TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Een local area netwerk voor THE KUNix machine

Peters, H.F.H.M.

Award date:
1985

Link to publication

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

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

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

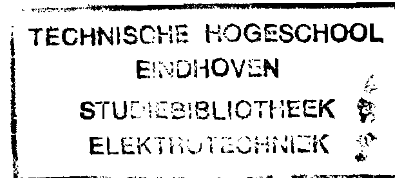
Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. May. 2025

4002 bse

ECB 943

TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling der Elektrotechniek
Vakgroep Digitale Systemen (EB)



Een local area netwerk
voor THE KUNix machine.

Afstudeerverslag van: H.F.H.M. Peters
Periode: Januari 1984 t/m december 1984
Afstudeerhoogleraar: Prof.ir.A.Heetman
Coach: Ing. L.A. van Bokhoven / Ir. M. Stevens

De Afdeling der Elektrotechniek van de Technische Hogeschool Eindhoven aanvaardt geen enkele verantwoordelijkheid voor de inhoud van stage- en afstudeerverslagen.

INHOUDSOPGAVE

1. INLEIDING 1
   1.1 THE KUNix machine 1
   1.2 Het netwerk 2

2. L.A.N Technologie 5
   2.1 Local Area Netwerken - een definitie 5
   2.2 Klassificatie van Local Area Netwerken 8
      2.2.1 Het ISO/OSI Model 8
      2.2.2 Het IEEE-802 Model 10
         2.2.2.1 Het physische medium 11
         2.2.2.2 Medium Attach Unit 13
         2.2.2.3 Physical signalling 15
         2.2.2.4 Medium Acces Control 16
         2.2.2.5 Logical Link Control 21
   2.3 LAN Standaards 22
      2.3.1 Vergelijking van de LAN standaards 24
   2.4 Beschikbare LAN hardware 34
      2.4.1 token passing chips 34
      2.4.2 CSMA/CD chips 34
      2.4.3 Overige chips 36
   2.5 De keuze van de hardware 38

3. De Ethernet hardware 39
   3.1 De INTEL 82586 LAN coprocessor 40
      3.1.1 Het gemeenschappelijk geheugen (shared memory) 42
      3.1.2 De receive frame area 44
      3.1.3 Command list 47
   3.2 De serieele interface 48
   3.3 De ethernet transceiver 49

4. Netwerk Software 51
   4.1 Enkele definities 52
   4.2 De physical Layer 55
   4.3 De MAC-laag 55
      4.3.1 MAC frame format 55
      4.3.2 MAC Service specificatie 57
   4.4 De LLC-laag 58
      4.4.1 LLC_protocol data unit 58
      4.4.2 LLC_service specificaties 59

5. De implementatie 62
   5.1 De structuur 65
   5.2 De LLC-MAC interface 69
   5.3 Het zend gedeelte 71
   5.4 Het receive gedeelte 74
   5.5 Het TIMER proces 76
   5.6 De LLC laag 77

6. VOORTGANG 78
A. Het CSMA/CD backoff -algoritme 80
B. Software listing 82
C. Het timer proces 108

SAMENVATTING

In een samenwerkings project tussen de Technische Hogeschool Eindhoven en de Katholieke Universiteit van Nijmegen wordt momenteel "THE KUNix" machine gebouwd. Dit is een modulair computer systeem, waarin de meeste I/O taken door "intelligente" I/O-controllers worden verzorgd.
Deze I/O-controllers zijn zelfstandige micro-computers, gebaseerd op de 80186 processor van Intel, en voorzien van een "multi-tasking" operating system.

In dit verslag wordt de ontwikkeling van een Local Area Netwerk beschreven voor THE KUNix machine.

Na een uitvoerige introductie in LAN technieken en methodes volgt een beschrijving van de hardware. Bij de realisatie hiervan is gebruik gemaakt van de Intel 82586 LAN coprocessor. Dit is een CSMA/CD (Ethernet) chip die voldoet aan de standaardisatie voorstellen van de IEEE-802 werkgroep. De communicatie tussen deze LAN coprocessor en de host processor is gebaseerd op het "shared memory" concept.

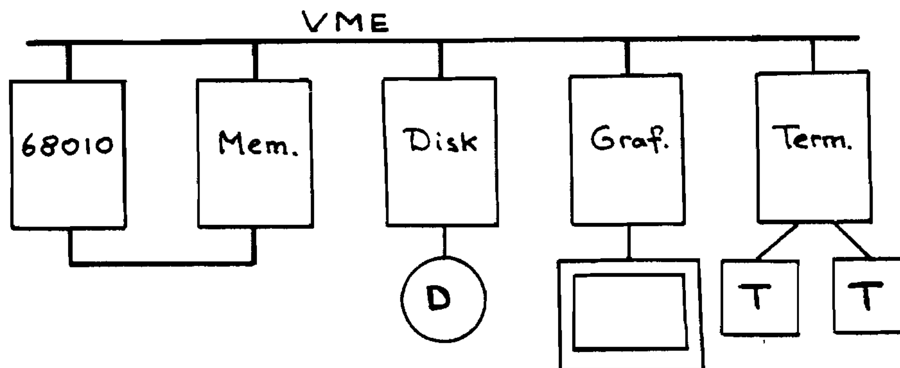
Bij de specificatie van de software is eveneens uitgegaan van het IEEE-802 model. De gerealiseerde software bestaat uit een aantal parallelle processen, werkend onder het "multi-tasking" systeem op een van de I/O-controllers.
De ontwikkelde software is een implementatie van de onderste laag van het netwerk model en bestaat voornamelijk uit de besturing van het "shared memory".

I INLEIDING

1.1 THE KUNIX MACHINE

Binnen de vakgroep EB wordt momenteel hard gewerkt aan de ontwikkeling van THE KUNix machine. Dit is een krachtig computer systeem, dat in samenwerking met de katholieke universiteit van Nijmegen (KUN) gerealiseerd wordt.
Het systeem is modulair opgebouwd en het werkt onder meer met het UNIX operating system.

De verschillende modulen (of kaarten) zijn onderling gekoppeld via een VME interface. Alhoewel de hardware een willekeurig aantal processoren toestaat, is in de huidige implementatie gekozen voor een centrale processorkaart ondersteund door verschillende (intelligente) I/O-controllers. Als centrale processor is gekozen voor de 68010 en de I/O controllers zijn gebaseeerd op de iAPX 186.
De reden om meerdere I/O processoren toe te passen is om de hoofdprocessor te ontlasten en zodoende de performance van het gehele systeem te verbeteren.



figuur 1.1 - THE KUNix machine

Op het ogenblik zijn er een aantal van deze modules gerealiseerd of in ontwikkeling, t.w:

- De 68010 kaart
  Op deze kaart zal het UNIX operating systeem worden geinplementeerd. De kaart bevat naast de 68010 processor, 1 Mbyte lokaal geheugen, 2 memory management units, 2 serieele I/O kanalen en een lokale extensie bus.

- Geheugenkaart
  Deze kaart bevat 2 Mbyte aan geheugen en is voorzien van error detektie en correctie. Naast de VME interface kan deze kaart ook via de lokale extensie bus worden aangestuurd.

- Disk Controller Kaart
  Deze kaart bevat een kompleet (micro) komputer systeem, gebaseerd op de iAPX 186 processor van Intel. Naast 128 kbyte RAM zijn twee disk interfaces aanwezig, t.w. een SASI interface voor aansturing van hard-disks en een SHUGART interface voor floppy disk drives.
  Verder bevat deze kaart nog 2 serie kanalen en uiteraard de VME interface.

- Terminal kaart
  Deze kaart is identiek aan de disk controller kaart, maar bevat i.p.v. de 2 disk interfaces 8 serieele controllers, bedoeld voor het aansturen van terminals, modems, etc.

- Grafische kaart
  Deze kaart bevat de NEC 7220 graphics display controller. Met deze kaart kan een grafisch (kleuren) werkstation worden gerealiseerd.

1.2 HET NETWERK

Doordat THE KUNix machine modulair van opbouw is en ieder module in feite een kompleet computer systeem is, ligt de toepassing van een netwerk zeer voor de hand. Het netwerk vormt dan de koppeling van een aantal verspreid opgestelde zelfstandige systemen.
Figuur 1.2 geeft een denkbeeldige configuratie bestaande uit 3 verschillende KUNix machines.
Configuratie A bestaat uit een kompleet systeem, dus processor, disk en terminal module.
Systeem B bevat 2 grafische werkstations + processor kaart, maar geen disk faciliteiten. Dit systeem is een zelfstandig geheel en gebruikt het netwerk als disk server.
Systeem C bevat alleen terminals en kan bijvoorbeeld een uitbreiding zijn behorende bij A. De terminal module bevat alle software om lokaal te kunnen editen en gebruikt het netwerk als file server.

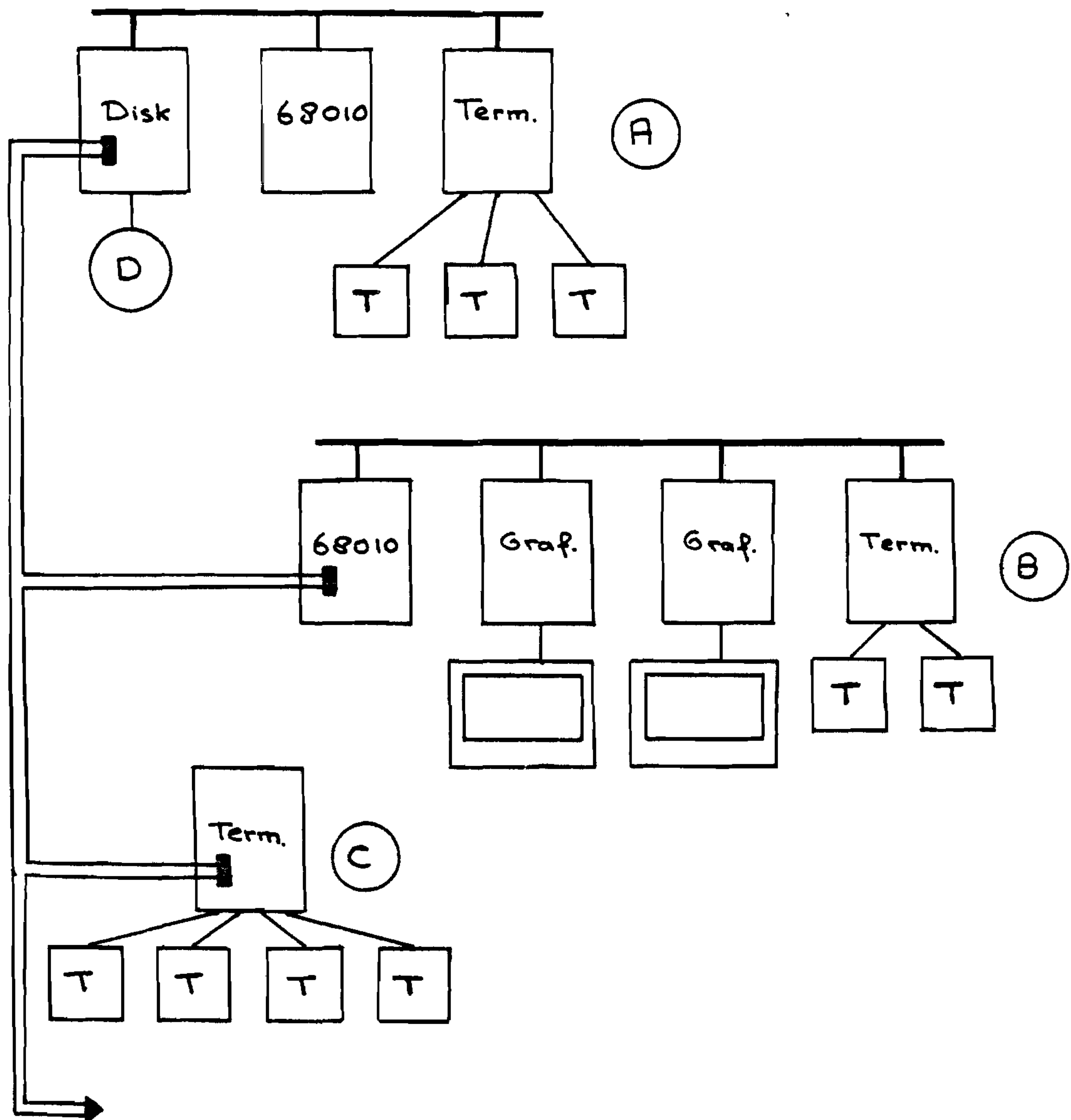
De voordelen van een dergelijk netwerk zijn nogal voor de hand liggend:

- Het uitsparen van dure hardware (disk drives, printers, streamers, etc.)

- Door de koppeling van de verschillende systemen ontstaat een krachtig multi-user, multi-processor systeem.

- Koppeling van dit netwerk aan een ander netwerk (THE-NET, DATANET-1) vergroot de mogelijkheden voor alle aangesloten gebruikers.

Kortom, een netwerk voor THE KUNix machine is zeer aan te bevelen.
Dit verslag behandelt de realisatie van een dergelijk netwerk.
In hoofdstuk 2 wordt een algemene beschouwing gegeven over standaards en prestaties van Local Area Netwerken.
Hoofdstuk 3 behandelt de hardware zoals die uiteindelijk gebruikt zal gaan worden.
In hoofdstuk 4 wordt de software organisatie die voor een dergelijk netwerk nodig is nader toegelicht.



figuur 1.2 - Een mogelijke netwerk configuratie

II L.A.N TECHNOLOGIE

2.1 LOCAL AREA NETWERKEN - EEN DEFINITIE

Een sluitende definitie van een LAN bestaat niet. In de literatuur wordt onder een LAN verstaan een netwerk dat in ieder geval aan de volgende eisen voldoet:

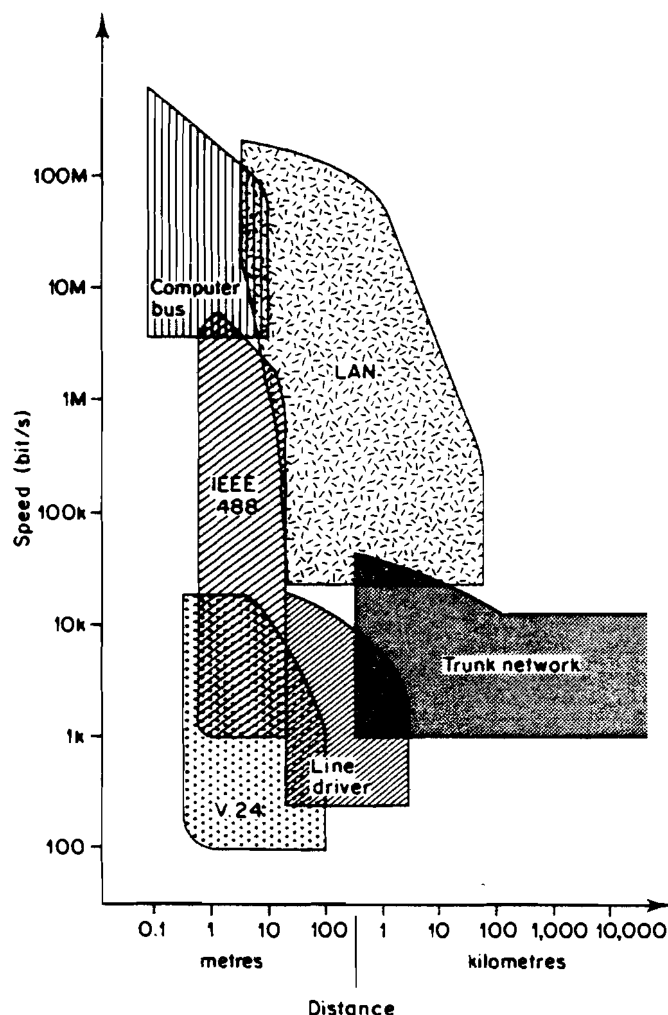
1 Een diameter van het net tussen de 1-2 km.

2 Een bitsnelheid van tenminste 1Mb/s. Deze snelheid is uiteraard afhankelijk van het doel waarvoor het netwerk gebruikt gaat worden.
Figuur 2.1 geeft de relatie tussen snelheid en grootte van een LAN in vergelijking met andere netwerken, zoals:

- De computerbus. Deze voorziet in een zeer hoge snelheid over zeer korte afstanden.
- De IEEE-488 (of ookwel H.P.-bus genoemd) is de oplossing om laboratorium instrumenten lokaal te koppelen met een vrij hoge snelheid.
- Line drivers worden in het algemeen gebruikt om terminals via een "twisted pair" met een komputer te kunnen koppelen. Snelheden tot 9600 baud zijn vrij algemeen.
- Trunk networks of ook wel Wide Area Networks (WAN) genoemd zijn in principe in grootte onbeperkt (worldwide), maar hebben een lage snelheid. Voorbeelden zijn X.25 en SNA.

3 Een gemeenschappelijk gebruik van het medium. (Hierdoor vallen circuit geschakelde netwerken buiten de definitie van een LAN.)

4 Eigendom van en beheerd door een organisatie.



figuur 2.1 - LAN versus overige netwerken

Omdat deze definitie niet veel houvast biedt bij het ontwerpen van een LAN, hetgeen ook blijkt uit de tientallen verschillende netwerken die momenteel op de markt verkrijgbaar zijn, is er in 1981 door IEEE een werkgroep opgericht met als taak enige standardisatie in het LAN gebeuren aan te brengen.
Deze IEEE werkgroep, met als kodenaam IEEE-802 heeft allereerst een aantal eisen opgesteld waaraan een LAN zou moeten voldoen.

Deze eisen hebben betrekking op de de onderstaande punten:

Toepassings gebieden

Een LAN moet een grote verscheidenheid aan datakommunikatie funkties voeren. Hieronder wordt verstaan: file transfer,

terminal support (inklusief highspeed grafische terminals), electronic mail, database access, voicegrams, etc.
Verder moet een LAN geschikt zijn om een groot aantal verschillende devices te kunnen aansluiten, zoals:

- Computers/Mass storage decives

- Terminals/printers/plotters

- Photocopiers/Facsimile transceivers

- "Gateways" naar andere LAN's

Physische eigenschappen

Een LAN moet transparant zijn voor ieder data type, m.a.w. mag geen eisen stellen aan het soort data dat verstuurd wordt.

Op een LAN moeten tenminste 200 gebruikers kunnen worden aangesloten.
Het verzorgings gebied van een LAN moet tot 2 km kunnen worden uitgebreid.

De transmissie snelheid moet tussen de 1Mbaud - 20Mbaud liggen.

Het in/uitloggen c.q. het aansluiten/verwijderen van devices mag een onderbreking van het net geven van ten hoogste 1 sec.

De verdeling van alle LAN faciliteiten (in het bijzonder de bandbreedte van het fysische medium) moet zo eerlijk mogelijk onder de verschillende devices verdeeld worden.

De afstand tussen de physische koppeling aan het net en het device moet minimaal 50 meter kunnen overbruggen.

