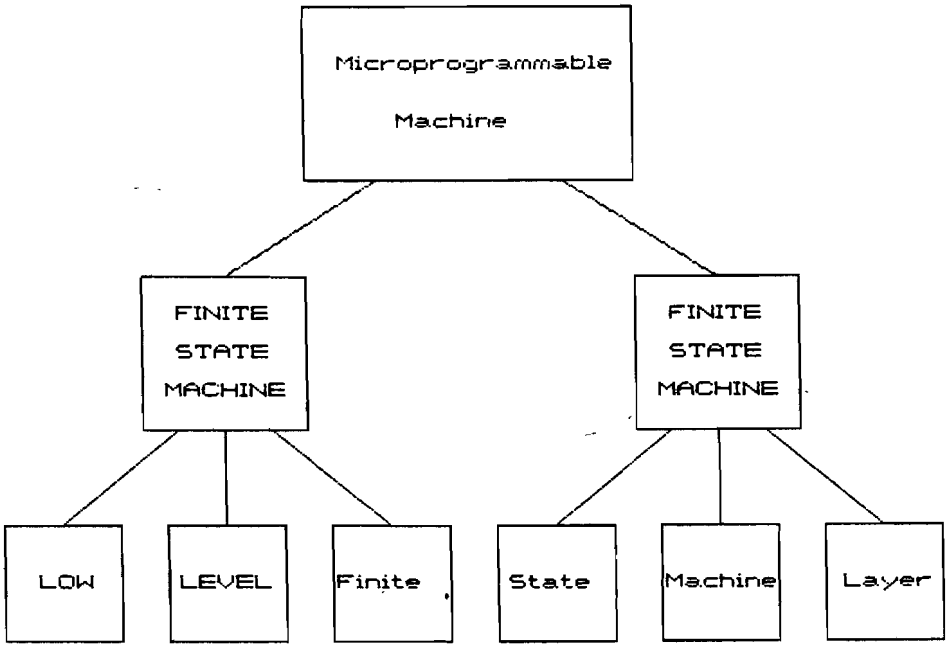


Fig 1. Hierarchic Machine structure.

5. What is X.25 ?

The X.25 protocol is used across worldwide packet-switched networks, to exchange information between computers and or terminals. Most X.25 networks are public, and the number of subscribers grows. This means that a cheap X.25 interface is wanted very much.

X.25 is based on a 3 layer architecture, corresponding to the bottom 3 layers of the OSI model. The first layer is the Physical Level, in which times, voltages and currents are defined for the transmitted data bits. This layer is based on X.21. See fig.2

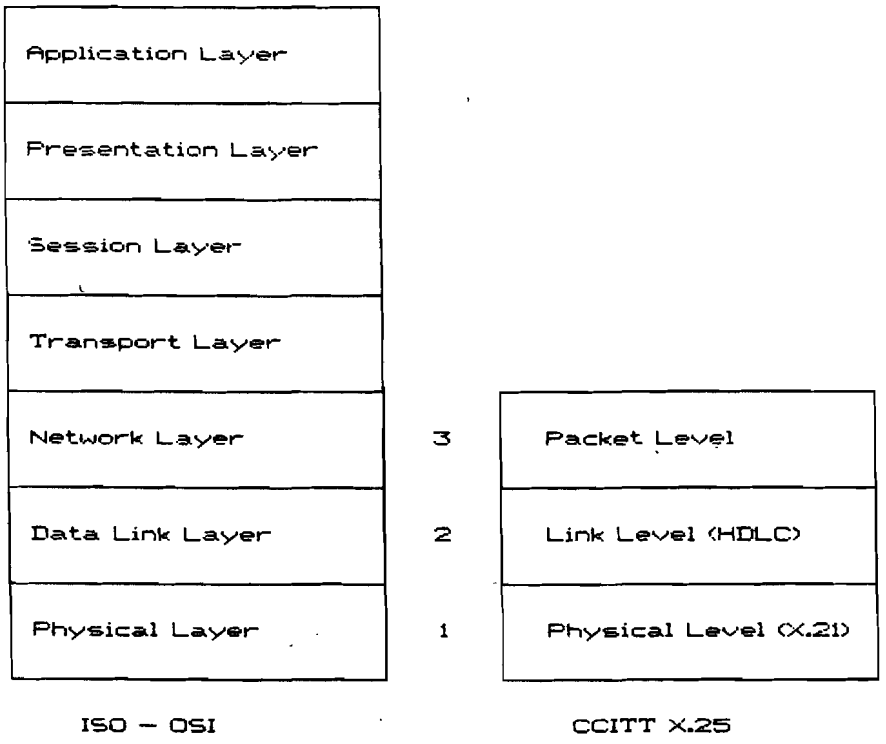


Fig 2. Comparison Layered Architectures OSI <--> X.25

The second layer, the Link Level, deals with frames. This layer provides for an errorfree link to the network. A checksum (FCS) on every frame will detect transmission errors, so that damaged frames may be retransmitted. This layer is based on the well known HDLC protocol.

The highest X.25 layer, the Packet Level, deals with packets that are transmitted through the network. Packets are sent in the information field of level 2 frames.

5.1 What happens to the user data ?

User data to be transmitted, will be sliced into a number of parts with the maximum byte length, and each slice will be put in the data field of a packet. (Packet Assembly) This packet will be passed on to the Link Level, and the frame will be passed on to the Physical Level. At the receiving end the process is reversed. (Packet Disassembly)

For more details on X.25 see (10), (11) or the more formal (1).

Not every packet or every frame contains user data. Those frames are used for maintaining the link, and for retransmission and flow control. The packets are used for call setup and clearing, and also for flow control.
Only Information Frames contain packets, and only data packets contain user data, if we do not count the user data that may be supplied in call request packets.

This can be visualised in figure 3.

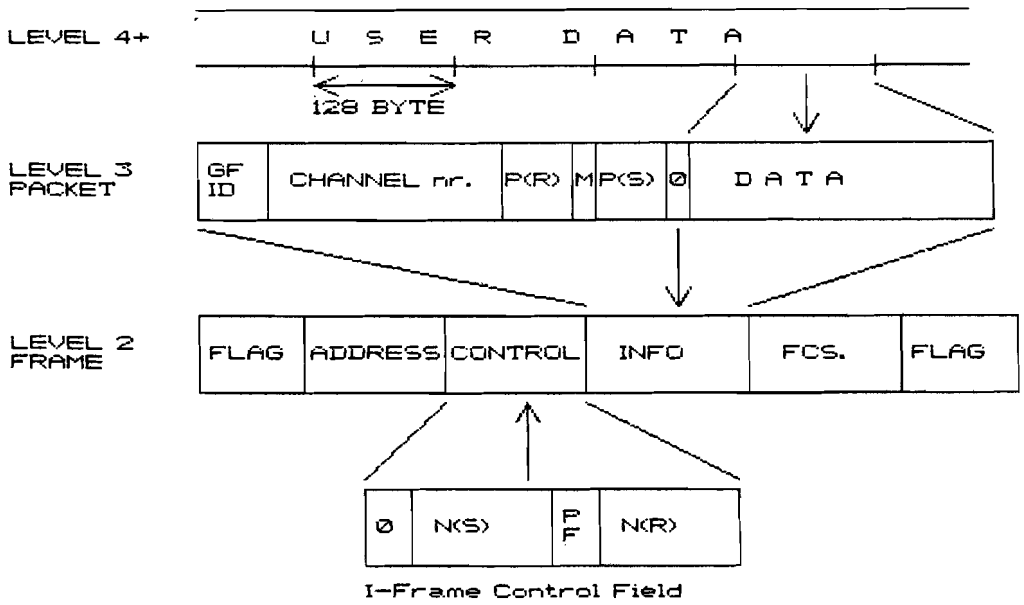


Fig 3. X.25 Packets and Frames

5.2 X.25 Virtual Channel concept.

X.25 level 2 deals with one physical link to the network. Level 3 uses the same link to offer a number of virtual circuits. This means in practice that a DTE may have a number of calls to remote DTE's active at the same instant. A short number (logical channel number) is included in every data packet to be sent, to address the correct remote DTE.

A call set-up procedure is used to make the virtual connection between two DTE's.

In this phase the destination DTE number is used, and assignment of the logical channel number is done.

The call clear phase does the reverse. The logical channel used is freed, and the channel number will no longer be used, until another call uses this number.

Figure 4. shows a DTE and a network with several calls active at a certain instant. A special kind of logical channel is the PVC, Permanent Virtual Circuit. This channel needs no call set up or clearing, but is a permanent link to a remote DTE. Such a type of channel has to be agreed upon by the administration at time of subscription.

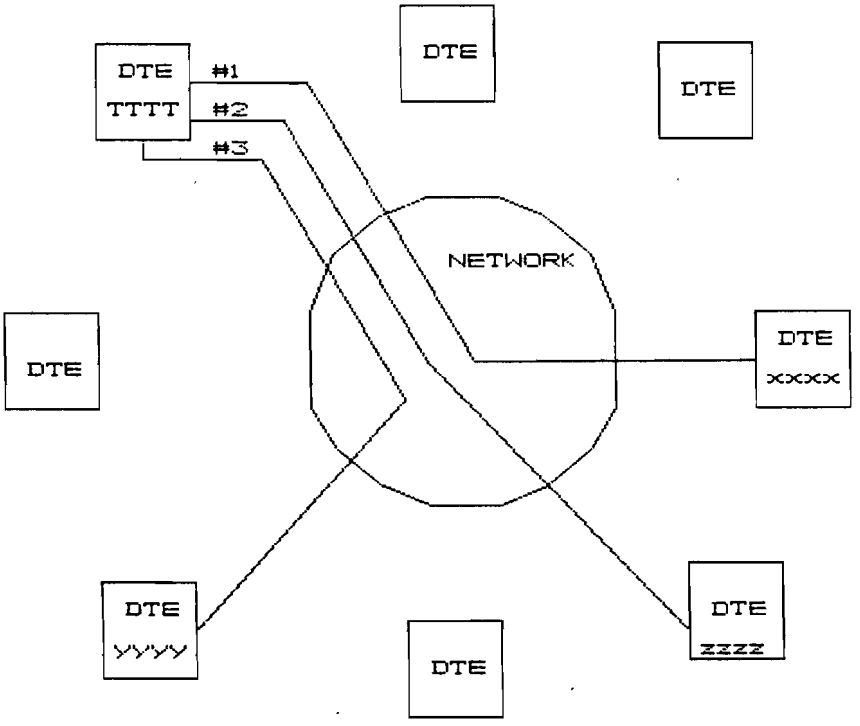


Fig 4. Network with a number of DTEs

6. Where will this chip be used ?

This chip can be used as a peripheral chip in any microprocessor system. The chip will function as a coprocessor dedicated to X.25. The chip will be connected to the system bus, and access the system memory using DMA. This shared memory between the chip and its Host microprocessor, is also used for communication between the chip and Host. The microprocessor-system employing this chip, will be a DTE, Data Terminating Equipment. A DTE is connected to the network via a DCE, Data Circuit terminating Equipment. Due to minor differences between DTE and DCE with a major impact on the architecture, this chip can only be used in a DTE. Figure 5 shows the architecture of a microprocessor with the X.25 chip.

The chip is designed for use in a simple DTE as well as in a sophisticated DTE. This means that the chip can be programmed to use only little memory and just one X.25 channel, or to use a full 64kByte memory with up to 250 virtual channels. The maximum number used in practice is around 40 channels because of throughput reasons (which is a limitation of X.25 rather than of the chip).

This chip should be useable with a large number of microprocessors of different make. Old 8 bit and new 16 bit microprocessors must be interfaceable to this chip, without need for additional circuits.

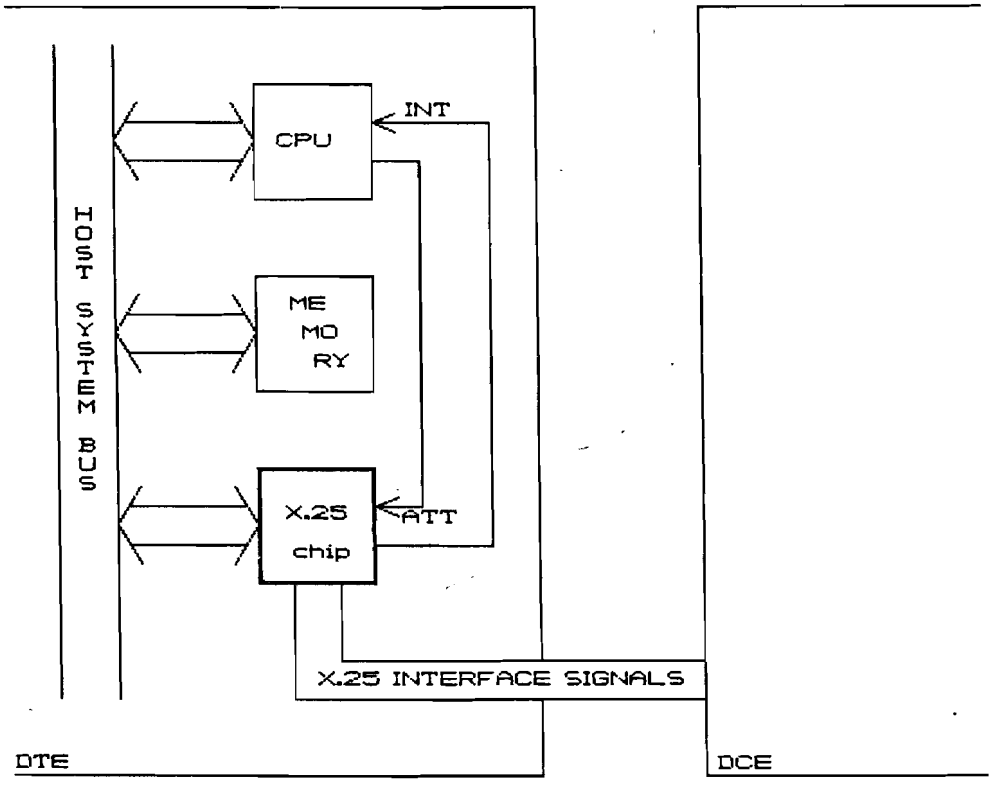


Fig. 5: MICROPROCESSOR SYSTEM ARCHITECTURE

7. What X.25 options and features will be implemented ?

The object is that the chip will relieve the host microprocessor of the bulk of work concerned with X.25. This means that level 1, 2 and most of level 3 functions will be implemented on chip.

Which level 3 functions should not be implemented? Those are the functions which will not consume much CPU time, because they occur only with a low frequency. Such functions are:

- decide whether to accept a call or not
- interpret facilities in a call request
- interpret level 2 or level 3 diagnostics

The facilities will have to be investigated by the host, and if necessary, parameters will have to be adjusted by the host. Also, not all of the requested facilities have to be accepted, such as the reversed charge facility.

X.25 knows a number of E and A services:

E - Essential
- virtual call service
- permanent virtual circuits

A - Additional
- datagram services.

Only the E-services will be implemented by the chip, these services are used in practice, whereas the A-service, datagrams, are not implemented in any network in the world, and this service will be omitted from future X.25 recommendations.

Also some more options are available per layer:

7.1 Physical level.

This level knows two variants:

- X.21
- X.21 bis.

The latter standard is adopted as a temporary standard to bridge the age between interfaces using old analog transmission links with modems, and true digital links. As X.21 bis is only temporary, this standard is not chosen. The physical level signals will be according to X.21 for the use of a dedicated circuit.(1)

The B signal, Byte timing will not be used by the chip, because level 3 is based

upon a bit oriented protocol that uses flags for synchronisation.

7.2 Link Level.

This level uses a protocol which is a subset of the HDLC protocol. Again two possibilities arise:

- LAP (Link Access Protocol)
- LAPB (Balanced Link Access Protocol)

In our implementation, LAPB will be used, since all DTE's must employ LAPB, but need only implement one of the two.(1)

7.3 Packet Level.

This level offers a large number of facilities, which may or may not be used by the DTE's. A limited number of facilities will not be implemented, which are:

- extended packet sequence numbering
- all options related to datagrams

The modulo 128 packet sequence numbering will not be implemented because just one network in the world employs it, where it is used to increase throughput with end to end acknowledgement on a packet basis. Implementing this feature would complicate the design of level 3 very much, which is not worth the effort.

8. Hardware requirements for the chip

As stated before, this chip must be useable with a number of Host microprocessors. This means that the physical interface must be very flexible. A number of those requirements are listed below, ensuring that most of the common microprocessors can be used with the chip, including modern 16 bit processors.

- 8 or 16 bit wide data bus.
- 24 bit wide address bus
- On chip 3 channel DMA controller.
- Motorola or Intel bus signals.
- Attention / Interrupt mechanism.

With a 24 bit wide address bus, this chip can address 16 Mbyte of memory. No I/O space will be used by this chip. The chip will use at most 64kByte of memory in maximum configuration, which segment may start at any multiple of 256. An on chip DMA controller is required, because memory will not be addressed contiguously, as found in simpler DMA applications. Three channels are needed for the transmitter, receiver and the updating of housekeeping information, which is largely held in memory.

All communication between the chip and its host will go via shared memory and an interrupt/attention mechanism. This means that the host need not access registers directly inside the chip, but can put commands in certain fields in the shared memory, whereafter an attention must be generated at the chip.

Reversely, status reports of the chip will be written into shared memory, and interrupts will be generated at the host. This scheme has the advantage that the hosts CPU overhead is minimised, and that communication between chip and host is on a functional basis, and not on a basis of byte transfer.

9. Functional interface between chip and host.

The chip offers a number of operation primitives, which are listed below:

- open communication
- close communication
- send data packet
- send interrupt data packet
- append empty receive buffers
- reset logical channel
- restart X.25 interface

As X.25 knows a virtual channel concept, the open and close primitives are used to build up and break down a communication link respectively. This is equivalent to picking up the receiver and dialling a number, and putting down the receiver after exchanging information. The open command needs a CCB, communication control block as a parameter. See figure 6. This CCB contains the number of the destination DTE, and may also contain facilities and some user data.

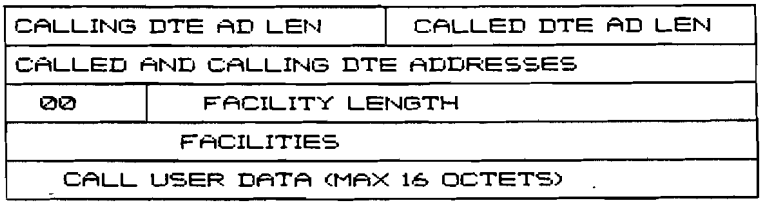


Fig 6. Communication Control Block (CCB)

The send command has the data itself as a parameter, where the channel to be used is implicit, because the command must be placed at the channel descriptor in concern. The host only needs to put a pointer to the data to be transmitted, and the chip will fetch data autonomously at the time of transmission, without intervenence of the Host.

Critical readers may have noticed that the command receive data is not present. In X.25 this could not be a command, because the initiative lies at the remote DTE. Such a command may perhaps be given at a higher level protocol (transport layer in OSI model), but not within X.25. To be able to receive data, receive buffers must be available, and the chip will receive data into such buffers autonomously. When no buffers are free, X.25 flow control ensures that no data is lost, but no further data can be received until new buffers are available to the chip.

The chip may report a number of statuses to the host, which are listed below:

- data packet received
- interrupt data packet received
- incoming call request
- call clear indication
- run out of receive buffers
- level 2 diagnostics
- level 3 diagnostics
- restart of X.25 interface
- reset of a certain channel

These reports are all a result from actions of the remote DTE or actions undertaken by the network, as a result of certain conditions.

10. Two global architectures.

Two global architectures have been designed, which can be seen in figures 7 and 8. The layered architecture of the X.25 protocol can clearly be recognized in both figures. The main difference between the two figures is, that in model 1 data is locally buffered on the chip, which is not the case with the second model. Functionally both architectures are the same. The reason the architecture of model 1 cannot be used is because this model requires a great deal of RAM memory on chip, which consumes very much chip area, and is therefore too expensive.

The second model processes the incoming bit stream on the fly, and passes the information on to the shared memory. Transmitted packets and frames are assembled at the time they are sent, thereby also eliminating the need for on-chip storage space.

Why the introduction of the first model ?

The first model shows the basic architecture in a clean cut fashion, without blurs from housekeeping signals. This model will be used for a first description of the functions of the chip.

Each level in the first model has three main processes, the receive process, send process and handler. Level 1 deals with bits, level 2 with frames and level 3 with packets. The physical level on chip only forms some interface signals as they are defined by X.25:

T - transmit
R - receive
C - control
I - indication
S - signal element timing
B - Byte timing (not used)
G - signal ground

The T and C circuits are outputs, where the C circuit indicates that data (level 2 data including flags and checksum) is being transmitted when c=1.

The R and I circuits are inputs from the DCE, where I has the same function as C.

S, Signal element timing, is the bit-clock delivered by the DCE. All bit transitions must be synchronous to this clock signal.

B, Byte timing, is also delivered by the DCE, but this signal will not be used. This signal is only used when the full X.21 protocol is used, which is not the case with X.25

Level 1 takes care of the bit-clock signal distribution inside the chip, and inspects the state of the level 1 interface. This level will usually be in the DCE ready and DTE ready phase. This means that the signals c and i will be in the logical high state.

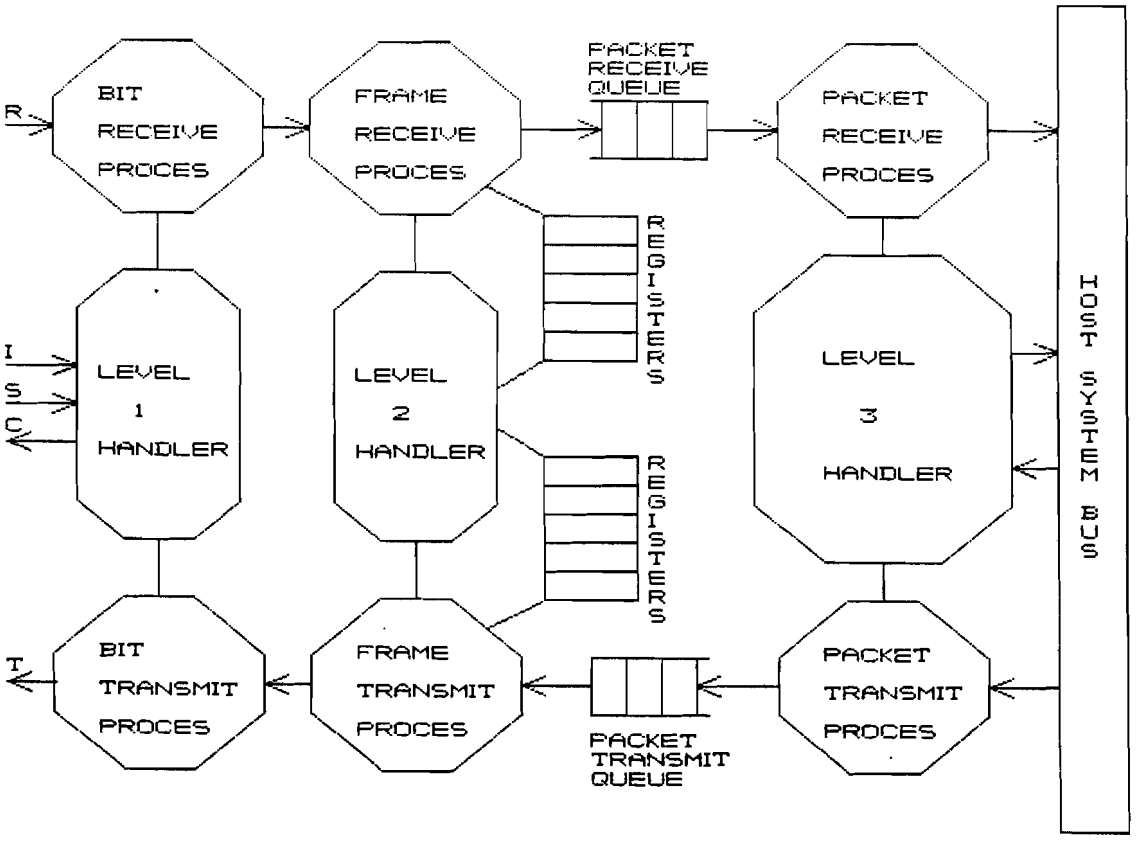


Fig.7: Global Chip Architecture ,Model 1

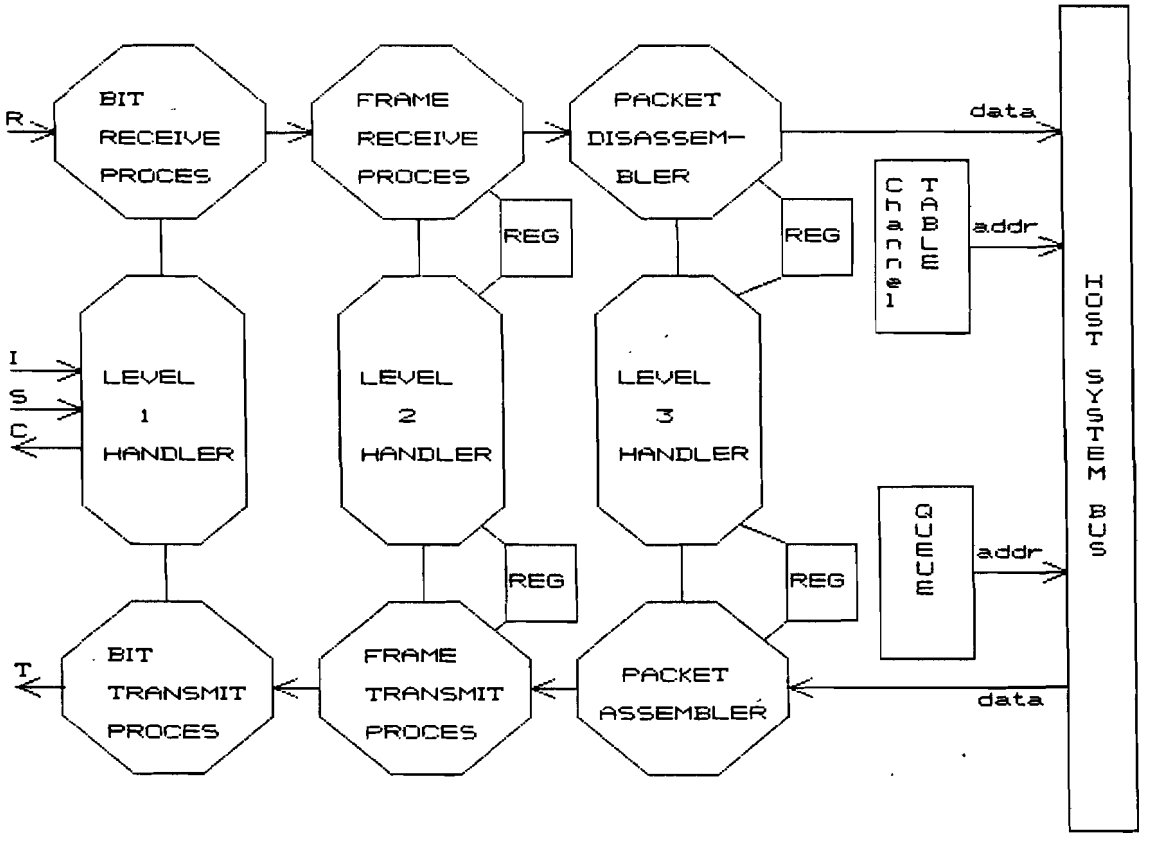


Fig.8: Global Chip Architecture, model 2

Level 2 is much more complicated. The receive process has to detect the start of frames, update the checksum, and save the information field in a queue between level 2 and 3. This information field contains a level 3 packet, including the header. The level 2 transmit process must take a packet from the transmit queue, and assemble the frame, after which it can be transmitted. The level 2 transmit and receive processes are coordinated by the third level 2 process: level 2 handler. This process inspects the state of level 2, takes care of error handling and frame retransmissions. Finally it also communicates with level 3 handler.

Level 3 is again more complicated than level 2, but it resembles level 2 very much. Instead of frame (dis)assembly, level 3 does packet (dis)assembly. The handler however is much more complex, since it must inspect the state of every active logical channel. The maximum number of channels active in X.25 is 4095, but the maximum used in practice is about 40. Furthermore, the level 3 handler has to communicate with both level 2 and the host. A memory access scheme must also be implemented in this process.

Level 3 receiver takes a packet from the receive queue, and disassembles it. The data field of this packet will be offered to the host. The transmitting process is the reverse.

What impact does model 2 have on these strategies ?

Since there is no queue between level 2 receiver and the level 3 receiver, those two processes must be synchronised on a bit level. This means that level 3 process must always be ready to accept a new packet shortly after receiving the last one. The minimum time between two packets is the frame overhead, which is:

- one flag
- control field
- address field
- 16 bit FCS

Only one flag is counted, because a frame closing flag may be used as an opening flag for the next frame.

This means that 5 bytes or 40 bit times is the rest period between packets. Assuming the maximum X.25 datarate of 48 kBits/sec, this gives approximately 1 millisecond time spacing per packet.

A problem is the queue between level 3 transmitter and level 2 transmitter. This queue is absolutely necessary, because level 2 must be able to retransmit. The name 'queue' used in this report is not the normal definition of a FIFO, since in our model elements can be read from the queue without removing them. This happens at transmission time. Elements can also be removed from the queue without reading them, which happens when an acknowledgement of reception occurs.

For instance the first element in the queue is read, the second is read and the third element is read. The first element is acknowledged, and therefore removed from the queue. The second element, which has now become the first one, has a

negative acknowledgement, and will be read again from the queue.

Still we do not want the queue to be stored on the chip, so it must be done elsewhere. The queue will be stored in system memory, and will be managed by the chip only. The Host need not bother about the queue, but may inspect its contents for diagnostic purposes.

11. Memory Management.

The memory shared between the chip and the host must be used efficiently. Level 3 is the closest one to the host, a reason for giving level 3 the task of managing the memory.

Another important thing, is to keep the copying of user data down to a minimum. This saves time, and minimises the number of accesses done on the shared memory.

A problem is that when a packet is being received, it is not yet known to which place in memory it should be written. This is so, because there is not time enough between reception of the header bytes and the user data bytes during which a search from tables could be performed.

The type of packets needing storage space in shared memory are the following:

- Incoming call
- call connected
- DCE data
- DCE interrupt
- reset
- restart
- diagnostics (level 3)

The packets that need no storage space in shared memory are:

- DCE RR
- DCE RNR
- DCE reset confirmation
- DCE restart confirmation
- DCE interrupt confirmation
- DCE clear confirmation

These last 6 packets will be dealt with completely in the chip, since they are only short flow control or confirmation packets.

In a special case also storage space is needed in shared memory, which is level 2 diagnostics. This is no packet information, and level 3 should not interpret this diagnostics.

As it not known in advance what kind of packet or in case of a data packet what kind of data will be received, and the fact that we do not wish to copy, clearly some pointer mechanism must be used. The first structure needed in shared memory is the Free Buffer Table (FBT).

11.1 Free Buffer Table.

This table will be held in shared memory, and accessed by both chip and host. This table contains pointers to empty buffers that the host has assigned to the chip for

receiving purposes. The chip has a number of internal registers that keep track of the state of this table. The Host must also have such an administration.

Important to be known to the chip is how many empty buffers are available, and which pointer is to be used next. The host must keep a complementary administration, which keeps track of the number of used buffers, and the first pointer to a used buffer, which the host has not yet processed.

The receiving strategy is that the chip should always have a copy of the pointer to be used next in a DMA receive register (also in the chip), which means that any data to be received will be put in an empty buffer. As soon as the reception is completed successfully, the pointer to the buffer just filled will be put in an appropriate table (to be discussed later), and the next pointer must be copied to the DMA work register. Furthermore, the number of empty buffers must be decremented by 1. When no buffers would be available, level 2 flow control will have taken care of the situation, so that no more data will have to written into memory buffers. Figure 9 shows the structure of the FBT.

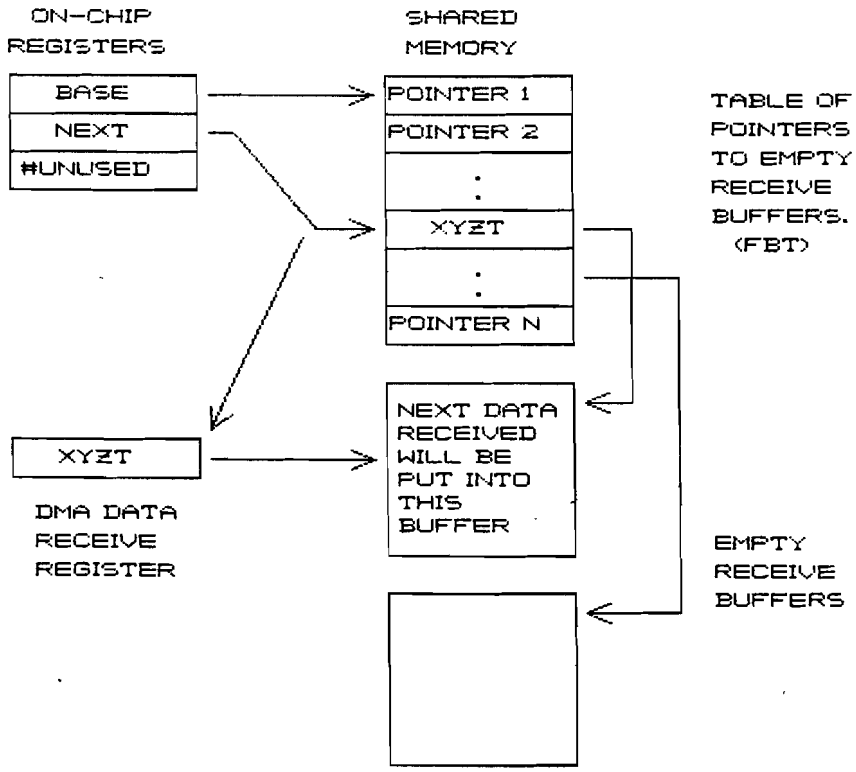


Fig 9. Empty buffer management

The table itself works in a wraparound fashion, as soon as the end of the table is reached, the next entry to be used is the one at the start.

Why use a double administration system, with a next and length pointer, instead of the more usual shared two pointer system, as found in implementation of queues? The reason for this is twofold: The first reason is one of speed. To be able to address the table fast enough, the descriptive registers must be placed inside the chip, which means that the host cannot access these registers, and must maintain its own administration. The second reason is that all of the processing done in the chip must be as simple as possible, where the zero testing of a counter is more easily implemented in hardware than the comparison of two pointers.

11.2 Logical channel table.

The second structure needed in shared memory, is a table of channel descriptors, of every logical channel that can be used (The number of channels is defined at time of subscription). This table is also accessed by the chip via a number of on-chip registers. The complete structure can be viewed in figure 10.

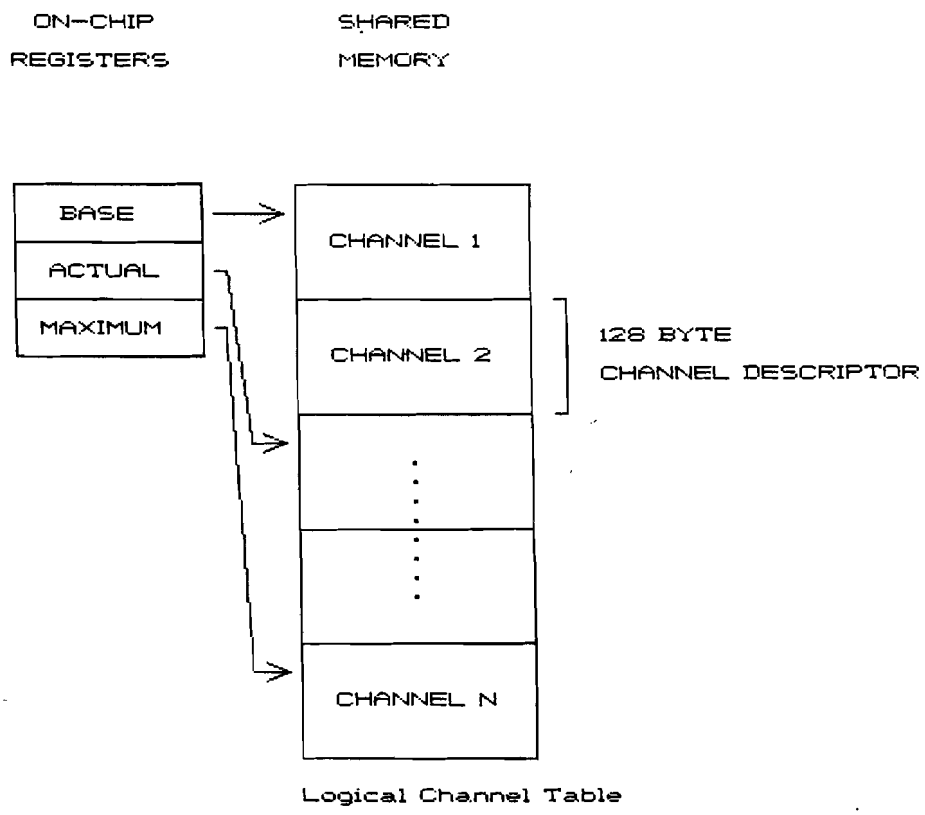


Fig 10. Logical channel table in shared memory

Each channel descriptor contains the following elements:

```
nr.bits
      3 - type of logical channel:
            - permanent virtual circuit
            - virtual call channel
                  - incoming
                  - outgoing
                  - twoway
      3 - receive sequence number
      3 - send sequence number
      1 - D bit
      1 - S bit
      4 - state of channel (12 possible states)

   2*16 - special purpose send pointer
   2*16 - special purpose receive pointer

 2*8*16 - 8 pointers to data-packet receive buffers
     16 - next pointer to be used
      3 - number of unused pointers
      2 - binary semaphores

 2*8*16 - 8 pointers to data-packet send buffers
     16 - next pointer to be used
      3 - number of unused pointers
      2 - binary semaphores

      8 - specific command field
      2 - specific response field

      8 - Timer
      3 - window size
      8 - pointer to next channel
      1 - DCE busy flag
      1 - in-chain flag
----------
minimum of 90 byte per channel
```

A channel descriptor contains the type of the channel, the state it is currently in, packet sequence counters and a timer, which will be updated by the chip. Most of those fields will be accessed by the chip only. Furthermore it contains two subtables of pointers to transmit- and receive data buffers. Those tables also work in a wraparound fashion.

To simplify addressing hardware, the number of bytes reserved per descriptor is 128, which is a round binary number. The length of the table, and therefore the number of channels is programmable. When a data packet is received, a pointer to the receive buffer is placed at the receive buffer pointer field. There is room for 8

receive data packets in this sub-table. When 1 or less pointer positions are available, at the current channel, level 3 flow control will stop further data packets on that channel from coming in, so that no data will be lost. New data can be received on the same channel when the host has copied the pointers or inspected the data itself, and has freed 1 or more receive pointer positions.

To transmit a packet, the host puts data in a buffer, and puts a pointer with length in the transmit table. Next, a transmit command should be given in case the channel is not active.

The special purpose transmit and receive pointer are not used for data packets, but for packets like Incoming call and reset indication. These events must be acknowledged by the host CPU as fast as possible, so that the acknowledgement can be made to the network or remote DTE. In such an event a high priority interrupt will be generated at the host. This in contrary to the reception of a data packet, in which case a low priority interrupt will be generated.

11.2.1 Type field.

This field is a code for the type of the channel, which can be:

1xx	- permanent virtual circuit
010	- incoming virtual call
001	- outgoing virtual call
011	- twoway virtual call

The used bit assignment simplifies testing.

11.2.2 Receive and send sequence number.

These number must be kept per channel, and are the P(R) and P(S) as specified by X.25 respectively. Since only modulo 8 sequence numbering will be used, 3 bits need to be reserved per number.

11.2.3 D and S bits.

These bits apply to the channel and must therefore also be placed in the channel descriptor. The D-bit, delivery confirmation bit, is used to force an end-to-end confirmation. This means that when a packet has been transmitted with the D-bit set to 1, the window cannot be updated until the acknowledgement has been received from the remote DTE (no local windowing). This puts a constraint on the throughput, and will therefore not be used much in practice.

When the D bit is set to 0, the S bit may be used. The Sequence bit indicates that the packet is part of a sequence when it is set to 1. This means that the network may block or unblock packets when S=1, which occurs when maximum packet sizes differ on both DTE's.

The S bit is used in conjunction with the F bit, which generates the M bit which is transmitted in a data packet.

11.2.4 Q and F bit.

The Q (Qualifier) and F (Final Packet) bit have to be specified per packet, and are therefore incorporated in the pointers to the data buffer of the packet. Such a pointer consists of a base pointer, containing the address of the first byte of the buffer, and a length counter. The base pointer is a true 16 bit value, but the length counter need only be 14 bits wide. A pointer combination is stored as follows:

```
bbbbbbbbbbbbbbbb     16 bit base pointer
QFllllllllllllll     14 bit length counter
```

The Q and F bits are stored at the two most significant positions of the length counter.

The Qualifier bit enables data transmission at two levels. This means that two packet sequences may be transmitted to the same destination DTE at the same time on the same virtual channel. This saves on the number of virtual channels to the same DTE, but puts a constraint on the throughput of a single sequence. In case data is only transmitted on a single level, the Q bit should be set to 0.

The F bit is only used when the D bit is set to 0, and when the S bit is set to 1. A complete packet sequence must be transmitted with the M (More data) bit set to 0. This bit is transmitted with the data packets. The F bit set to 1 indicates the final packet in a sequence, and causes the M bit to be set to zero.

A table clarifies the use of the bits

S	F	M
0	x	0
1	0	1
1	1	0

where M is a function of the S and F bits.

11.2.5 state field.

This field encodes the current state of the channel. The state must also be kept per channel, so that this field must be put in a channel descriptor. Figure 11. shows the possible states of a channel.

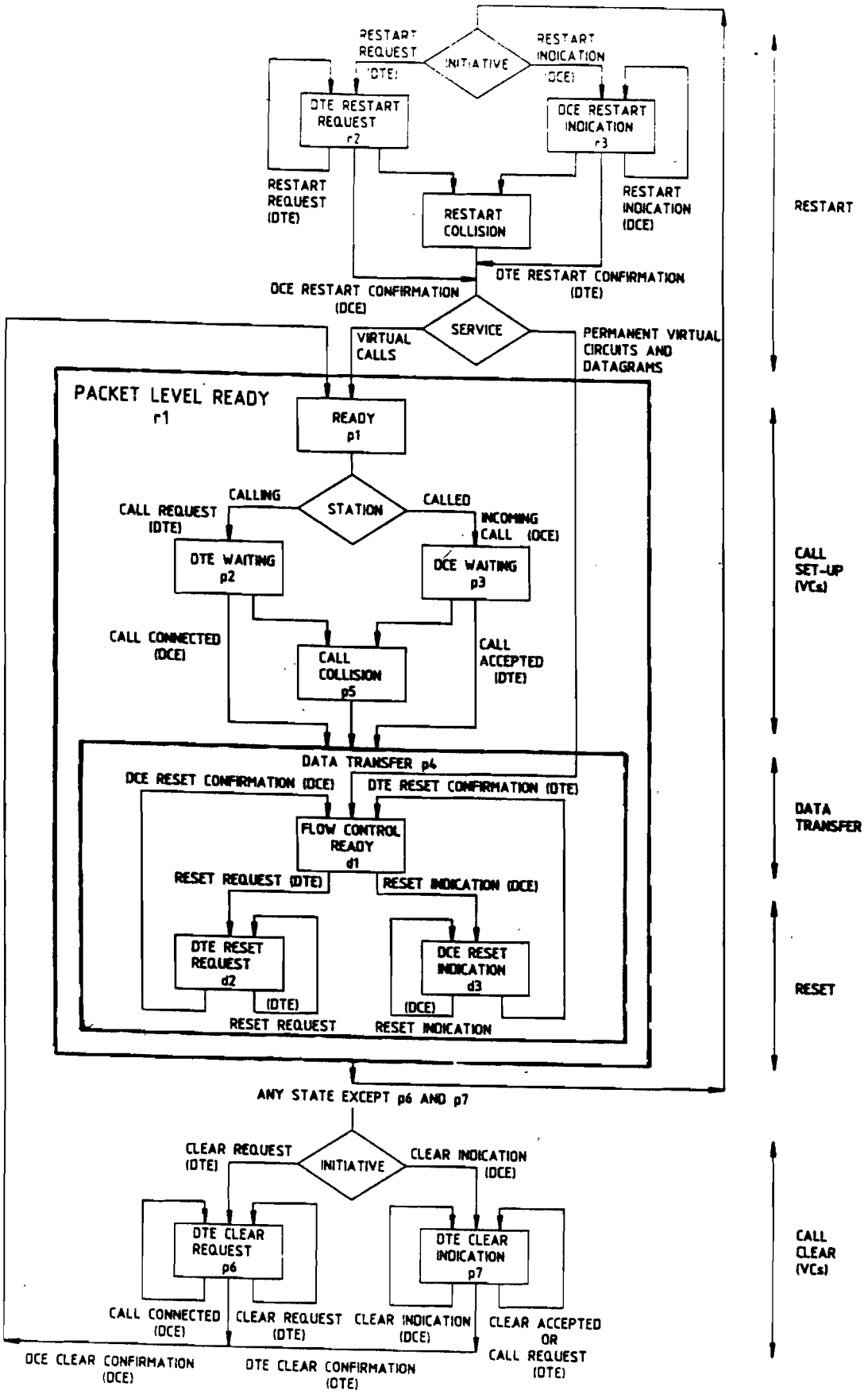


Fig 11. X.25 Level 3 states

11.2.6 Special purpose send pointer.

This pointer combination points at the buffer containing a non data packet. A command must be given in such a case. This will be discussed in detail further on.

```
bbbbbbbbbbbbbbb     16 bit base pointer
  lllllllllllll     14 bit length counter
```

11.2.7 Special purpose receive pointer.

This points at a received non-data packet, and will be accompanied by a response. This will also be discussed in detail further on.

```
bbbbbbbbbbbbbbb     16 bit base pointer
  lllllllllllll     14 bit length counter
```

11.2.8 Timer field.

According to X.25, a timer must be used in certain states of the channel. This timer is channel specific, so that it must also be incorporated in the channel descriptor. Once every 4 seconds, level 3 will update all timers in the channel descriptors. This leads to a 2.5 % inaccuracy. A zero value of this field means that the timer is not running. A non-zero value means that it is running, and when the resultant value after decrementing is zero, the timer expires, and action must be taken.

11.2.9 Window size.

This parameter is channel specific, and therefore incorporated in the channel descriptor. Usually its value remains constant, but with the window size negotiation facility, the host may alter its value.

11.2.10 Pointer to next channel field.

This pointer points at the next channel descriptor where data packets have to be sent. This will also be discussed further on.

11.2.11 DCE busy flag.

This flag indicates that the DCE has sent a RNR frame, which means that the chip cannot send data packets until a RR packet from the DCE is received. This condition is retained by the DCE busy flag.

11.2.12 In chain flag.

This flag indicates that this channel descriptor is part of the circular linked list. This structure is used for the data transmit strategy, which will be discussed further on.

11.3 The queue

The transmit queue is also held in shared memory. It consists of 8 packet descriptors, which contain packet headers and pointers to the data fields. Again some registers in the chip contain pointers to entries in this queue. This can be seen in figure 12.

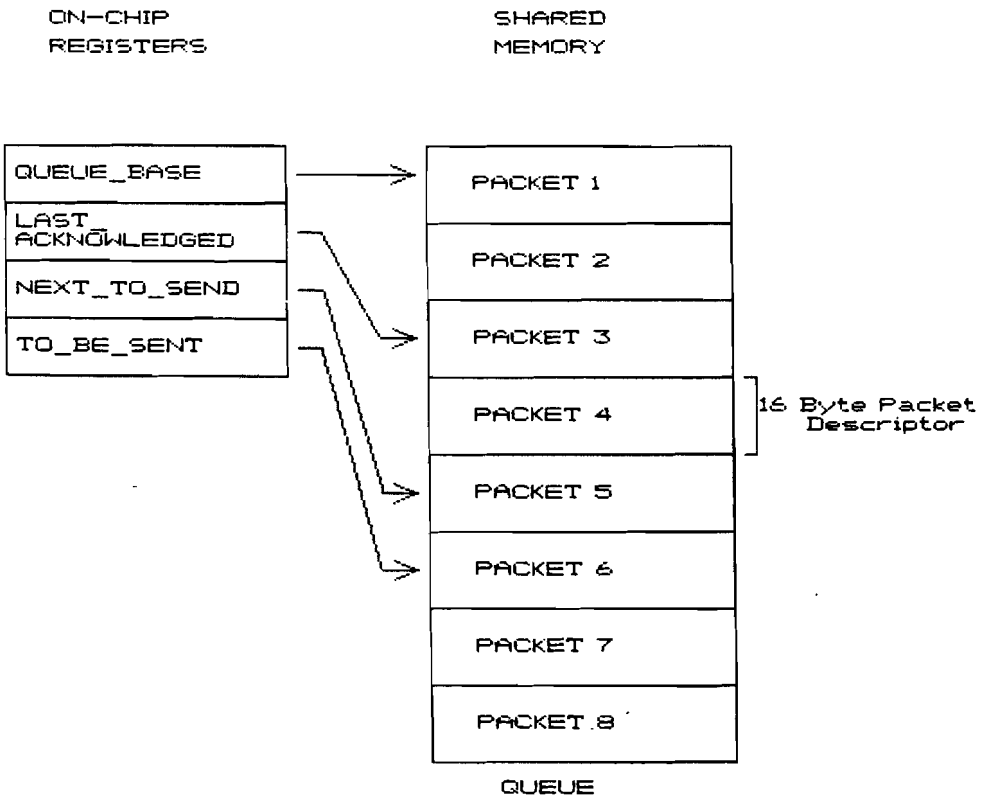


Fig 12. Queue management in shared memory

Efficient use is made of memory space, since each entry in the queue consumes 16 bytes only. Packet headers do not consume much space, and the data itself is accessed via pointers.

Since data is not copied, provisions must be made to free the buffer used to hold the data to be transmitted. As soon as an acknowledgement of a packet occurs, the packet descriptor can be removed from the queue. To free the buffer in use, a pointer is present in the packet descriptor which points at the originating logical

channel. The descriptor of this channel must be updated, so that the buffer is freed. This is only a minor drawback for not allowing copying of the packet data to an intermediate buffer.
The items per packet descriptor are listed below:

nr. of bits
- 2*16 - pointer to data field, if there is any.
- 16 - pointer to originating logical channel
- 24 - header of packet, which consists of:
  - general format identifier
  - logical channel number
  - packet type identifier
- 40 - diagnostics information (5 octets)

---

16 byte per channel as first neat value

11.4 The communication registers.

These registers are fields in a structure also held in shared memory. This last structure provides for the exchange of housekeeping information between the chip and the host. Its use is to transfer commands from the host to the chip, and statuses from the chip to host. Furthermore it contains pointers to data to be transmitted or already received.
Some fields within this structure are chip internal use. A drawing of the communication registers can be seen in figure 13.

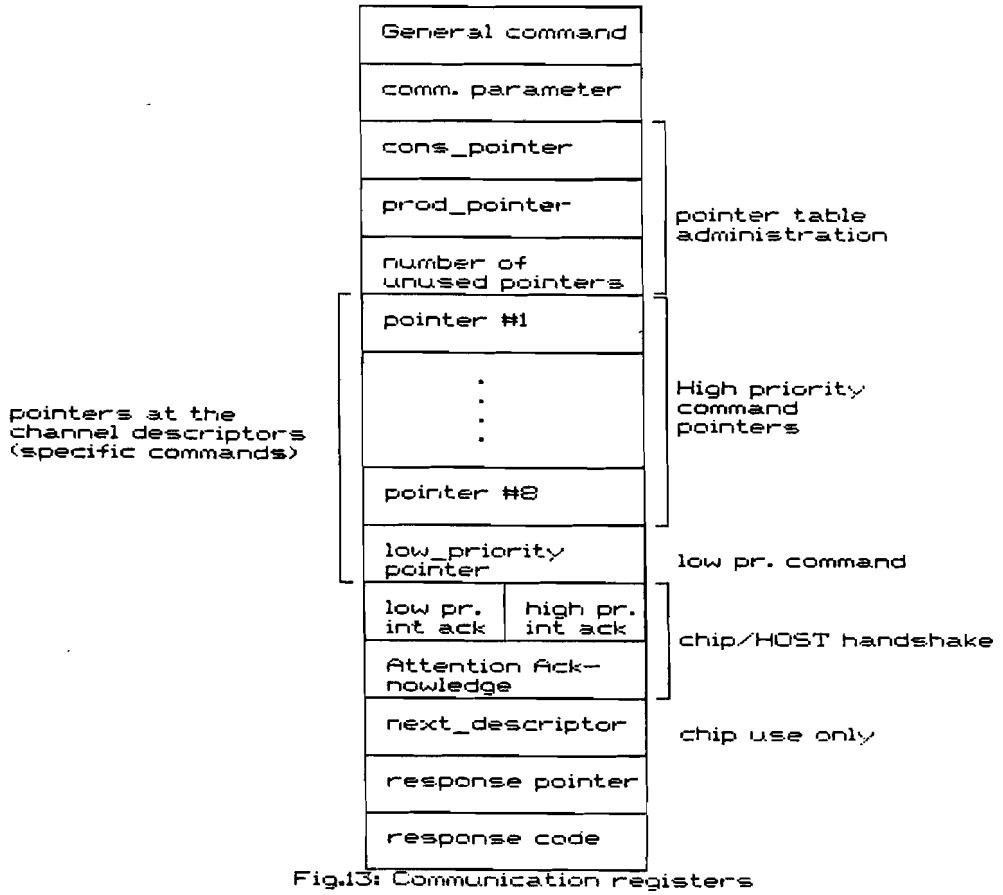


Fig.13: Communication registers

12. Use of the communication registers.

Commands and responses can have a general form, or a channel specific form. The general form will be discussed next.

See figure 13 for a look at the communication registers.

A number of command types are defined, which are listed in decreasing order of priority, meaning that the chip will execute a higher priority command, or list of commands, before a lower priority command will be executed.

- general command(s)
- specific command, high priority
- specific command, low priority

12.1 General commands.

These commands apply to the interface in general, meaning that they apply to every channel simultaneously. These commands are listed below:

- initialize chip (power on start-up)
- start up X.25 communication
- stop X.25 communication
- restart X.25 interface
- set diagnostics mode
- append receive buffers.

The first command will be described separately in a following paragraph. The second and third command are used to initialize the interface itself, to start up the communication. This is also a non-trivial matter, as each of the three levels has to move to the proper state, and a higher level must wait until the lower one is ready. To stop such a communication, the reverse action must take place, after it has been made sure that all virtual calls are cleared.

The restart X.25 interface command, is used as a reset of the level 3 communication, which resets all channels simultaneously. All Permanent Virtual Circuits are reset, and all virtual calls are cleared. Furthermore all the packets that were still in the network will be removed. This command will only be given in the very unlikely situation that the X.25 level 3 protocol crashes.

The last command, append receive buffers, is used by the host to inform the chip that a number of receive buffers have been appended to the list. These buffers can be used by any channel, as the data received belongs to any channel.

To give a general command to the chip, the general command field must be filled with the code for this command. A zero value in this field means that no command is present, and only when this field is zero, the host may enter such a general command in this field. This field will be cleared by the chip as soon as the command has been executed; the host may never clear this field (this field also has

a binary semaphore function, it is set by the host, and cleared by the chip only). Optionally, the chip can be programmed to generate an interrupt at the host as soon as a command is completed.

The last command in the list, append receive buffers, needs a parameter, that will have to be written into the parameter field. When the command has been written into those registers, an attention must be generated at the chip, so that the execution may start.

12.2 Channel specific commands.

Again a number of commands can be given, but they apply to each channel separately. These commands are listed below:

- send data packet
- send interrupt packet.
- remove received packet pointer
- reset channel

To virtual call channels, also the following commands apply:

- open communication (call request)
- close communication (call clear)
- accept incoming call

These commands will have to be put in the command field in the channel descriptor of the channel in concern. The send data packet command only has to be given when the channel is not active. A channel is active when one or more data packets have to be sent on the same channel. To be sure that the chip will transmit the additional packet, the send data packet command must be given when no or one data packet still has to be sent.

Specific commands can be given to the chip with either a high or a low priority. High priority commands are all but the data transmit command. High priority commands can be put in a list of such commands, which is a list of pointers to the channel descriptors containing the commands. For low priority purposes there is only one field available, which will be used to point at a channel containing a data transmit command.

This scheme is developed in this way for a number of reasons:

- general commands will not be given frequently, and are more important than specific commands.

- The division of specific commands into a high and low priority portion, is because the open and close commands occur less frequent than the data transmit commands, but are more important. A data transmit command cannot be given when a call is not set up.

- Only one field for data transmit commands is available, because of the linked list transmit structure used by the chip.

12.3 Data transmit strategy.

Taking into consideration that in general a large number of data packets have to be sent, which will be divided across a number of channels, and to minimise the hosts overhead, the following strategy will be employed by the chip.

The channel descriptors which need data transmission, form a circular linked list, which may grow and shrink with the number of channels needing data transmission. When all higher priority matters are dealt with, the chip will resume data transmission. The chip will remember the next channel descriptor where a data packet has to be sent, and will transmit this packet. When no packets remain to be sent on this channel, its descriptor will be removed from the chain, when one or more data packets remain to be sent, its descriptor will not be removed from the chain, so that these packets will be dealt with in future.

When the host has data to be transmitted on a certain channel, it must put a pointer to the databuffer in the correct field of the channel descriptor, and when two or more packets have to be transmitted, it need not do anything else. When only one or no data packets had to be transmitted on that channel, a pointer to that descriptor will have to be written in the low priority pointer field, whereafter an attention must be generated.

This scheme also has the advantage, that all channels have equal priority, and more important, no search algorithm is needed to find channels that need transmission. This saves on the bus load.

12.4 Other fields in the communication registers.

The interrupt and attention acknowledge registers form status registers. The Low- and High priority interrupt acknowledge registers contain status bits which will be set to corresponding events, and may be cleared by the host as an acknowledgement. The attention acknowledge field will be cleared, as soon as the action of the chip following the attention has completed.

A number of registers in the communication registers are chip use only (used for linked list management).

The table of high priority pointers also needs some administration, which is done in:

- consumer pointer
- producer pointer
- number of unused pointers

The chip does not use (not access) the producer pointer field, but only uses consumer pointer and the number of unused pointer fields. Whenever the hosts decrements the number of unused pointers field, this action must be performed as an indivisible operation, because of sharing problems. The chip increments this field also in an indivisible action.

12.5 Responses from the chip.

Other fields in the communication registers are used for responses from the chip to the host. The responses can also be divided into general and specific. General responses are:

- restart indication
- level 3 diagnostics
- level 2 diagnostics
- specific response
- no receive buffers left

Channel specific responses are:

- Incoming call
- call connected
- clear indication
- data packet received
- interrupt data packet received
- reset indication

Specific responses contain a pointer to the concerning channel descriptor.

Those responses will generate an interrupt at the host, where all responses accept data packets received, will generate a high priority interrupt.