Netwerk management

Het moet mogelijk zijn om een data packet een individueel-, een groeps- of een broadcastadres mee te geven. Deze adressering moet flexibel en door de gebruiker eenvoudig te wijzigen zijn.

De packet lengte moet (binnen zekere grenzen) variabel zijn.

De "throughput" van een LAN moet in geval van overbelasting stabiel blijven.

De maximale vertraging die een data packet kan ondervinden moet deterministisch zijn, d.w.z. moet vooraf berekend kunnen worden.

Fouten & Onderhoud

De kans op een niet te detekteren fout in een data packet mag niet meer dan 1 maal per jaar voorkomen, hetgeen bij een bitrate van 5Mb/s een foutenkans van 10**-14betekent. De werkelijke en te detekteren foutenkans mag 10**-8 bedragen.

De errordetektie moet zodanig zijn uitgevoerd dat een maximum van 4 bitfouten per packet gedetekteerd kan worden.

De betrouwbaarheid van het net moet erg groot zijn. Een "down" tijd van minder dan 20 minuten per jaar (0,03%) is acceptabel.

Een LAN moet over faciliteiten beschikken om (hardware) fouten in het net te kunnen detekteren en eventueel te lokaliseren.

2.2 KLASSIFICATIE VAN LOCAL AREA NETWERKEN

In deze paragraaf zal ik enige begrippen behandelen die karakteristiek zijn voor een LAN en die inzicht verschaffen om tot een uiteindelijke keuze van een netwerk te komen.

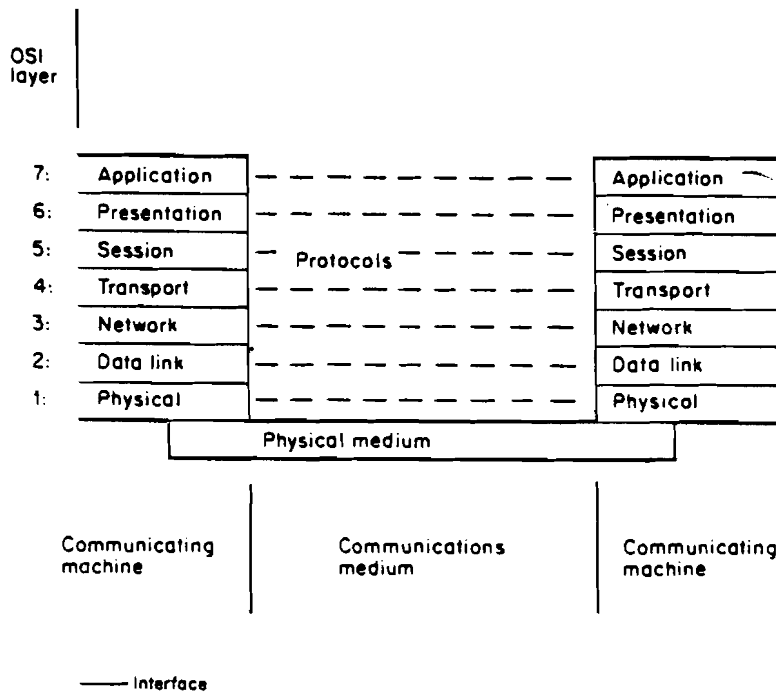
2.2.1 Het ISO/OSI Model

Door de ISO (International Standards Organization) is rond de zeventiger jaren een algemeen model gepresenteerd dat zou moeten leiden tot een gestandardiseerde aanpak bij de ontwikkeling van data netwerken. (figuur 2.2) Het z.g. OSI (Open Systems Interconnection) model bestaat uit 7 lagen, waarvan tot op heden (medio 1984) alleen de eerste vier lagen duidelijk zijn vastgelegd. Voor laag 5 is een voorstel ingediend en de lagen 6 en 7 zijn nog leeg.

In het kort kunnen de lagen van het ISO/OSI model als volgt worden omschreven:

PHYSISCHE LAAG

Deze laag beschrijft het physische medium en de methode waarmee de "bit stream" wordt verstuurd. Het betreft hier dus zaken als signaal nivo, baudrate, soort medium (kabel, fiber), basisband of broadband, etc.



figuur 2.2 - Het Open System Interconnection Model

DATA LINK LAAG

Hierin zijn de regels vastgelegd op welke wijze de data wordt verstuurd.
De data link laag beschrijft:
De opbouw van het frame en op welke wijze het medium wordt gebruikt. Dit staat ook wel bekend als Medium Acces Methode. (MAC)
De methode waarop een verbinding wordt gelegd (CONNECT/DISCONNECT) en de regulering van de frames (FLOWCONTROL).

NETWERK LAAG

Deze Laag verzorgt de "routing" van de afzondelijke frames tussen de verschillende netwerken.
Omdat "station-to station" kommunikatie in een netwerk gelijk is aan een "point- to point" verbinding, is de netwerk laag overbodig zolang alle kommunikatie binnen een netwerk verloopt.

TRANSPORT LAAG

De Transport laag is de eerste laag die medium onafhankelijk is en waarbij sprake is van "peer-to peer" kommunicatie. De

funktie van de transport laag is hoofdzakelijk fout detektie, korrektie en het reguleren van de data flow.

SESSION LAAG

Dit is de "overall" manager en zorgt voor het openen en sluiten van de virtuele verbinding die de transport laag aanbiedt. Ook verzorgt deze laag de vertaling van logische stations namen naar netwerk adressen.

PRESENTATIE LAAG

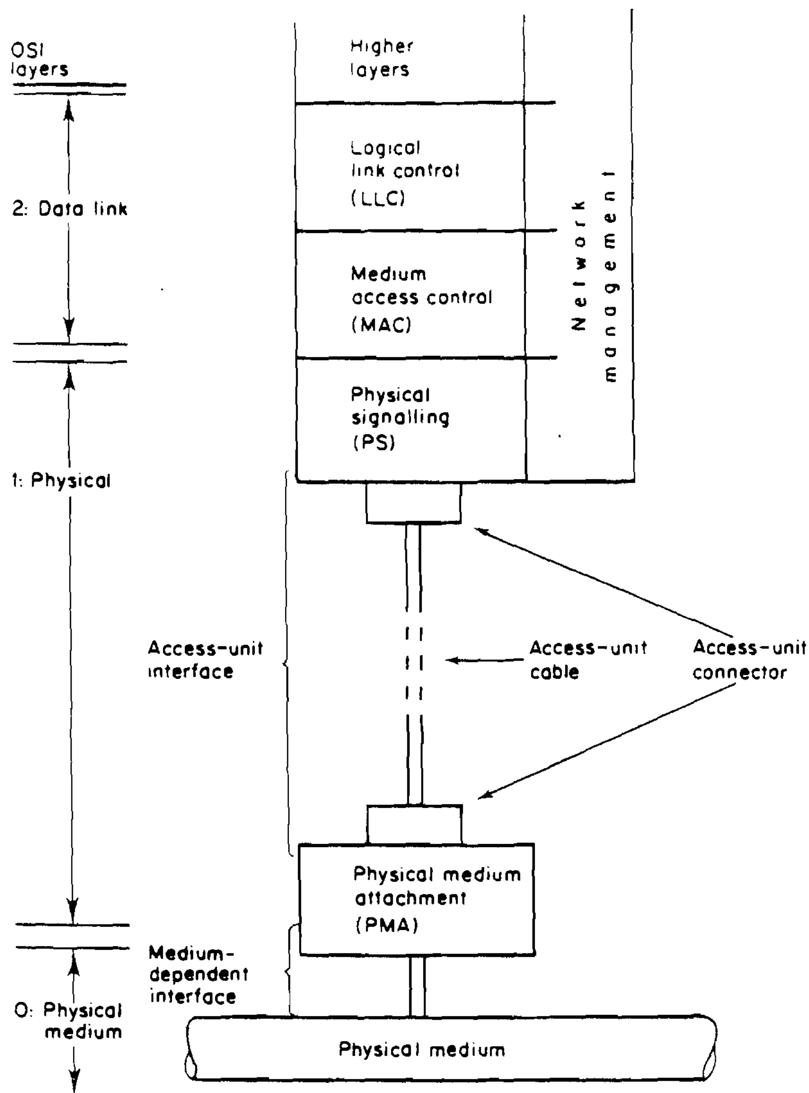
Deze laag zorgt voor een eventuele vertaling en/of code konversie, zodat de data in een bepaalde vorm aan de gebruiker wordt aangeboden.
Een echt duidelijke omschrijving van deze laag is nog niet gerealiseerd.

APPLIKATIE LAAG

Dit is de interface naar de gebruiker en deze laag behoort de verschillende netwerk funkties aan te bieden. Zoals reeds eerder genoemd, zijn dit zaken als: electronic mail, file server, etc.
Evenals bij de presentatie laag is er nog geen duidelijk afgebakend idee over de inhoud van deze laag.

2.2.2 Het IEEE-802 Model

Naast het ISO/OSI model bestaat er voor de lagen 1 en 2 een meer gedetailleerd model dat door de IEEE-802 werkgroep als nieuw LAN model is voorgedragen en zo goed als zeker als standaard model zal worden aangenomen. Het model is weergegeven in figuur 2.3.
Aan de hand van dit model zullen de verschillende lagen wat verder worden uitgediept, waarbij tevens op de praktische realisatie nader wordt ingegaan.



figuur 2.3 - IEEE 802 Reference Model

2.2.2.1 Het physische medium

Voor het physische medium zijn er een aantal mogelijke oplossingen:

Twisted pair: Dit is de goedkoopste en eenvoudigste oplossing. (zie ook tabel 2.1) Doordat de kabel niet afgeschermd is, is deze vrij gevoelig voor storingen. Ook de

maximale bitsnelheid ligt relatief laag, zodat dit type kabel zelden voor een LAN wordt toegepast.

Multi core kabel: Door meerdere kabels parallel te zetten wordt de kapaciteit van de totale kabel vergroot. Multi core kabels worden in een aantal netten toegepast (Cluster One, Econet), maar hebben als nadeel de hoge kosten.

Coax: Dit is verreweg het meest toegepast medium voor LAN's. Mede vanwege de toepassing in CATV systemen, word ook bij LAN steeds meer broadband technieken toegepast, met een bandbreedte die gelijk is aan een TV-kanaal.(5Mhz)

Glasvezel: Waarschijnlijk het medium dat in de naaste toekomst het meest gebruikt zal gaan worden. Snelheden tot 1Gb/s zijn hiermee mogelijk. Vooralsnog is toepassing van glasvezel een zeer dure oplossing. (konnektor !)

Radio: Radio verbindingen hebben als voordeel dat de stations mobiel kunnen zijn maar worden vrijwel alleen voor militaire toepassingen gebruikt.

Overige: Hieronder vallen een aantal (nog) weinig toegepaste media, zoals: Infrarood, Twinax en Triax kabel (Een sterk verbeterde coax).

De kapaciteit van het physische medium wordt vaak aangegeven als het product van de maximale baudrate en de daarbij behorende maximale overbruggings afstand. Voor de meeste gebruikte media ligt deze kapaciteit in orde van:

Twisted pair : 1 Gmeter bit/s
Coax: 15 Gmeter bit/s
Glasvezel: 400 Gmeter bit/s

Medium	UK cost in 1981: Per metre	UK cost in 1981: Per physical connector	Notes
Unshielded twisted pair	10p	—	Phone wire
Shielded twisted pair (2 pair)	90p	—	
4-wire unshielded	50p	£6	CCITT V.24 (50ft maximum range)
7-wire unshielded	65p	£6	
Nestar Clusterbus (16 wire)	£1	£3	For Cluster One
16-wire unshielded	£1.3	£7	CCITT V.24
25-wire unshielded	£1.8	£7	CCITT V.24
25-wire shielded	£2.3	£8	V.24 (extended 500ft maximum range)
Ethernet coax cable	£1.5	£12	50 ohm
CATV coax (RG62U)	20p	£4.5	75 ohm
CATV semi-rigid	75p	—	75 ohm
Optical fibre	50p	£30	Bandwidth > 200 MHz
Polynet	45p	—	

Tabel 2.1 - Kosten van het Physische medium

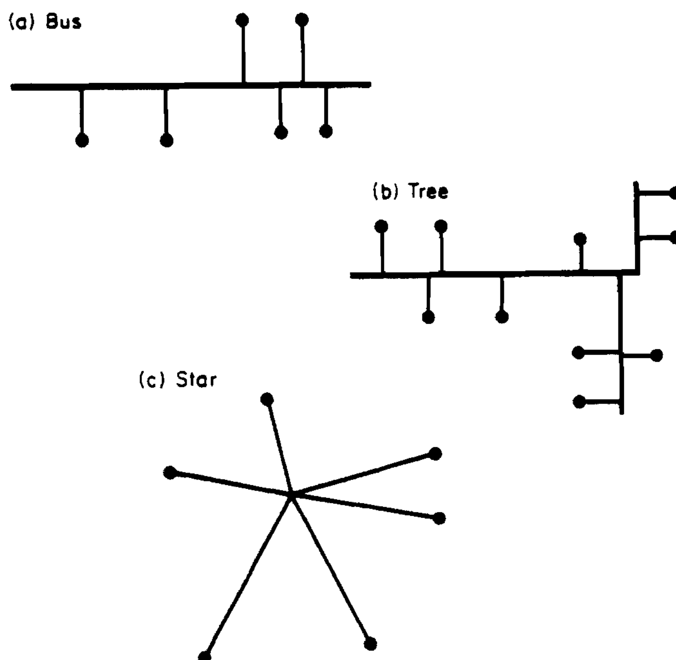
2.2.2.2 Medium Attach Unit

De PMA (Physical Medium Attachement) verzorgt de koppeling aan het medium, d.w.z. zet de elektrische signalen om naar signalen die door het medium verwerkt kunnen worden. (bijv. optisch, broadband) In het geval dat het medium uit metallieke geleiders bestaat, verzorgt de PMA de galvanische scheiding en vormt een beveiliging tegen te hoge spanningen op het medium.

De PMA wordt naast het medium sterk bepaald door de topologie van het netwerk.

Er zijn in principe twee klassen van topologieen, t.w: broadcast en sequential.

De verschillende broadcast topologieen zijn in figuur 2.4 weergegeven. Eigenschap van een broadcast topologie is dat ieder knoopppunt een direkte verbinding heeft met elk ander knooppunt.



figuur 2.4 - Broadcast topologieen

Dit betekent dat de PMA veel vermogen moet kunnen leveren om alle andere knooppunten te kunnen bereiken. In de praktijk betekent dit dat de lengte van een segment beperkt is tot zo'n 500-600 meter en moeten meerdere segmenten d.m.v. repeaters worden gekoppeld.

Een ander probleem bij broadcast topologieen is de overhead die onstaat doordat slechts een station het medium kan gebruiken. Deze overhead (ook wel "slot- tijd" genoemd) is een sommatie van de volgende tijden:

- De propagatie tijd van het signaal over het netwerk

- De tijd nodig om een controle bericht te versturen

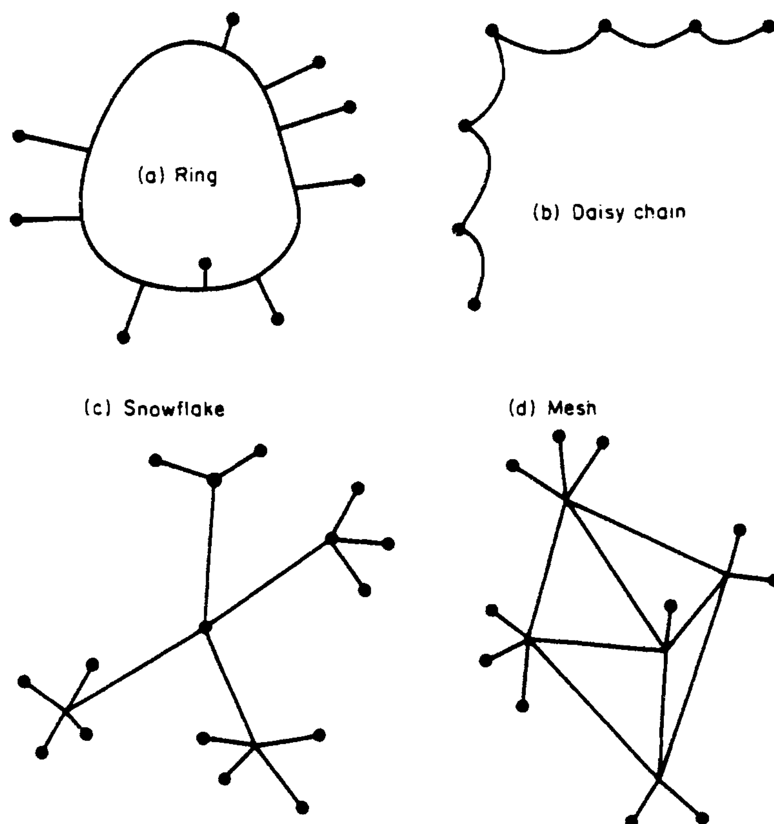
- De response tijd van de bestemming

- De vertraging van de repeater

Omdat deze overhead aanwezig is bij ieder bericht dat verstuurd wordt, is het zaak om de gemiddelde bericht lengte een veelvoud van de slot-tijd te kiezen.

In een sequentiele topologie (figuur 2.5) wordt telkens naar slechts een ander station gezonden.

Voordelen zijn het lage(re) zendvermogen en de mogelijkheid om het netwerk op te bouwen uit verschillende physische media.
Ondanks deze voordelen wordt tot nu toe in de LAN techniek hoofdzakelijk de bus topologie toegepast.



figuur 2.5 - Sequentiele topologieen

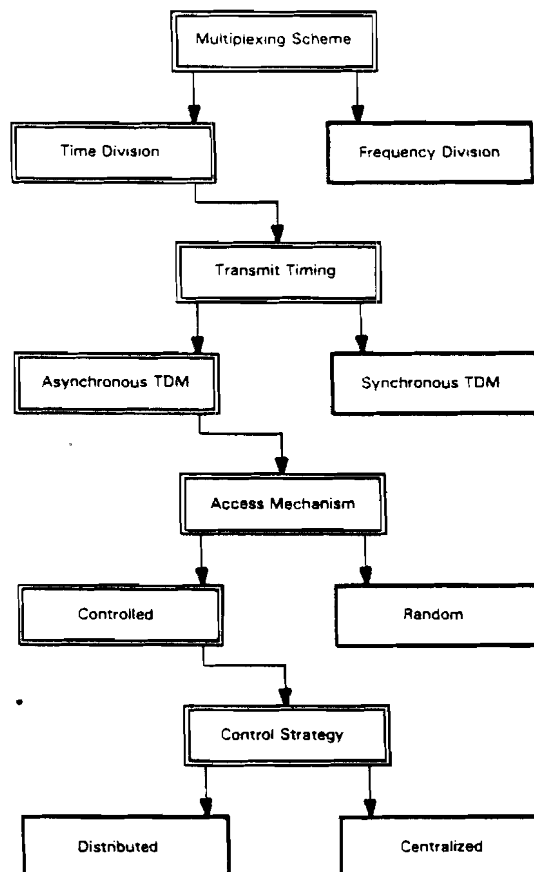
2.2.2.3 Physical signalling

Physical signalling beschrijft de manier waarop de "individuele" bits gekodeerd worden. In de meeste gevallen wordt er gebruik gemaakt van baseband- of broadband technieken. Bij baseband wordt hoofdzakelijk (differential) Manchester kodering toegepast. Manchester kodering heeft als voordeel dat het ontvangende station het oorspronkelijke klok signaal eenvoudig kan reconstrueren.

Bij broadband signalling wordt vrijwel altijd gebruik gemaakt van CATV technieken, vanwege de ruime ervaring die hiermee is opgedaan en de relatief goedkope hardware.

2.2.2.4 Medium Acces Control

De MAC laag beschrijft op welke wijze het medium wordt gebruikt. Hiervoor zijn verschillende methoden ontwikkeld, die verklaard zullen worden aan de hand van figuur 2.6.



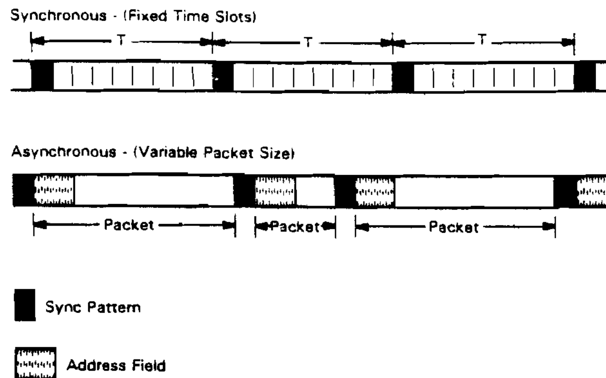
figuur 2.6 - Medium Access Methoden

Omdat het physische medium gedeeld wordt door meerdere gebruikers, moet er een of andere vorm van multiplexing worden toegepast. De twee bekendste methoden hiervoor zijn: verdeling in tijd (TDM) en in frekwentie (FDM).

FDM is zeker niet de definitieve oplossing, omdat verdeling van de beschikbare bandbreedte op technische gronden beperkt is tot zo'n 40-50 kanalen (bij een acceptabele bandbreedte per kanaal) en een LAN minimaal 200 aansluitingen moet kunnen realiseren. Dit komt er in de praktijk op neer dat ieder FDM kanaal d.m.v. een of andere vorm van TDM weer verder wordt verdeeld.(bijv het THE-NET)

TDM kan worden onderverdeeld in synchrone en asynchrone TDM. Synchroon wil zeggen dat alle stations dezelfde klok gebruiken. Dit is alleen mogelijk als het klok-signaal via een apart kanaal aan alle stations wordt doorgegeven.

Een afgeleide methode van synchrone TDM is het gebruik van vaste tijd-slots (figuur 2.7). Er wordt dan voor ieder station een vaste tijdshoeveelheid (minimaal een slot) gereserveerd. Een nadeel hiervan is uiteraard de verspilling van tijd doordat er ook slots gereserveerd moeten worden voor die stations welke geen gebruik willen maken van het medium. Deze MAC methode heet reservation methode.



figuur 2.7 - TDM synchroon/asyncroon

De meeste LAN-MAC methoden zijn echter asynchroon, met als groot voordeel dat de bericht lengte variabel mag zijn. (De tijdsduur is nu niet meer van belang) Asynchroon werken betekent echter ook dat er een mechanisme moet zijn dat de aanwezigheid van twee of meer kanalen dient te voorkomen. De twee mogelijke strategieen zijn: random access en controlled access.

Random access berust op de methode dat ieder station zelf bepaalt wanneer het wil gaan zenden. Het kriterium hiervoor is, dat alvorens het medium te bezetten, elk station eerst moet verifieren dat geen ander station bezig is een bericht uit te zenden.
Random access werd het eerst toegepast onder de naam ALOHA.

Bij controlled access is er een duidelijk protocol dat de volgorde van zenden onder de aànwezige stations verdeelt.

Controlled access kan worden verdeeld in distributed of centralized. access.

Gecentraliseerd wil zeggen, dat er een master station aanwezig is. Deze master regelt het gehele verkeer op het medium.(Polling)

Bij distributed control is er geen master station meer maar wordt de toestemming om het medium te gebruiken bepaald door een z.g. token. Een token is een uniek kenmerk dat onder de stations rouleert en dat het recht verschaft om het medium te gebruiken. Na gebruik wordt het token doorgegeven aan een volgend station. Een token bestaat meestal uit een zeer kort uniek bericht.
Tabel 2.2 geeft een overzicht van de meest gebruikte netwerken, ingedeeld naar topologie & access methode.

MAC Methode	BUS	RING
Random Access	CSMA/CD	Register insertion
Centralized C.	Multipoint	Loop
Distributed C.	Token	Token(zonder addressering)

tabel 2.2 - Typen netwerken

De werking van deze verschillende netwerk-typen zal nu kort worden toegelicht.

CSMA/CD

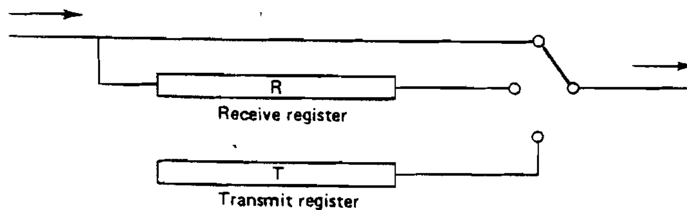
Dit staat voor Carrier Sensed Multiple Access/Collision Detection, ofwel ieder station moet, voordat het wil gaan zenden testen of het medium vrij is (carrier sensing). Pas dan mag het station gaan zenden. Omdat dit voorschrift niet garandeert dat de overige stations niet gelijktijdig (of binnen een tijdsinterval, gelijk aan de looptijd van het signaal over het net) met zenden zijn begonnen, moet er ook gedurende het zenden konstant worden getest of het bericht niet "botst" met een bericht van een ander station. (Collision Detection) Indien zo'n collision optreedt, wordt het zenden onmiddellijk gestopt en er wordt dan een random tijd gewacht, alvorens het opnieuw te proberen.

CSMA/CD is een veelvuldig toegepaste access methode, die ook wel de naam ETHERNET draagt. Een meer gedetailleerde beschrijving is terug te vinden in [lit4]

Register insertion

Wanneer er sprake is van een ring topologie heet de daarbij behorende random access methode register insertion. Bij deze methode bezit ieder station twee registers, elk ter grootte van een frame. In normaal bedrijf zijn beide registers uit de ring geschakeld.

Als een station wil gaan zenden, wordt het register T met de te verzenden data gevuld en op het moment dat de lijn vrij komt wordt het register op de lijn geplaatst. (Schakelaar op T) Eventuele data die gedurende het zenden binnenkomt, wordt gebufferd in het schuifregister R, zodat geen data verloren gaat.

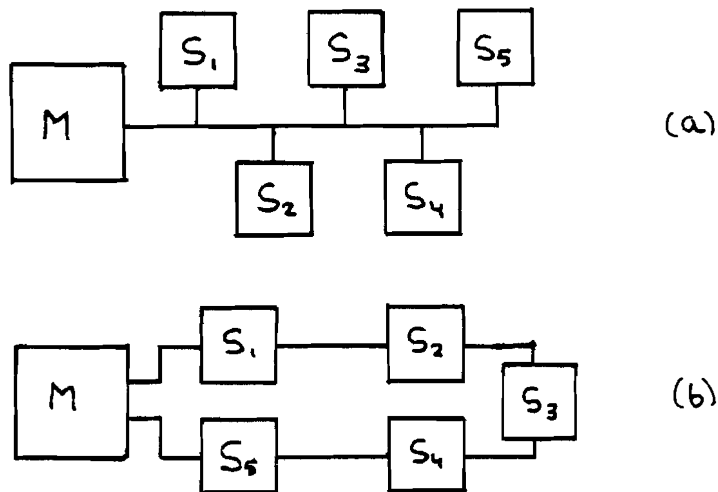


figuur 2.8 - Register insertion

Nadat het register T verzonden is, wordt de schakelaar op R gezet. Register R fungeert dan als shuifregister.
Vervolgens wordt er gewacht totdat het frame langs alle stations geweest is , m.a.w. weer terugkeert in register R. Op dat moment wordt de schakelaar weer op N gezet, waardoor het frame uit de ring wordt verwijderd.
Opgemerkt dient te worden dat de "round-trip" tijd van een frame evenredig is met het aantal in de ring geschakelde registers.

Multipoint

Figuur 2.9 geeft de schematische opbouw van een multipoint netwerk. Het master station (M) vraagt aan alle andere stations d.m.v. een poll frame of deze een bericht te versturen hebben. Alle stations ontvangen om de beurt een poll frame en mogen dan een bericht zenden of sturen een ontkenning terug naar de master, ten teken dat het station niets te zenden heeft.



figuur 2.9 - Multipoint (a) en Loop (b) configuratie

Loop

Bij een loop netwerk wordt er telkens een broadcast poll verstuurd, dat via de ring langs alle stations loopt.
De stations antwoorden op dezelfde wijze als bij het multipoint netwerk, met dit verschil dat de volgorde van antwoord geven nu is vastgelegd door de physische plaats in de ring.
Het voordeel van de loop t.o.v. multipoint is een efficienter gebruik van het medium, doordat slechts een poll frame per cyclus wordt gegenereerd.

Tokenbus

Bij een tokenbus is er geen master station meer, maar in plaats hiervan circuleert er een uniek bericht over de bus, het z.g. token.

Een station dat het token in bezit heeft, mag gebruik maken van de bus en behoort nadien het token door te geven aan het volgende station. Doordat het token rouleert langs alle stations, wordt de token bus ook wel een "logische ring" genoemd. (alle stations hebben een logisch volgnummer)

Alhoewel het principe vrij simpel is, is de realisatie van het protokol nogal komplex. De oorzaak ligt in het feit dat alle fouten die kunnen ontstaan door de stations gezamelijk moeten worden opgelost. Dat dit problemen kan veroorzaken wordt door de onderstaande voorbeelden verduidelijkt.

- Als een station wil inloggen op het netwerk zal dit station niet automatisch een token ontvangen, omdat geen van de overige stations zich bewust is van zijn aanwezigheid. Dit betekent dat ieder nieuw station d.m.v. een stoorsignaal het token bericht moet vernietigen, waarna het netwerk opnieuw dient te worden opgebouwd.

- Een eenmaal ingelogged station raakt door een storing plotseling "off- line". Het gevolg hiervan is dat het token bestemd voor dit station niet meer beantwoord wordt en het dient door een ander station (welk ???) verwijderd te worden.

- Er zijn twee of meer tokens op de bus aanwezig. Welk station neemt het initiatief om weer orde op zaken te stellen ?

Kortom, de token bus is voor wat betreft het MAC protokol niet eenvoudig te realiseren. De meeste LAN's die gebruik maken van de token techniek hebben dan ook meestal een master station die het token beheert, waardoor de tokenbus in de meeste gevallen degradeert tot een multipoint netwerk, dat een lagere performance heeft.

De Tokenring

De stations zijn nu opgenomen in een ring. Omdat een impliciete adressering van het token nu niet meer nodig is (logische ring=physische ring) is het probleem van in/uitloggen vervallen.

Om het netwerk echter niet te traag te maken mogen de stations de frames niet bufferen, maar moet het frame "real-time" (of met max. 1-2bit vertraging) worden uitgelezen. Dit heeft nogal wat konsekwenties voor de hardware, die dan ook vrij duur is.

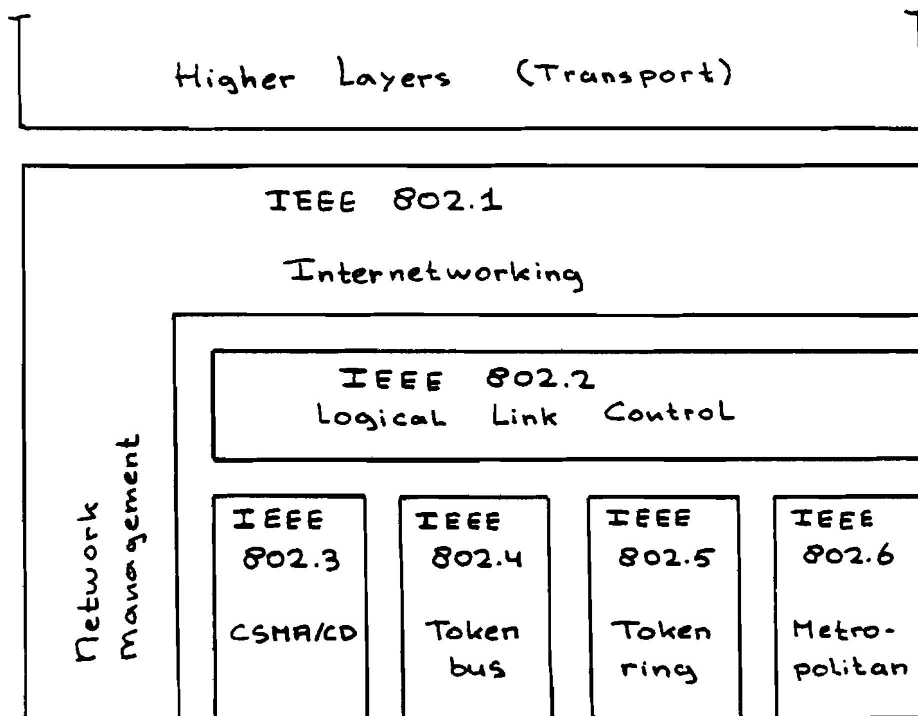
Token ring netwerken worden op het ogenblik door IBM ontwikkeld als netwerk voor hun mainframes.

2.2.2.5 Logical Link Control

Deze laag is bedoeld als universele interface naar de hogere lagen. De LLC laag is onafhankelijk van de gebruikte access methode.
Op de funktie van de LLC laag wordt in een van de volgende hoofdstukken nader ingegaan.

2.3 LAN STANDAARDS

Medio 1982 verschenen de eerste voorstellen vanuit de IEEE-802 werkgroep, welke een eerste stap in de richting van een nieuwe standaard betekenden.
Op het ogenblik (medio 1984) bestaat het komplete werk uit 6 hoofdstukken (IEEE-802.1 t/m IEEE-802.6), waarbij ieder hoofdstuk een bepaald, afgebakend deel beschrijft. Figuur 2.10 geeft een overzicht van het het IEEE-802 werk en laat de 4 nieuwe standaards zien, t.w.



figuur 2.10 - De IEEE-802 standaards

CSMA/CD Ookwel ETHERNET genoemd en is oorspronkelijk ontwikkeld door XEROX. De CSMA/CD methode wordt tegenwoordig sterk ondersteund door firma's als INTEL en Digital.

Tokenbus Waarschijnlijk als standaard gekozen vanwege het succes van het ARCNET van DATAPIONT.

Tokenring Deze standaard is mede "op aanraden" van IBM in het IEEE-802 voorstel opgenomen.

Metropolitan
Dit is een standaard voor een regionaal netwerk. De omvang van een dergelijk netwerk bedraagt 20 tot 40 km. Een uitgewerkt voorstel van deze nieuwe standaard bestaat er nog niet.

Op het ogenblik is de CSMA/CD en de Tokenbus standaard definitief door IEEE goedgekeurd en ook de LLC-standaard is grotendeels aanvaard. Tabel 2.3 geeft een meer gedetailleerd overzicht van de diverse voorstellen en laat ook de verschillende topologieen en signallings methoden zien.

	OSI layer	IEEE sublayer	Option							
2	Data link	Logical link control (LLC)	Station classes: I —connectionless operation only II —connectionless and connection-orientated operation III* —connectionless and acknowledged connectionless operation							
			Service access points: One (null address is used) Several							
			Duplicate MAC service access point address detection procedure*							
		Medium access control (MAC)	Addresses: 16 bit 48 bit—locally administered —universal product code							
			Capacity sharing algorithm: CSMA/CD			Token bus			Token ring	
1	Physical		Speed (bit/s)			Speed (bit/s)			Speed (bit/s)	
										40M
								20M		20M
			10M	10M	10M		10M	10M		
				5M*			5M	5M	4M	4M
				1M*		1M		1M	1M	
0	Medium		Cable: Coax		Optical fibre	Coax			Shielded pair	Coax
			Topology: Tree	Rooted tree	?	Bus with stubs	Bus with drops	Rooted tree	Ring	Ring
			Encoding: Manchester	Miller	?	Differential Manchester			Differential Manchester	
			Signalling: Baseband	Vestigial sideband	?	Phase continuous	Phase coherent	Phase shift keying	Baseband	
				Broad band		Frequency shift keying		Broad band		

Key:
*At draft C (IEEE, 1982) of May 1982 these items were provisional.

tabel 2.3 - Mogelijke uitvoeringen van de IEEE standaard

2.3.1 Vergelijking van de LAN standaards

Bij de vergelijking van de verschillende "access" methoden wordt veelvuldig gebruik gemaakt van de grootheden kanaalkapaciteit en de delay dat een bericht of pakket ondervindt.

Delay analyse wordt in de literatuur ook wel performance genoemd en is een maat voor de vertraging die een bericht ondervindt.
Bij de delay analyse wordt gebruik gemaakt van de wachttijd-en stagnatie theorie, waarbij het netwerk wordt gemodelleerd in een verzameling van "servers" en "queue's".
Het artikel van L.Li [lit5] gaat uitvoerig in op de performance-analyse van de tokenring en de tokenbus. Tevens wordt er een vergelijking gemaakt met de CSMA/CD methode.

De Kanaal kapaciteit zegt iets over de efficiency waarmee het physische medium wordt benut. Andere termen die men in de literatuur regelmatig tegenkomt en waar hetzelfde mee bedoeld wordt zijn grootheden als: throughput, maximum mean data rate en efficiency.

In het vervolg van dit hoofstuk zal ik een berekening maken van de maximum mean datarate , ofwel hoe efficient wordt het physische medium gebruikt.

De maximum mean datarate (MMD) van de 3 access methoden is uitvoerig geanalyseerd door de IEEE-802 werkgroep en vastgelegd in een 250 pagina's tellend rapport. [lit6]
Aangezien dit rapport nog niet gepubliceerd is, heb ik zelf een schatting gemaakt van de MMD, waarbij ik ben uitgegaan van de methode zoals gepresenteerd in het artikel van B.W.Stuck. [lit7]
Dit artikel geeft tevens enige resultaten uit het 802 rapport, zodat verifikatie van mijn resultaten mogelijk was.

Bij de afleiding ben ik uitgegaan van een eindig aantal stations N, waarbij ieder station zich in twee toestanden kan bevinden, nl idle of active.

idle: wil zeggen dat het station ingelogd is op het netwerk, maar niets te zenden heeft en dus alleen maar luistert.

active: wil zeggen dat het station aan het zenden is, of een bericht wil gaan verzenden. In de verdere afleiding wordt er vanuit gegaan dat een actief station nadat het een bericht verstuurd heeft onmiddellijk weer een bericht heeft dat verstuurd moet worden.