Other response fields are the level 2 and 3 diagnostics pointers. These fields point at received diagnostics buffers, which may be interpreted by the host. Whenever diagnostics are received, the code for this response will be written to the general response field, and a pointer to the buffer containing the diagnostics data will be put in the diagnostics pointer field.

13. Evaluation of the memory management, comparison with 82586.

The memory management employed by the chip should be:

- efficient
- simple
- flexible
- easy to use

To look at efficiency, the buffer size must be examined. In this architecture the buffer size must be as large as the largest packet size. Maximum packet size depends on the maximum user data field length, which can be up to 1024 bytes big. The header of this packet will not be stored in the buffer in memory, so that this buffer must be at least as big as the maximum user data field size.

The buffer must also be able to hold the CCB (fig. 13). The CCB has a maximum size of 96 bytes. The default user data field length is 128 byte. Other packets the buffer must be able to hold are diagnostics, clear, restart and reset which are only small.

Most of the packets sent are the data packets and acknowledgement packets (flow control). The flow control packets do not need any buffer storage, so that in normal cases the buffers will be used completely. Only when a call request or call clear packet arrives, the buffers will not be completely used, but these events do not occur very frequent.

Level 3 does not check if a packet exceeds the maximum length, as this is already done by level 2. This level has a maximum frame length, which depends on the maximum packet length. Any frame that is too long will be discarded (1), and the frame reception is aborted as soon as the maximum length is exceeded.

This implies that level 3 will never be offered a packet which is too long, so that level 3 need not check the maximum length.

A completely different memory management scheme is used by the INTEL 82586 LAN controller (13). This is a chip that implements the Ethernet protocol used on certain local area networks. The complexity of this chip is comparable with the complexity of the X.25 chip. The memory management employed by the 82586 is more complex than ours, which is inherent with the Ethernet protocol.

This scheme can be seen in figure 14. This scheme allows a frame to be received into a number of buffers. A frame descriptor points at a linked list of buffer descriptors, and those buffers together hold a single frame. This scheme is used because of efficiency reasons: A data frame may hold some 1500 bytes, but all acknowledgement frames must also be held in memory buffers. About half of all frames are acknowledgement frames which need about 6 bytes of storage. If all frames had to be received into single buffers, this would lead to a very inefficient memory usage. With the X.25 chip there is no such a problem, because the acknowledgement packets do not need memory storage space at all.

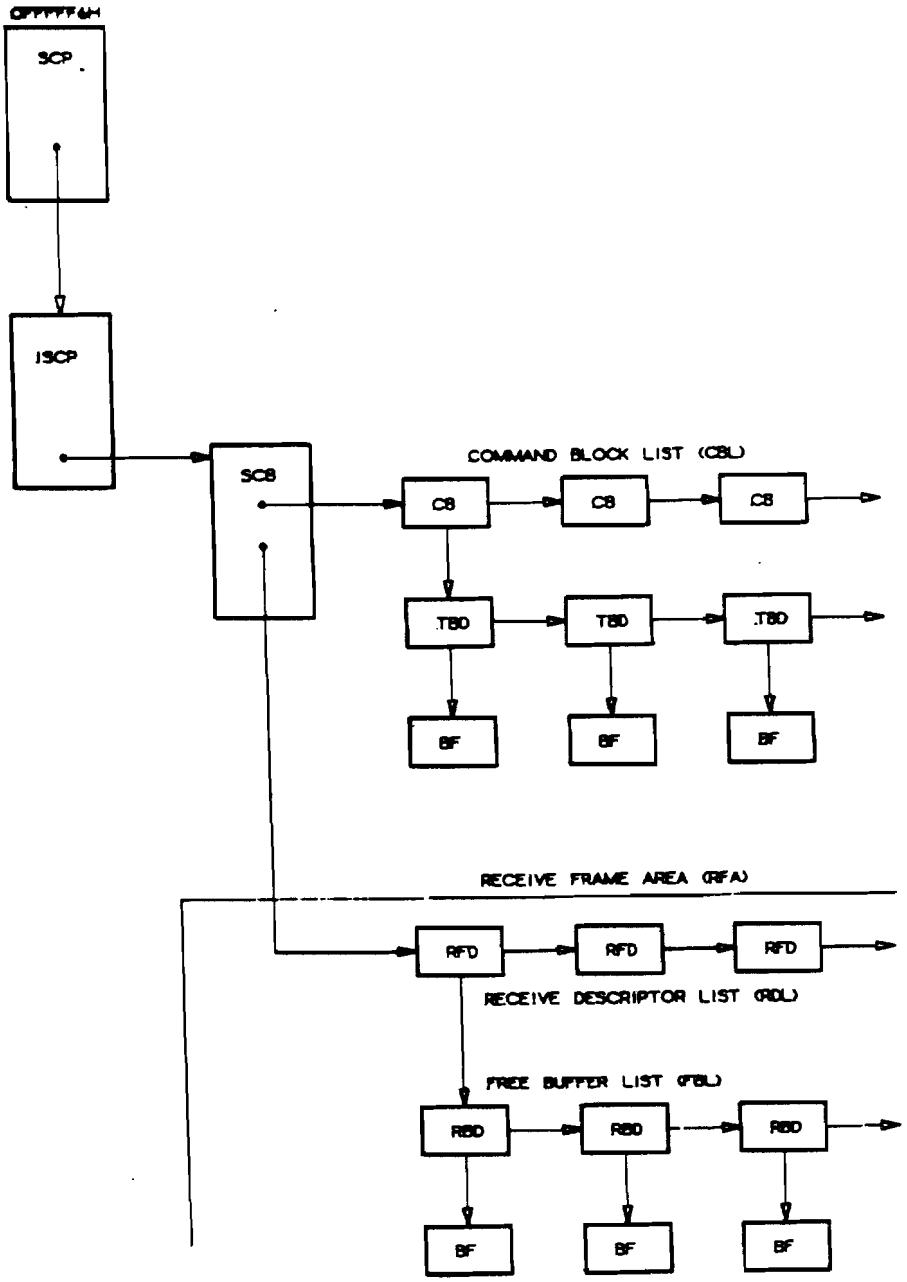


Fig 14. Intel 82586 LAN controller Memory Management

The command structure used by the 82586, is also based upon a linked list. However, when a chain of commands have been presented to the 82586, it is dangerous to append a new command at the end of this linked list. The 82586 might not see this new command when the chain has been completely processed before appending the new command. On the other hand, a long list of executed commands is hanging around in memory, which cannot be deleted from the list. The list must be removed all in one. This means that the 82586 host must perform garbage collection of all those descriptors hanging around in memory. That again results in a not very smooth stream of commands given at the 82586, which is not a benefit for the throughput.

The command structure used by the X.25 chip however, needs no garbage collection, because of the wraparound tables used.

Concluding it can be stated that the X.25 chip memory management is efficient and simple compared with the memory management as performed by the 82586. The X.25 memory management is flexible enough, as the buffer size can be varied per application, as well as the number of channels used.

14. DMA-controller.

The chip incorporates its own DMA controller, because of the following reasons:

- to minimise hosts overhead
- because of X.25 structures
- to communicate via shared memory
- number of DMA-channels needed

As stated before, an object in this project was to minimise the actions needed by the Host for X.25 protocol handling. If data bytes are written into memory without Host intervenence, this would save a considerable amount of CPU time.

The X.25 structures imply that data written to memory need not be contiguous. Data belonging to different channels may be received interleaved, and with the structures held in memory, also housekeeping data has to be accessed. For this reason a 3 channel DMA controller is incorporated in the chip with channels for:

- data writing
- data reading
- housekeep information updating

This implies that these three processes may happen independently.

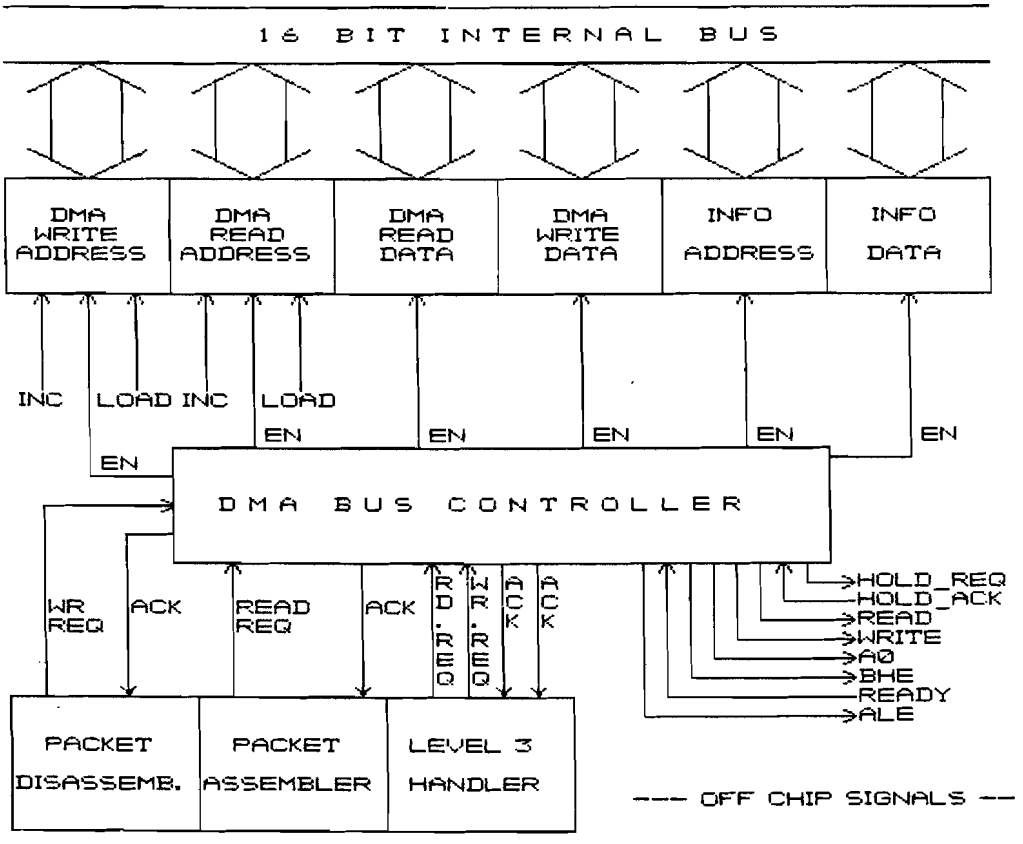


Fig 15. DMA controller (part 1)

The structure of the DMA controller can be seen in figure 15. The processes above are listed in decreasing order of priority. The processes have to initialize the address register, in case of writing also initialize the data register, and request the DMA transfer. The DMA controller will deal with the arbitration. The priorities of the DMA requests are related with the processes in the following way:

DMA	reading	writing
data	2	1
housekeeping information	4	3

where 1 is the highest priority. It must be noticed that the packet disassembler process can only write, the packet assembler process only read, but the level 3 handler process can do both kinds of operation.

14.1 Address space, segmented addressing.

To let the chip be of use for a number of host processors, the address range addressable by the chip is 16 Mega Byte. This implies that a 24 bit address bus is required. Only 64kByte of 16 Mbyte range will be actively addressable by the chip, but the start of this segment can be any multiple of 256, meaning that the last byte of the address is zero. This address forms the start of the structures which are held in shared memory. The segmented addressing scheme has the advantage that only 16 bit addresses have to be stored in the structures, and only a 16 bit wide bus is required in level 3. To use 24 bits throughout would consume too much chip area and too much memory.

With 64 kBytes of memory available for X.25, 250 channels may be defined for use, consuming 32 kByte. The other half would be used for data buffering. The largest number to be used in practice would be 40 channels, so that only 64 kByte available to the chip should be sufficient.

Within the chip, and structures in shared memory, use will be made of offset values within the current segment. The offset together with the segment will form the 24 bit physical address. The physical addresses used can be found from:

$$Phys = offset + (segment * 256)$$

where offset is the address within the segment, and 'segment' is the contents of the segment descriptor register. Logically it means that the 16 bit segment value is shifted left 8 bits, before the offset value is added to yield the Physical address.

```
  OOOO
SSSS00
------ +
PPPPPP
```

Each letter represents a four bit nibble.

PPPPPP is the 24 bit physical address.

This segmented addressing scheme has the advantage that with simple hardware, not needing any multiplyers, the structures in physical memory may start at any position with the least significant 8 bits zero.

Figure 16 shows amongst others the hardware needed for this segmented addressing scheme. The box labeled '+' is a 24 bit address adder.

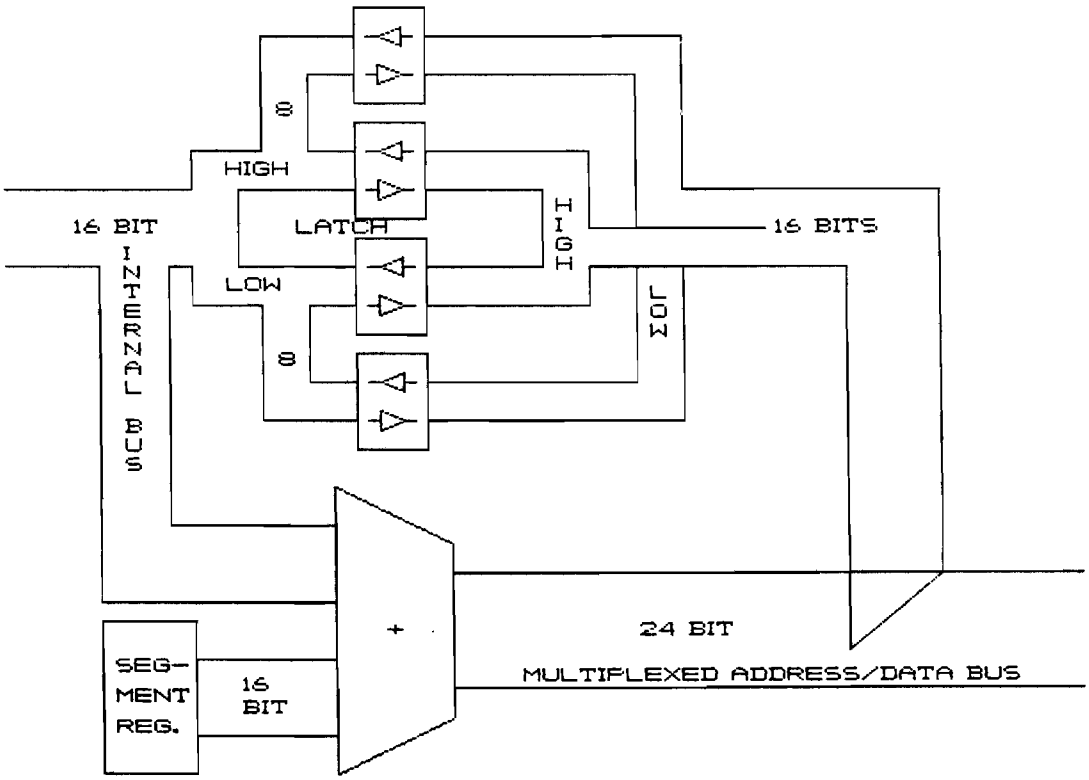


Fig 16. Data bus mapper (DMA part 2)

14.2 Data bus mapping.

Again to be compatible with a number of different host microprocessors, the databus is configureable to be used 8 or 16 bits wide. The 'data bus mapper' has to do packing and unpacking of bytes into 16 bit words when the databus is configured for 8 bits wide. Every word will have to be accessed in two bus cycles.

Another possible and necessary conversion, is the one between the Intel and the

Motorola way of storing a word. Intel stores the low byte first, and Motorola stores the high byte first. In case of Intel, bytes must be swapped when transferred to or from memory. This is done by data bus mapping hardware, which can be seen in figure 16.

The upper four boxes in this figure are tri-state buffers, where the two center boxes are latches. These need to be latches to be able to read 16 bit data with an outside bus of 8 bits.

15. Memory map and mapping hardware.

The structures which are stored in memory are:

- Queue
- Logical channel descriptor table
- free buffer table
- communication registers.

The queue contains 8 packet descriptors of 16 byte each, resulting in 128 bytes in total. 8 entries in this table is enough to support a window size of 7, which is the maximum level 2 window size.

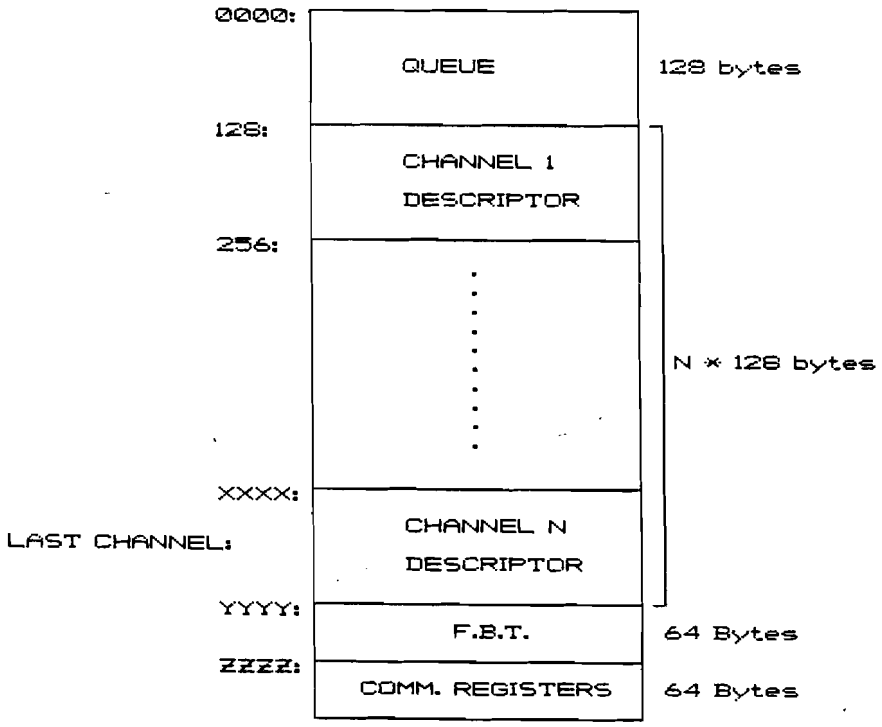


Fig 17. Memory Map

The number of channels to be used is programmable, so the size of the logical

channel table depends on the number of channels. The size of one channel descriptor is 128 bytes, so that a maximum of 254 channels results in 32 kbyte used for structures.

The free buffer table holds 32 entries of 2 byte, resulting in 64 bytes, leaving 64 bytes for the communication registers.

The minimum amount of memory needed is 384 bytes for structures, and a minimum of 3 receive buffers with the size of the maximum packet size. With a total of 1024 bytes the minimum system should be operational.

The mapping of those structures in memory can be seen in figure 17. The base address of this map is the zero address within the segment used by the chip.

The hardware needed for addressing the structures from the chip is given in figure 18. Again there is made use of little area consuming hardware. Use has been made of multiplexers rather than of adders.

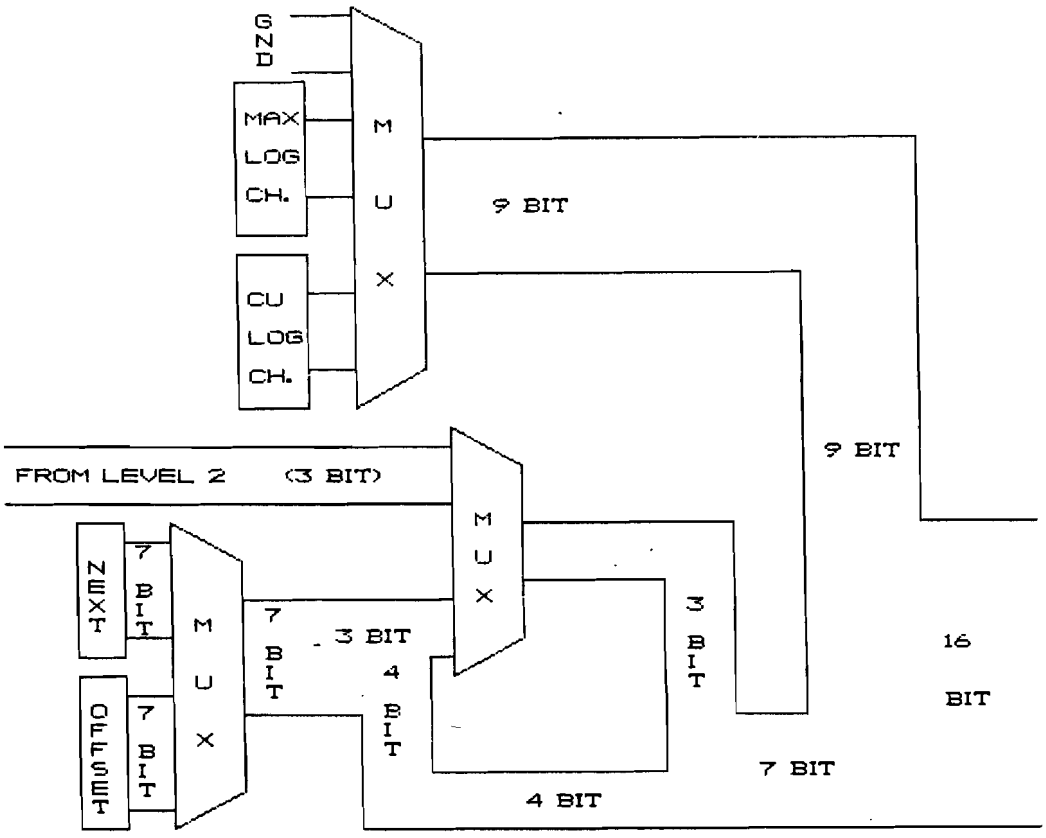


Fig 18. Address mapping hardware

Level 3 uses 16 bit pointers within the segment, where the structures are accessed by:

	Msb															Lsb
bits	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Queue	0	0	0	0	0	0	0	0	0	s	s	s	o	o	o	o
Channel Table	0	c	c	c	c	c	c	c	c	o	o	o	o	o	o	o
Free Buffer	0	m	m	m	m	m	m	m	m	0	f	f	f	f	f	f
Communication	0	m	m	m	m	m	m	m	m	1	d	d	d	d	d	d

The queue is accessed by making bits 7 to 15 zero, the 3 bit sss selects one of the 8 packet descriptors, where the four bit oooo selects one of the 16 bytes of that descriptor.

The channel table is accessed by putting the sequence number into the 8 bit cccccccc, and thereby accessing the desired channel descriptor. The individual bytes may be accessed by the 7 bit ooooooo offset. The 8 bit cccccccc is not the logical channel number itself, but a number derived from it. Channels in X.25 need not be assigned contiguously, but there may be some gaps between different types of channels, which can be seen in figure 19.

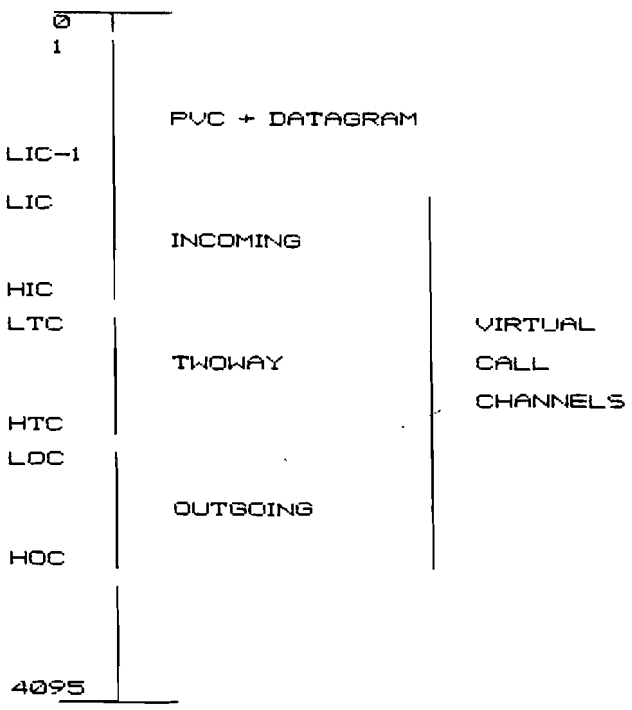


Fig 19. Logical Channel assignment

The free buffer table and the communication registers are accessed by mmmmmmmm which is the number of channels used + 1.
The number of active channels is found from:

NAC = HOC - (LOC - HTC -1) - (LTC - HIC -1)
= HOC - LOC - LTC + HTC + HIC + 2

so,

mmmmmmm = HOC - LOC - LTC + HTC + HIC + 3

fffff and ddddd are used to access the individual bytes of the free buffer table and communication registers, respectively.

To find out the value of ccccccc from the logical channel number, the following algorithm should be used.

```
IF ch_numb <= HOC THEN
  BEGIN
    IF ch_numb > HTC then
      BEGIN
        IF ch_numb < LOC THEN error
        ELSE ccccccc:=ch_numb - LOC - LTC + HTC + HIC + 2
      END ELSE
      BEGIN
        IF ch_numb > HIC THEN
        BEGIN
          IF ch_numb < LTC THEN error
          ELSE ccccccc:=ch_numb - LTC + HIC + 1
        END ELSE
        BEGIN
          ccccccc:=ch_numb;
        END
      END
  END ELSE error;
```

The error occurs when a channel number is higher than the maximum defined, or when the channel number lies in a gap. In this case the received packet is discarded, and an error must be generated.

16. Level 1 implementation

Level 1 is implemented functionally only on chip. The voltage levels and currents are to be taken care of by off-chip drivers, which conform to the X.26 and X.27 standards. Their names are better known as RS-423 and RS-422 drivers/receivers respectively.

The functions of level 1 are simple enough to be implemented by means of a single FSM. This machine must be able to detect the DCE not ready state, and also be able to action two commands. Those commands are the loopback command and the DTE controlled not ready command.

Those commands are given to level 1 by two signals, Enable_loopback and disable_level_1 respectively. When those signals are not active, level 1 should aim at the data transfer state. This can be achieved as soon as the DCE signals DCE ready. Appendix A shows a specific definition of this machine.

17. Level 2 implementation.

The functions to be performed by level 2 are:

- Flag generation and hunting
- FCS generation and checking
- Address generation and checking
- zero insertion/zero deletion
- frame sequence numbering
- sequence error detection
- retransmission
- acknowledge
- flow control

Difference will be made between high level and low level functions within level 2. The low level functions are those as found in HDLC controllers, and are:

- Flag generation and hunting
- FCS generation and checking
- zero insertion/deletion
- Address generation and checking
- frame disassembly

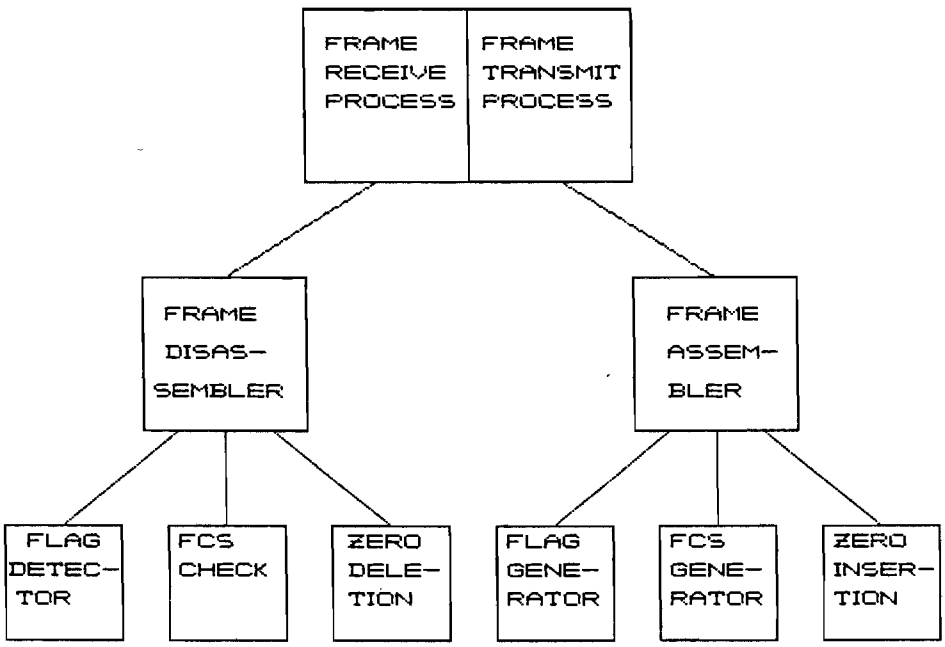


Fig 20. Level 2 Hierarchic Machine structure.

The other functions are considered high level functions. The low level functions are implemented by means of finite state machines. Finite state machines (FSM) will be defined for:

T - Flag / abort sequence generation
R - Flag / abort sequence detection
T - FCS generation
R - FCS detection
T - zero insertion
R - zero deletion

The 'T' machines will be used in the transmitter, and the 'R' machines in the receiver.

To coordinate the three T (R) machines, a medium level finite state machine will be used to assemble (disassemble) the frames, which will be called the low level transmitter (low level receiver).

The high level functions will be performed by a micro programmable machine, which will be called the high level transmitter (high level receiver). The hierarchy of those machines can be seen in figure 20.

Appendix A shows definitions of the finite state machines.

17.1 Functions of the low level transmitter and receiver.

The low level transmitter and receiver within level 2 have the function of frame assembly and disassembly respectively.

In case of the receiver, status signals from the low level 'R' machines are used to disassemble the frame, which means that the header bytes will be put into registers, and the databits will be passed on to level 3. Also control signals will be fed into some low level 'R' machines.
In turn, the low level receiver has to give status signals to the high level receiver and also to level 3.