Verder wordt er van uitgegaan dat alle stations een verwaarloosbaar kleine respons-tijd bezitten.

Het aantal aktieve stations wordt aangeduid met n. (n = 0, .. ,N)
Tevens wordt verondersteld dat ieder bericht bestaat uit een vast aantal control- en databits. Controlbits zijn o.a de header van het bericht (bevat synchronisatie en adres gegevens) en de trailer. (voor error detektie en end-of-frame synchronisatie)
Het aantal controlbits wordt aangeduid met Nc, het aantal databits met Nd.

De MMD wordt nu als volgt gedefinieerd:

$$\text{MMD} = \text{werkelijke bericht lengte} * \text{aantal berichten/sec}$$

Uit deze definitie blijkt dat MMD een soort effectieve baudrate is, zoals deze wordt waargenomen op het medium.

Het aantal berichten per sekonde is sterk afhankelijk van de gebruikte access methode, zoals in de verdere afleiding zal blijken.
Voor alle access methoden geldt echter dat:

$$\text{aantal berichten/sec} = \frac{1}{\text{Tmess} + \text{Tacces}}$$

Waarbij Tmess gelijk is aan:

$$\text{Tmess} = \text{Tdata} + \text{Tcontrol} = \frac{\text{Nd} + \text{Nc}}{\text{B}}$$

met B = de baudrate.

en Taccess de extra "overhead" is die volledig bepaald wordt door de gebruikte access methode.

Gebruik makend van bovenstaande definitie, zullen we nu de MMD van de verschillende access methoden berekenen.

TOKENRING

Voor de tokenring definieren we twee konstanten, nl:

Tprop = de looptijd van een bit over de totale lengte van het medium.

Tinterface = de tijd die een station nodig heeft voor "signaal processing". Bij een goed ontworpen token ring is dit 1 bittijd.

Het aantal berichten per sekonde dat door n aktieve stations verstuurd word is gelijk:

$$\#berichten/sec = \frac{1}{Tmess + Ttoken}$$

Ttoken is de tijd die verloopt tussen het versturen (=doorgeven) van de token en het weer opnieuw in bezit komen van de token. Bij de tokenring is vrij eenvoudig in te zien dat

$$Ttoken = \frac{N}{n} \left(\frac{Tprop}{N} + Tinterface \right) = \frac{Tprop}{n} + \frac{N}{n.B}$$

Waarbij Tprop konstant wordt verondersteld, hetgeen betekent dat alle stations met gelijke onderlinge afstand over de ring verdeeld zijn.
Dit alles ingevuld in de definitie van de MMD geeft:

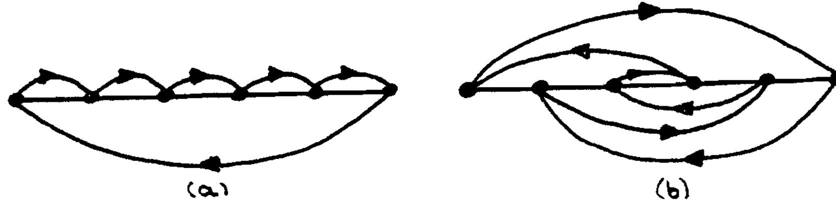
$$MMD = \frac{Nd}{\frac{1}{B} \left(Nd + Nc + \frac{N}{n} \right) + \frac{Tprop}{N}} \qquad (1)$$

TOKENBUS

Hiervoor geldt uiteraard dezelfde redenering als bij de tokenring. De tijd Ttoken is nu echter niet meer zo eenvoudig te bepalen, omdat de volgorde waarin de token wordt doorgegeven nu willekeurig kan zijn. Dit heeft tot gevolg dat Tprop niet meer konstant is ! Wat echter altijd geldt is:

$$Ttoken = \frac{1}{n} Tloop + (N + 1 - n).Tinterface$$

met Tloop de fysische looptijd van de token over het medium. Voor deze tijd is een onder- en een bovengr aan te geven.



figuur 2.11 - tokenbus best-case (a) en worst-case (b)

De ondergrens verkrijgen we als de token telkens aan het dichtsbijzijnde station wordt doorgegeven, terwijl de bovengrens van Tloop onstaat als de token telkens aan het verst afgelegen station wordt doorgegeven. In figuur 2.11 is e.e.a. weergegeven.

Voor de ondergrens geldt:

$$Tloop = (N - 1)\left(\frac{Tprop}{N - 1}\right) + Tprop = 2\ Tprop$$

hetgeen onmiddellijk volgt uit fig. 2.11a

Voor de bovengrens moeten we nog een onderscheid maken tussen een even en een oneven aantal stations. Uitgaande van fig. 2.11b is de afleiding vrij eenvoudig op te schrijven en omdat het een cyclisch probleem is maakt het niet uit bij welk station we beginnen te tellen. Altijd zal gelden:

$$Tloop = \left[(N - 1) + (N - 2) + \ldots.. + 1 + \frac{N}{2}\right] \frac{Tprop}{N - 1}$$

voor N = even. Als N = oneven geldt er:

$$Tloop = \left[(N - 1) + (N - 2) + \ldots.. + 1 + \frac{N - 1}{2}\right] \frac{Tprop}{N - 1}$$

Dus

$$Ttoken = \frac{2}{n}\ Tprop + (N + 1 - n)Tinterface \qquad \text{(lowerbound)}$$

$$T_{token} = \frac{N^2}{2n(N-1)} T_{prop} + (n+1-n)T_{interface} \quad \text{(upperbound)}$$

zodat

$$MMD = \frac{Nd}{\frac{1}{B}(Nd + Nc) + (N+1-n)T_{int} + \frac{N^2}{2n(N-1)} T_{prop}} \quad (2)$$

$$MMD = \frac{Nd}{\frac{1}{B}(Nd + Nc) + (N+1-n)T_{int} + \frac{2}{n} T_{prop}} \quad (3)$$

waarbij (3) de bovengrens en (2) de ondergrens van de MMD aangeeft.

CSMA/CD (Ethernet)

Hiervoor geldt:

$$\# \text{ berichten/sec} = \frac{1}{T_{mess} + T_{interframe} + T_{cont}}$$

Tinterframe
: is de minimale tijd tussen twee opeenvolgende frame's. Voor ethernet is deze waarde vastgelegd op 96 bits, hetgeen overeenkomt met 9,6us voor een 10Mb/s systeem.

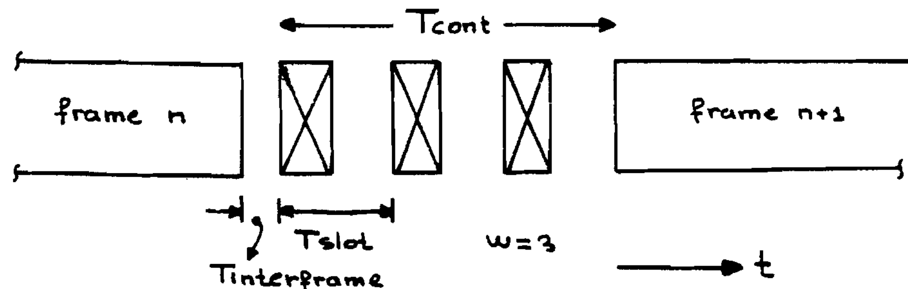
Tcont
: is de tijdsduur van het contention-interval, ofwel de totale botsings-tijd. (collision-tijd) Deze tijdsduur is uiteraard afhankelijk van het aantal aktieve stations n. Volgens het ethernet protokol moet ieder station na k opeenvolgende botsingen een tijd w * Tslot wachten, alvorens het opnieuw te proberen.

Het getal w is een random getal uit de reeks: 1 .. 2k. Tslot is bij ethernet gestandariseerd als 2 * Tprop + Tinterface, zodat geldt: Tcont = W * Tslot met W het gemiddeld aantal collisions. (fig. 2.12)

Dit gemiddeld aantal botsingen W bij n aktieve stations kan als volgt worden berekend:
Omdat gedurende het contention interval ethernet zich gedraagt als "slotted ALOHA", is de kans Q dat er slechts een station start met zenden in een gegeven slot gelijk:

$$Q = n * p (1 - p)^{n-1}$$

met n het aantal aktieve stations en p de kans dat een aktief station start met zenden.



figuur 2.12 - CSMA/CD contention interval

Bij ethernet is geprobeerd om Q maximaal te laten worden door voor ieder station p aan te passen aan de belasting van het net. Deze procedure bij ethernet heet Binary Exponential Backoff (BEB) en is in bovenstaande tekst reeds verklaard. (Na k botsingen wordt p gelijk 2**-k)
Het BEB algoritme zal voor het stationaire geval de optimale waarde van p (= 1/n) altijd bereiken. Dit geldt in feite voor ieder backoff algoritme, het verschil zit in de snelheid waarmee dit gebeurt. Appendix A geeft een verklaring van het feit dat ieder backoff algoritme een waarde voor p gelijk aan 1/n oplevert.

Als we uitgaan van deze veronderstelling, kunnen we de kans op i botsingen berekenen. We definieren hiertoe:

w(i) = de kans dat het contention interval uit i slots bestaat.
w(0) = Q (per definitie !)
w(1) = Q(1 - Q)
.
.
$w(i) = Q(1 - Q)^i$

en dus $W = \sum_{i=0}^{\infty} i\,w(i) = \sum_{i=0}^{\infty} i\,Q(1 - Q)^i = \frac{1 - Q}{Q}$

ingevuld in de formule voor MMD geeft dit:

$$MMD = \frac{Nd}{\frac{Nd + Nc}{B} + \frac{96}{B} + \frac{1 - Q}{Q}\left(2 * Tprop + \frac{96}{B}\right)} \qquad (4)$$

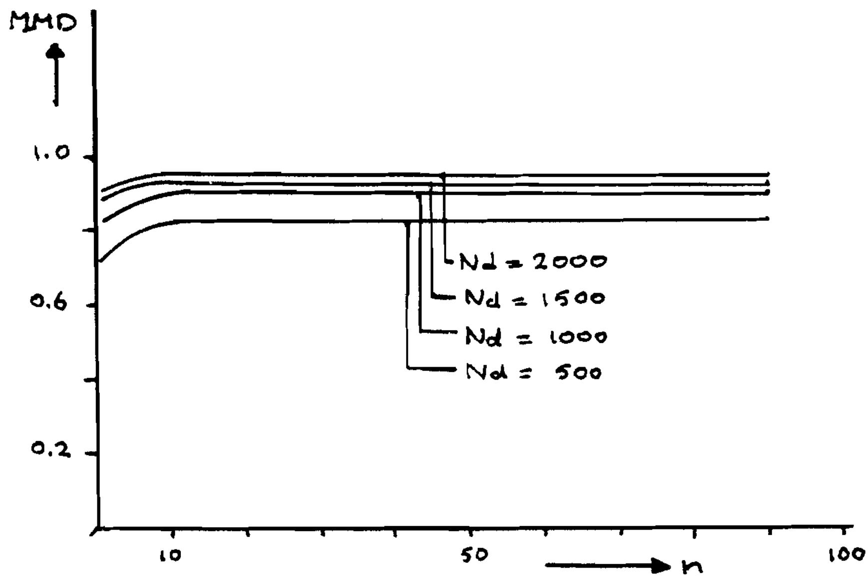
$$\text{met } Q = (1 - 1/n)^{n-1}$$

Resultaten

In de figuren 2.13 t/m 2.16 zijn de resultaten van de verschillende access methoden uitgezet. Voor alle resultaten geldt de volgende aanname:

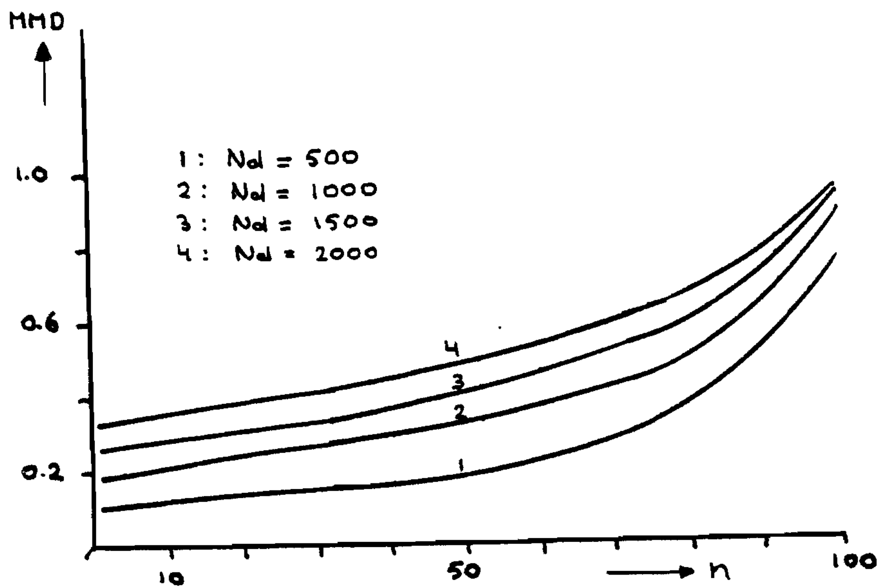
- Een baudrate van 10 Mb/s
- Een propagatie tijd van 10 us, hetgeen overeen komt met een kabelsegment van zo'n 300 meter.
- Een totaal van 100 ingelogde stations.
- Het aantal control bits per bericht = 96 (Nc=96)

Wat onmiddellijk opvalt is dat de efficiency van alle access methoden beter wordt bij grote bericht lengte. Dit is ook vrij logisch omdat de overhead voor wat betreft "token passing" c.q. het "contention interval" relatief kleiner wordt.

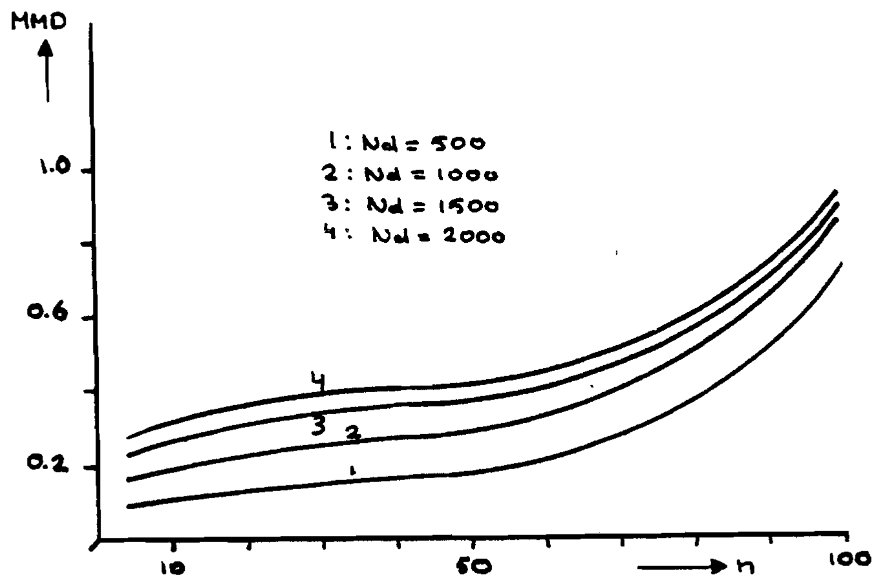


figuur 2.13 - performance tokenring

Verder blijkt dat de tokenring de beste performance heeft. (uitgezonderd het geval van slechts een aktief station)



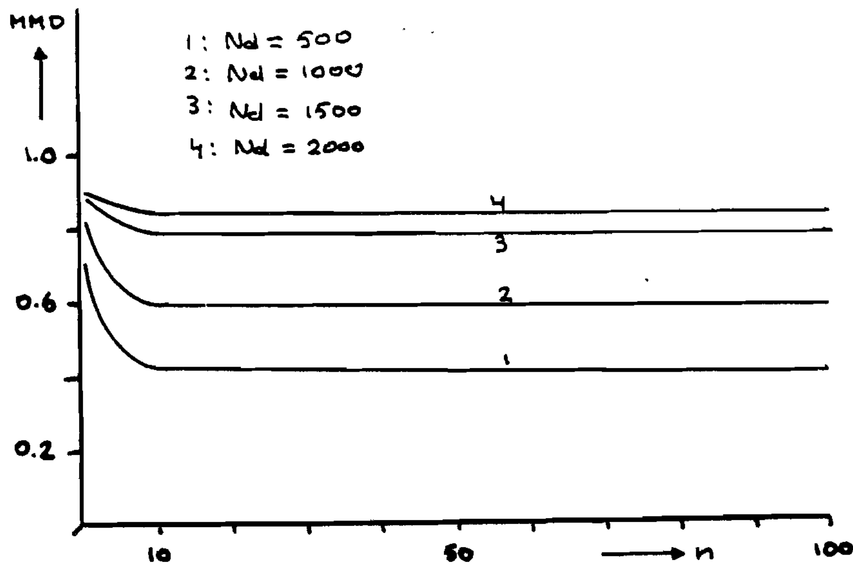
figuur 2.14 - performance tokenbus bestcase



figuur 2.15 - performance tokenbus worstcase

Bij de tokenbus zien we dat de volgorde waarin de token wordt doorgegeven alleen bij een klein aantal gebruikers enige invloed heeft. Algemeen geldt voor de tokenbus dat de efficiency erg gevoelig is voor het aantal gebruikers.

CSMA/CD geeft bij een aktieve gebruiker de beste efficiency, hetgeen vrij logisch is vanwege de minimale overhead.
Boven een bepaalde grenswaarde van het aantal aktieve stations blijft de MMD ongeveer konstant, wat een gevolg is van het BEB algoritme.
In vergelijking met de tokenbus blijkt dat boven een bepaalde grens CSMA/CD slechter is dan de tokenbus. Opvallend is dat deze grenswaarde niet afhankelijk is van de bericht lengte, maar uitsluitend bepaald wordt door het aantal aktieve stations.
Bij de gekozen parameters, blijkt dat tot 88 aktieve stations CSMA/CD efficienter is dan de tokenbus.



figuur 2.16 - performance CSMA/CD

Tabel 2.4 geeft samengevat de verschillende resultaten nog eens weer.

	+	-
CSMA/CD 802.3	Als standaard aanvaard en gesupport door Intel Xerox en Digital. Hardware beschikbaar.	Geen real-time. Bij zeer veel aktieve gebruikers matige performance.
Tokenbus 802.4	Bij zeer veel aktieve gebruikers betere performance dan CSMA/CD	(Nog) geen hardware beschikbaar. complexe MAC laag.
Tokenring 802.5	Beste performance. gesupport door IBM.	Nog geen uitgewerkte standaard. Dure en complexe hard-nodig.

tabel 2.4 - Vergelijking IEEE standaards

2.4 BESCHIKBARE LAN HARDWARE

Dit hoodstuk tracht een overzicht te geven van de hardware (chips) die op dit moment beschikbaar zijn, of door de fabrikanten zijn aangekondigd. Voor wat betreft beschibare LAN hardware kan ruwweg een onderscheid worden gemaakt tussen 3 katagorien van chips, t.w:

- token passing chips
- CSMA/CD chips
- overige, niet tot de standaard behorend.

2.4.1 token passing chips

ARCNET Tokenpassing controller SMC 9026

ARCNET is een veel toegepast LAN dat door DATAPOINT midden 70-jaren ontwikkeld is. Het is een 2,5Mb/s "modified" token passing controller voor gebruik op een bus of een ring, waarbij tot 255 stations op het netwerk kunnen worden aangesloten.
De chip maakt gebruikt van een extern dual ported RAM buffer, waarin maximaal 4 frames van ieder 508 bytes kunnen worden opgeslagen. De SMC 9026 bevat een hardware oplossing voor het "token management" probleem.
In [lit8] staat een beschrijving van een netwerk rondom deze chip.

WD 2840 van Western Digital

Ook dit is een niet tot de IEEE standaard behorende token controller. De chip heeft een maximale baudrate van 1Mb/s. (Bij deze chip behoort nog een manchester coder/decoder type HD 6409 van Harris)
Verder wordt er bij Western Digital momenteel hard gewerkt aan een standaard IEEE-802.4 chip die medio 1986 op de markt moet komen.

2.4.2 CSMA/CD chips

Voor CSMA/CD toepassingen zijn momenteel 6 chip sets op de markt dan wel aangekondigd.

MOSTEK MK68590/AMD AM7990

Deze chips zijn identiek en voldoen geheel aan de IEEE-802.3 standaard. Naast deze controller chips leveren beide firma's

ook nog de encoder/decoder chips. Dit zijn de MK3991 van Mostek en de AM7791 van AMD.
De extra hardware nodig om deze chips te koppelen met een 68000 of 186 processor is minimaal. (1 pal 16L8)
AMD heeft tevens een IEEE-802.3 transceiver aangekondigd, met als type aanduiding AM7995/6. Deze transceiver is echter niet leverbaar voor 1986.

Intel's 82586
Dit is momenteel de meest flexible 802.3 chip op de markt. De chip lijkt erg veel op die van MOSTEK/AMD, maar is veel uitgebreider programmeerbaar. In [lit9] staat een uitgebreide vergelijking van deze chips. Naast de standaard konfiguratie kan deze chip voor een groot aantal verschillende CSMA/CD methoden worden geprogrammeerd.
De bijbehorende coder/decoder is de 82501 van Intel.

Seeq 8001/8002 en 8003/8023
De 8001 was de eerste chip op de markt die geschikt was voor ethernet toepassingen. (ethernet versie 0) Een verbeterde uitvoering is de 8003 en deze chip is 802.3 compatible.
Beide chips vergen nogal wat extra hardware, zoals een eigen "prive" data buffer.
De 8002 en 8023 zijn de bij deze chips behorende Manchester coder/decoders.

National Semiconductor
Deze firma heeft 3 chips aangekondigd die aan de nieuwe IEEE standaard voldoen. Het zijn de typen 8390/8391 en 8392.
Vooral de 8392 is een interessante chip, omdat dit de eerste transceiver chip (PMA) op de markt is.
De chip kan rechtstreeks een coax kabel aansturen en verzorgt ook de collision detektie.
De 3 chips zijn door N.S. ontwikkeld voor de z.g. <u>cheapernet</u> standaard. (tabel 2.5)
Cheapernet is een "gestripte" versie van Ethernet en maakt gebruikt van eenvoudige (en dus goedkopere) hardware. Het grote verschil met Ethernet is voornamelijk de goedkopere coaxkabel en het ontbreken van de dure ethernet transceiver.

Parameter	IEEE-802.3	Cheapernet
Data Rate	10 Mbit/s	10 Mbit/s
Segment lengte	max. 500 meter	max. 180 meter
Tot. grootte	2,5 km	1 km
Nodes per segment	100	30
Nodes per netwerk	1024	1024
Node afstand	2,5 meter	0,5 meter
Kabel koppeling	0,4 inch 50 ohm coax met speciale N-type connectors	0,25 inch 50 ohm coax verbonden met BNC-connectors
Transceiver-interface	0,38 inch multiway kabel met D-type connector	N.V.T.

tabel 2.5 - Ethernet versus Cheapernet

Fujitsu

Deze firma levert de ethernet controller type MB8795A. Het is een eenvoudige chip en te vergelijken met de SEEQ 8001 qua prestatie.
Naast deze chip levert Fujitsu nog de coder/decoder MB502A

Rockwell 68802

Van deze chip heb ik geen informatie kunnen bemachtigen, alleen maar de mededeling dat de chip IEEE 802.3 compatible zou zijn.

2.4.3 Overige chips

Omdat alle bovengenoemde LAN chips aan het begin van mijn afstudeer periode niet of zeer moeilijk verkrijgbaar waren, is ook nog gezocht naar een alternatieve oplossing, gebruik makend van wel verkrijgbare "communicatie" chips.
Deze oplossing zou gerealiseerd moeten worden met zo weinig mogelijk extra hardware en een baudrate tot 1Mb/s. Onderstaande

chips zijn mogelijke kandidaten om zo'n alternatief netwerk te realiseren.

Intel 82530 en 8274

Dit zijn twee Serial Multiprotocol Controllers. Beide chips hebben een aantal gemeenschappelijke funkties, zoals:

- Standaard SDLC en HDLC frame handling
- X.25 compatible
- automatische CRC-generatie en controle.

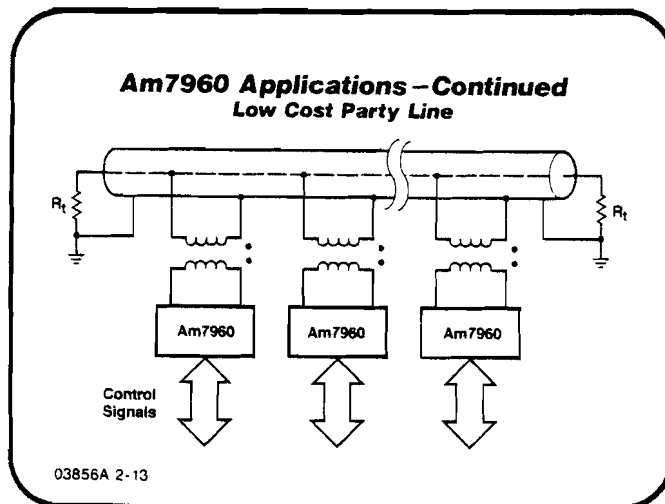
De 82530 heeft nog wat extra faciliteiten, waaronder:

- "On-chip" baudrate generator.
- Diagnostic Local loopback
- Manchester coder/decoder + digitale PLL voor clock recovery

Dit laatste ontbreekt bij de 8274, zodat deze chip nog een extra chip vereist.
Een interessante chip hiervoor is de HD 6409 van Harris. Dit is een 1Mbaud Manchester coder/decoder met eveneens een digitale PLL.

AMD 7960 Coded Data Transceiver

Deze chip is speciaal bedoelt voor communicatie systemen met een gemeenschappelijke bus struktuur.



figuur 2.17 - AMD 7960 bus struktuur

De chip bevat een Manchester coder/decoder en heeft een maximale bitrate van 3 Mbaud De 7960 kan genoeg vermogen leveren om rechtstreeks een coaxiale kabel aan te sturen (tot 1 km)

Signetics 5080 en 5081

Dit is een high speed FSK modem transmitter (5080) en receiver (5081). De chips zijn bedoeld als PMA-device in de IEEE-802.4 standaard en kunnen rechtstreeks aan een coax kabel worden gekoppeld.
Bij een maximale snelheid van 2 Mbaud kan tot 1,5 km een standaard coax kabel worden aangestuurd. Bij gebruik van een kwalitatief hoogwaardige kabel is een bereik tot 4,5 km mogelijk.

2.5 DE KEUZE VAN DE HARDWARE

Als uiteindelijke oplossing is gekozen voor de IEEE-802.3 standaard (Ethernet). Deze keuze is mede gemaakt op grond van de volgende overwegingen:

- Het netwerk moest voldoen aan een van de 3 IEEE standaard voorstellen.

- Het netwerk dient voor de koppeling van een aantal microcomputers (THE KUNix machine) en is daardoor vrij beperkt van omvang. Gedacht wordt aan zo'n 20-40 gebruikers. Op grond hiervan zijn de tokenbus en ethernet het meest geschikt.
  De tokenring is een te dure oplossing en is meer bedoeld voor de grotere mainframes.

- Bij de afweging tussen tokenbus en ethernet is uiteindelijk voor de laatste gekozen, omdat:

  a Er alleen nog maar hardware beschikbaar was voor de ethernet standaard (Intels's 82586) en nog niet voor de tokenbus.

  b De performance van ethernet is bij een relatief klein aantal gebruikers beter dan de tokenbus.

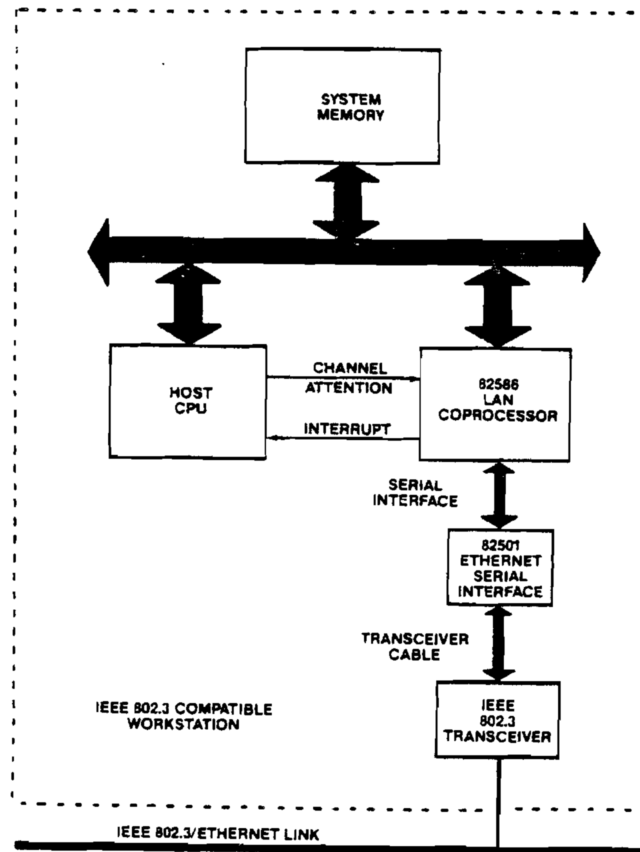
  c Er zijn steeds meer fabrikanten die ethernet als standaard LAN gebruiken (IBM PC).

  d De Intel 82586 is op het moment de meest "sophisticated" ethernet chip op de markt.

III DE ETHERNET HARDWARE

Dit hoofdstuk behandelt de hardware die gebruikt is om de ethernet node te realiseren.
Zoals reeds vermeld, is gekozen voor de Intel 82586 LAN coprocessor. (In het vervolg aan te duiden met 586)
Met de keuze van deze chip ligt ook de hardware struktuur van de ethernet node vast. In figuur 3.1 is deze struktuur weergegeven.

Omdat het netwerk bedoeld is voor de THE KUNix machine, kan de host CPU van het type 68000 of iAPX 186 zijn. In eerste instantie is echter alleen de implementatie met de iAPX 186 gerealiseerd. De reden hiervoor was dat er alleen iAPX 186 hardware beschikbaar was (prototype 0).



figuur 3.1 - De Ethernet node met de 82586

In [lit10] wordt een beschrijving gegeven van deze implementatie en wordt een oplossing gegeven voor de "tijdelijke" koppeling van twee ethernet nodes. Op deze in dit verslag beschreven en gerealiseerde hardware oplossing zijn gedurende de test fase nogal wat vereenvoudigingen aangebracht en vanaf prototype 1 is er een zeer eenvoudige oplossing gevonden, gebruik makend van slechts 2 extra IC's om de 586 met de 186 te koppelen.

Naast de 586 is er nog een serial interface noodzakelijk. Deze interface zorgt voor de lijn codering en de klok-generatie.

De transceiver of PMA verzorgt de koppeling aan het medium. Bij baseband CSMA/CD bestaat dit medium uit een 50 ohm coaxiale kabel.

De eigenschappen en werking van de 586, de serieele interface en de transceiver zal in de volgende hoofdstukken worden besproken.

3.1 DE INTEL 82586 LAN COPROCESSOR

De 586 is een universele CSMA/CD controller, die voldoet aan de IEEE 802.3 eisen.
De 586 is een coprocessor in die zin dat de chip geen (besturings) registers bevat zoals een normale periferie-chip, maar kommuniceert met de host processor via een gemeenschappelijk deel van het geheugen.
De chip werkt dan ook volledig parallel aan de host en de enige "rechtstreekse" kommunicatie lijnen zijn de interrupt (van 586 -> host) en de channel attention (CA) lijn. (CA = interrupt van host -> 586)

De 586 kan twee verschillende taken verrichten (en dat dan ook weer parallel), t.w:

1) Het ontvangen van frame's en de data van deze frame's in vooraf gespecificeerde receive buffers plaatsen.
   Dit receive proces wordt afgehandeld door de Receive Unit (RU).

2) Het uitvoeren van door de host opgedragen commando's.
   De verwerking van deze commando's gebeurt door de Command Unit (CU) en er zijn een 8-tal verschillende commando's beschikbaar, nl:

   NOP — Dit commando lijkt op het eerste gezicht niet erg zinvol, maar zoals bij de behandeling van de software zal blijken is dit in sommige gevallen toch een nuttig commando.

Setup Individual Address
: Met dit commando kan het unieke stations-adres worden geladen.

Configure
: Hiermee kan de 586 worden geprogrammeerd, zodat ook andere modes dan IEEE 802.3 mogelijk zijn. (De default configuratie van de 586 is conform de IEEE-802.3 voorschriften).

Setup Multicast Address
: Voor het laden van multicast en/of groeps-adressen.

Transmit
: Met dit commando kan telkens een frame worden verzonden. Indien er meerdere frames verzonden dienen te worden, kunnen d.m.v. een "linked list" meerdere transmit commando's aan elkaar worden gekoppeld.

TDR
: Dit is het Time Domain Reflectometry commando. Hiermee kan de coaxiale kabel worden getest op defekten, zoals kortsluiting en onderbrekingen.

Diagnose
: Dit is een zelf-test commando waarmee de interne hardware van de 586 kan worden gekontroleerd.

Dump
: Met dit commando worden alle (werk)registers van de 586 in het geheugen gedumpt. (Totaal zo'n 160 bytes)

Verder bevat de chip alle logika om het CSMA/CD protokol af te handelen, waarbij dan m.b.v. het configure commando uit verschillende mogelijkheden gekozen kan worden. [lit12] geeft een zeer uitgebeide beschrijving van dit commando en alle opties die mogelijk zijn.

3.1.1 Het gemeenschappelijk geheugen (shared memory)

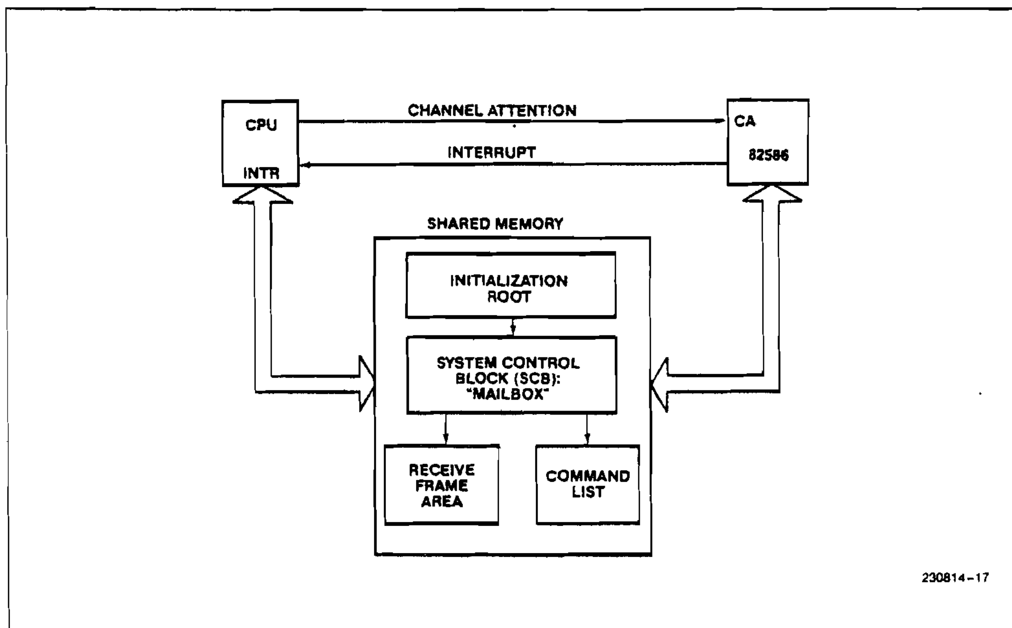


Figure 2-4. 82586/Host CPU Interaction

figuur 3.2 - Het gemeenschappelijk geheugen.

Bovenstaande figuur geeft schematisch de opbouw van het gemeenschappelijk geheugen, zoals dat door de 586 en 80186 gebruikt wordt.
De kern wordt gevormd door het System Control Block (SCB) dat als een mailbox fungeert. Het SCB kan via een initialisatie sequence op een willekeurige plaats in het geheugen worden gefixeerd.
De opbouw van het SCB is als volgt:

STAT	CUS	RUS	Status word
ACK	CUC	RUC	Command word
Pointer naar CL			
Pointer naar RFA			
# CRC-errors			Statistics
# Alignment-errors			"
# Resource-errors			"
# Overrun-errors			"

Het status veld en de 4 error velden, worden alleen door de 586 beschreven. Het statusveld is weer onderverdeeld in 3 kleinere velden, t.w

Receive Unit Status (RUS)
Command Unit Status (CUS)
Interrupt status (STAT)

Het commando woord en de beide pointers worden door de host in het SCB geplaatst. Evenals het statusveld is ook het commandveld weer in 3 afzonderlijke velden gesplitst. Het ACK veld dient als acknowledge voor de interrupts aangegeven door het STAT veld.

De kommunikatie tussen 586 en host verloopt volgens een "handshake" protokol, waarbij alle berichten via het SCB worden uitgewisseld. De procedure is als volgt:

a) Van host naar 586
De host plaats de commando's voor RU en/of CU in het commando veld en genereert een channel attention.
De 586 op zijn beurt zal vervolgens dit commando lezen en uitvoeren.
De 586 zal als "handshake" het commando woord onmiddellijk na het lezen wissen. Voor de host is dit het teken dat het SCB door de 586 gelezen is en opnieuw gebruikt mag worden.

b) Van 586 naar host
Er zijn 4 oorzaken waardoor de 586 door middel van een interrupt de attentie van de host zal vragen, nl:

- Er is een commando uitgevoerd waarvan het interrupt bit was gezet. (I-bit)

- Er is een frame ontvangen.