Control signals will also operate the multiplexers used, which can be seen in figure 21, which shows an overview of level 2.
The datapath in this figure is rudimentary only, it gives a global overview of functions performed within level 2. More detailed diagrams can be found in figure 24. Figure 21 does not show the low level receiver and low level transmitter sequencers. It does show a box labeled 'control' which is not connected to any other box, except by the data bus. All signals from and to sequencers in this diagram go to or from the higher level sequencers just mentioned.

Starting from level 1, the received data bits will be fed into the Flag detection FSM, which will indicate if a flag or abort sequence has occurred. The same bits are also fed into the zero deletion FSM, together with the bit clock signal. Output from this FSM is fed into the FCS-checker FSM, and into a demultiplexer which is controlled by the low level receiver. At the outputs of this demultiplexer we find

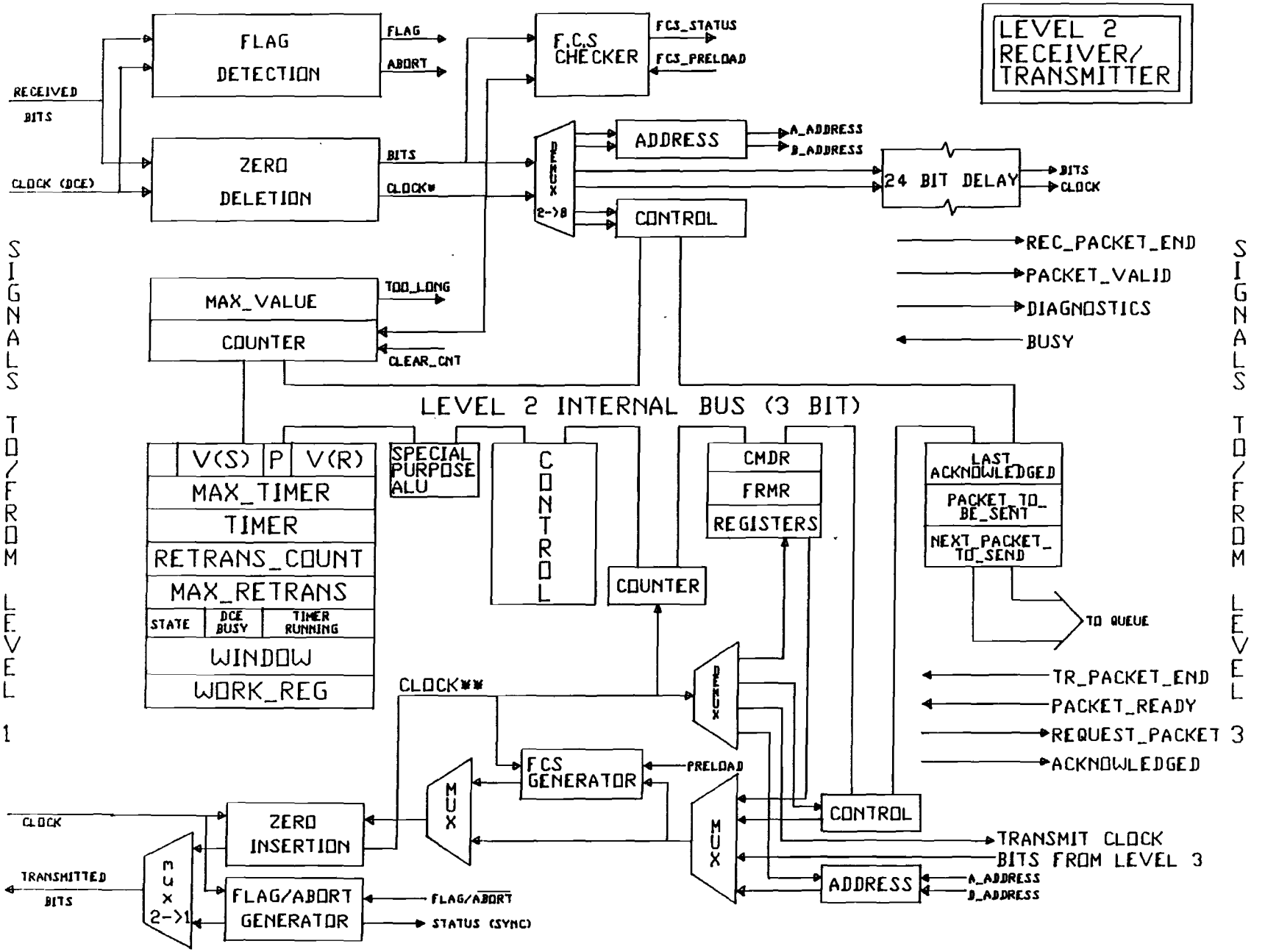


Fig 21. Level 2 Datapath Architecture Overview

an address register, a control register and a 24 bit delay line, which output is fed into level 3.

The Address register also contains some logic, that will indicate whether the A- or the B-address is present or not. Only two addresses are used in X.25 level 2, which however are 8 bits wide each:

A- 00000011
B- 00000001

where the rightmost bit is least significant.

The 24 bit delay line in figure 21. is necessary because databits only must be offered to level 3, and only at time of the end of the 24 bit frame trailer, it is known that the frame ends, so this non-data information will be captured in this delay line. This is a simple 24 bit shift register.

A counter with a max_value register is used to detect frames which exceed the maximum length, and must therefore be discarded. The counter will be reset by the low level receiver, as soon as a flag has been detected.

The transmitter consists of more multiplexers, which all are controlled by the low level transmitter. In fact the reverse process just described takes place. Important in this concept, is that when the high level transmitter decides to transmit an I-frame, which by definition contains a packet, it must request level 3 to prepare this packet. Level 3 must fetch information from the queue, and initialize the transmit data DMA channel. As soon as this is complete, level 3 signals 'packet ready', and the low level transmitter can start to transmit.

Another important fact is, that the high level transmitter must inform the low level transmitter of the type of frame to be transmitted, which consists of the following types:

- I-frame
- all U and S frames except FRMR and CMDR
- FRMR and CMDR

The I frame needs data from level 3, where the other two types do not. The CMDR and FRMR-Frames however, need transmission of the CMDR/FRMR register contents, which is put into the I-field of such frames.

17.2 High level functions within level 2.

The high level transmitter and receiver will be implemented through the use of a microprogrammable machine, because of complexity reasons. Those machines function independently for the greater part, but the operate on the same datastructure.

PROBLEM:
Two independent processes use the same datastructure, which is a problem known as 'sharing'. Furthermore these processes may need to be synchronised at certain instants.

17.2.1 Alternatives for high level receiver and transmitter implementation.

1-
Use of only one sequencer, which will have to perform both transmitter and receiver tasks in a single microprogram. This will probably lead to a difficult and not well structured machine. Another problem could be long response times on outside events.

2-
Use of one sequencer which via some kind of scheduling mechanism allows processes to run concurrently. In this architecture, a microprogram interrupt is required, which also requires a microprogram stack. This implies difficult hardware, and does not yet solve the sharing problem.

3-
Use of two sequencers that operate upon the same datapath, but with a double accu ALU. A bus arbitration mechanism is needed, and also the sharing problem is not yet solved.

4-
Use of two sequencers, and dual ported registers, which only leaves the sharing problem. A disadvantage is the large chip area consumption.

5-
Use of two sequencers and two sets of registers which will have to be updated to contain equal information. This requires even more chip area than alternative 4.

In case of alternative 3, two problems have to be solved, bus-arbitration and sharing. The bus arbitration problem can be solved elegantly through the use of a four phase clock, where the transmitter process runs at two phases and the receiver process at the other two. This also saves on chip area, because sequencer hardware can also be shared between the two processes, such as an incrementer.

A clear disadvantage of this solution is that the processes will be slowed down by a factor of 2. The question must be asked if fast processing is required at all, which most probably is not required.

The other problem is sharing. A number of registers are shared, and when both processes need to write in the same register, some sharing mechanism must be used. Semaphores are generally used for such a purpose (12), and in this case binary hardware semaphores will be used. The groups of registers to be protected by one semaphore are:

- * frame_to_send(1..2)
  * I_to_send
  * AB_seq
  * P_F_bit
  * command

- * in_betwener
  * V(S)
  * next_packet_to_send

- * FRMR/CMDR registers

The accumulator has to be shared as well, but if that is done by a semaphore, most of the time one of the processes is waiting for the other to release it. The accu will implemented twofold, one accu assigned to each process.

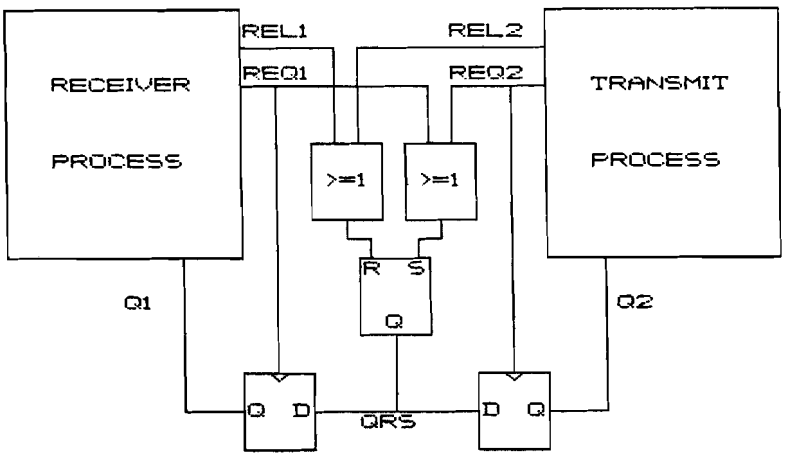


Fig 22. Binary Semaphore hardware.

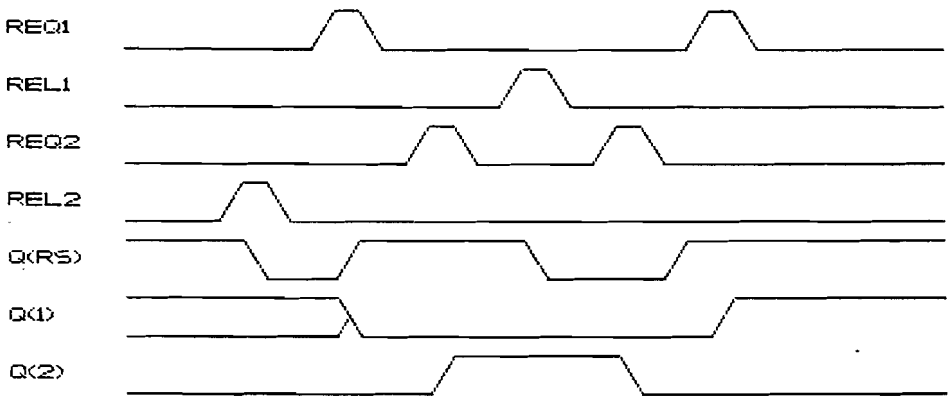


Fig 23. Timing of Binary semaphore hardware.

3 Semaphores are needed to share 3 groups of registers. Binary semaphores will be implemented in hardware, as shown in figure 22. This hardware can be fairly simple because the 2 processes will never run at exactly the same instant, as they run on an interleaved basis. A time diagram in figure 23. shows when the semaphores can be set or cleared by a process. A set command can be given at all times, but when the semaphore was already set, the process that tried to set it, did not acquire the desired registers.
The semaphore is set when Q(SR) is high. When the other process makes a REQuest, it will find its own Q line high, which latches the previous Q(RS) state. When a process RELeases, the Q(RS) goes low, and with a next REQuest this low state will be latched into its Q. This indicates that the request was successful. As long as a request is done with Q=high, the request should be repeated, which leads to a busy form of waiting.

With the above solutions, alternative 3 clearly has fewer disadvantages than the other alternatives, and does not consume very much chip area, and will therefore be chosen in the level 2 implementation.

17.3 Operation types in level 2 microprograms.

Appendix B. contains a number of programs of which the first is the microprogram that should be run by the high level receiver and transmitter within level 2. This program has been written in a Pseudo Pascal language, and has been converted to a more microprogram-like form in B.2. This is not a notation as found with 'real' micro-assemblers, but still has pascal like assignment statements to improve readability. This program has been analysed to find the different types of operations, and those types are listed below:

BRANCH INSTRUCTIONS

- jmp :unconditional jump
- jfalse :jump if selected condition is false
- jtrue :jump if selected condition is true

- jz :jump if accumulator contents is zero
- jnz :jump if contents is not zero

- call :unconditional call, affects stack
- ret :unconditional return, affects stack

ASSIGNMENT STATEMENTS (3 bit wide)

- reg:=reg :register to register transfer
- reg:=const:constant to register transfer

ARITHMATICS (3 bit wide)
- part one: ALU operations
  - accu:=accu + register
  - accu:=accu + constant
  - accu:=accu - register
  - accu:=accu - constant

  The zero flag will be affected by these operations.

- part two: IN_BETWENER operations
  - flag:= low_bnd <= tested <= high_bnd

MISCELLANEOUS

- select:=...	:select condition to test
- set(..)	:set output
- clear(..)	:clear output
- signal(..)	:signal semaphore
- wait(..)	:wait on semaphore

As can be seen from these operations, all arithmatic and assignment statements are 3 bit wide, which results directly from the fact that level 2 employs modulo 8 sequence numbering. For this reason a 3 bit wide databus will be used in level 2, which saves again on consumed chip area. Also the ALU can be of a very simple type, since only addition and subtraction are required operations.

As disadvantage of the 3 bit wide bus and 3 bit registers used, it can be stated that it will be impossible to use modulo 128 sequence numbering in level 2, since this requires 7 bits. The advantage of low chip area consumption outgrows the disadvantage. The modulo 128 sequence numbering is not very much used in practice, and all networks should offer modulo 8 sequence numbering, so that this chip can be used without that option.

17.4 Level 2 datapath.

As shown from the instruction repertoire, most registers and operations are 3 bit wide only. A datapath has been designed which uses a 3 bit wide bus. See figure 24a. Besides from the registers (See Appendix B), a number of flags are used, which are implemented as a number of Flip/Flops, that will be grouped to form a status register (these are not drawn in figure 24a). The three registers containing queue information, are grouped together, and are dual ported so that level 3 can use them as well. No sharing problems will occur, since only one level writes in a certain register where the other will only read its contents.

A number of registers, such a Max_retrans and max_length, are more then 3 bits wide (figure 24b). Those registers will only be written to at power up initialization, since these registers contain level 2 parameters, that remain constant during operation. Installing a wide bus for those registers would not be worth the effort. For this reason a serial bus will be used, that will not consume

much chip area. A disadvantage is slow speed, but that should not be a problem. The registers containing the parameters need to be shift registers, but a shift register does not need very much more chip area than a normal register.

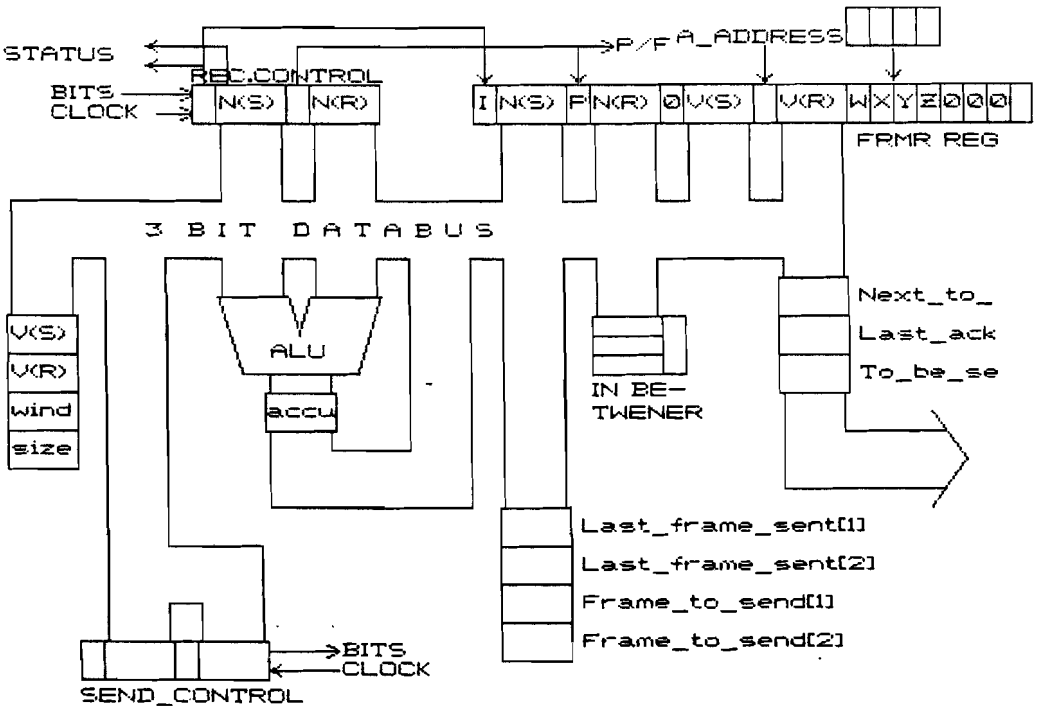


Fig 24a. Level 2 Datapath part 1

Those registers are used in dedicated machines for checking the length of a frame, the maximum number of retransmissions and a time out on transmitted frames. Those machines are used by level 2 higher level machines.

17.4.1 In_betwener.

A dedicated piece of hardware employed by level 2, is the in_betwener (see figure 24). This machine has 3 registers each 3 bits wide, and performs the following operation:

Flag := low_bound <= tested_value <= high_bound

The flag is a status line that indicates if a modulo 8 value lies between two modulo 8 limits. To give some examples:

0 <= 3 <= 3 : true
1 <= 6 <= 4 : false
4 <= 6 <= 1 : true

Implementing this operation with an 8 bit microprocessor would be a tedious job, in hardware however, this becomes more simple, because the hardware is tailored down to the size of the modulo 8 values: 3 bits.

This piece of hardware is also the main reason that only a very simple ALU is required.

17.4.2 FRMR / CMDR register.

This register is used to transmit level 2 diagnostics, and contains the information field of FRMR and CMDR frames (1). This register contains the received control field, which copied in two 4 bit slices, and also the state of the level 2 internal variables V(S) and V(R). The last part contains a code for the type of detected error (1). This code is held in a separate 4 bit status register, which is copied into the FRMR/CMDR register before transmission. This allows new error detection of new received frames during FRMR/CMDR frame transmission.

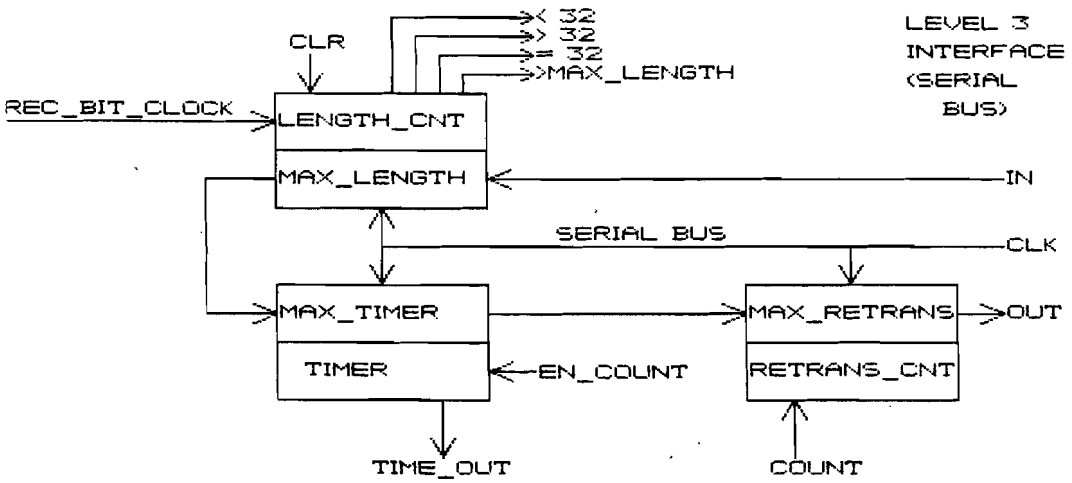


Fig 24b. Level 2 datapath, part 2 (serial bus)

18. Interface signals between level 1 and level 2.

Only a small number of interface signals cross the level 1 / level 2 border. Those signals are:

- receive data bits
- receive data bit clock

- transmit data bits
- transmit data bit clock

- disable level 1 (from level 2)

Only the last signal needs explication. This signal from level 2 indicates to level 1 that it should go to the 'DTE controlled not ready' state(1). This means that on level 1 it can be signalled that the chip is not ready for data exchange. This will only be signalled when power is applied to the chip, until the initialization is complete, or when a 'stop communication' command is given.

No special status signal is required from level 1 to level 2, since the clock signals to level 2 will be stopped when level 1 is not ready for data transmission.

19. Interface signals between level 2 and level 3.

The interface signals between level 2 and level 3 are:

- receive data bits
- receive data clock
- receive packet end
- receive packet valid
- receive diagnostics
- busy (from level 3)

- transmit data bits
- transmit data clock
- request packet
- packet ready
- transmit packet end
- packet acknowledge
- queue index (3 bits)
- read/write/select

- initialize data input
- initialize data output
- initialize clock

- level 2 ready
- enable level 2

Most of these signals can be seen in figure 21. These signals are on chip signals, that cannot be seen from outside the chip. In case however, that the chip should be integrated in two parts, this would be the place to cut the chip. The number of signals is limited, and this means that there is a nice line of demarcation between level 2 and level 3.

A description of these signals is given below.

19.1 Receive data bits and receive data bit clock.

These signals transfer the received packets to level 3. Since this is in a serial bitstream, the clock must also be given. This clock signal cannot be shared with the transmit clock, because of the zero insertion and deletion. This mechanism causes the clock signal to be irregular at times of zero deletion. The data bits may either contain:

- Packet bits (headers included)
- level 2 diagnostics.

Level 2 indicates when diagnostics data is transferred.

Level 2 also indicates when a packet is valid, and when the packet ends.

19.2 Receive packet end and packet valid signals.

The Receive packet end signal is used as indication that the last bit has been transferred. To indicate if this packet is valid, the Packet Valid signal will be activated before the packet end signal. As level 2 does not know if a packet is valid until the last bit has been received, and because it is impossible to store the packet in level 2, data bits transferred to level 3 may be invalid. According to the X.25 protocol, only valid packets will be transferred from level 2 to level 3. In practice, this is almost impossible, since storage space will consume far to much chip area.

19.3 level 2 diagnostic signal.

In case that level 2 receives diagnostic information, (FRMR or CMDR frame), it must put the I-field of such a frame somewhere into hosts memory. Level 2 does not have its own DMA controller, and therefore cannot access memory. Whenever level 2 receives diagnostics, the Level 2 Diagnostic signal will be activated before the first bit is transferred to level 3. Level 3 knows in advance that this not a packet, and should not be treated as such. All that it has to do is write it into a buffer, and adjust some pointers. Certainly, it must not disassemble the packet, and try to interpret the headers.

In this case also the packet end and packet valid signals are of importance. The contents of the diagnostics frame must be thrown away if the Packet Valid signal is not activated before the Packet End signal.

Level 2 can derive the Level 2 Diagnostic signal directly from the control field register, where one bit indicates the frame type to be diagnostic or not.

19.3 Busy signal.

To enable level 2 flow control, level 3 must indicate to level 2, that there is no more, or almost no more receive buffer storage space available. If this is done in time, level 2 can transmit a RNR frame to stop the reception of Information frames that contain packets. Because an I-frame can be on its way, the RNR command must be given in advance. Level 3 must activate the busy signal when one or no receive buffers are present.

19.4 Transmit data bits and clock signals.

The transmit data bit signal is fed into level 2, but the clock signal is delivered to level 3 by level 2. This is the case because the DCE delivers the clock signal to the chip, to which all data transitions have to be synchronised. The Transmit data bit signal always contains packet information.

19.5 Request Packet and Packet ready signals.

These signals are needed because level 2 cannot access hosts memory. Instead, when it needs to transmit a packet, it must activate the request packet signal to ask level 3 to prepare a packet for transmission. When level 3 has completed this work, it will signal this to level 2 by activating the packet ready signal.

19.6 Queue index and read/write/select signals.

This is a three bit index register, which must be used by level 3 to prepare the packet desired by level 2. This queue index signal is used together with the read/write/select signals (from level 3), to control the second port of the dual ported registers, which contain:

- Last_acknowledged
- Next_to_send
- to_be_sent.

When level 3 prepares a packet, it must use the Next_to_send register as an index. It may not alter its contents however. It may only alter the contents of the To_Be_sent and the Last_Acknowledged register, when a new packet has been added to the queue or a packet is removed from the queue respectively.

19.7 Transmit Packet End signal.

This signal will be activated by level 3, to indicate level 2 that the last bit of a packet is being transferred. This depends on packet length.

19.8 Packet Acknowledge.

The Packet Acknowledge signal indicates to level 3, that one packet from the queue has been acknowledged, and may therefore be removed from the queue. This means that level 3 can free the buffer used for its data field by using the originating channel pointer.

19.9 Initialize data and clock signals.

This is a serial bus, which transfers the parameters to level 2. Both signals are delivered by level 3. This clock signal is independent from any other clock signal discussed before.

19.10 Level 2 ready and Enable Level 2 signals.

The level 2 ready signal indicates to level 3 that level 2 is in the data transfer state. The Enable signal from level 3 indicates to level 2 that is may maintain the link or not. In the Enable active state, level 2 must go to the 'Information Transfer state' and when Enable is not active, it must go to the 'Disconnected state'.

20. Interface signals between level 1 and level 3.

This very unusual interface signal passes level 2. It is used for diagnostic purposes:

- Enable_loopback

This signal from level 3, indicates to level 2 that the internal loopback mode should be used. This implies that no communication with the DCE will occur, but an external clock signal (S-signal) must be applied to the chip.

Level 1 will signal 'DTE controlled not ready', and the loop will be made at the side of level 2 within level 1.

21. Interface signals between chip and host.

These signals are really the signals between level 3 and level 4+. These signals will be external pins of the chip.

- AD0..AD15, Bidirectional
These signals form the multiplexed address/ data bus. When an 8 bit data bus is used, only AD0..AD7 will be used for the data bus. The address can be demultiplexed with use of an address latch external to the chip.

- ALE, output
This signal latches the lower 16 address bits into the address latch.

- A16..A23, output
Upper 8 bits of the address lines. These are not multiplexed.

- Bus_hold, output
  Bus_ack , input
These signals request (and grant) the shared bus to perform accesses to shared memory on DMA basis.

- Int/Mot, input
This signal indicates to the chip, which bus protocol should be used. This can be either the Intel or the Motorola protocol.

- Ready, input
This signal indicates to the chip, that a bus cycle has completed. It allows for insertion of wait states in memory accesses.

- 8/16, input
This signal indicates to the chip the width of the data bus, which can be 8 or 16 bits wide.

- BHE, output
This signal indicates that the high byte of the databus is used in a transfer. Together with the A0 line this indicates:

A0	BHE	
0	0	16 bits
1	0	high byte only
0	1	low byte only

- Read, Write, output
These signals indicate that a read or a write transfer to memory occur. These signals are the R/W_ and Enable in case of the motorola bus protocol.

- PHI1, PHI2, input
These form a high frequency two phase clock input to the chip, which is used internally for the sequencers and finite state machines. This clock is independent from the S-signal, which forms the received and transmitted bit clock input from the DCE.

- INT, output
These are two signals (High and Low priority) to the host, which are used as interrupt signals.

- ATT, input
The attention signal from the host functions as an interrupt signal to the chip.

- GND
Ground potential input.

Other signals which are not part of the level 3 level 4 interface signals are the X.25 interface signals, and are also listed below:

T - Transmit, output
R - Receive , input
C - Control , output
I - Indication,input
S - signal element timing (clock), input

These signals are functional only, and do not conform to the voltage levels as defined by (1). These voltage levels can be converted to and from using drivers and receivers for the RS-422 or RS-423 (X.27 and X.26) respectively.

22. Packet Level implementation.

The functions of level 3 are very much the same as those of level 2. The difference is that level 2 deals with only one physical link, where level 3 deals with up to 4095 virtual channels. On each of those channels level 3 performs the functions listed below:

- packet assembly / disassembly
- packet sequence numbering
- sequence error detection
- flow control
- acknowledgement
- virtual call set up
- virtual call clearing

The decomposition of this level is done in the same way as that of level 2. There are a number of finite state machines that perform the following functions:

- packet disassembly
- packet assembly
- grouping received data bits
- ungrouping data bits to be transmitted

These functions are again low level functions within level 3. The other functions are the high level functions within level 3.
The packet disassembly process controls the demultiplexer, that directs the bitstream to a number of registers. These registers contain the packet header, such as:

4 - GFID General Format Identifier
12 - Channel number
8 - packet type identifier
16 - data shift register

These registers are connected to a 16 bit wide bus, which is used throughout level 3.

The packet assembly process does the reverse, and also uses the same set of registers, as can be seen in figure 25.

The grouping and ungrouping of data bits is the conversion from serial to parallel and vica versa. The word length is programmable, and only applies to user data. The user data can be grouped into words of 1 to 16 bits. The word length is programmable, but applies to all channels and cannot be altered per channel. This cannot be done on a channel basis, because when a data packet is received, in advance it is not known to which channel it belongs. Another function is to transfer the received bytes to memory. A handshake mechanism with the DMA controller will transfer these bytes to the receive buffer.

When no user data is handled, the grouping and ungrouping will be done bytewise. The fact that user data will be received is indicated directly by bit 0 of the packet type ID field.