- De CU heeft alle commando's uitgevoerd en wacht op de volgende opdracht.

- De RU is gestopt met het ontvangen van frames. (bijv. doordat er onvoldoende buffer ruimte is om de frames in op te slaan.)

Alvorens de interrupt te plaatsen, zal de 586 in het STAT veld aangeven welke interrupt is opgetreden. (merk op dat er ook meer dan een interrupt tegelijkertijd kan optreden !)
De host moet vervolgens deze interrupt "acknowledgen". Dit gebeurt door in het ACK veld aan te geven welke interrupt(s) geacknowledged worden en vervolgens een CA te genereren.

Zoals uit figuur 3.2 reeds blijkt bestaat het gemeenschappelijk geheugen naast het SCB nog uit een Receive Frame Area (RFA) en een Command List (CL).

3.1.2 De receive frame area

De RFA bevat alle data strukturen die de RU gebruikt voor het ontvangen van frames. De RFA wordt initieel door de host processor aangemaakt en aan de 586 bekend gemaakt d.m.v. een pointer in het SCB.
Een overzicht van het RFA geeft figuur 3.3. De RFA bestaat naast de feitelijke receive buffers (waar uiteindelijk de ontvangen data terecht komt) uit 2 typen <u>descriptors</u> , t.w: <u>receive frame descriptors</u> (RFB) en <u>receive buffer descriptors</u> (RBD).
De RFD's vormen een linked-list, waarvan het begin wordt aangegeven door een pointer in het SCB. Het laatste RFD wordt gekenmerkt doordat het z.g. end-list bit in deze descriptor is gezet.
Na ieder ontvangen frame vult de RU een RFD met het source adres en type field van het frame. Ook eventuele fouten die gedurende ontvangst van het frame zijn opgetreden worden in de RFD vastgelegd.
Na ontvangst van een frame wordt automatisch het volgende RFD door de RU "geclaimed". Indien er geen RFD's meer beschikbaar zijn stopt de RU en komt deze in de "NO RESOURCE" state.
Het is dus zaak dat de host de RFA tijdig van nieuwe (lege) RFD's voorziet.

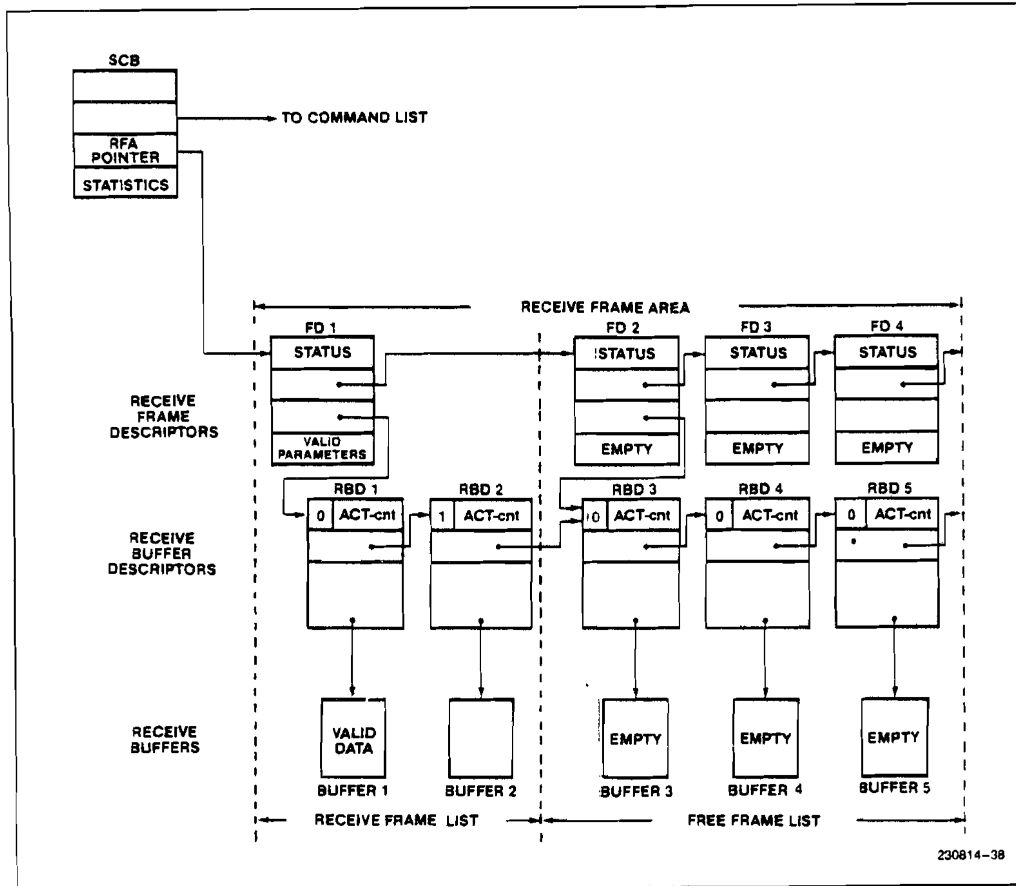


Figure 2-22. The Receive Frame Area

figuur 3.3 - Receive Frame Area

Evenals de RFD's vormen ook de RBD's een linked-list. Ieder RBD geeft een beschrijving van het bijbehorend receive buffer (plaats in het geheugen en grootte). Nadat een buffer door de RU met data is gevuld, wordt in het bijbehorend RBD aangegeven hoeveel data het buffer bevat en of dit het laatste buffer is dat bij het ontvangen frame behoord.

Dit laatste geeft o.a de kracht van deze 586 chip aan in vergelijking met andere ethernet controllers. Een ethernet frame heeft een variabele lengte (64 .. 1512 bytes). Doordat de 586 gedurende ontvangst van een frame nieuwe buffers "fetched", kan worden volstaan met een aantal "kleine" buffers i.p.v. buffers die altijd de maximale frame lengte moeten bezitten. Dit betekent een efficienter gebruik van de geheugen ruimte.

Welke RBD's op een gegeven moment nu bij een bepaald RFD behoren wordt door de RU aangegeven door in ieder RFD een pointer naar het eerste RBD te plaatsen. (Een uitzondering is de link tussen de

eerste RFD en eerste RBD, die door de host wordt gezet alvorens de RU te starten.)

Kort samengevat:

- De host kreeert initieel de RFA data struktuur, bestaande uit een linked list van RFD's en RBD's.

- De host zet in het SCB een pointer naar het eerste RFD en in dit RFD weer een pointer naar het eerste RBD.

- Hierna kan de RU worden gestart. De RU zal nu zelfstandig de receive buffers vullen, de RBD bijwerken, deze RBD's toewijzen aan een RFD en vervolgens deze RFD vrijgeven aan de host.

- De enige taak van de host is nu nog om tijdig nieuwe RFD's en RBD's toe te voegen om te voorkomen dat de RU in de NO RESOURCE state komt.

3.1.3 Command list

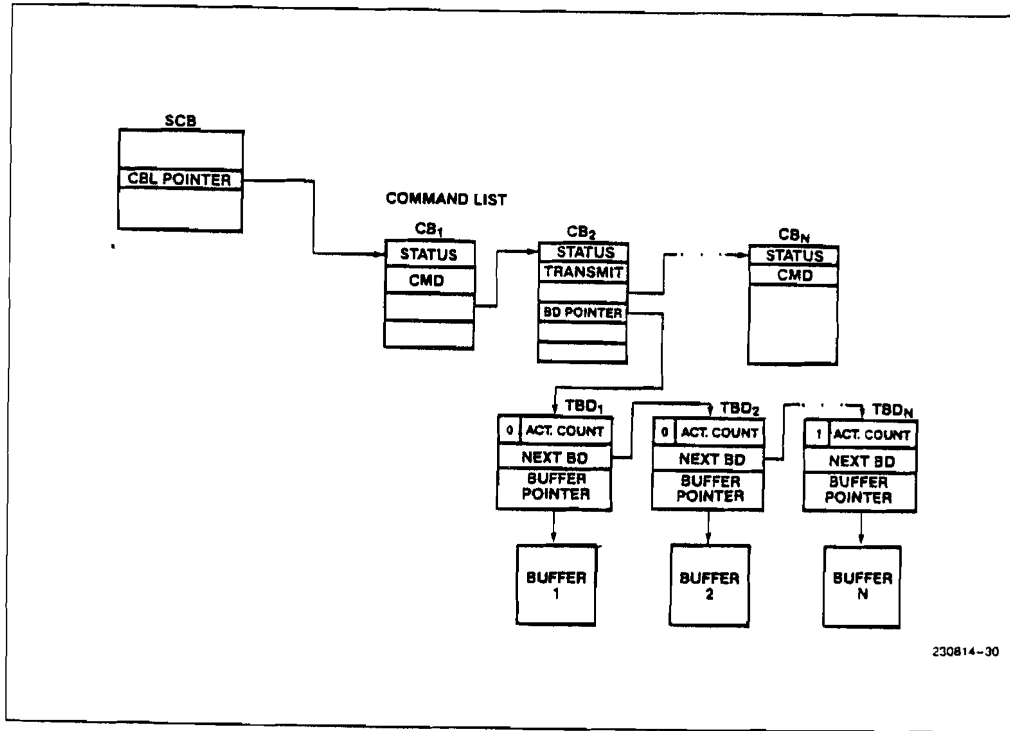


Figure 2-17. The Transmit Command Data Structure

figuur 3.4 - De Command List

De command list (CL) bevat de werk voorraad (commando's) voor de CU. Evenals bij de RFA bestaat deze werkvoorraad weer uit een linked list van command blocks (CB). Zoals reeds aangegeven zijn er 8 verschillende commando's en dus ook 8 typen CB's.

In figuur 3.4 is ook een transmit command block aangegeven. Bij zo'n commando horen transmitbuffers die de te versturen data bevatten. Ieder transmitbuffer wordt beschreven in een Transmit Buffer Discriptor (TBD), analoog aan de RBD's.
Deze TBD's worden gelinked tot een list en in ieder transmit commando wordt een pointer naar het eerste TBD gezet.

Merk op dat hier in tegenstelling tot de RBD's, de TBD's telkens een kleine keten vormen en behoren tot slechts een transmit commando, terwijl de RBD's een lange lijst vormen en de toewijzing aan een bepaald RFD door de RU gebeurt.

Het grote verschil met de RFA is dat de CL "initieel" leeg is (dus nog geen commando's bevat), terwijl de RFA start met een "maximale" lijst van (lege) RFD's.
In de praktijk betekent dit dat de gemiddelde snelheid waarmee nieuwe commando's aan de CL worden toegevoegd nooit hoger mag zijn dan de snelheid waarmee de CU deze kan verwerken. Dit zou namelijk een oneindig lange lijst opleveren.
Voor het receive proces geldt het omgekeerde, nl: de RFD's moeten door de host sneller kunnen worden verwerkt dan dat deze door de RU kunnen worden aangeleverd. (Met hierbij de opmerking dat dit natuurlijk bepaald wordt door het aantal berichten dat ontvangen wordt.)

3.2 DE SERIEELE INTERFACE

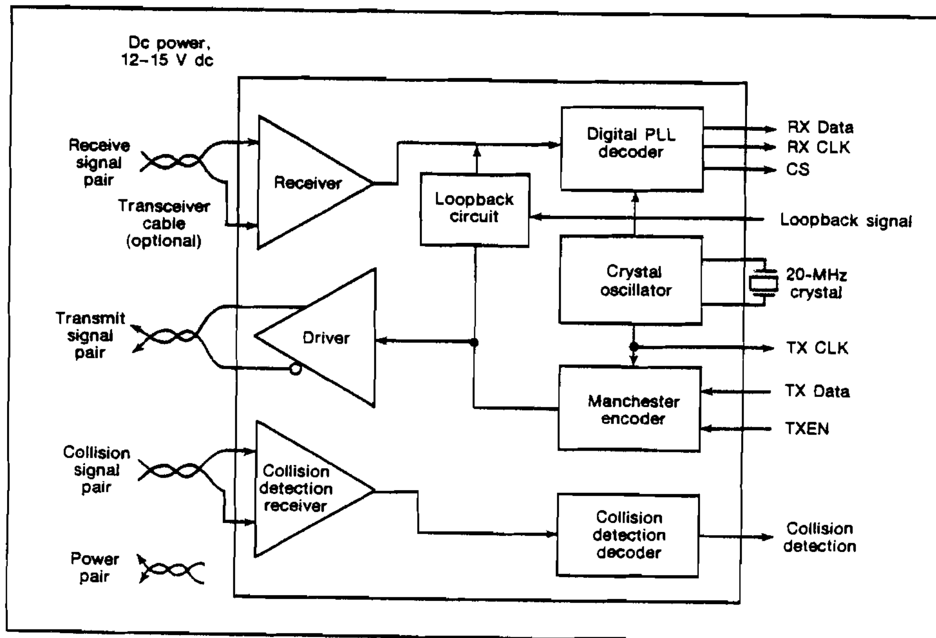
De serieele interface of ook wel coder/decoder genoemd verzorgt de lijn codering en decodering voor de 586 en genereert de collision en carrier-sense signalen.
Als serieele interface voor de 586 levert Intel de 82501. Omdat deze chip nog niet leverbaar was is voorlopig gekozen voor de 8002 chip van Seeq. Deze chip is pin-compatible met de 82501.
Een blokschma van de serieele interface is in figuur 3.5 weergegeven en bestaat uit:

- Een Manchester encoder en decoder. De decoder genereert tevens het clock- signaal voor de RU van de 586. Hiertoe beschikt de 8002 over een 10Mhz PLL schakeling.

- Een 20 MHz kristal oscillator (+ 2 deler) voor de genaratie van de transmit clock en voor het "op frekwentie houden" van de PLL.

- Een mogelijkheid tot z.g loopback mode. Hiermee wordt via een multiplexer de uitgang van het transmit gedeelte doorverbonden met de ingang van het receive gedeelte en kunnen alle hardware functies worden getest zonder gebruik te maken van het medium.

- Collision detection.



figuur 3.5 - De serieele interface

Tevens bevat zowel de 8002 als de 82501 een z.g. <u>watchdog timer</u>. Deze timer zal na 25 ms alle zend operaties afbreken. Dit is een extra beveiliging opdat een station nooit voor een onbepaalde tijd het medium kan bezetten.

3.3 DE ETHERNET TRANSCEIVER

De Transceiver, of in IEEE termen de PMA, vormt de koppeling met de coax kabel.
Zoals uit figuur 3.6 blijkt bestaat de verbinding tussen serieele interface en transceiver uit 4 symmetrische aderparen, tw:

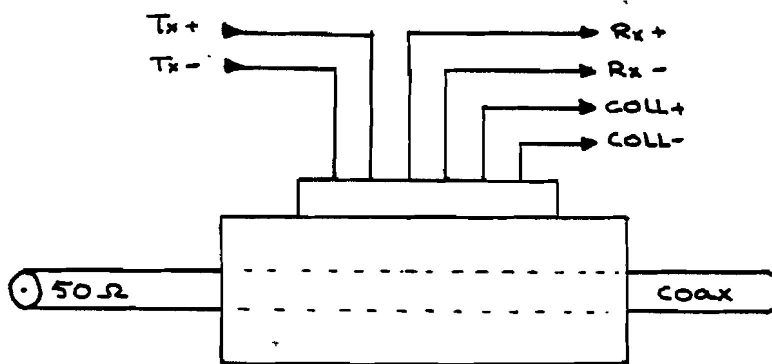
1) Het transmit signaal, 10 Mbaud en Manchester gecodeerd.

2) Het receive aderpaar, met hierop het signaal zoals dit op de coax aanwezig is.

3) Het collision signaal. Hierop plaatst de transceiver een 10 Mhz signaal wanneer een collision op het medium wordt waargenomen.

Een collision wordt alleen gegenereerd tijdens het zenden en de transceiver herkent een collision indien het signaal zoals dat wordt waargenomen op het medium niet overeenkomt met dat op het transmit aderpaar.
Dit heeft als nare konsekwentie dat ook reflekties op de coaxiale kabel (t.g.v. misaanpassingen) ook gedetekteerd worden als collision. Om deze reden wordt er veel aandacht besteed aan de kwaliteit van de ethernet kabel en mag de bevestiging van de transceiver aan deze kabel geen noemenswaardige verstoring van de kabel-impedantie geven.

4) Het voedings aderpaar (12 volt).

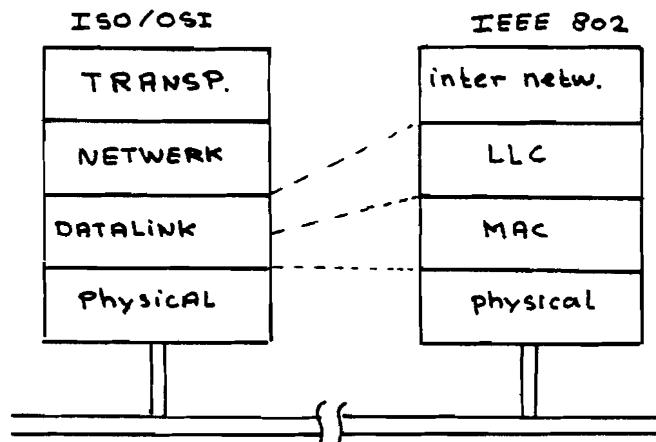


figuur 3.6 - De Ethernet transceiver

IV NETWERK SOFTWARE

In hoofdstuk 2.2.1 en 2.2.2 is een korte uiteenzetting gegeven van het OSI resp, het IEEE-802 model. Een dergelijk model is bij de huidige stand van de techniek slechts ten dele (alleen de onderste lagen) als een komplete harware funktie realiseerbaar. De hogere lagen dienen d.m.v. software oplossingen gerealiseerd te worden

Zowel het OSI model als het IEEE-802 model zijn op dit moment slechts voor een deel gedefinieerd. Van het OSI model zijn de lagen 1 t/m 4 (tot aan session laag) volledig in een standaard vastgelegd. De hogere lagen zijn nog in studie.
Het IEEE-802 model is speciaal bedoeld voor een LAN en heeft t.o.v. het OSI model een afwijkende organisatie. Zoals in fig 4.1 blijkt heeft IEEE de netwerk- en transport- laag in een laag ondergebracht. (inter networking) De reden hiervoor is dat bij LAN's een netwerk laag meestal overbodig is, omdat een LAN een lokaal netwerk is, dat meestal bestaat uit slechts een (physisch) segment.



figuur 4.1 - Het OSI en IEEE-802 reference model

Verder heeft het IEEE model een zeer uitgebreide link- laag, opgesplitst in een Medium Acces Control (MAC) en een Logical Link Control (LLC). Deze laatste laag bevat veel meer funkties dan de link- laag zoals gespecificeerd in het OSI model.

Van het IEEE-802 model zijn op het ogenblik de LLC en MAC lagen voor zowel CSMA/CD alsmede de tokenbus gerealiseerd. De LLC-MAC laag voor de overige standaards alsmede de internetworking laag is reeds in een vergevorderd stadium.