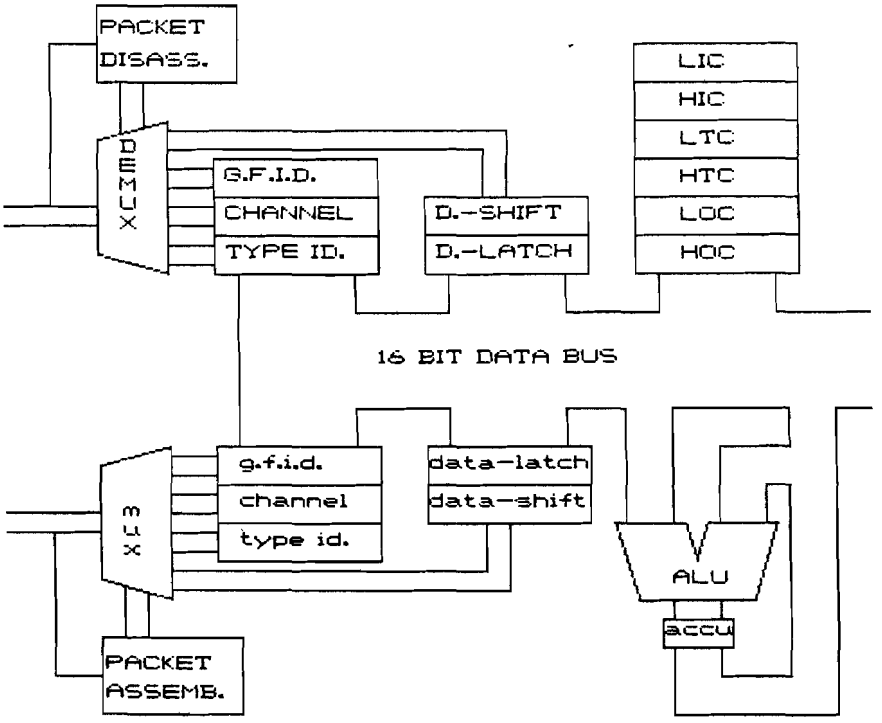


Fig 25. Level 3 Datapath (without DMA)

The High level functions within level 3 are implemented in a microprogrammable machine.

22.1 Level 3 software.

The task of this software is comprised of the following elements:

- maintaining the state of every channel
- respond to inputs on a channel
- queue management
- keeping track of data to be transmitted
- updating timers
- communication register management

22.1.1 Maintaining the state of every logical channel.

Each channel is in one of 12 states, of which 3 are nested according to the X.25 definition. Actions to be taken on the reception of packets in certain states can found in tables 1/X.25 through C.4/X.25 of Annex C to X.25 (1).These tables are very large, and it would certainly be a tedious job to implement these in software. In Pascal it probably would lead to a complicated construction of CASE and IF THEN ELSE statements. To store a large table would consume too much chip area. The solution to this problem is a Next_State_generator, which is a PLA that generates the next state and output vectors on the following inputs:

- current state
- Packet type ID
- a number of input flags which are:
  * length < 2 octets
  * incorrect GFID
  * unassigned logical channel
  * restart request/confirmation with nonzero channel
  * packet type identifier too short
  * invalid packet type for PVC channel
  * REJ facility not subscribed
  * Timer expired.

Appendix C contains a table with the inputs and outputs of the next_state_generator. This table should be minimised and implemented in a PLA. All diagnostics code in the table can be omitted, because the DTE cannot transmit a diagnostic packet. The X.25 level 3 protocol is not symmetrical, and the table also contains the DCE elements.

22.1.2 Level 3 Operation types.

A program has been written in a pseudo Pascal language, which globally covers the level 3 protocol. This program can be found in appendix B. From this program the following operation types are found:

- 3 bit: modulo 8 P(R) and P(S)
- 4 bit: offset addressing
- 6 bit: pointer index operations
- 7 bit: offset addressing
- 12 bit: logical channel number manipulation
- 16 bit: pointer manipulation

For all types the ALU must be able to perform the following functions:

- addition
- subtraction
- and
- or
- shift

For this reason a 16 bit ALU will be used, that can be configured by microprogram parameter to the word size to be used. This means that the carry status can be programmed to the nr of bits of the current wordsize.

22.1.3 Level 3 processes.

Important are the packet assembly and disassembly process, which are implemented by FSM's. These two processes run in parallel, and use the same host system bus to access shared memory. With maximum X.25 data rate, 6000 bytes per second will have to be written to memory, and 6000 bytes per second will have to be read from memory, yielding a busload of 12kbyte per second. This is the highest data bus load. The access to the structures held in memory must be added to that number, and is estimated to be 20 percent of the data transfer, yielding an average of 15 kByte per second. As this is not a very high transfer rate, no FIFO's are needed.

The high level processes that have to be performed are:

- processing the received packets, and
- updating the logical channel
- initializing the low level transmitter upon request from level 2.
- updating tables from acknowledged packets
- sending household packets (non-data)
- sending data packets
- updating receiver pointer table
- updating timers and take appropriate actions

These processes are listed in decreasing order of priority. All of these processes have to share the third DMA channel (House keeping information), and also have to share the same on chip registers. In principle, all of these processes could run in parallel, but because they need to share so much this has not much advantage. To simplify the hardware, use will be made of one sequencer, where a dispatcher schedules the processes to run. The sharing problem is solved when no process can start before another has finished the kernel of its task.

A status register within level 3 indicates which process needs to run. One of those bits is set by the Attention signal from the host.

22.2 Updating channels on packet reception.

This task has the highest priority, since this task has to be completed before another packet can be received. In case of data packets, pointers to the receive buffer must be put in the channel descriptor, and the acknowledgement must be initiated. If there are no more unused receive buffer pointers in that channel descriptor, a RNR must be transmitted. When data has to be sent on the same channel, the piggy back acknowledge may be used, and no further action is

necessary. When no data has to be transmitted (on the same channel), and receive buffers are available, a RR must be transmitted. Finally a low priority interrupt may be generated at the host.

Other packets need updating of the special purpose pointers in the channel descriptor. A high priority interrupt must be generated at the host.

In any case, the state of the channel must be fed into the next state generator, together with status bits from the packet disassembler. The output vector tells the channel updating process what to do exactly. A multiway branch instruction could speed up this process(2). Also, when the packet has been received, the next DMA receive pointer must be initialized, or must be put back to its old value when level 2 indicates this packet to be invalid.

22.3 Initializing the packet assembler.

Whenever level 2 decides to transmit an I-frame, the packet to be transmitted must be ready for transmission, as the packets and frames are assembled on the fly. Before frame transmission, level 2 asks level 3 to prepare a packet by activating the interface signal 'request packet'. The packet to be selected from the queue is found from the 'next_to_send' register. When the transmission of the requested packet is ready, the 'packet ready' signal will be activated.

The initialization is comprised of the following elements:

- fetching the packet header, and putting them in the header registers.
- fetching the pointer and initializing the DMA read and length counter.
- activating the packet ready signal.

This process also has a high priority, because it forms a bottleneck for packet transmission. Throughput is seriously affected when this process is held up.

22.4 Updating the queue from level 2 acknowledgements

When level 2 indicates that a packet from the queue is acknowledged, level 3 must remove it from the queue, and free the pointers. This is always done in the queue and for certain packet types, a pointer traces back to the originating logical channel, where the pointer must also be released.

22.5 Sending non-data packets

This process has a higher priority than the data transmission itself, because without these packets there will be a serious limitation on data transmission throughput on virtual call channels. The table of high priority pointers will be processed, and causes a packet to be put into the queue. After one packet this

process will be exited, to enable higher priority processes. When there are none ready to run, this process will be activated once more.

This goes on until there is only one entry in the queue available, then this process also becomes blocked. This gives opportunity for lower priority processes to run.

22.6 Sending data packets.

The next channel descriptor in the circular linked list will be accessed, and a data packet will be added to the queue. The working pointer will be shifted to the next channel descriptor, and in some cases the old channel descriptor may be removed from the linked list. Also after one entry to the queue this process will be exited, thereby enabling higher priority processes to run.

22.7 Updating timers and take actions on time out.

A hardware timer within level 3 will set a bit in the level 3 status register every four seconds. When this process is active and that particular bit is set, the timer in the first channel descriptor will be updated. A work pointer register keeps track of the current channel, and after processing a channel this process will be exited, thereby enabling higher priority processes. When the last channel has been processed, the bit in the status register will be cleared.

This process will also be used as the idle process, that is constantly swapped in and out by the dispatcher in case that none of the processes needs to run and is blocked.

22.8 The dispatcher.

This is a very simple process, that works on basis of the status register. This process takes the highest priority bit from that register and jumps to the corresponding process.

Processes that consider theirselves ready, must clear the corresponding status bit.

22.9 Communication register management.

As soon as an attention is generated at the chip, this process examines the communication registers. Bits in the status register will be set to start corresponding processes. This process will have to do some of the functions by itself. These functions are for instance: software implementing diagnostic modes such a loopback, de-activation of level 3 and the register dump.

Other tasks are the starting and stopping of X.25 communication.

23. Power up initialization.

After power on reset, all communication is stopped, and no access to the shared memory will be done. The chip will signal 'DTE controlled not ready' at level 1. The first parameter the host must prepare, is the segment base, which forms the base address for the structures held in memory (see the memory map). The first chip attention will cause the chip to fetch the segment descriptor from memory with all address bits set to zero, resulting in a physical address 000000.

The host must prepare the initial parameters in this segment, which are:

level 2:
- N1
- N2
- T1
- window size

level 3:
- wordwidth
- LIC
- HIC
- LTC
- HTC
- LOC
- HOC

Again an attention must be generated at the chip, and all these parameters will be loaded into chip internal registers. As soon as that has happened, an interrupt will be generated at the host, so that it can initialize all structures in memory. This means that all fields in each channel descriptor must be initialized, And a number of receive buffers must be appended to the (empty) free buffer list. This is done via the 'append receive buffer' command. When all of this has completed, an initialize X.25 command must be given, which will cause all three levels to go to the ready state. This also implies a 'restart' command at level 3.

Hereafter the chip initialization is complete, and X.25 communication starts.

24. Testability.

A very important item of chips is their testability. In this phase of the design, no technology to produce the chip has been chosen, so that the testability can only be a functional one.

Depending on technology, the scan path could be implemented for testing.

On a functional level, the items listed next will be implemented:

- test loops
- register dump
- de-activation of level 3

24.1 Test loops

The chip may incorporate 1 test loop, which will be located at the physical level. Other test loops, as defined by X.25, will have to be placed outside the chip, and will therefore not be very important in chip testing. The chip can be programmed to work in loopback mode, and the X.25 interface signals will not be used, except for the clock input.

24.2 Register Dump.

This general command will cause every internal register to be dumped into a memory buffer. During this dump no X.25 communication can take place, so the 'stop X.25 communication' command must be given in advance, if no violation of the X.25 protocol may occur. Such a register dump checks whether the registers contain the desired values, as a number of those registers contain parameters that remain constant during operation. Other registers may be used to find out the state of a certain level and exactly what is happening.

24.3 De-activation of level 3.

In this mode, the level 2 protocol is the only one tested, where packets are passed on to memory buffers as they are received. Only the level 2 protocol is handled by the chip. The level 3 protocol may be implemented by software, but that is not an easy task. In this case the host must write packets into the queue, and use the channel attention to indicate that a packet has been appended.

24.4 Other ways of testing.

Another way of testing can be performed without special commands given to the chip. This is done by examining the contents of the queue when the chip is in normal operation. As this queue is placed in shared memory, that is very easy.

24.5 Diagnostic programming.

To command the chip to perform certain diagnostics, the specific command 'diagnostics mode' must be given, with the mode number in the parameter register.

25. Additional features.

A number of features could be added to the chip, but they require on chip registers that consume extra area, and they can only be implemented when there is chip area available. Sofar no decision has been made whether to implement those features or not. This is more of an indication of things that could be done extra.

The feature discussed here has to do with statistics. A counter could be added in level 2, which counts the total number of frame retransmissions since power on reset. This could be useful in determining the quality of the physical link. Another counter could count the number of discarded frames as a result of FCS errors. The contents of such registers could be read by using the register dump facility.

Another feature would be to expand the diagnostics with a self test command, where registers and ALU's are tested yielding a diagnostics code as output. This would increase the size of the microprogram, and therefore again the chip area.

26. Examples.

In this paragraph an example will be given of how user data is received into memory.

Figure 3 shows an I-frame, where the information frame contains a data packet, where the data field contains the desired user data.

In advance, the pointer to the next free receive buffer must be copied into the DMA write register. When the frame arrives, level 2 will be synchronised by the opening flag, and the header bytes such as address and control field will be placed in the level 2 receive registers. The low level receive process will pass the remaining bits on to level 3. In the mean time, level 2 may evaluate the header registers, and when the frame is completed, the last 24 bits of that frame will be caught in the 24 bit delay shift register, so that the 16 bit FCS and te closing flag are not passed on to level 3. Also the clock signal will stop at the input of level 3. If the FCS and the sequence numbers of that I-frame check out, the Valid_Packet signal will be activated together with the Packet_End signal. Level 3 now knows the packet to be valid.

When the first bits of a packet enter level 3 at time of reception, the packet header bytes, which are GFID, channel nr. and packet type field, are received into level 3 internal registers. The following bits will be converted from serial to parallel, and written into the prepared receive buffer on DMA basis.

When the Packet_valid and the Packet_End signals are activated, level 3 will copy the pointer to the base of the receive buffer into the table of receive pointers at the channel descriptor. The channel descriptor is accessed by taking the channel number from the channel number register, and use this number to access the logical channel table at the desired descriptor. The length of the packet can be found from the base register and the DMA receive pointer register, and must also be placed in the descriptor.

Finally, the number of unused pointers must be decremented, and the next pointer in the FBT must be copied into the DMA receive register, to be able to receive the next packet.

Optionally an interrupt can be generated at the host, so that it si notified of this event. In that case a pointer to the channel descriptor in concern will be placed in the communication registers.

Another example will be given in a moment, where an incoming call packet will be received. In this case all actions in level 2 remain the same, because all packets are transmitted within I-frames.

In case of an incoming call packet, no user data will be received, but a CCB, communication control block, will be received into a receive buffer. Again the channel number will be used to find the correct descriptor, and the pointer/length combination will be written into the special purpose receive pointer field. A pointer to the channel descriptor will be written into the communication registers,

and a high priority interrupt will be generated at the host.
The host inspects the CCB, and determines whether to accept the call or not. The host must give a open or close communication command, and the chip will transmit the corresponding call accept or call clear packets. Also the pointer to the next free buffer will be updated.

27. Conclusions.

Clearly this project only just has started off, but sofar a number of conclusions can be drawn anyway.

The first conclusion to be drawn is, that it is possible to implement the full X.25 protocol on a single chip. This chip will however be of great complexity, of which this report contains a functional specification only. This conclusion is based on the fact that the complexity of the X.25 chip is comparable to that of a 68000 microprocessor, which has been integrated in custom design.

The separation into layers of the X.25 protocol can be maintained for the greater part. A small number of concessions had to be made, to minimise the chip area. This is the fact that level 3 is offered packets which may or may not be damaged, which means that extra information is fed into level 3 by level 2, informing level 3 if a packet is correct or not. The advantages clearly outweigh the disadvantages.

The chip is designed to be used for small systems, as well as big systems. This also means that a flexibility must be built in. Furthermore, it is useable with a large number of different microprocessors which can be of either the 8 or the 16 bit generation.

The decomposition of X.25 into a number of subtasks seems fruitful. Two microprogrammable machines have been defined, and about 15 finite state machines will be used. Very interesting are the special purpose machines, such as the in_betwener and the next_state generator. These two machines save enormously on processing, and on consumed hardware. These machines are a typical example of one of the benefits of VLSI.

A word must also be said about verification of this design. In practice verification without CAD tools is almost impossible, since the human element is not removed when verification is done by hand. In the long run, by looking at such a design frequently, and discussing it with other people, a large number of inconsistensies have been removed.

A final conclusion to be drawn would be that a specification of an intelligent I/O controller has been designed. It is a chip that is used as a dedicated coprocessor, which will relieve the host processor of the tedious X.25 task. A powerful memory management system minimises overhead, and upgrades the chip - host software interface to a functional one.

28. Work that remains to be done.

First a verification cycle by use of CAD tools would be recommended. That would consume a great deal of time already, but it would show more formally that this design has no flaws.

From this point on, an electrical diagram would have to be designed, which would be more or less a translation of all diagrams into gates. At that point timing analysis would have to be done, after which a layout can be made. This layout should not be too difficult, since regular structures only, such as registers, multiplexers and PLA's have been applied only. All these tasks should be performed with the use of CAD tools.

A more difficult task would be the programming of the microprogrammable machines. A first indication of what has to be done is given in appendix B.

29. Final word.

At this place I would like to thank all persons who have helped me during my study, and especially those who helped me during my final work at the University.

Special thanks go to ir. M. Stevens, who proved to be a reliable coach during this project.

30. Literature:

(1) CCITT Yellow Book Fascicle VIII.2
Recommendations X.25 and X.21
Geneva 1980

(2) Mick J. and Brick J.
Bit Slice Microprocesor Design
New York: Mc Graw Hill, 1980

(3) Grass W.
Steuerwerke, Entwurf von Schaltwerke mit festwert speichern
Springer Verlag, Berlin Heidelberg New York 78

(4) Gill A.
Introduction to the theory of finite state machines
Mc Graw Hill Electronic Science Series 82

(5) Hill F.J. and Peterson G.R.
Introduction to switching theory and logical design.
Wiley, third edition

(6) Mead C. and Conway L.
Introduction to VLSI Systems
Addison-Wesley 1980

(7) Mavor J, and Jack M.A. and Denyer P.B.
Introduction to MOS LSI design
Addison Wesley 1983

(8) Newkirk J. and Mathews R.
The VLSI designers library
Addison-Wesley 1983

(9) The 8086 Family users manual
chapter 3: 8089 Input Output Processor
Intel, oct 79

(10) Meijer A. and Peeters P.
Computer Network Architectures
Pitman

(11) Tanenbaum A.S.
Computer Networks
Prentice Hall, 1981

(12) Lister A.M.
Fundamentals of Operating Systems
The Macmillan Press LTD, second edition, 1979

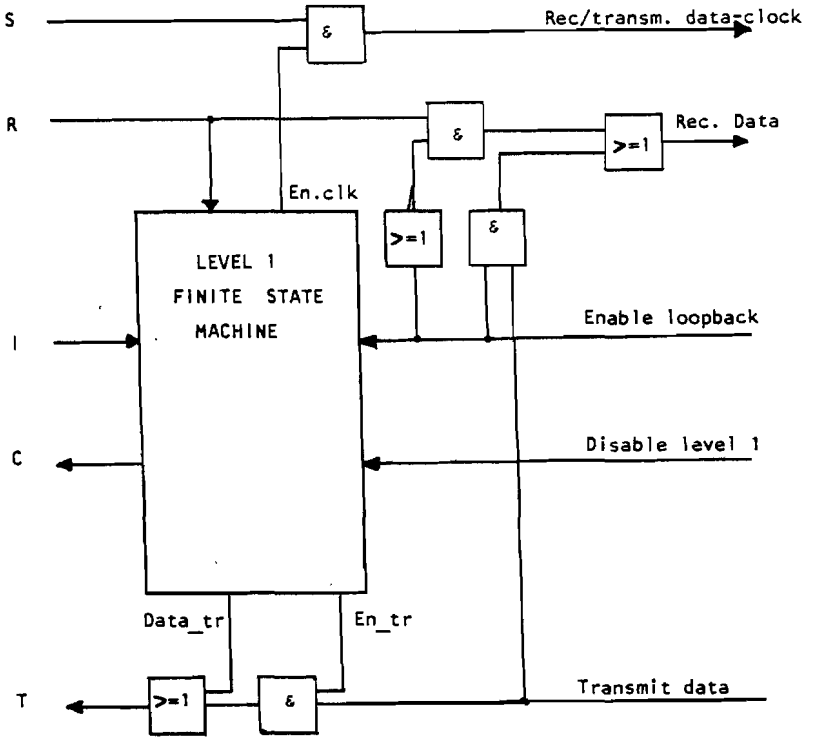
(13) Intel 82586
Reference Manual
Intel corporation, Januari 1983

APPENDIX A: state machine definitions.

Contents

A.1 Level 1 Finite State Machine.

Level 1 can be implemented by means os simple hardware, together with a Finite State Machine. The diagram of that machine can be seen below:



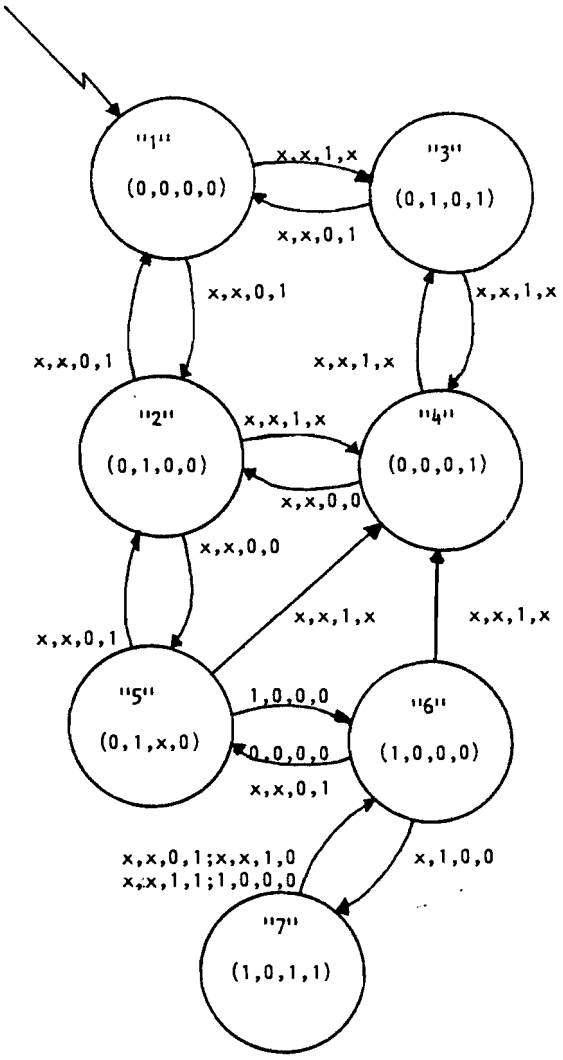
Signal list:

S : Signal element timing on X.25 interface
R : Received data bits (X.25)
I : Indication (X.25)
C : Control (X.25)
T : Transmit (X.25)

Other signals cross the chip internal border between level 1 and level 2.

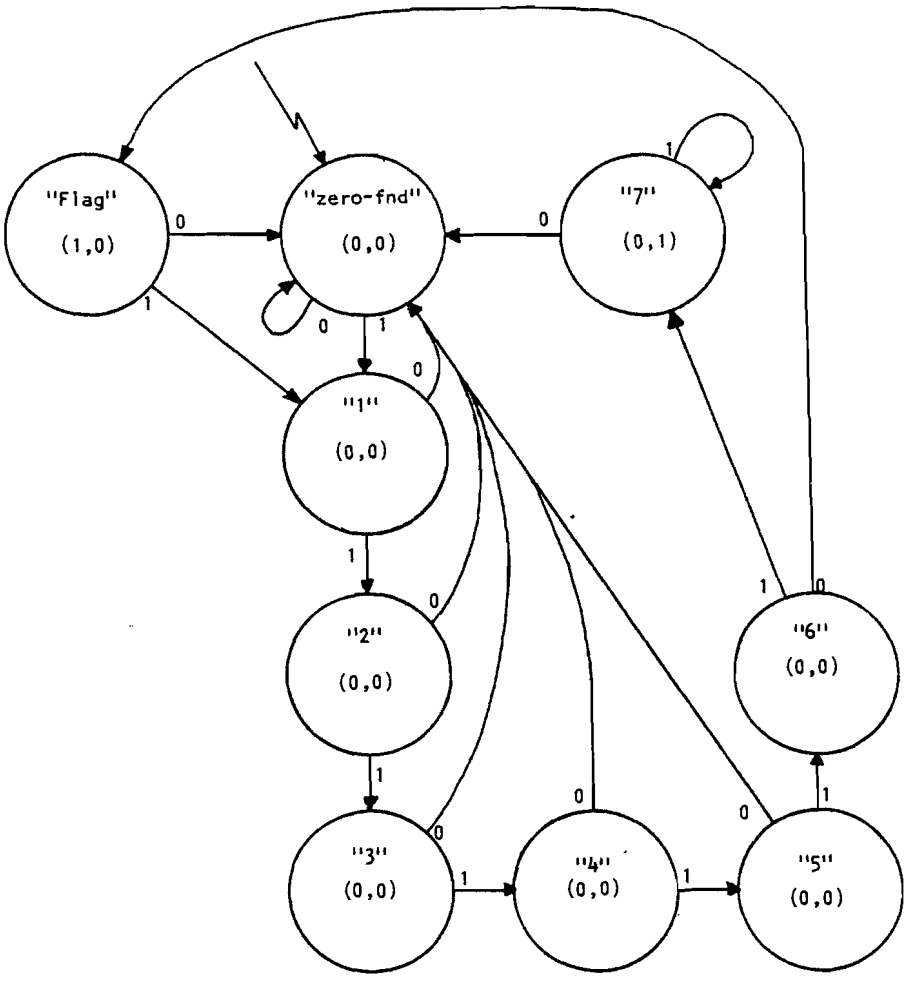
Level 1 Finite State Machine Definition.

Inputs: R,I,Enable_loopback,Disable_level_1
output: C,data_tr,En_tr,En_cl
type: Clocked Mode Moore



A.2 Flag/Abort detection machine.

Inputs: Received bits
Output: Flag,abort
Type: clocked mode Moore

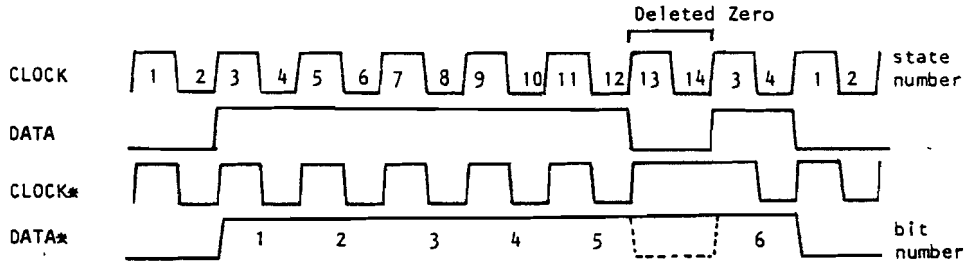


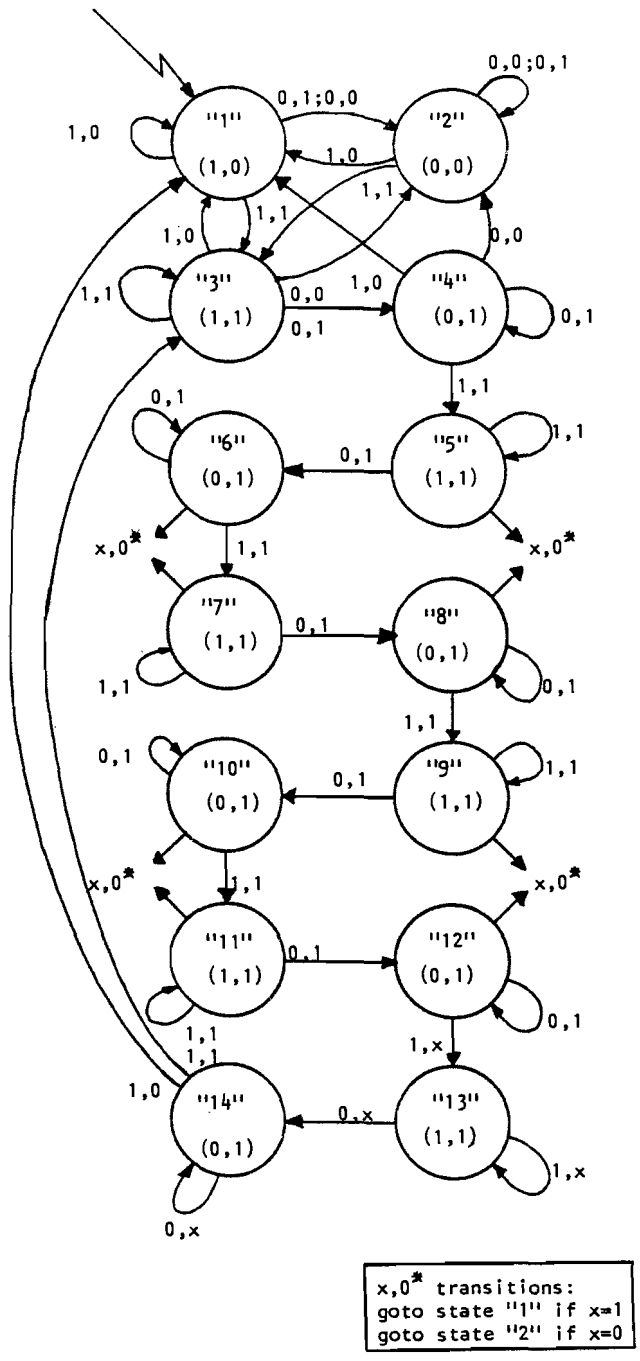
A.3 Zero Deletion Machine.

The timing of this zero deletion machine can be seen in the diagram below. To delete a zero, the clock signal fed to the receiver must skip one transition, where the data signal is not important at that transition.