In dit hoofdstuk zal ik, uitgaande van het IEEE model, een beschrijving geven van de tot nu toe gerealiseerde lagen, alsmede enige definities.

4.1 ENKELE DEFINITIES

Iedere laag in het netwerk model kan worden omschreven als:

> Een verzameling van "entities" die d.m.v. een "peer-to-peer protocol" de "data-units" verwerken, hierbij gebruik makend van de service-primitieven van de eronder liggende laag.

Een dergelijke algemene omschrijving is weinig zinvol, daarom enige toelichting van de gebruikte begrippen.

<u>protocollen en entities</u>
Iedere laag bestaat uit een aantal aktieve elementen, ook wel "entities" (eng.) genoemd. Deze entities communiceren met:

- peer entities. Dit zijn entities in dezelfde laag, maar (meestal) van een ander systeem.
- entities van aangrenzende lagen in het zelfde systeem.

Communicatie tussen peer-entities van twee verschillende systemen is uiteraard alleen maar mogelijk indien alle peer-entities hetzelfde communicatie voorschrift of protocol gebruiken. Dit is vrij logisch als men bedenkt dat alle informatie eerst verwerkt wordt door de lager gelegen lagen tot aan het physische medium. Vervolgens via dit medium verstuurd wordt naar een ander systeem, waar de informatie via telkens een hoger gelegen laag wordt doorgegeven naar de oorspronkelijke laag.
Een protocol dat hieraan voldoet heet peer-to-peer protocol en betekent dat rechtstreekse communicatie in een bepaalde laag mogelijk is, zonder zich van de eronder liggende lagen bewust te zijn.

<u>Data Units</u>
In zowel het OSI als IEEE model vindt alle communicatie tussen entities plaats d.m.v. data units. Er bestaan 3 klassen van data units, nl:

1) (N)-protocol-data-units (N-PDU)

2) (N)-interface-data-units (N-IDU)

3) (N)-service-data-units (N-SDU)

waarbij (N) een prefix is voor de N-laag.

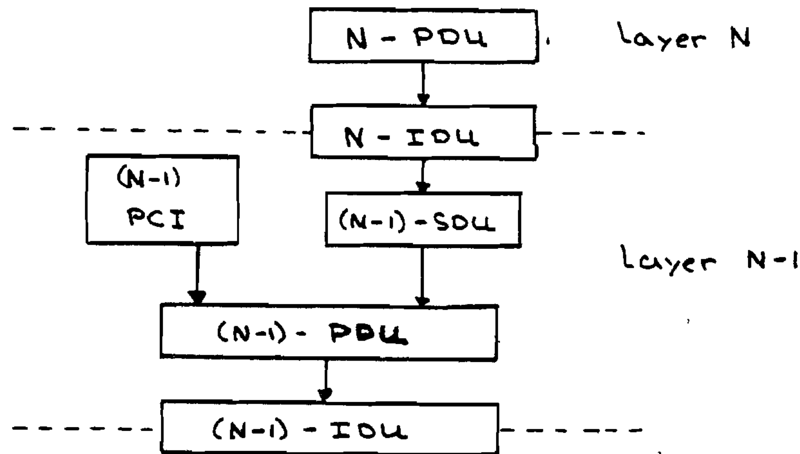
Deze 3 klassen kunnen verder worden onderverdeeld in 2 subklassen, t.w. (N)-control information en (N)-user data
Control information bestaat uit alle noodzakelijke informatie om twee entities met elkaar te laten communiceren.
(N)-user data is de data welke twee (N)-entities met elkaar uitwisselen en die afkomstig is van (N+1)-entities.
Onderstaande tabel toont de relatie tussen de verschillende data units.

	Control	Data	Combinatie
(N) - (N) peer-entities	(N)-protocol control information	(N)-user data	(N)-protocol data units
(N) - (N-1) entities zelfde systeem	(N-1)- interface control information	(N-1)- interface data	(N-1)- interface control data unit

tabel 4.1 - relatie tussen entity <-> data unit

Het gebruik van de data-units is weergegeven in figuur 4.2. Een (N)-protocol data unit wordt eerst afgebeeld op een (N)-interface data unit. Dit is a.h.w. de (ontwerp) interface tussen de beide lagen. (In het meest eenvoudige geval wordt een (N)-PDU meteen gekopieerd in een (N-1)-SDU.)
Van dit (N)-IDU wordt het user data gedeelte gekopieerd in het (N-1)-SDU. De inhoud van dit (N-1)-SDU wordt door de (N-1)-laag niet aangetast en vormt samen met de (N-1)-protocol informatie een nieuw (N-1)-PDU.

Het protocol moet zodanig zijn dat uit een (N-1)-PDU altijd weer het oorspronkelijke (N)-PDU kan worden gereconstrueerd.



figuur 4.2 - Het gebruik van de data-units

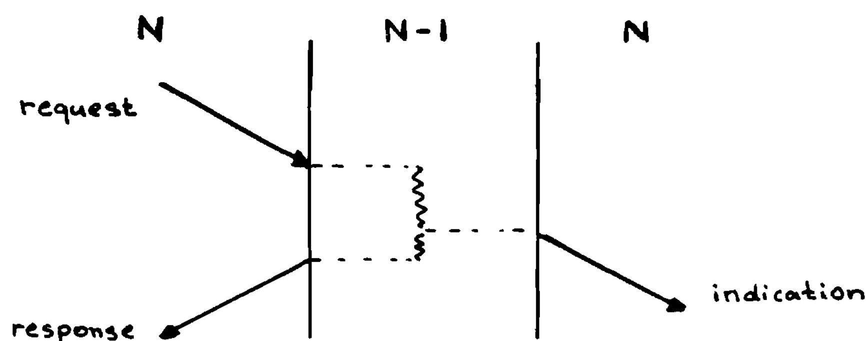
Service primitieven
Iedere laag in het model biedt een bepaalde hoeveelheid "service" aan de volgende hoger gelegen laag. De beschrijving van deze service gebeurt d.m.v. service primitieven en parameters.
De primitieven geven aan hoe de service wordt aangeboden en de parameters wat er aan service wordt aangeboden.

Er zijn 3 soorten primitieven, die in alle lagen worden toegepast, t.w.

request: Dit is de aanvraag van de service die de (N)-laag plaatst bij laag (N-1).

indicate: Dit is een melding van laag (N-1) aan laag (N). Deze kan afkomstig zijn van een (N)-laag request uit een ander systeem, of wordt intern gegenereerd in de onderliggende (N-1)-laag.

response: Dit is een terugmelding van de (N-1)-laag aan de (N)-laag op een eerder aangevraagde service request.



figuur 4.3 - De service primitieven

Gebruik makend van bovenstaande definitie zullen nu de lagen 1 t/m 3 worden besproken, zoals deze door de IEEE werkgroep in [lit13] en [lit14] gedefinieerd zijn

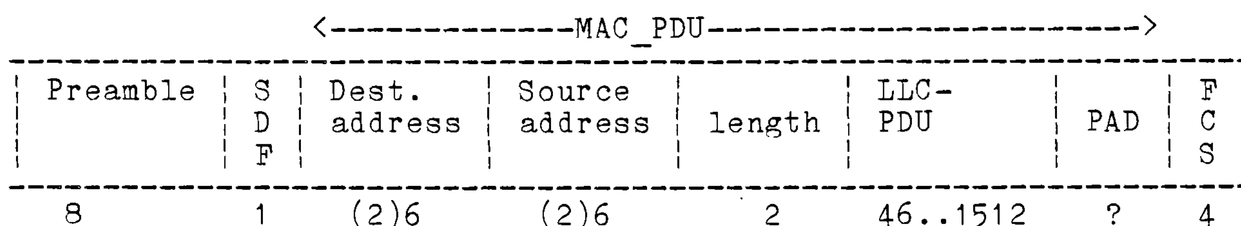
4.2 DE PHYSICAL LAYER

Deze laag is volledig d.m.v. hardware gerealiseerd (zie hoofdstuk 3) en valt dan ook buiten het kader van dit hoofdstuk.
Een volledige beschrijving van het physische protocol is te vinden in [lit4] en [lit13]

4.3 DE MAC-LAAG

De MAC-laag geeft een beschrijving van: De acces methode, de opbouw van het MAC_SDU (of frame) en van de service die deze laag moet bieden aan de LLC-laag.
Deze laag is, evenals de physische laag, voor het overgrote deel in hardware uitgevoerd. Dit betreft dan in bijzonder de access methode, die volledig door de 82586 wordt afgehandeld.

4.3.1 MAC frame format



<------------MAC_PDU---------------------->

Preamble	SDF	Dest. address	Source address	length	LLC-PDU	PAD	FCS
8	1	(2)6	(2)6	2	46..1512	?	4

figuur 4.4 - Het MAC frame format (MA_PDU)

Het MAC-frame bestaat uit de volgende elementen:

preamble:
Dit zijn 8 bytes bestaande uit het patroon AAH = 10101010 Deze zijn bedoeld om de physische laag gelegenheid te geven om te synchroniseren.

SDF Dit is de Start Frame Delimiter (13H) en geeft de start van het frame aan.
Zowel het SDF alsmede het preamble veld behoren in feite niet tot het MAC_PDU, maar moeten worden beschouwd als protocol informatie van de physiche laag. (P_PCI)

Adres velden
De adres velden zijn 2 of 6 bytes groot. Deze keuze is arbritair en niet door IEEE gestandaardiseerd. De meeste implementaties gebruiken echter een 6 bytes adres.
Er zijn twee soorten adressen, t.w. een individueel en een groepsadres. Dit wordt aangegeven door bit 0 van het eerste adresbyte. (0=individueel, 1=groepsadres)
Een bijzonder groepsadres is het broadcast- address en dit bestaat uit 2 of 6 bytes met 0FFH.
In het geval er gebruik gemaakt wordt van een 6 bytes adres, geeft bit 1 van het eerst byte aan of het een globaal of lokaal adres betreft. Dit onderscheid is zuiver een administratief probleem. De 586 kan "real-time" het adres decoderen en maakt daarom geen gebruik vam dit tweede bit.

lengte
Dit 2 bytes grote veld geeft het aantal geldige bytes (de lengte) van het MAC informatie veld.

LLC_PDU
Dit veld bevat de MAC informatie. Indien de lengte hiervan kleiner is dan de minimum framesize moet dit veld worden aangevuld met z.g. PAD-bytes. Deze bytes behoren dus niet tot de LLC_PDU en worden dan ook niet geteld in het lengte veld.

FCS Bevat de 32-bit CRC-check. Deze wordt berekend over de velden: source/dest. adres, lengte, LLC_PDU en PAD.

4.3.2 MAC Service specificatie

De MAC service primitieven zijn als volgt gedefinieerd:

```
MA_DATA_request ( destination_adres
                  MAC_SDU
                )
```

Deze primitieve wordt gegenereerd door een LLC-entity. De MAC laag dient alle overige velden, zoals weergegeven in figur 4.4 toe te voegen.

```
MA_DATA_response ( transmission_status)
```

Dit is de terugmelding aan de LLC op een eerder geplaatste request. Het aantal mogelijke waarden van de "transmission_status" die bij deze primitieve behoort, is afhankelijk van de implementatie.
Opm: Bij de implementatie moet er tevens zorg gedragen worden dat het voor de LLC laag duidelijk is welke respons behoort bij welke request. In het geval de MAC laag volgens de first-in-first-out methode werkt, is geen verdere voorziening vereist.

```
MA_DATA_indicate ( dest. address
                   source_address
                   MA_SDU
                   reception_status
                 )
```

Deze primitieve definieert de data transfer van MAC naar LLC.

```
MA_RESET_request
```

Deze primitieve, zonder parameters, heeft tot doel alle data units binnen de MAC te vernietigen en de MAC laag in een gedefinieerde (lege) toestand te brengen.

```
MA_RESET_response ( reset_status)
```

Terug melding aan de LLC laag of de RESET aanvraag wel of niet succesvol is geweest.
Omdat de RESET primitieven geen peer-to-peer communicatie inhouden, is een MA_RESET_indicate niet nodig.

4.4 DE LLC-LAAG

De LLC-laag is door het IEEE voorzien van een groot aantal mogelijkheden, waardoor deze laag een veel complexer aanzien heeft dan de oorspronkelijke Data Link laag zoals voorgesteld door de ISO. Doordat de laag zo uitgebreid is worden er in de standaard 2 klassen gedefinieerd:

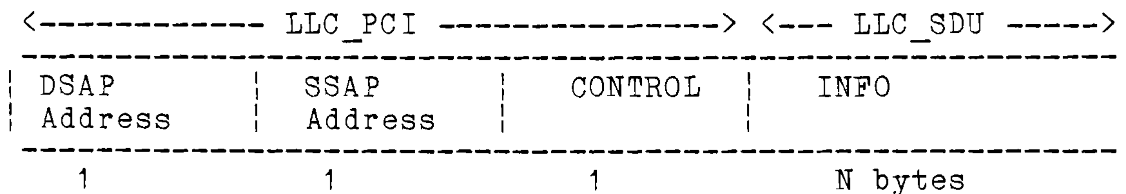
Klasse 1 is de eenvoudigste en voorziet in dezelfde service primitieven als de MAC-laag. (Dit heet dan unacknowledged-connectionless service) Als extra voorziet de LLC laag in z.g. Service Access Points (SAP). Dit zijn de "end-points" van de LLC-laag en vormen de verbinding met de hoger gelegen lagen. Een in de literatuur veel gebruikte terminologie voor SAP is: Poort.
In de LLC-laag kunnen maximaal 128 SAP's worden gedefinieerd. Om met deze poorten te kunnen werken zijn een aantal management functies aan de LLC-laag toegevoegd.

Klasse 2 is nog breder opgezet en bevat, naast de functies van klasse 1 ook nog functies om een verbinding (connection) tussen twee SAP's tot stand te brengen. Hierdoor kunnen zaken als framenummering en error-recovery worden toegepast.
Tevens bestaat de mogelijkheid om ieder LLC-PDU te bevestigen.
De complete set van service primitieven zal verderop worden toegelicht.

Een klasse 3 laag is in studie. Wat betreft de mogelijkheden ligt deze tussen klasse 1 en 2 in en maakt een goede kans om als standaard te worden aanvaard.

4.4.1 LLC_protocol data unit

Een LLC_PDU bestaat uit de volgende elementen.



<------------ LLC_PCI ---------------> <--- LLC_SDU ----->

DSAP Address	SSAP Address	CONTROL	INFO
1	1	1	N bytes

figuur 4.5 - LLC_PDU

DSAP Address
Dit is het destination Service Access Point. (1 byte) Hier worden 7 bits van gebruikt als adressering en het LSB voor de aanduiding van een individueel of groeps-adres. (identiek aan

het MAC adres)

SSAP Address
Identiek aan DSAP Address. Het LSB dient nu om aan te geven of het control veld beschouwd moet worden als een commando (0) of een respons (1).

Control veld
Dit veld wordt gebruikt door twee LLC entities voor bijv. frame nummering en LLC commando's. (alleen klasse 2) De opbouw lijkt erg veel op het control veld in een X.25 packet. Er zijn drie typen control-velden, nl:
I-format: wordt gebruikt om frame's te nummeren (Klasse 2)
S-format: Voor besturings functies in klasse 2. (acknowledge, hertransmissie, etc.)
U-format: Voor klasse 1 en 2. Wordt gebruikt voor het versturen van LLC- management commando's.

Info-veld
Bevat het LLC_service data unit, afkomstig van een hogere laag.

4.4.2 LLC_service specificaties

De LLC service primitieven kunnen we in 4 klassen onderverdelen:

1) Unacknowledged connectionless.
Deze primitieve wordt gebruikt in de LLC klasse 1 definitie en is een "een-op-een" afbeelding van de MA_DATA primitieve.

2) Acknowledged connectionless.
In klasse 2 bedoeld om de omvangst van een frame te bevestigen.

3) Connection oriented.
Eveneens voor klasse 2. M.b.v. deze primitieven kan een LLC data link verbinding worden gemaakt. Zo'n (vaste) verbinding heeft als voordeel dat op eenvoudige wijze framenummering, flowcontrol en error recovery kan worden toegepast.

4) Management services
Deze functies worden in beide klassen gebruikt. (Een uitzondering hierop vormt de LLC_SAP_FLOWCONTROL primitieve die alleen in klasse 2 gebruikt wordt)
TEST en STATUS zijn bedoeld als controle functies om de werking van de LLC laag te testen.
Met ACTIVATE/DEACTIVATE kan een hogere laag de LLC laag (en dus het gehele station) in-en uitschakelen.
LLC_SAP_(DE)ACTIVATE is bedoeld om een SAP te creeeren en weer te verwijderen.

UNACKNOWLEDGED CONNENTIONLESS SERVICE			
Service	request	response	indication
LLC_DATA	X	0	X

ACKNOWLEDGED CONNECTIONLESS			
Service	request	response	indication
LLC_DATA_ACK	X	0	X

CONNECTION ORIENTED			
Service	request	response	indication
LLC_CONNECT	X	0	X
LLC_DATA_CONNECT	X	0	X
LLC_DISCONNECT	X	0	X
LLC_RESET	X	0	X
LLC_CONNECTION_FLOWC.	0	-	0

MANAGEMENT SERVICES			
Service	request	response	indication
LLC_TEST	X	X	-
LLC_STATUS	X	X	X
LLC_SAP_ACTIVATE	X	0	-
LLC_SAP_DEACTIVATE	X	0	0
LLC_LAYER_ACTIVATE	X	0	0
LLC_LAYER_DEACTIVATE	X	0	0
LLC_SAP_FLOWCONTROL	0	-	0

X = verplicht   0 = optioneel   - = n.v.t.

tabel 4.2 - LLC service primitieven

Zoals uit bovenstaande tabel blijkt, is de LLC laag uit een groot aantal service primitieven opgebouwd. Deze behoeven in een bepaalde implementatie niet allemaal gerealiseerd te worden. De syntax van deze LLC primitieven zal ik niet alle behandelen, maar als voorbeeld de LLC-CONNECT service wat nader toelichten.

```
LLC_CONNECT_request ( local address
                      remote address
                      quality )
```

Met deze aanvraag kan een verbinding tussen 2 SAP's tot stand worden gebracht. De "quality" parameter is voor de LLC laag niet gedefinieerd, maar is bedoeld om bepaalde opties mee te geven.

```
LLC_CONNECT_response ( local address
                       remote address
                       status
                       quality )
```

Dit is de bevestiging van de LLC verbinding. De status parameter geeft aan of de aanvraag succesvol is of niet. (De LLC_CONNECT_indicate primitieve heeft dezelfde syntax als de response primitieve)

De behandelde specificaties voor de MAC en de LLC-laag zijn afkomstig uit een niet volledig IEEE-802 rapport (Draft C) en is, voor wat betreft het LLC gebeuren zeker nog niet definitief. Met name de klasse 3 protocol definitie moet nog verder worden uitgewerkt.