State machine definition:

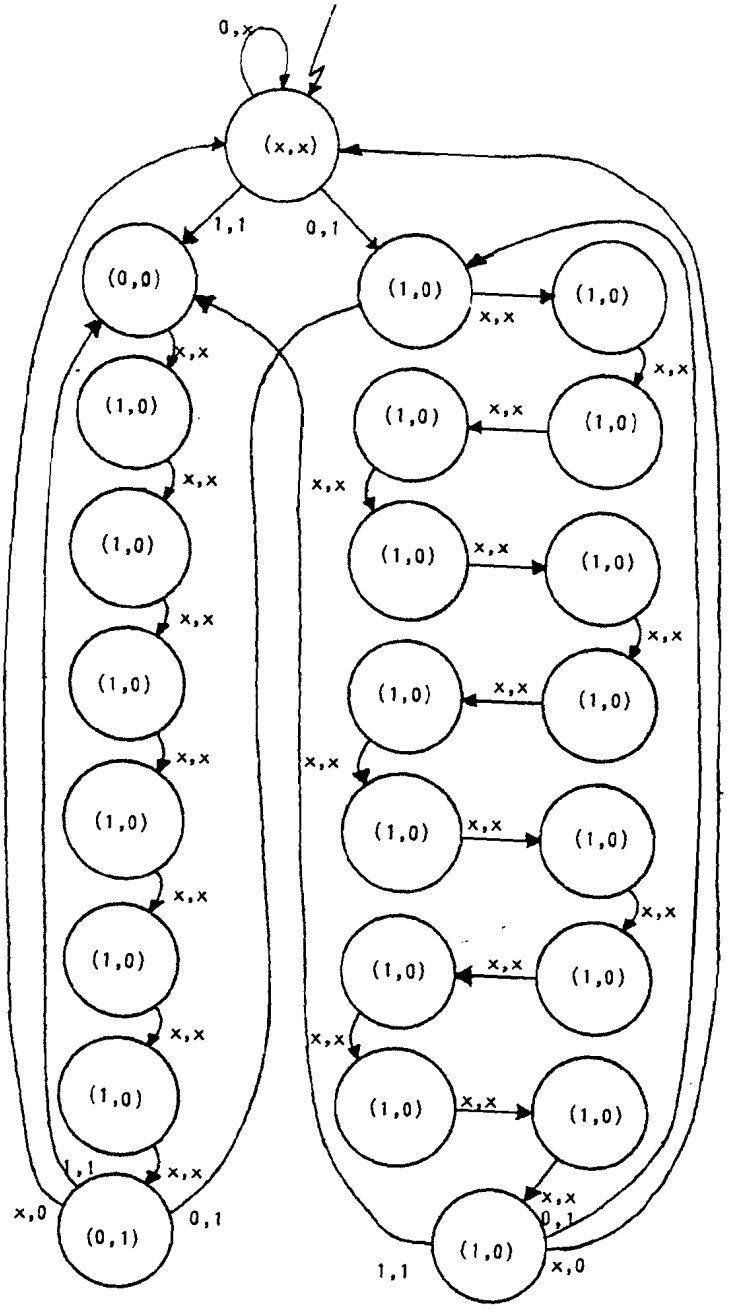
Inputs: Clock,Data
Output: Clock*,Data*
Type: Level Mode Moore




"1" (1,0)
"2" (0,0)
"3" (1,1)
"4" (0,1)
"5" (1,1)
"6" (0,1)
"7" (1,1)
"8" (0,1)
"9" (1,1)
"10" (0,1)
"11" (1,1)
"12" (0,1)
"13" (1,1)
"14" (0,1)
0,0;0,1
0,1;0,0
1,0
1,1
0,0
0,1
x,0*
1,x
0,x
x,0* transitions:
goto state "1" if x=1
goto state "2" if x=0

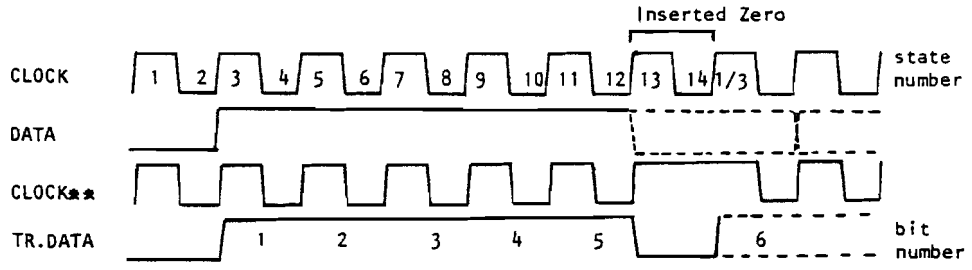
A.4 Flag/Abort generator.

Inputs: Flag/abort, Enable
Output: Flagdata,Sync
Type: Clocked mode Moore



A.5 Zero insertion machine.

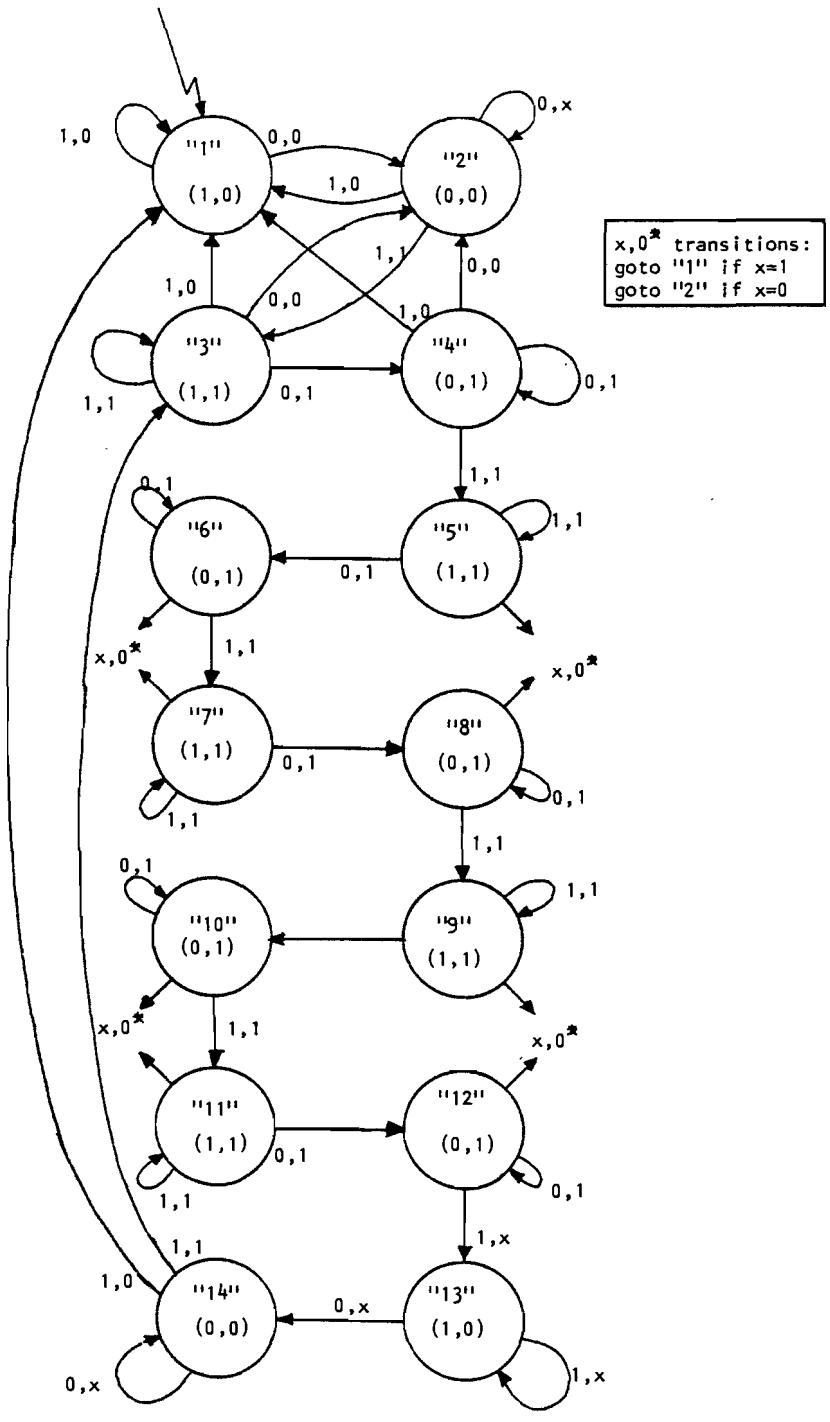
This machine looks very much like the zero deletion machine, but a few minor diferences can be seen in the timing diagram below:



The Clock signal and the Data signal are inputs, and the inserted zero can be realised by means of skipping one transition in the clock signal (clock**) ed to the low level transmitter, and forcing the output TR.data low during that period.

State machine definition:

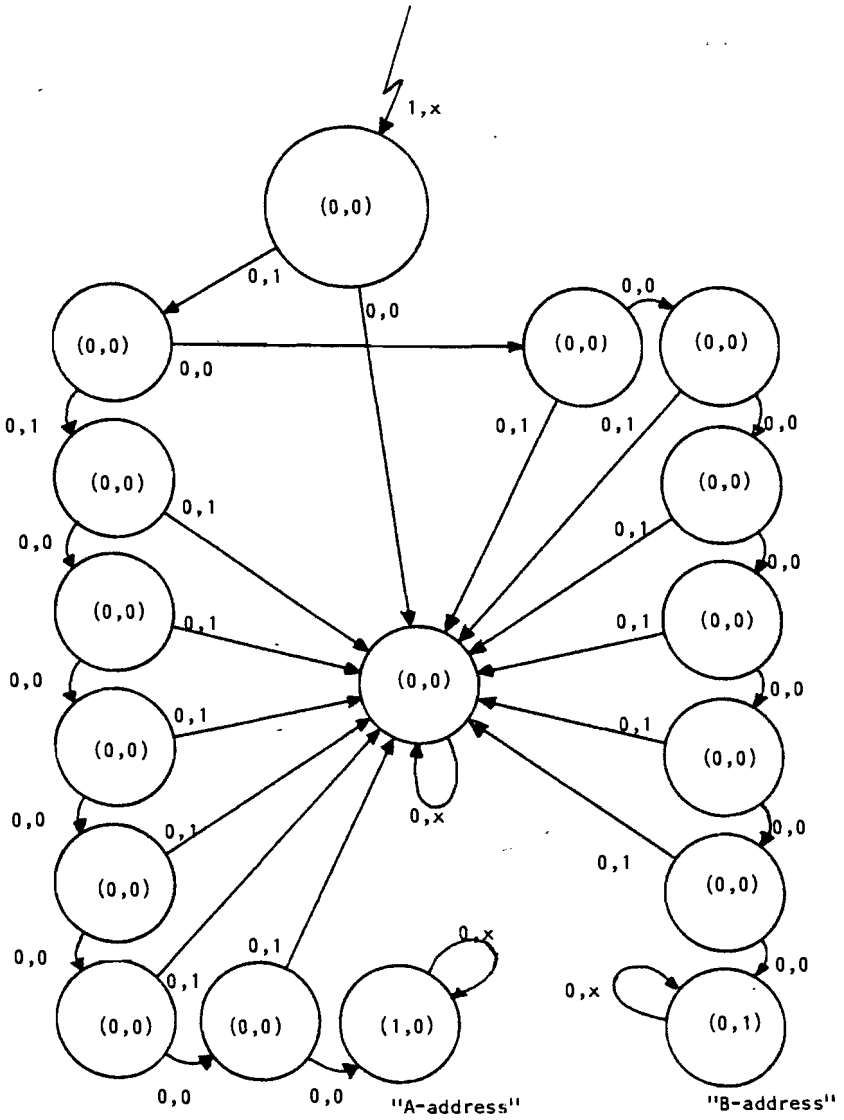
Inputs: Clock,Data
Output: Clock**,TR.data
Type: Level mode Moore


"1" (1,0)
"2" (0,0)
"3" (1,1)
"4" (0,1)
"5" (1,1)
"6" (0,1)
"7" (1,1)
"8" (0,1)
"9" (1,1)
"10" (0,1)
"11" (1,1)
"12" (0,1)
"13" (1,0)
"14" (0,0)
x,0* transitions:
goto "1" if x=1
goto "2" if x=0

A.6 Received Address checker.

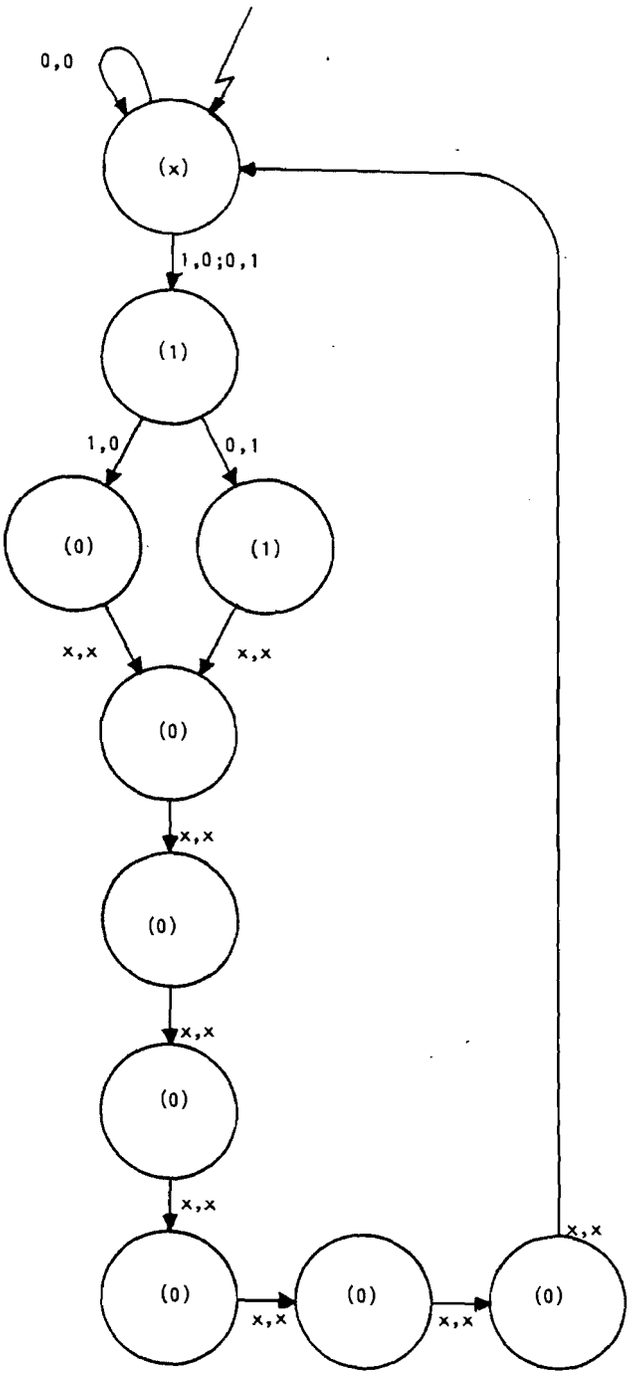
The activation of the reset input causes the machine to go to the power on state.

Inputs: Reset,data
Output: A_addr,B_addr
Type: Clocked Mode Moore



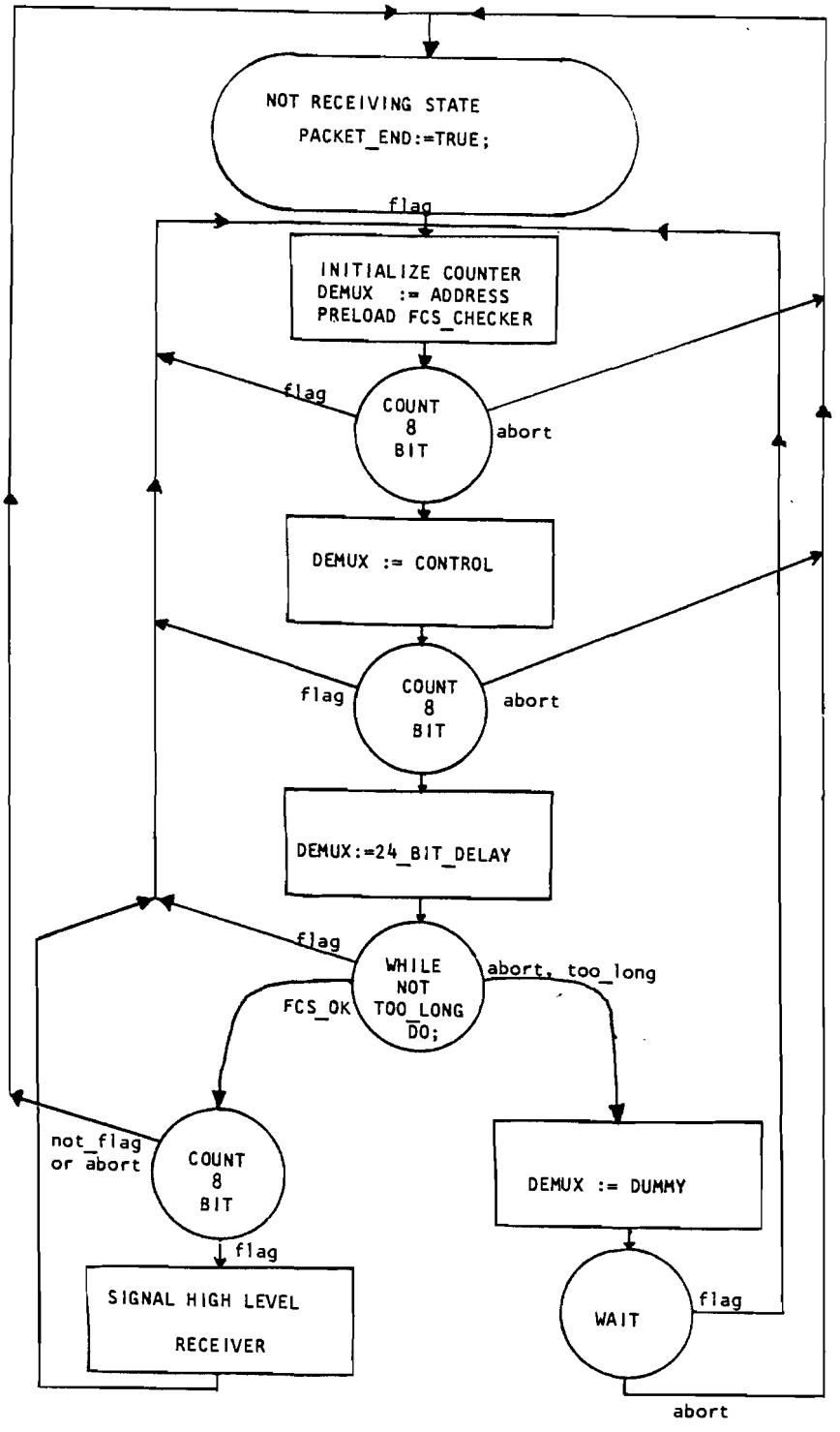
A.7 Address generator.

Inputs: A_addr,B_addr
Output: Data (address data)
Type: Clocked Mode Moore



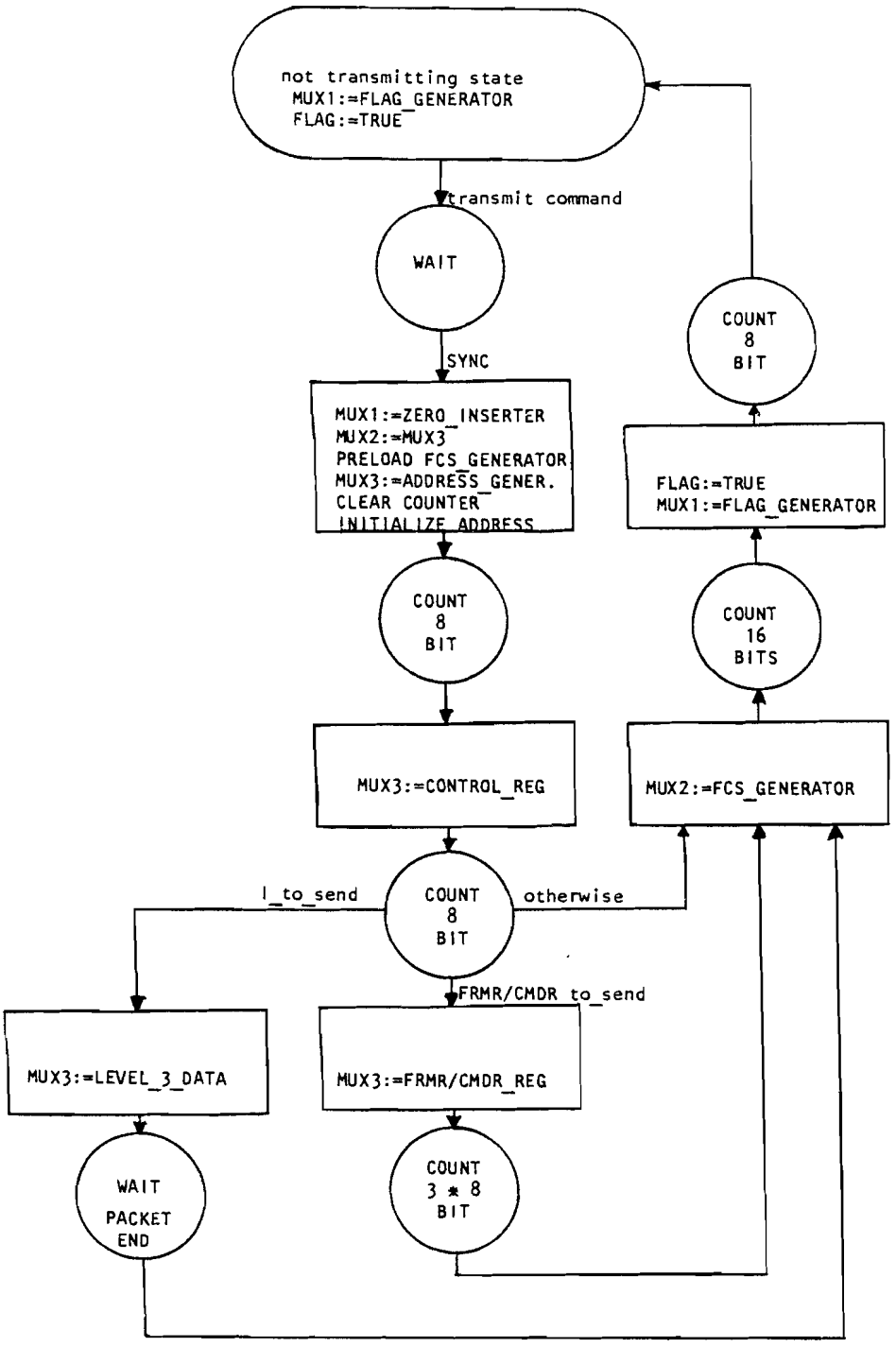
A.8 Low Level Receiver.

The diagram below contains a global description of the level 2 low level receiver process. A square contains assignment statements, and a circle a state.



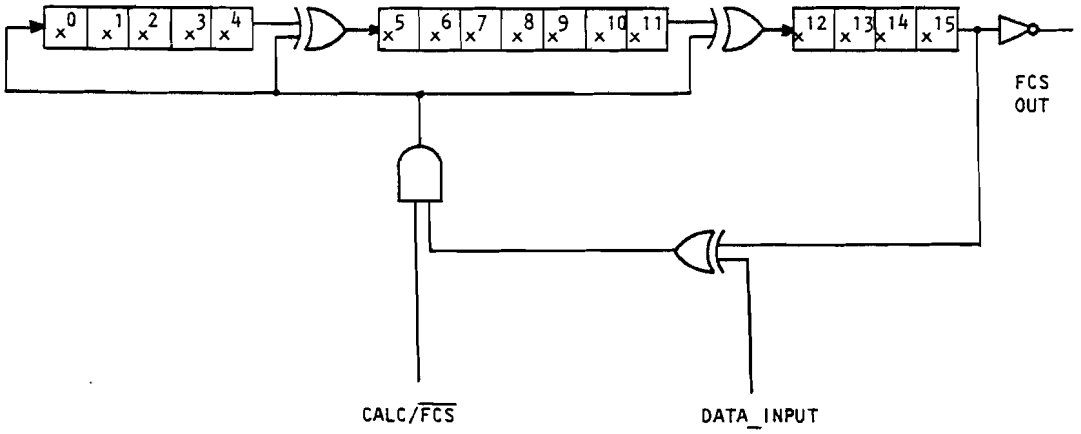
A.9 Low level transmitter.

This diagram contains the actions to be performed by the level 2 low level transmitter. The meaning of the squares and circles is shown in the previous paragraph.



A.10 Frame check sequence checker/generator.

Inputs: Data_in, Calc/FCS, Preload
Output: FCS_out
Type: shift register with feedbacks.



Generator polynomial: $X^{16} + X^{12} + X^{5} + X^{0}$

Before FCS checking or generation, the shift register must be preloaded with all ones (1).

To generate or check the FCS, the calc/$\overline{FCS}$ signal must be active high.

To transmit the FCS itself, the calc/$\overline{FCS}$ signal must be low, there by disabling the feedbacks, and simply shift out the actual shift registers contents, inverted at FCS_out.

Logic to check the residu which means that the check is correct is not shown, which can be a simple PLA checking that the shift register contains: 0001110100001111, where the leftmost bit is the most significant.

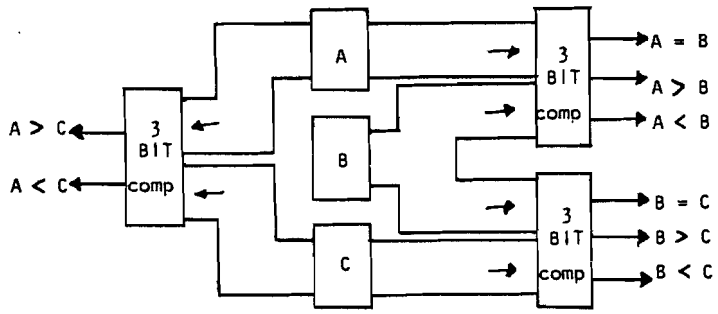
The AND gate with the calc/$\overline{FCS}$ signal may be left out from the FCS checker, together with the FCS_out inverter gate.

A.11 In_betwener.

Three registers contain a 3 bit value A, B and C respectively, where the modulo 8 test A <= B <= C must be performed.

A=low bound
B=value to be tested
C=high bound

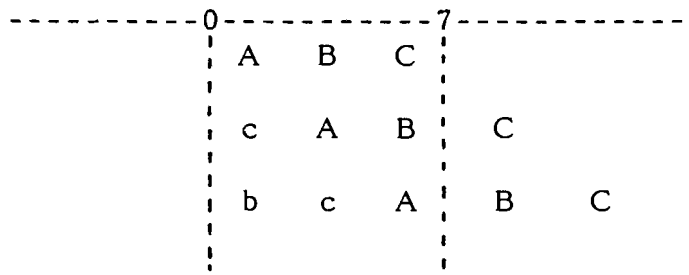
The basis for the inbetwener are three 3 bit comparators, with outputs as shown in the diagram below:



Outputs from these comparators go to a PLA with function:

STATUS = ((A=B) AND (B=C)) OR
((A<B) AND (B<C) AND (A<C)) OR
((A<B) AND (B>C) AND (A>C)) OR
((A>B) AND (B<C) AND (A>C)

This formula can be derived from the pictured below:



Where capitals show the normal relation, and the lower case characters show the same values on a modulo 8 basis.

APPENDIX B

Level 2 and level 3 programs

Contents

B.1 Level 2 Pseudo Pascal Program

The program on the following pages, is a pseudo Pascal program that implements level 2 of the X.25 protocol. All level 2 functions are found in this program, but the current version is not up to date with the final hardware.

This program is used as a base for the level 2 implementation which consists of a number of finite state machines, found in appendix A, and a microprogram that should be run on the level 2 microprogrammable machine. This can be found in appendix B.2.

```
SYSTEM X.25;    (*********************************************
                  This program implements the X.25 protocol.
                  Dedicated hardware performs low-level
                  functions. This program describes func-
                  tions to be performed in additional
                  hardware (sequencers, PLA's), and the micro
                  program for the level 2 microprogrammable
                  machines.
                *********************************************)


TYPE
   modulo8  =  0..7;

VAR             (* these var's are level 2 - 3 interface signals*)
   diagnostics : boolean;
   packet_valid: boolean;
   packet_end  : boolean;
   busy        : boolean;

   last_acknowledged  : modulo8;
   next_packet_to_send: modulo8;
   packet_to_be_sent  : modulo8;


program level_2;

CONST
   A = 11000000;        (* Frame addresses level 2 *)
   B = 01000000;

TYPE
   octet = packed array(1..8) of 0..1;
   frame = (SABM,DISC,FRMR,DM,RR,RNR,REJ,none);


VAR
   frame_rec : semaphore;  (* indication for received frame *)
   add-rec   : semaphore;  (* address field received *)
   contr_rec : semaphore;  (* control field received *)
   transmitter:semaphore;  (* transmitter action semaphore *)

   rec_addr  : octet;
   rec_cont  : octet;

   snd_addr  : octet;
   snd_cont  : octet;

   invalidnr    : boolean;

   state        : (information_transfer,disconnected);
```

```
    event          : (timeout,any_frame,send_packets,stop_I);

    frame_to_send      :frame
    last_frame_sent    :frame;     (* all except an I-Frame *)
    DCE_busy           :boolean;
    command            :boolean;
    V(R)               :modulo8;
    V(S)               :modulo8;
    window             :modulo8;   (* I-Frame to b acknowledged *)
    window_size        :modulo8;   (* host programmable *)
    retrans_count      :integer;
    max_retrans        :integer;   (* host programmable )*
    PF_bit             :boolean;
    P_bit_sent         :boolean;
    rej_sent           :boolean; (*this boolean remembers that a *)
                                   (*  reject has been sent *)


proces receiver;

CONST
  standard = 32;  (* standard non-I non CMDR frame length *)

VAR
  length : integer; (* hardware bitcounter *)


(*************************************************************)

procedure retransmit;

begin

  (* initiate retransmission *)

  if in_between then
  begin
    if N(R)-window <> 0 then retrans_count:=0;
    last_acknowledged:=last_acknowledged + (N(R)-window);
    next_packed_to_send:=last_acknowledged + 1;
    window:=N(R);
    V(S):=N(R);
    retrans_count:=retrans_count+1;
    if retrans_count>max_retrans then
    begin
      signal(level_3);
      stop(level_2);
    end;
  end else
  begin
    invalidnr:=true;
```

```
      reject_frame
    end;
end;

(*************************************************************)

procedure update;

begin

   (* update window;some outstanding frames are acknowledged *)

   if in_between then
   begin
      if N(R)-window <> 0 then retrans_count:=0;
      last_acknowledged:=last_acknowledged + (N(R)-window);
      window:=N(R);
   end else
   begin
      invalidnr:=true;
      reject_frame;
   end;
end;

(*************************************************************)

procedure send(kind:frame);

(* this procedure initiates the transmission of the frame
    KIND. This means that the low level transmitter is given
    that assignment.                                          *)

begin
   frame_to_send:=kind;
   PF_bit:=(rec_cont(5) and rec_addr=A) or PF_bit;
   event:=any_frame;
   signal(transmitter);
end;

(*************************************************************)

procedure abort;
begin

   (* an invalid frame or frame-type is detected
       this means that the cuurent transmitting I
       frame can be aborted, by asserting the
       packet_end signal. The low level transmit-
       ter acts upon that signal.                  *)

   packet_end:=true;   (* stop level 3, invalid packet *)
```

```
end;

(************************************************************)

procedure send_next_I;

(* this procedure initiates the transmission of the next I- *)
(* frame from the queue. *)

begin
  if not DCE_busy                    and
     window<=V(S)<=window+window_size and
     next_packed_to_send =packet_to_be_sent then
    send(I)
end;

(************************************************************)

procedure acknowledge;

(* this procedure acknowledges the correctly received I-
   frame. It is taken into account if receive buffer
   are available, in which case a RR can be sent, or
   piggy-back acknowledge my be used. If not available an
   RNR must be transmitted                                  *)

begin
  if not DCE_busy                    and
     window<=V(S)<=window+window_size and
     last_acknow<=next_packed_to_send<=packet_to_be_sent then
    send(I) else if busy then send(RNR) else send(RR);
end;

(************************************************************)

procedure reject_frame;
begin
  (* an invalid frame with correct FCS is found     *)
  (* the FRMR/CMDR registers are used in this case *)

  initialize FRMR register;
  send(FRMR);
end;

(************************************************************)

procedure start_timer;
begin

  (* this procedure will start the time out timer *)
```

```
   Timer_value:=0;
   timer_running:=true;
end;

(*************************************************************)

procedure stop_timer;
begin

   (* this procedure will stop the time_out timer, but
      will start it again when some frames have not been
      acknowledged. (Outstanding I-frames )                  *)

   timer_running:=false; (* stop it *)
   if V(S)<>window then start_timer;
end;

(*************************************************************)

function in_between:boolean;
begin

   (* this function makes a modulo 8 in between test of *)
   (* Window<= N(R) <= V(S) *)

   in_between:="test"; (* done by special hardware *)
                       (* "in-betwener" *)
end;

(*************************************************************)

procedure increment(var target:modulo8);
begin

   (* this procedure does a modulo8 increment of the target *)
   (* it uses special hardware to do so.*)

   if target=7 then target:=0 else target:=target+1;
end;

(*************************************************************)
(*       R E C E I V E R     P R O C E S       (level 2) *)
(*************************************************************)

begin                                  (* receive proces body *)
while true do
begin
 PF_bit          := false;
 command         := false;
 wait(add_rec);                    (* wait for address field *)
 If rec_addr=A or rec_addr=B then
```

```
begin

wait(frame_rec);                    (* wait for complete frame *)
 if fcs=good and length>= standard then
 begin
  if state=information_transfer then
  begin
   if rec_cont=0xxxxxxx then     (* I-Frame *)
   begin
     if rec_addr=A then           (* must be command *)
     begin
       if N(S) = V(R) then
       begin
         packed_valid:=true;     (* level 3 knows this data is*)
         acknowledge;                    (* correct !!! *)
         increment(V(R));
         rej_sent:=false;        (* reset rejection condition *)
       end else
       begin
         update;
         if not rej_sent then send(REJ);
         rej_sent:=true;          (* SET rejection condition *)
       end;
     end else
     begin
       abort;
       send(FRMR);                  (* I-Frame must be command *)
     end;
   end else
   begin
     if rec_cont=x0xxxxx then (* S-frame *)
      begin
        if length<>standard then
        begin
          too_long:=true;
          reject_frame;
        end else
         case rec_cont(3..4) of
          00 : begin                (* RR *)
                 DCE_busy := false;
                 if rec_addr=A then (* received as command *)
                 begin
                   acknowledge;
                 end else
                 begin             (* received as response*)
                   update;
                   send_next_I;
                   if last_frame_sent=none then stop_timer;
                   if P_bit_sent and rec_cont(5) then
                   begin
                     P_bit_sent:=false;
```

```
                stop_timer;
              end;
            end;
          end;

    10 : begin                (* RNR *)
           DCE_busy := true;
           if rec_addr=A then (* received as command *)
           begin
             if busy then send(RNR) else send(RR);
           end else
           begin              (* received as response*)
            if P_bit_sent and rec_cont(5) then
             begin
               P_bit_sent:=false;
               stop_timer;
             end;
           update;
           if last_frame_sent=none then stop_timer;
         end;

    01 : begin                (* REJ *)
           event:=stop_I;(* abort I-fr.being sent *)
           signal(transmitter);
           DCE_busy:= false;
           if rec_addr=A then (* received as command *)
           begin
             if busy then send(RNR) else send(RR);
           end else
           begin              (* received as response*)
             if P_bit_sent and rec_cont(5) then
             begin
               P_bit_sent:=false
               stop_timer;
             end;
           end;
           retransmit;
         end;

     11 : reject_frame;     (* Undefined S-frame; reject *)

  end; (* case *)
end else
begin                        (* A U-Frame has been received*)
  if length=32 or rec_cont(3..8)=10x001 then
  begin
   If rec_addr = A and last_frame_sent<>rec_cont
      and last_frame_sent<>none then
   begin
     send(DM);             (* just answer with DM response *)
     state:=disconnected; (* COLLISION OF COMMANDS *)
```

```
end;
case rec_cont(3..8) of

 11x100 : if rec_addr = A then
            begin          (* SABM *)
              V(S):=0;
              V(R):=0;
              DCE_busy:=false;
              next_packed_to_send:=last_acknowledged+1;
              retrans_count:=0;
              if Last_frame_sent<>SABM then
                if last_frame_sent= DISC or
                     last_frame_sent= CMDR or
                     last_frame_sent= FRMR then
                      state:=disconnected else send(UA);
            end else reject_frame;

 00x010 : if rec_addr = B then
            begin          (* DISC *)
              state:=disconnected;
              send(UA);
            end else reject_frame;

 10x001 : if rec_addr = B then
            begin          (* CMDR,FRMR *)
              command:=true;
              send(SABM);
            end else reject_frame;

 11x000 : if rec_addr = B then
            begin          (* DM *)
              if P_bit_sent and rec_cont(5) then
              begin
                P_bit_sent := false;
                stop_timer;
              end;
              command:=true;
              send(SABM);
              if last_frame_sent=DISC or
                  last_frame_sent=SABM then
              begin
                last_frame_sent:=none;
                stop_timer;
              end;
            end else reject_frame;

 00x110 : if rec_addr = B then
            begin          (* UA *)
              stop_timer;
              command:=false; (* command acknowledged*)
              retrans_count:=0;
```

```
                        DCE_busy:=false;
                        if last_frame_sent=DISC then
                        begin
                          state:=disconnected;
                          command:=true;
                          send(SABM);
                        end else last_frame_sent:=none;
                        if rec_cont(5) and P_bit_sent then
                                    P_bit_sent:=false else
                        begin
                          command:=true;
                          if busy then send(RNR) else send(RR);
                        end;
                      end else reject_frame;

          otherwise reject_frame;

        end; (*  case *)
      end else
      begin
        too_long:=true;
        reject_frame;
      end;
    end;  (* U-frame *)
 end;(* U-frame or S-frame *)
end (* state=information transfer *) else
begin (* state=disconnected *)
  if rec_addr = A then case rec_cont of

    1111x110 : begin            (* SABM received *)
                  V(S):=0;
                  V(R):=0;
                  DCE_busy:=false;
                  retrans_count:=0;
                  stop_timer;
                  next_packed_to_send:=last_acknowledged + 1
                  state:=information_transfer;
                  if last_frame_sent<>SABM then
                    if last_frame_sent= DISC or
                       last_frame_sent= CMDR or
                       last_frame_sent= FRMR then
                      state:=disconnected else send(UA);
               end;

      1100x010: begin           (* DISC received *)
                   send(DM);
                end;

      otherwise if rec_cont(5) then
                begin
                  PF_bit:=true;
```

```
                    send(DM);(* tell DCE we are disconnected*)
                  end else abort;

       end; (* case *)
      end; (* state=disconnected *)
    end (* FCS=good *) else abort;
  end (* addressfield is A or B *) else abort;(* invalid addr*)
  packet_end:=true;
 end; (* while true do *)
end; (* receive proces *)

(*************************************************************)
(*     T R A N S M I T T E R   P R O C E S  (level 2) *)
(*************************************************************)

proces transmitter;

VAR
    frmr_reg : array(1..24) of 0..1;
    frame_kind:frame;

begin  (* transmitter proces body *)
  while true do
  begin
     wait(transmitter);
     case event of
        timeout: begin
                    if last_frame_sent = none then
                    begin
                       V(S):=window;
                       Next_packed_to_send:=last_acknowledged+1;
                       frame_to_send:=I;
                    end else frame_to_send:=last_frame_sent;
                    retranscount:=retrans_count + 1;
                    if retrans_count>max_retrans then
                    begin
                       signal(level3);
                       stop(level2);
                    end;
                    event:=any_frame;
                    PF_bit:=true;
                    signal(transmitter);
                 end;

       any_frame:begin
                    if command or frame_to_send=I then snd_addr:=B
                                                  else snd_addr:=A;
                    snd_cont:=frame_to_send;
                    if frame_to_send=I then
                    begin
                       snd_cont(2..4):=V(S);
```

```
                    increment(next_packed_to_send);
                    increment(V(S));
                  end;
                  if frame_to_send=I      or
                    frame_to_send=RR      or
                    frame_to_send=RNR     or
                    frame_to_send=REJ  then snd_cont(6..8):=V(R);
                  snd_cont(5):=PF_bit;

                  if frame_to_send=FRMR then
                  begin  (* initialize the FRMR/CMDR register *)
                    frmr_reg(1..8)        :=rec_cont;
                    frmr_reg(9)           :=0;
                    frmr_reg(10..12)      :=V(S);
                    frmr_reg(13)          :=rec_addr=A;
                    frmr_reg(14..16)      :=V(R);
                    frmr_reg(17)          :=aborted;
                    frmr_reg(18)          :=too_long;
                    frmr_reg(19)          :=I_too_long;
                    frmr_reg(20)          :=invalidnr;
                    frmr_reg(21..24)      :=0000;
                    snd_addr:=A;   (* response *)
                  end;
                  frame_kind:=frame=to_send;
                  start_low_level_transmitter;
                  if frame_to_send=I then last_frame_sent:=none
                       else last_frame_sent:=frame_to_send;
                  if command then start_timer;
                  if next_packed_to_send<=packet_to_be_sent
                  then begin
                         event:=any_frame;
                         frame_to_send:=I;
                         signal(transmitter)
                       end;
                  if frame_to_send = I and not timer_running
                  then start_timer;
                end;

      send_packets:begin
                     frame_to_send:=I;
                     event:=any_frame;
                     signal(transmitter);
                   end;

      stop_I:begin
                if last_frame_sent=I and not_finished then
                  start_sending_abort_sequence;
                end;

    end; (* case *)
  end; (* while true do *)
```

```
end;    (* transmitter proces body *)

(*************************************************************)

Begin  (* Level 2 program body *)

  (* initialize the variables *)

  start receiver;
  start transmitter;

end;    (* level 2 program body *)

(*************************************************************)
(*************************************************************)

program level_3;


(*************************************************************)
(*       R E C E I V E R     P R O C E S     (level 3)   *)
(*************************************************************)

proces receiver_3;
begin
end;


(*************************************************************)
(*     T R A N S M I T T E R     P R O C E S    (level 3)
(*************************************************************)

proces transmitter_3;
begin
end;

(*************************************************************)

Begin (* level 3 program body *)
  (* initialize *)
  start receiver_3;
  start transmitter_3;
end;  (* level 3 program body *)

(*************************************************************)
(*************************************************************)
(*************************************************************)

BEGIN  (* system body *)

  start level_3;  (* start up level 3 program *)
```

```
   start level_2;   (* start up level 2 program *)

END.

(* that's all folks !   *)
```

B.2 Level 2 microprogram

The following pages contain the microprograms that run on the level 2 microprogrammable machines. This program is not written in a micro assembler syntax, but in a more readable pascal like form. Multiple lines in this program may form one micro-assembler line.

The sharing and command passing to the transmitter will be described in the following paragraph.

The sharing of the registers needed for passing the frame to be send to the transmitter (Frame_to_send registers), is done via a semaphore, so that it can be shared between the receiver and the transmitter process. When the receiver fills the register with a frame to be sent, the transmitter process must be awaked via setting of the Flip-Flop 'transmitter'. As soon as the transmitter has used these registers, it will clear 'transmitter'. The receiver process must never write to these registers, even if the semaphore allows it, when 'transmitter' is high, because that means that the transmitter process has not yet processed the previous command. To wait until the 'transmitter' is low, the 'call w' mnemonic is inserted.

The transmitter process will copy the 'frame_to_send' register into the control field register of the actual frame to be sent next, by the low level transmitter. Of course the transmitter process must wait for the low level transmitter to be ready with a frame, before the control field register can be accessed. Again a handshaking mechanism is used for that purpose, using 'start_low_level_transmitter' and the 'low_level transmit_ready' signal.

That mechanism allows for pipelining, so that no process (be it transmitter or receiver) is blocked via a semaphore unnecessarily. Because of the structure of the level 2 protocol, no stagnation of the receiver process should occur. Only one frame may be sent normally as a response to a non I-frame.

```
none:           equ 0
none1:          equ 0
none2:          equ 0
I:              equ 0

;*****************************************************
;* This subroutine initializes the in betwener regis-
;* ters, and inputs the status from that machine to
;* the microprogrammable machine status test line.
;* The in_betwener will be requested (semaphore wait)
;*****************************************************
sub1:           call getbetw            ;request betwener
                betwener(1):=window
                betwener(2):=N(R)
                betwener(3):=V(S)
                select:=in_between      ;input select
                ret

;*****************************************************
;* This routine requests the in_betwener register
;* block. The in_betw_sem semaphore is used for this
;* this sharing purpose.
;*****************************************************
getbetw:        select:=in_betw_sem
gbwait:         pulse(in_betw_sem)
                jtrue gbwait
                ret

;*****************************************************
;* This subroutine checks if any frames are still
;* outstanding, and updates the outstanding frames
;* that have been acknowledged.
;* The in betwener must be requested before calling
;* this routine.
;*****************************************************
sub2:           accu:=N(R)
                accu:=accu - window
                jz rtr0
                initialize_retranscounter
rtr0:           accu:=accu + last_acknowledged
                last_acknowledged:=accu
                ret


;*****************************************************
;* this routine initiates retransmission
;* The in_betwener semaphore is requested / released
;*****************************************************
retransmit:     call sub1
                jfalse rtr1
               call sub2
```

```
                    accu:=accu+1;
                    next_packet_to_send:=accu
                    window:=N(R)
                    V(S):=N(R)
                    clear(in_betw_sem)
                    pulse(retranscounter)
                    select:=max_retrans
                    jfalse rtr2
                    call getbetw
                    set(command)
                    clear(betw_sem)
                    call w
                    frame_to_send(1):=SABM1
                    frame=to_send(2):=SABM2
                    call send    ;maximum number of retransmissions
                    retranscounter:=0
rtr2:               jmp rtrx
rtr1:               clear(In_betw_sem)        ;release in betwener
                    set(invalid_nr)
                    call reject_frame
rtrx:               ret

;*****************************************************
;* This routine waits until the transmitter is ready
;* and must be called before writing to any register
;* that is needed to pass a frame to send to the
;* transmitter process.
;*****************************************************
w:                  select:=transmitter
                    jtrue w
                    ret


;*****************************************************
;* this routine updates the sequence numbers on re-
;* ception of an I-frame.
;* The in_betwener semaphore is requested / released
;*****************************************************
update:             call sub1
                    jfalse ud2
                    call sub2
                    window:=N(R)
                    clear(in_betw_sem)
                    jmp udx
ud2:                clear(in_betw_sem)
                    set(invalid_nr)
                    call reject_frame
udx:                ret
```

```
;*****************************************************
;* this routine initiates the transmitter process
;* to send a frame.
;* If  P bit was set or A address or rec_P bit was 1
;* then the  F bit will be set.
;* IF the transmitter is not ready, this routine will
;* wait before giving the command.
;*****************************************************
send:               select:=PF_bit
                    jtrue se2                ;if P/F bit was set
                    select:=rec_cont_5       ;then do nothing
                    jfalse se2
                    select:=rec_adr_A
                    jfalse se2
                    set(PF_bit)
se2:                select:=transmitter
wait_transm:        jtrue wait_transm        ;wait until last command
                    set(Transmitter)         ;has been completed
                    ret

;*****************************************************
;* In case an abort is received, level 3 must be no-
;* tified of the packet end, where packet is nonvalid
;*****************************************************
abort:              pulse(packet_end);       ;notify level3
                    ret

;*****************************************************
;*  check:  window <= V(S) <= window + window_size
;* The in_betwener will be requested (In_betw_sem).
;*****************************************************
sub3:               call getbetw
                    betwener(1):=window      ;check if inside window
                    betwener(2):=V(S)
                    accu:=window
                    accu:=accu + window_size
                    betwener(3):=accu
                    select:=in_between
                    ret

;*****************************************************
;*    check:
;* last_acknow <= next_to_send <= to_be_sent
;*  status select will not be changed by this routine
;* THE IN_betwener must be requested before calling
;*****************************************************
sub4:               betwener(1):=last_acknowledged
                    betwener(2):=next_packet_to_send
                    betwener(3):=packet_to_be_sent
                    ret
```

```
;*****************************************************
;* this routine sends off next to send I-frame, if
;* within window
;*****************************************************
send_next_I:        call sub3                  ;do in between test
                    jfalse snex                ;no more frames to do?
                    call sub4
                    jfalse snex                ;in betwener was select
                    clear(In_betw_sem)         ;release in betwener
                    select:=DCE_busy
                    jfalse snex2
                    set(I_to_send)
                    call send
                    jmp snex2
snex:               clear(In_betw_sem)         ;release in betwener
snex2:              ret


;*****************************************************
;* This routine initiates the acknowledgement of
;* received I-frames
;*****************************************************
acknowledge:        call sub3
                    jfalse nack                ;is there room left ?
                    call sub4
                    jfalse nack                ;in_between was select
                    clear(in_betw_sem)
                    select:=DCE_busy
                    jfalse nack2
                    call getreg
                    set(I_to_send)             ;piggy back acknowledge
                    call send
                    jmp acex
nack:                clear(in_betw_sem)
nack2:               call rr_rnr                ;normal acknowledge
acex:               ret


;*****************************************************
;* this routine sends a RR if receive room left, and
;* a RNR if no receive room was left
;* the getreg_sem is requested and released
;*****************************************************
rr_rnr:             call getreg
                    select:=busy               ;busy yourself ?
                    jfalse ack2
                    call w
                    frame_to_send(1):=RNR      ;load immediate
                    call send
                    jmp rcex
ack2:               call w
```

```
                    frame_to_send(1):=RR      ;load immediate
                    call send
rcex:               clear(getreg_sem)
                    ret


;*****************************************************
;* A frame reject or command reject frame will be
;* sent by this routine
;* The getreg semaphore is requested and released
;*****************************************************
reject_frame:       call getreg               ;request register
                    call w
                    frame_to_send(1):=FRMR1 ;load immediate
                    frame_to_send(2):=FRMR2
                    clear(getreg_sem)         ;release getreg_sem
                    call getcmdr               ;request cmdr
                    frmr(1):=N(S)
                    frmr(2):=N(R)
                    frmr(3):=V(S)
                    frmr(4):=V(R)
                    frmr(5):=wxyz
                    clear(getcmdr_sem)
                    call send
                    ret


;*************************************************************
;* This routine waits for the release of Frame to send reg's.
;*      Protected by a hardware semaphore.
;*************************************************************
getreg:             select:=getreg_sem        ;select semaphore
grwait:             pulse(getreg_sem)         ;obtain it
                    jtrue grwait               ;wait until successful
                    ret

;*****************************************************
;* This routine requests the FRMR register.
;*****************************************************
getcmdr             select:=getcmdr_sem
gcwait:             pulse(getcmdr_sem)
                    jtrue gcwait
                    ret

;*****************************************************
;* The level 2 timer will be started by this routine *
;*****************************************************
start_timer:        pulse(initialize_timer)
                    set(timer_running)
                    ret
```

```
;*****************************************************
;* The timer will be stopped. If any I-frames are
;* still outstanding, it will be restarted
;*****************************************************
stop_timer:         clear(timer_running)
                    accu:=window
                    accu:=accu - V(S)          ;I-frames outstanding?
                    jz stex
                    call start_timer
stex:               ret



;*****************************************************
;* M A I N   R E C E I V E R   P R O C E S S
;*
;*****************************************************
receiver:           clear(PF_bit)
                    call getreg
                    clear(command)
                    clear(reg_sem)
r0:                 select:=add_rec             ;wait for adress field
w1:                 jfalse w1
                    select:=adres_A
                    jtrue r2
                    select:=adres_B
                    jfalse r0
r2:                 select:=frame_rec          ;wait for complete frame
w2:                 jfalse w2
                    select:=fcs_good           ;FCS is OKAY ??
                    jfalse receiver
                    select:=smaller32
                    jfalse receiver             ;frame too short
                    select:=state
                    jfalse no_inf_transf
                    select:=rec_cont_1
                    jtrue no_I_frame             ;I frame ?
                    select:=adres_A              ;I frame received
                    jfalse invalid
                    accu:=N(S)
                    accu:=accu-V(R)
                    jnz retrans
                    select:=greater_max        ;too_long in fig 21
                    jfalse Iokee
                    set(I_too_long)
                    call reject_frame           ;I-field too long
                    call update
                    jmp okee
Iokee:              pulse(packet_valid)
                    call acknowledge
                    accu:=V(R)
```

```
                accu:=accu + 1
                call getbetw
                V(R):=accu
                clear(in_betw_sem)      ;release register
                clear(rej_sent)
                jmp okee
retrans:        call update
                select:=rej_sent
                jtrue no_more           ;REJ already sent
                call getreg
                frame_to_send:=REJ      ;load immediate
                clear(getreg_sem)       ;release semaphore
                call send
                set(rej_sent)
no_more:         jmp okee
invalid:        call abort              ;must be command
                call reject_frame       ;reject this one
                jmp okee
no_I_frame:     accu:=N(S)              ;test rec.control bit
                test_bit:=1             ;S frame received
                jnz u_frame             ;'1' means U-frame
                select:=equal32
                jtrue U_valid
                set(too_long)           ;incorrect length
                call reject_frame
                jmp okee
U_valid:        test_bit:=2
                jz first_nil
                test_bit:=3
                jz U_RNR
                call reject_frame
                jmp okee
first_nil:      test_bit:=3
                jz U_RR
                call getreg
                call w
                set(AB_seq)             ;REJ received
                set(transmitter)        ;ABORT !!!
                clear(DCE_busy)
                select:=adres_A
                clear(getreg_sem)
                jfalse rej_b
                call rr_rnr             ;send RR or RNR
                jmp rej_1
rej_b:          select:=P_bit_sent
                jfalse rej_1
                select:=rec_cont_5
                jfalse rej_1
                clear(P_bit_sent)       ;reset P bit-sent
                call stop_timer         ;positive acknowledge
rej_1:          call retransmit
```

```
                jmp okee                    ;frame process ended

U_RR:           clear(DCE_busy)             ;no more busy
                select:=adres_A
                jfalse u_rr_b
                call acknowledge
                jmp okee
u_rr_b:         call update
                call send_next_i
                accu:=none                  ;load immediate
                accu:=accu - last_frame_sent
                jnz u_rr_b1
                call stop_timer
u_rr_b1:        select:=P_bit_sent
                jfalse okee
                select:=rec_cont_5
                jfalse okee
                clear(P_bit_sent)
                call stop_timer
                jmp okee

U_RNR:          set(DCE_busy)               ;DCE is busy
                select:=adres_A
                jfalse U_RNR_b
                call rr_rnr                 ;send RR or RNR
                jmp U_rnr_ex
U_rnr_b:        select:=P_bit_sent
                jfalse U_rnr_u
                select:=rec_cont_5
                jfalse U_rnr_u
                clear(P_bit_sent)
                call stop_timer
u_rnr_u:        call update
u_rnr_ex:       accu:=none                  ;load immediate
                accu:=accu-last_frame_sent
                jnz okee
                call stop_timer
                jmp okee

S_frame:        select:=equal32             ;length frame correct ?
                jtrue lokee                 ;this is a U frame
                accu:=N(S)                  ;check if type = FRMR
                accu:=accu - frmr1
                jnz wrong
                accu:=N(R)                  ;check second half
                accu:=accu - frmr2
                jz lokee
wrong:          call reject_frame           ;incorrect frame
                jmp okee
lokee:          select:=adres_A             ;A_address found ?
                jfalse handle               ;no command
```

```
                accu:=last_frame_sent(1)
                accu:=accu-none1
                jnz outstand
                accu:=last_frame_sent(2)
                accu:=accu-none2
                jz handle
outstand:       accu:=last_frame_sent(1)
                accu:=accu - N(S)
                jnz collision
                accu:=last_frame_sent(2)
                jz handle
collision:      call getreg
                frame_to_send:=DM          ;load immediate
                clear(getreg_sem)
                call send
                clear(state)               ;enter disconnected sta
                jmp okee
handle:         accu:=N(S)                  ;case rec_cont(3..8)
                test_bit:=2
                jnz fnd1
                test_bit:=3
                jnz fnd01
fnd00:          accu:=N(R)                  ;fetch second half of
                test_bit:=1                ;field
                jnz fnd00x1
                test_bit:=2
                jnz fnd00x01
fnd00x000:      call reject_frame          ;exit of case statement
                jmp okee
fnd1:           test_bit:=3
                jnz fnd11
fnd10:          accu:=N(R)                  ;fetch second half
                accu:=accu - 001           ;test immediate
                jnz fnd00x000              ;if not--3 reject
                select:=adres_B
                jfalse fnd00x000
                call getreg
                set(command)
                frame_to_send:=SABM        ;load immediate
                clear(getreg_sem)
                call send
                jmp okee
fnd00x1:        test_bit:=2
                jz fnd00x000               ;reject
                test_bit:=3                ;here 00x11
                jnz fnd00x000
fnd00x110:      select:=adres_B
                jfalse fnd00x000           ;must be command
                call stop_timer
                call getreg
                clear(command)
```

```
                clear(reg_sem)
                pulse(initialize_retranscounter)
                clear(DCE_busy)
                accu:=last_frame_sent(1)
                accu:=accu - DISC1
                jnz none
                accu:=last_frame_sent(2)
                accu:=accu - DISC2          ;again immediate
                jnz none
                clear(state)                ;disc received,
                call getreg                  ;state:=disconnected
                set(command)
                frame_to_send:=SABM
                clear(getreg_sem)
                call send
                jmp ack_chk
none:           clear(all_acknowledged)
ack_chk:        select:=rec_cont_5
                jfalse okee
                select:=P_bit_sent
                jfalse okee
                call getreg
                set(command)
                clear(getreg_sem)
                call rr_rnr                  ;send RR or RNR
                jmp okee

fnd00x01:       test_bit:=3
                jnz fnd00x000                ;if zero--> reject
                select:=adres_B
                jfalse fnd00x000            ;must be command
                clear(state)                ;DISC received
                call getreg
                frame_to_send:=UA
                clear(getreg_sem)
                call send
                jmp okee

fnd11:          accu:=N(R)                   ;fetch second part
                test_bit:=1
                jz fnd11x0
                test_bit:=2
                jnz fnd00x000
                test_bit:=3
                jnz fnd00x000                ;incorrect, reject
                select:=adres_A             ;SABM, must be command
                jfalse fnd00x000
                call getbetw
                V(S):=0                      ;load immediate
                V(R):=0                      ;load immediate
                clear(DCE_busy)
```

```
                    accu:=last_acknowledged
                    accu:=accu + 1
                    next_packet_to_send:=accu
                    clear(in_betw_sem)
                    initialize_retranscounter
                    accu:=last_frame_sent(1)
                    accu:=accu - sabm1          ;test immediate
                    jnz check
                    accu:=last_acknowledged(2)
                    accu:=accu - sabm2
                    jz okee
check:              accu:=last_frame_sent(1)
                    accu:=accu - disc1
                    jnz ch_go_on
                    accu:=last_frame_sent(2)
                    accu:=accu - disc2
                    jz jammer
ch_go_on:           accu:=last_frame_sent(1)
                    accu:=accu - frmr1
                    jnz sendua
                    accu:=last_frame_sent(2)
                    accu:=accu - frmr2
                    jnz sendua
jammer:             clear(state)                ;collision
                    jmp okee
sendua:             call w
                    frame_to_send(1):=ua
                    call send
                    jmp okee

no_inf_transf:      select:=adres_A             ;here disconnected
                    jfalse okee
                    select:=rec_cont_1
                    jtrue otherwise
                    accu:=N(S)
                    accu:=accu - sabm1
                    jnz next_ni
                    accu:=N(R)
                    accu:=accu - sabm2
                    jnz next_ni
                    call getbetw
                    V(S):=0                      ;SABM received
                    V(R):=0                      ;load immediate
                    clear(DCE_busy)
                    initialize_retranscounter
                    call stop_timer
                    accu:=last_acknowledged
                    accu:=accu + 1
                    next_packet_to_send:=accu
                    set(state)
                    clear(in_betw_sem)
```

```
                    accu:=last_frame_sent(1)
                    accu:=accu - sabm1
                    jnz s_1
                    accu:=last_frame_sent(2)
                    accu:=accu - sabm2
                    jz okee
s_1:                accu:=last_frame_sent(1)
                    accu:=accu - disc1
                    jnz s_2
                    accu:=last_frame_sent(2)
                    accu:=accu - disc2
                    jz s_3
s_2:                accu:=last_frame_sent(1)
                    accu:=accu - frmr1
                    jnz s_4
s_3:                accu:=last_frame_sent(2)
                    accu:=accu - frmr2
                    jnz s_4
                    clear(state)
                    jmp okee
s_4:                call w
                    frame_to_send(1):=UA
                    call send
                    jmp okee

next_ni:            accu:=N(S)
                    accu:=accu - disc1
                    jnz otherwise
                    accu:=N(R)
                    accu:=accu - disc2
                    jnz otherwise
                    call w
                    call getreg
                    frame_to_send(1):=DM1
                    frame_to_send(2):=DM2
                    clear(getreg_sem)
                    call send
                    jmp okee
otherwise:          select:=rec_cont_5
                    jfalse okee
                    call w
                    call getreg
                    set(PF_bit)
                    frame_to_send(1):=DM1
                    frame_to_send(2):=DM2
                    clear(getreg_sem)
                    call send
okee:               pulse(packet_end)          ;signal level3
                    jmp receiver
```

```
;*************************************************************
;*      T R A N S M I T T E R      P R O C E S S      *
;*************************************************************
tmineen:            clear(in_betw)              ;release in betwener
TRANSMIT:           select:=transmitter       ;check if a frame has
                                                    ;to be sent
                    jtrue any_frame
                    select:=time_out
                    jtrue timeout
                    select:=ab_seq              ;abort sequence to do?
                    jtrue do_abort
                    select:=in_betw_sem         ;share in betwener
twait:              pulse(in_betw_sem)
                    jtrue twait                  ;busy form of waiting
                    accu:=last_acknowledged
                    betw(1):=accu
                    accu:=next_P_to_send
                    betw(2):=accu
                    accu:=to_be_sent
                    betw(3):=accu
                    select:=in_betw             ;get status
                    jfalse tmineen              ;if not, poll again
                    clear(in_betw)              ;release in betwener
                    select:=reg_sem
twait2:             pulse(reg_sem)
                    jtrue twait2                 ;wait on reg (sharing)
                    Next_to_send(1):=I
                    clear(reg_sem)              ;release it

;*************************************************************
; This part of the transmit program sends a frame of any type
; it does not send until the last frame ha been sent.
;*************************************************************
any_frame:          select:=low_level_tr_ready
                    jfalse transmit           ;wait until last frame
                                                    ;has been sent.
                    select:=command            ;command has to be sent
                    jtrue do_b                   ;then B address
                    accu:=frame_to_send(1)
                    accu:=accu-I                ;I frame to be sent ?
                    jz do_b
                    set(send_A_addr)            ;no, set A address
                    jmp skip_do
do_b:               set(send_B_addr)
skip_do:          accu:=frame_to_send(1)
                    send_contr(2..4):=accu
                    accu:=frame_to_send(2)
                    send_contr(6..8):=accu
;the first bit of send_contr (send_contr(1)) is automatically
;set by a FF which contains the flag 'I_to_send', and bit 5 is
;also set by the 'p_to_send' FF.
```

```
                    accu:=frame_to_send(1)
                    accu:=accu-I                ;I frame to be sent ?
                    jfalse skipVS
                    accu:=V(S)
                    send_contr(2..4):=accu
                    select:=in_betw
twait3:             pulse(in_betw_sem)
                    jtrue twait3
                    accu:=accu+1
                    v(s):=accu
                    accu:=next_P_to_send
                    accu:=accu+1
                    next_P_to_send:=accu
                    clear(in_betw_sem)          ;release again
                    jmp do_vr
skipvs:             accu:=frame_to_send(1)
                    accu:=accu-RR               ;is it a RR frame ?
                    jz do_vr
                    accu:=frame_to_send(2)
                    accu:=accu-RNR
                    jz do_vr
                    accu:=frame_to_send(1)
                    accu:=accu-REJ
                    jfalse skipvr
do_vr:              accu:=V(R)
                    send_contr(6..8):=accu ;init acknowledge number

skipvr:             accu:=frame_to_send(1)
                    accu:=accu-FRMR1
                    jnz no_frmr
                    accu:=frame_to_send(2)
                    accu:=accu-FRMR2
                    jnz no_frmr
                    select:=frmr_sem
twait4:             pulse(frmr_sem)
                    jtrue twait4                ;wait on semaphore
                    accu:=rec_contr(2..4)
                    frmr(2..4):=accu
;bit 1 of FRMR is loaded automatically with bit 1 of rec_contr
                    accu:=rec_contr(6..8)
                    frmr(6..8):=accu
;bit 5 of frmr is automatically loaded with bit 5 of rec_contr
                    accu:=V(S)
                    frmr(10..12):=accu
;bit 9 is automatically set to zero (as defined by X.25)
                    accu:=V(R)
                    frmr(14..16):=accu
;bit 13 is set to the A_address value
                    pulse(frmr_init)
;bits 17, 18, 19 and 20 are set to
;                   -aborted
```

```
;                   -too_long
;                   -I_too_long
;                   -invalid_nr
;respectively, by the frmr_init signal
;rest of the frmr register is set to zero (bits 21..24)

                    clear(frmr_sem)           ;release this register
                    set(snd_A_addr)
                    set(FRMR_to_send)         ;tell low level transm
no_frmr:            pulse(LOW_LEVEL_TRANSMITTER)
                    ;the low level transmitter is started

                    select:=reg_sem
twait5:             pulse(reg_sem)
                    jtrue twait5              ;request registers
                    select:=command
                    jfalse do_none
                    accu:=frame_to_send(1)  ;get type of frame
                    accu:=accu-I              ;I frame sent ?
                    set(timer_running)        ;start timer
                    jnz skip_I
do_none:            last_frame_sent(1):=none1 ;remember last
                    last_frame_sent(2):=none2 ;command frame !
                    jmp test_cmd
skip_I:             accu:=frame_to_send(1)
                    last_frame_sent(1):=accu
                    accu:=frame_to_send(2)
                    last_frame_sent(2):=accu;remember frame type
test_cmd:           clear(reg_sem)            ;release semaphore
                    clear(transmitter)        ;transmit program ready
                    jmp transmit              ;transmitter started


timeout:            accu:=last_frame_sent(1)
                    accu:=accu-none1
                    jnz elsenone
                    accu:=last_frame_send(2)
                    accu:=accu-none2
                    jnz elsenone
;here a timeout on an I frame is detected, which means that
;retransmission must be initiated
                    select:=in_betw_sem
twait6:             pulse (in_betw_sem)
                    jtrue twait6
                    accu:=window
                    v(S):=accu
                    clear(in_betw_sem)
                    select:=reg_sem
twait7:             pulse(reg_sem)
                    jtrue twait7
                    accu:=last_acknowledged
```

```
;last_acknowledged contains last_acknowledged packet + 1
                    next_P_to_send:=accu
                    clear(reg_sem)
                    jmp inc_retrsn
;here a time out on a command frame with poll/final bit is
;detected
elsenone:           select:=reg_sem
twait8:             pulse(reg_sem)
                    jtrue twait8
                    accu:=last_frame_sent(1)
                    frame_to_send(1):=accu
                    accu:=last_frame_sent(2)
                    frame_to_send(2):=accu
                    clear(reg_sem)
inc_retrns:         pulse(retranscounter)
                    select:=max_retrans
                    jfalse transmit
;here the maximum number of retransmissions has been reached,
;a DISC must be made.

                    select:=reg_sem
twait9:             pulse(reg_sem)
                    jtrue twait9
                    frame_to_send(1):=DISC1
                    frame_to_send(2):=DISC2
                    clear(reg_sem)
                    jmp transmit

do_abort:           accu:=frame_to_send
                    jnz transmit              ;abort I-frames only
                    pulse(trans_abort)        ;activate low level
                                                   ;abort line
                    jmp transmit


;************************************************************
```

SIGNALS from finite state machines.

adr_rec	- address field received
frm_rec	- frame received
adres_A	- A address detected
adres_B	- B address detected
Smaller32	- length frame smaller than 32 bits
equal32	- length frame equal to 32 bits
greater32	- length frame greater than or equal to 32 bits
greater_max	- length frame greater than maximum value N1
FCS_good	- Frame Check Sequence is correct
busy	- signal from level 3 indicating if receive buffers are available
rec_cont_1	- bit 1 of receive control field
rec_cont_5	- bit 5 of receive control field (Poll/Final)
in_between	- status signal from in_betwener.
zero_flag	- ALU accu is zero/ selected bit of ACCU
max_retrans	- maximum number of retransmissions occurred

SYNCHRONISATION SIGNALS from and to transmitter process

transmitter:	- This signal indicatse to the transmitter, that a frame has to be transmitted HIGH: tranmitter has not completed LOW : idle
get_reg:	-
let_reg:	-
reg_sem:	- These signals are used in the semaphore mechanism. They are used for sharing purposes of the register groups. A process trying to request a group of registers (needed for writing only), has obtained them when reg_sem = low. The let_reg signal releases the obtained regsiters.
ab_seq	- This signal indicates to the transmitter that an I-frame which is transmitted can be aborted.
I_to_send:	- This signal indicates to the transmitter that an I-Frame must be transmitted.

B.3 Level 3 program

The following pages contain an outline of actions to be taken by level 3. The program given here is provisionary only, and is only an indication of what has to be done.

The program is partly written in Pseudo Pascal, and partly in micro-assembler, a found in the first part of this appendix.
A switch between the Pseudo Pascal and the micro-assembler statements is made by means of a slash character '/'.

```
/
;This routine initialises all register needed for packet
;transmission, where the data is fetched from the queue.

send_packet:        adres:=queue
                    offset:=0
                    select:=next_to_send
                    call read_info
                    zend_veld_1:=info
                    offset:=2
                    call read_info
                    zend_veld_2:=info
                    offset:=4
                    call read_info
                    zend_veld_3:=info
                    offset:=6
                    call read_info
                    DMA_read:=info
                    offset:=8
                    DMA_length:=info
                    signal_low_level_transmitter
                    signal_packet_ready


;This routine removes a packet from the queue through updating
;of the pointers, and updating queue administration.

acknowledge:        adres:=queue
                    offset:=10
                    select:=last_acknowledged+1
                    call read_info
                    work_channel:=info
                    adres:=channel_table
                    offset:=nr_unused_send
                    call read_info
                    accu:=info
                    accu:=accu+1
                    call write_info
                    accu:=last_acknowledged
                    accu:=accu + 1
                    last_acknowledged:=accu


/

procedure verwerk_pakket;

(* This procedure is called as soon as a packet is received, and processes the
packet.*)

begin
```

```
if pakket = invalid then discard else
begin
   packet_type_ID to next_state_generator
   toggle(next_state_generator);
   if diagnostics_to_send then send_diag(diagcode);
   else begin
            /
            ;update state of channel
            offset:=state_field
            info:=state of generator
            call write_info
            accu:=DMA_write_reg
            offset:=pointer_to_be_used
            call read_info
            offset:=info
            info:=accu
            call write_info
            offset:=pointer_to_be_used
            call read_info
            accu:=info
            accu:=accu + 2
            call write_info
            offset:=nr_unused
            call read_info
            accu:=info
            accu:=accu - 1
            call write_info
            /
         end;
         if nr_unused > 2 then send(RNR)
         clear_interrupt_field;
         generate_low_priority_interrupt;
```

(*
differences between the packet types must be made:
ordinary data packet
interrupt packet, call clear, call request, diagnostics, etcetera, must be treated specially. The Next_state_generator indicates what action must be taken on reception of a packet, which can be: discard, normal, where normal can be subdivided into three kinds, and also ERROR handling may be performed, which again is divided into three kinds.

This must be maintained per channel, and also a timer must be updated per channel.

An initialization routine must start the X.25 interface, load parameters from the shared memory. The transmission of a packet is done by putting that packet, or pointers to that packet, in the queue, where level 2 will send it off in its own time. Queue management is performed by level 3.*)

```
procedure low_pr_int_handler;

begin
   - fetch logical channel number
   - use that to address the channel descriptor
   - is this descriptor in the chain ?(chain flag set ?)
   jmp skip
   - Then address previously accessed descriptor
   - and enter the next data pointer to piont at the added desc.
   - The previous value is put into the added descriptor.
skip:
   - Remember the last accessed descriptor, by putting its addr.
   - in the work pionter field in the communication registers.

end;


procedure put_in_queue;

begin
   - assemble the packet
      - initialize the different fields
      - pointers to data buffers
   - put this in the queue at (to_be_sent + 1) mod 8
   - increment to_be_sent by one, modulo 8
   - signal level 2
end;



procedure high_priority_int_handler;

begin
   - analyse the general command field
   - do we have to look at a specific channel ?
   - if so jmp channel_hand
   - execute the command:
                  - either send diagnostics
                  - or send restart packet
   - jmp acknowledge
channel_hand:
   - fetch pointer from table to channel descriptor.
   - set priority flag, indicating that non-data packets
     must be sent, which have a higher priority.
acknowledge:
   - put the attention acknowledge field in the communication
     registers
end;
```

```
procedure add_new_data_buffers;

(* This procedure takes care of adding a number of receive
    buffers, the host does this by adding new pointers to the
    FBT, and this is told to the chip by a command.
    The number of added pointers is  given in the parameter
    Register. If the table was empty at first, the DMA write
    register must be initialized before further packet reception
*)

begin
   - Is the amount of added pointers > 0
   - if not so, exit the procedure
   - was table empty ?
   - if not jmp skip2
   - fetch first pointer from table
   - put value int the DMA write register.
   - decrement the number of unused  pointers
skip2:
   - write the number of unused pointers in the
     'unused' register.
   - and signal level 2
   - if table was empty, enable level 3 receiver
end;



procedure verlaag_tellers;


(* This routine decrements the timers in each logical
    Channel descriptor in case the timer runs          *)

begin
   for i:=1 to max_log_channel do
   begin
      address:=channel_table
      logical channel:=i;
      accu := timer value
      if accu<>0 then
      begin
         accu:=accu-1
         timer:=accu
         if timer = 0 then update_state_timer
      end;
   end;
end;



procedure update_state_timer;
```

```
(* This routine takes the appropriate actions when a time
    out occurs *)

begin
   with actual_channel do
   begin
      read state;
      set timer flag;
      toggle next_state_generator
      store state
      send packet indicated by next_state_generator
   end;
end;
```

The following page contains a table of actions to be take by the DTE on reception of certain packets. The condition of the received packet, like length or time out, will be input to the sequencer, which will be in a certain channel state, as defined by X.25. The contents of the table has been extracted from tables C.1/X.25 to C.4/X.25 of annex C to X.25 (1). These tables define actions to be taken by the DCE on reception of packets. The table in this appendix contains diagnostics codes and indicate when diagnostics have to be transmitted. The DTE however cannot transmit a diagnostics packet, so that these entries in the table may be ignored.

This table has to be implemented in a sequencer, and must be minimised for that purpose. This has been tried with a minimising program available at our university, but the program suffered from memory overflow problems.

	r1 p1	r1 p2	r1 p3	r1 p4 d1	r1 p4 d2	r1 p4 d3	r1 p5	r1 p6	r1 p7	r2	r3
Restart request & PVC					N2 R2					DISC R2	N2 D1
Restart request & VC					N2 R2					DISC R2	N2 P1
DTE RESTART CONFIRMATION & PVC					ER2 R3 17					ER2 R3 18	N2 D1
DTE RESTART CONFIRMATION & VC					ER2 R3 17					ER2 R3 18	N2 P1
CALL REQUEST & NOT OUTGOING	N3 P2	ER3 P7 21	N3 P5		ER3 P7 23		ER3 P7 24	ER3 P7 25	DISC P7	ER2 R3 18	DISC R3
CALL REQUEST & OUTGOING	N3 P2	ER3 P7 21	ER3 P6 34		ER3 P7 23		ER3 P7 24	ER3 P7 25	DISC P7	ER2 R3 18	DISC R3
CALL ACCEPTED	ER3 P7 20	ER3 P7 21	N3 D1		ER3 P7 23		ER3 P7 24	ER3 P7 25	DISC P7	ER2 R3 18	DISC R3
CLEAR REQUEST	N3 P6	N3 P6	N3 P6		M3 P6		M3 P6	DISC P6	N3 P1	ER2 R3 18	DISC R3
DTE CLEAR CONFIRMATION	ER3 P7 20	ER3 P7 21	ER3 P7 22		ER3 P7 23		ER3 P7 24	ER3 P7 25	N3 P1	ER2 R3 18	DISC R3
RESET REQUEST	ER3 P7 20	ER3 P7 21	ER3 P7 22	N4 D2	DISC D2	N4 D1	ER3 P7 24	ER3 P7 25	DISC P7	ER2 R3 18	DISC R3
DTE RESET CONFIRMATION	ER3 P7 20	ER3 P7 21	ER3 P7 22	ER4 D3 27	ER4 D3 28	N4 D1	ER3 P7 24	ER3 P7 25	DISC P7	ER2 R3 18	DISC R3
DATA/DATAGRAM/INTERRUPT/FLOW-CONTROL	ER3 P7 20	ER3 P7 21	ER3 P7 22	N4 D1	ER4 P7 28	DISC D3	ER3 P7 24	ER3 P7 25	DISC P7	ER2 P7 18	DISC R3
RESTART WITH NON-ZERO CHANNEL NR.	ER3 P7 41	ER3 P7 41	ER3 P7 41	ER4 D3 41	ER4 D3 41	DISC D3	ER3 P7 41	ER3 P7 41	DISC P7	ER2 R3 41	DISC R3
PACKET TYPE IDENTIFIER TOO SHORT	ER3 P7 38	ER3 P7 38	ER3 P7 38	ER4 D3 27	ER4 D3 28	DISC D3	ER3 P7 38	ER3 P7 38	DISC P7	ER2 R3 38	DISC R3
UNSUPPORTED PACKET TYPE	ER3 P7 33	ER3 P7 33	ER3 P7 33	ER4 D3 27	ER4 D3 28	DISC D3	ER3 P7 33	ER3 P7 33	DISC P7	ER2 R3 33	DISC R3
CALL REQUEST/CALL CONNECTED/CALL-CLEAR/CLEAR CONF & PVC	DISC P1	DISC P2	DISC P3	ER4 D3 35	ER4 D3 35	DISC D3	DISC P5	DISC P6	DISC P7	DISC R2	DISC R3
REJECT FACILITY NOT SUBSCRIBED	DISC P1	DISC P2	DISC P3	ER4 D3 37	ER4 D3 37	DISC D3	DISC P5	DISC P6	DISC P7	DISC R2	DISC R3
TIME OUT	DISC P1	ER3 P6 49	DISC P3	DISC D1	N4 D2	DISC D3	DISC P5	N3 P6	DISC P7	N2 R2	DISC R3
SMALLERTHANTWO	DIAG P1 38	DIAG P2 38	DIAG P3 38	DIAG D1 38	DIAG D2 38	DIAG D3 38	DIAG P5 38	DIAG P6 38	DIAG P7 38	DIAG R2 38	DIAG R3 38
ICORRECT GFID	DIAG P1 40	DIAG P2 40	DIAG P3 40	DIAG D1 40	DIAG D2 40	DIAG D3 40	DIAG P5 40	DIAG P6 40	DIAG P7 40	DIAG R2 40	DIAG R3 40
WRONG_CH	DIAG P1 36	DIAG P2 36	DIAG P3 36	DIAG D1 36	DIAG D2 36	DIAG D3 36	DIAG P5 36	DIAG P6 36	DIAG P7 36	DIAG R2 36	DIAG R3 36
DIAGNOSTICS	N3 P1	N3 P2	N3 P3	N4 D1	N4 D2	N4 D3	N3 P5	N3 P6	N3 P7	N2 R2	N2 R3

DISC : DISCARD ---> IGNORE PACKET
ERN : ERROR N ---> GENERATE ERROR N 2 <= N <= 4
NK : NORMAL K---> NORMAL RESPONSE K 2 <= K <= 4
DIAG : DIAGNOSTICS

DECIMAL NUMBER IN CENTER BOTTOM OF BOX IS DIAGNOSTIC NUMBER IN CASE OF ERROR N

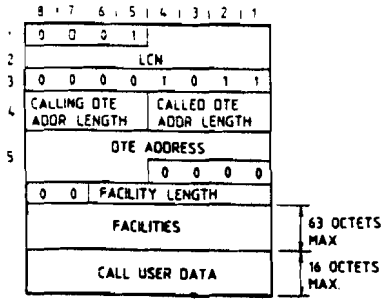
PACKET TYPE (MODULO 8)		OCTET 1 BITS	OCTET 3 BITS
FROM DCE TO DTE	FROM DTE TO DCE	8 7 6 5	8 7 6 5 4 3 2 1
CALL SET-UP AND CLEARING			
INCOMING CALL	CALL REQUEST	0 D 0 1	0 0 0 0 1 0 1 1
CALL CONNECTED	CALL ACCEPTED	0 D 0 1	0 0 0 0 1 1 1 1
CLEAR INDICATION	CLEAR REQUEST	0 0 0 1	0 0 0 1 0 0 1 1
DCE CLEAR CONFIRMATION	DTE CLEAR CONFIRMATION	0 0 0 1	0 0 0 1 0 1 1 1
DATA AND INTERRUPT			
DCE DATA	DTE DATA	Q D 0 1	P(R) M P(S) 0
DCE INTERRUPT	DTE INTERRUPT	0 0 0 1	0 0 1 0 0 0 1 1
DCE INTERRUPT CONFIRMATION	DTE INTERRUPT CONFIRMATION	0 0 0 1	0 0 1 0 0 1 1 1
DATAGRAM *			
DCE DATAGRAM	DTE DATAGRAM	0 0 0 1	P(R) 0 P(S) 0
DATAGRAM SERVICE SIGNAL		1 0 0 1	P(R) 0 P(S) 0
FLOW CONTROL AND RESET			
DCE RR	DTE RR	0 0 0 1	P(R) 0 0 0 0 1
DCE RNR	DTE RNR	0 0 0 1	P(R) 0 0 1 0 1
	DTE REJ *	0 0 0 1	P(R) 0 1 0 0 1
RESET INDICATION	RESET REQUEST	0 0 0 1	0 0 0 1 1 0 1 1
DCE RESET CONFIRMATION	DTE RESET CONFIRMATION	0 0 0 1	0 0 0 1 1 1 1 1
RESTART			
RESTART INDICATION	RESTART REQUEST	0 0 0 1	1 1 1 1 1 0 1 1
DCE RESTART CONFIRMATION	DTE RESTART CONFIRMATION	0 0 0 1	1 1 1 1 1 1 1 1
DIAGNOSTIC			
DIAGNOSTIC *		0 0 0 1	1 1 1 1 0 0 0 1

* NOT NECESSARILY AVAILABLE ON EVERY NETWORK

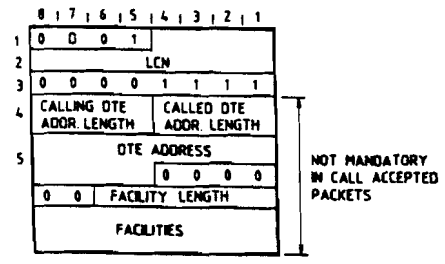
0 0	NOT ASSIGNED
0 1	MODULO 8
1 0	MODULO 128
1 1	EXTENSION

PACKET TABLE

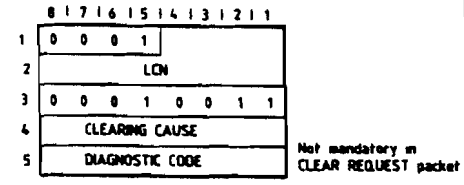
CALL SET-UP AND CLEARING



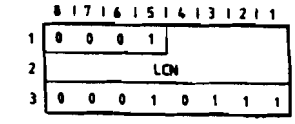
CALL REQUEST AND INCOMING CALL



CALL ACCEPTED AND CALL CONNECTED

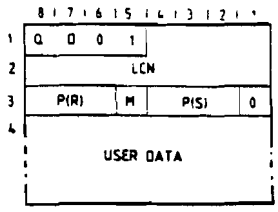


CLEAR REQUEST AND CLEAR INDICATION



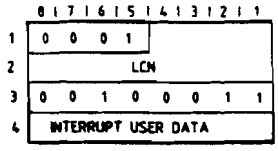
DTE AND DCE CLEAR CONFIRMATION

DATA AND INTERRUPT

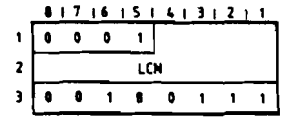


MODULO 8

DTE AND DCE DATA

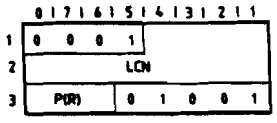


DTE AND DCE INTERRUPT



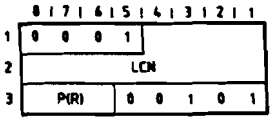
DTE AND DCE INTERRUPT CONFIRMATION

FLOW CONTROL AND RESET



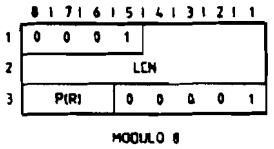
MODULO 8

DTE AND DCE RR



MODULO 8

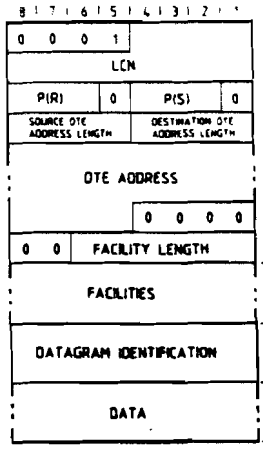
DTE AND DCE RNR



MODULO 8

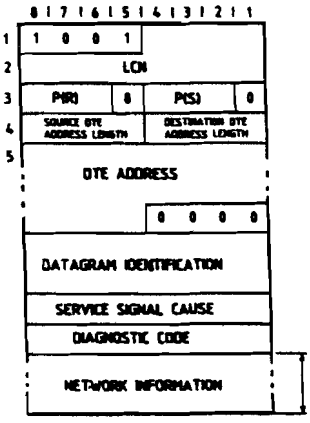
DTE REJ

DATAGRAM



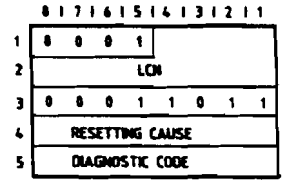
MODULO 8

DTE AND DCE DATAGRAM

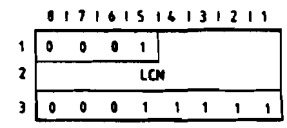


MODULO 8

DATAGRAM SERVICE SIGNAL

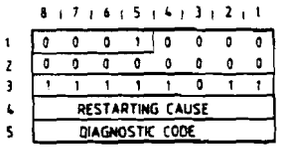


RESET REQUEST AND RESET INDICATION



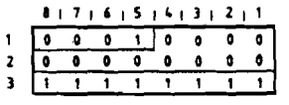
DTE AND DCE RESET CONFIRMATION

RESTART



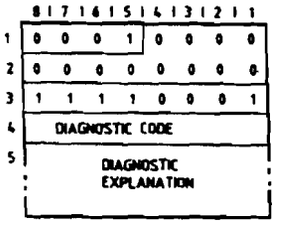
NOT mandatory in RESTART REQUEST PACKETS

RESTART REQUEST & RESTART INDICATION



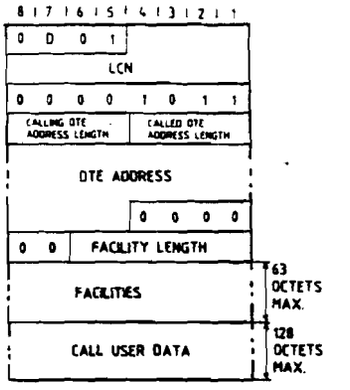
DTE & DCE RESTART CONFIRMATION

DIAGNOSTIC

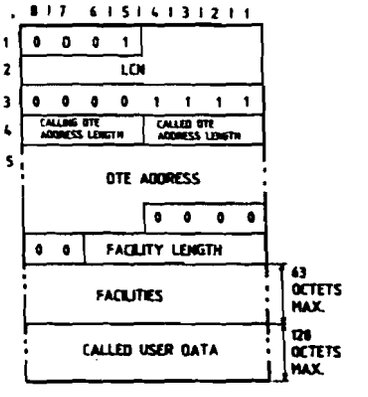


DIAGNOSTIC

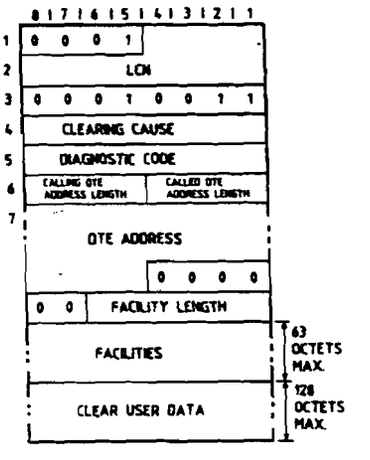
FAST SELECT FACILITY



CALL REQUEST AND INCOMING CALL



CALL ACCEPTED AND CALL CONNECTED



CLEAR REQUEST AND CLEAR INDICATION