De uiteindelijke implementatie van alle LLC functies zal ook sterk afhankelijk zijn van de erboven liggende lagen. (met name de transport laag)
Zo bestaat er een ECMA voorstel [lit15] waarbij een LAN gerealiseerd dient te worden m.b.v. een klasse 1 LLC laag. (of althans hiermee vergelijkbaar) Functies zoals flowcontrol en error- recovery dienen door de transportlaag, zoals deze is gedefinieerd door de ISO, te worden uitgevoerd. Dit ISO-transport protocol is een algemene definitie welke niet onmiddellijk aansluit op de IEEE LLC standaard.

Verwacht kan worden dat het IEEE voorstel, betreffende de internetworking wel goed zal aansluiten op het LLC model. Daar ik nog in het bezit ben van dit voorstel, kan ik hierover geen uitspraken doen en dit voorstel dient daarom nog nader bestudeerd te worden.

V DE IMPLEMENTATIE

Dit hoofdstuk behandelt de tot nu toe gerealiseerde software. Alvorens deze software stap-voor-stap te bespreken wil ik eerst nog enkele zaken betreffende de implementatie toelichten.

De LEX

De ontwikkelde software maakt gebruik van de faciliteiten van de LEX. Dit is een Local Executive en is ontwikkeld door L.Borger [lit16] en geimplementeerd door R.Deliege. [lit17]
De LEX is geschreven in C (80%) en in 186 assembler (20%) en is bedoeld als operating systeem voor de 186 I/O controller kaarten van THE KUNix machine.
De door mij ontwikkelde software is tevens de eerste "grote" test voor de LEX, die tot dan toe alleen nog maar getest was met zeer eenvoudige (test) programma's.

Een uitgebreide behandeling v/d LEX zal ik in dit verslag niet geven, maar verwijs hiervoor naar [lit16]. Op deze plaats zal ik alleen de belangrijkste zaken van de LEX even aanstippen.

De LEX maakt het mogelijk om onafhankelijke processen te definieren. Zo'n proces is in feite een "oneindige" lus. De processen kunnen onderling "communiceren" d.m.v. semaphoren en mailboxes, maar kunnen ook "geisoleerd" blijven, zodat in feite meerdere applicaties semi-parallel onder de LEX kunnen werken.

Semaphoren worden gebruikt voor onderlinge synchronisatie van de processen alsmede voor afgrendeling van bepaalde gemeenschappelijke faciliteiten.

Een mailbox werkt analoog aan de semaphore, maar men kan nu tevens een bericht meegeven (vmail-operatie), dat vervolgens door een ander proces gelezen kan worden (pmail-operatie). Indien een mailbox leeg is kan men opgeven of een proces al dan niet op een bericht moet wachten.
Verder heeft de LEX de mogelijkheid om prioriteiten aan processen te geven.

In zijn eenvoudigste vorm ziet een proces er als volgt uit:

```
process1()
{
/* declaratie autovariabelen */

initialisatie();

while(TRUE) { /* eindeloze loop
                 met proces acties */
            }
}
```

Het initialisatie gedeelte heeft tot doel om bij bijv. de semaphoren en (lege) mailboxes te definieren.
Deze eindeloze loop moet in ieder geval een operatie op een semaphore of mailbox bevatten, zodat proces-switching mogelijk is.

Naast processen bestaan er nog handlers. Dit zijn (eindige) functies die aan een van de 256 interrupt vectors van de 186 processor kunnen worden gekoppeld. De LEX onderscheid 2 soorten handlers, nl: DIRECT en INDIRECT ATTACHED handlers.
Direct attached handlers krijgen onmiddellijk de besturing na het optreden v/d interrupt, d.w.z. de gebruiker moet zelf alle processor registers redden en mag geen LEX system calls op semaphoren en mailboxes uitvoeren. (Direct attached handlers zijn bedoeld voor zeer snelle interrupt verwerking en zijn meestal geschreven in assembler.)
Indirect attached handlers zijn de door de LEX opgestarte (interrupt) functies en zij mogen in principe alle LEX system calls gebruiken. (Het gebruik van bijv. een vmail-operatie op een volle mailbox moet met de nodige omzichtigheid gebeuren, omdat hierdoor de handler geblokkeerd kan raken.)
Een indirect attached handler kan er als volgt uitzien:

```
handler1()
{
/* declaratie auto variabelen */

/* interrupt service acties,
   waarbij meerdere LEX calls gebruikt mogen worden */

intret(vectornummer) /* acknowledge */
}
```

Ieder proces kan een handler koppelen aan een bepaalde interrupt vector.

Zoals reeds gesteld, is de LEX nog niet uitvoerig getest. Met name de mogelijkheid om de processen in verschillende (code) segmenten onder te brengen en het gebruik van meerdere data-segmenten is in zijn geheel nog niet getest.

In eerste instantie is de software dan ook ontwikkeld door alle processen (en data) onder te brengen in een segment.
De complete applicatie bestaat dan ook uit een groot C-programma (+ enige LEX assembler routines) met hierin:

- Een main() functie. Deze initialiseert de LEX en geeft aan welke functies als proces moeten worden beschouwd. Na deze initialisatie is de main() functie overbodig geworden.

- Een aantal proces-functies, in het vervolg te noemen: processen.

- De interrupt handlers. (Er is alleen gebruik gemaakt van indirect attached handlers)

- Alle LEX functies.

Door deze opzet bevat iedere applicatie de volledige LEX-code, hetgeen moet worden gezien als een tijdelijke oplossing.

De 586 "shared memory" structuur

Bij de implementatie is zoveel mogelijk getracht het "shared memory" concept van de 586 te benutten. Zoals in hoofdstuk 3 is uiteengezet is de 586 in staat om meerdere commando's gelijktijdig te accepteren (linked-list), zodat ook de besturings software in staat moet zijn om meerdere aanvragen gelijktijdig in behandeling te nemen.
Tevens voorziet het "shared memory" model in een mogelijkheid bij iedere send en receive opdracht meerdere databuffers te gebruiken. Dit betekent o.a. dat bij een MAC_DATA_request de data niet behoeft te worden gekopieerd in een MA_SDU, maar dat het voldoende is om alleen de plaats en de grootte van de databuffers vast te leggen in een buffer descriptor.

Software onwikkelingen & testen

Alle software is in Nijmegen op de UNIX computer van de Katholieke Universiteit ontwikkeld. Dit systeem beschikte nl. over een C-compiler die als output iAPX 186 code opleverde. Op dit systeem werd een complete LEX applicatie gemaakt, waarna code file, assembly listing en symbol table via een modem werden overgestuurd. De assembly listing en symbol table, samen met de monitor op de 186 kaart, werden gebruikt als "debugging tool".
Deze omslachtige methode plus het feit dat de LEX nog een aantal "bugs" bevatte, heeft ertoe geleid dat het testen van de software meer tijd heeft gevergd dan oorspronkelijk was voorzien.

De komplete applicatie bestaat uit de volgende files:

```
MACSTRUC.H   - Header file, bevat alle konstanten
               en definities van de structures.
MAC.C        - Alle C-sources
LEXINIT.C    - Bevat de main() functie. Initialiseert de LEX
               en geeft aan welke functies in MAC.C
               de status van proces krijgen.
LEX FILES    - Een verzameling van files die
               gezamelijk de LEX vormen. Dit zijn de files:
               STRUCTS.H, SYSDATA.H = alle declaraties
               LEXCALL.C            = LEX system calls
               LEXUTIL.C            = LEX C-utility's
               LEXKERN.A86          = LEX kernel
               LEXDECL.A86          = declaratie LEX data
               LEXSUPP.A86          = diverse assembler functies
```

5.1 DE STRUCTUUR

Figuur 5.1 geeft de globale structuur van de gerealiseerde software, voorstellende de MAC-laag + 586 driver (rechts van de onderbroken lijn) en een eerste aanzet tot de LLC laag. Dat een dergelijke structuur, gegeven het "shared memory" model van de 586, zeer voor de hand liggend is, zal ik in deze paragraaf proberen uit te leggen.

1)
Volgens de definitie van LLC en MAC-laag is er tussen deze twee lagen maar 1 communicatie punt, m.a.w. de MAC laag kent geen SAP's. Vertaald naar de LEX is zo'n communicatie middel een mailbox (MB1), waarin de LLC laag opdrachten (request primitieven) kan plaatsen.

2)
De LLC kan meerdere SAP's bevatten, hetgeen betekent dat de LLC uit meerdere "opdracht gevers" kan bestaan. Dit kan worden voorgesteld door meerdere processen. In figuur 5.1 is dit aangegeven door de processen P1 .. Pn.
Elk van deze processen kan onafhankelijk van elkaar met de MAC-laag communiceren. Hoe deze communicatie precies verloopt wordt in een volgende paragraaf uiteengezet.

3)
De service die de MAC-laag moet aanbieden is het versturen en ontvangen van LLC_PDU's en het RESETTEN van de MAC-laag.

De MAC-laag zelf bestaat uit een aantal processen die gezamelijk de 586 en zijn "shared memory" structuur besturen en de gewenste service kunnen aanbieden. (N.B. Deze "shared memory" structuur is in figuur 5.1 niet aangegeven. De figuur toont alleen de processen en hun onderlinge relatie.)

Welke processen zijn er nu nodig ???
De 586 bestaat uit een RU en een CU, die onafhankelijk van elkaar werken en ieder een eigen data structuur gebruiken. Als we er vanuit gaan dat de CU alleen gebruikt wordt voor zend-commando's, impliceert dit dat er minimaal 2 processen nodig zullen zijn, nl: een send en een receive proces. Het send-proces "bewerkt" de CL en het receive-proces de RFA.

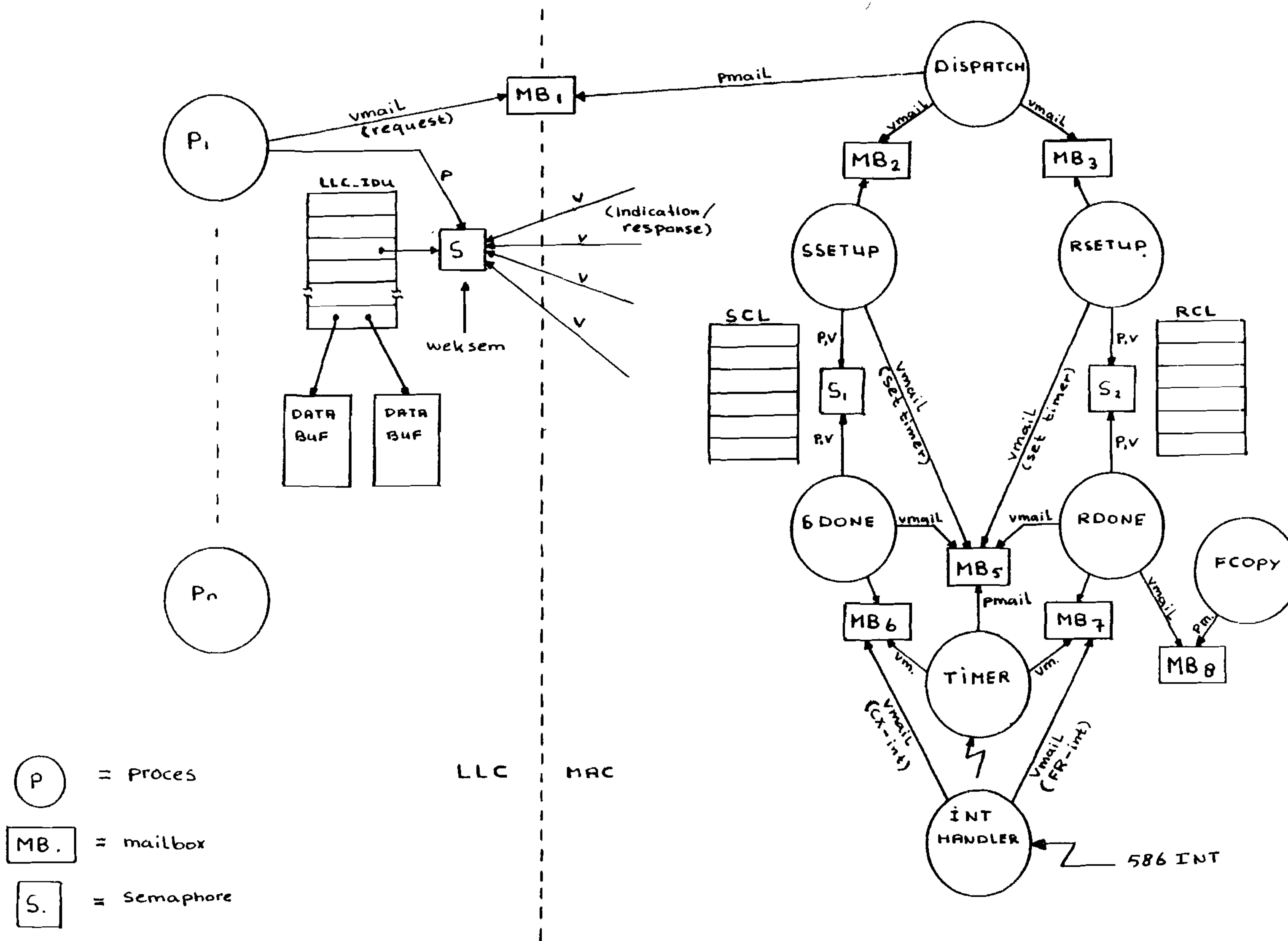
Dit send-proces is vervolgens ook weer gesplitst in twee deel processen. De gedachte hierachter is als volgt:
Omdat de CL bestaat uit een linked-list en werkt volgens het FIFO principe is gekozen voor een soort input- en output proces. Het "input" proces accepteerd de aanvragen vanuit de LLC en voegt deze toe aan de CL. Het "output" proces verwerkt de door de 586 vrijgegeven datastructuren en verwijdert deze uit de CL.
Een zelfde gedachte gang is ook toegepast bij het receive-proces, zodat er in totaal dus 4 processen zijn die het "shared memory" van de 586 "besturen", t.w.

SSETUP: Dit proces zal alle binnenkomende aanvragen om data te verzenden (MAC_DATA_request) aannemen en vervolgens een transmit command block aan de CL van de 586 toevoegen.

RSETUP: Als SSETUP, maar dit proces heeft betrekking op de receive aanvragen. Merk op dat een receive-aanvraag geen service primitieve is. Een dergelijke aanvraag moet dan ook worden geinterpreteerd als een LLC_entity, dat in staat is om een MAC_DATA-indication te accepteren.

SDONE: Dit proces zorgt voor de afhandeling van de transmit-commando's, verwijdert deze commando's uit de CL en genereert een MAC_DATA_response.

RDONE: Verwerkt de door de 586 vrijgegeven framedescriptors en bepaalt aan welk extern LLC proces de ontvangen data wordt toegewezen.



figuur 5.1

In figuur 5.1 zijn naast deze vier (elementaire) processen nog drie processen een interrupt handler + twee datastructuren opgenomen. De functie hiervan is:

DISPATCH:
Dit proces accepteert alle aanvragen vanuit de LLC-laag en geeft een send- of receive aanvraag door aan SSETUP resp. RSETUP. Verder verzorgt dit proces de initialisatie van de MAC-laag (datastructuren) en hardware. (586 diagnose, configure & individual address command) Ook de MAC_RESET_request wordt door dit proces uitgevoerd.

TIMER: Dit is een algemeen timer proces dat meerdere aanvragen tegenlijkertijd kan verwerken en is bedoeld voor time_out detectie. In een van de volgende paragrafen zal nader op deze functie worden ingegaan.

FCOPY: Dit is een Frame COPY proces en wordt gestart vanuit RDONE. De functie is om de ontvangen data naar een extern proces te copieren. (Een van de processen P1 .. Pn. Welk proces wordt bepaald door RDONE.)
De reden om de ontvangen data te kopieren, is het feit dat een ethernet-frame een variabele lengte heeft en dat de lengte van een binnenkomend frame in principe niet bekend is. Dit betekent dat er een situatie kan ontstaan waarbij een extern proces te weinig buffer ruimte heeft om een frame te ontvangen. Om dit te voorkomen worden alle binnenkomende frame's intern (in de MAC-laag) gebufferd, waarna RDONE bepaalt welk extern proces in staat is om dit frame te ontvangen. Vervolgens wordt FCOPY geactiveerd.
Merk op dat deze kopieer-slag bij het zenden niet noodzakelijk is, omdat nu exact bekend is hoeveel data er verzonden gaat worden. Bij een zend commando wordt de data door de 586 rechtstreeks uit de data buffers van de externe processen gelezen. (De buffers worden "gelinked" aan de transmit buffer descriptors.)

INT HANDLER:
Dit is een INDIRECT ATTACHED handler die alle 586 interrupts afhandelt. De handler geeft alle vrijgegeven frame descriptors aan het RDONE proces en alle afgehandelde transmit commando's aan het SDONE proces.

SCL/RCL:
Deze afkorting staat voor Send Command List en Receive Command List. Het zijn beiden lineaire lijsten met informatie over de externe processen.
De SCL bevat informatie over alle processen die een zend opdracht hebben gegeven. In de SCL wordt een pointer opgeslagen, die naar de bij het proces behorende LLC_IDU

wijst en een pointer die naar het (transmit) command block wijst.
De SCL wordt aangemaakt door het SSETUP proces en weer "afgebroken" door het SDONE proces.
RCL heeft dezelfde functie met het verschil dat hierin alle processen die een frame wensen te ontvangen worden geregistreerd.

Tot zover een korte beschrijving van de structuur. In de volgende paragrafen zal nader op de diverse onderdelen worden ingegaan.

5.2 DE LLC-MAC INTERFACE

Zoals uit figuur 5.1 blijkt verlopen alle "requests" van LLC laag naar MAC laag via mailbox 1.
De terugmelding van MAC aan LCC (de response en indication) gebeurt via een v-operatie.
De interface data unit (LLC_IDU) heeft de volgende struktuur:

Veld	Woorden
COMMAND	1
TIME_OUT/STATUS	1
& WEKSEMAPHORE	1
BUFFER INFO.	1
VARIABLE PART	0 ..18

De eerste vier velden worden altijd gebruikt en zijn ieder 16 bits (=1 woord) groot. De grootte van het VARIABLE PART is n woorden, met n=0..18 en deze waarde is afhankelijk van het gebruikte commando.
Bij de beschrijving van de velden worden de logische namen gebruikt zoals deze zijn gedefinieerd in de file: MACSTRUC.H

COMMAND
De tot nu toe geimplementeerde commando's zijn:

MSEND = zend een frame
MRECEIVE = meld aan de MAC dat een frame ontvangen kan worden
MRESET = reset hardware en meld dit vervolgens aan alle wachtende processen.

TIME_OUT/STATUS

In dit veld kan de gewenste timeout waarde worden gezet, waardoor een proces kan opgeven hoelang een zend opdracht mag duren of hoelang een proces wil wachten op de ontvangst van een frame. Als er geen time_out waarde wordt ingevuld (0) wordt automatisch de maximale waarde genomen.
Nadat het commando is uitgevoerd geeft het status veld het resultaat aan. Dit veld kan de volgende waarden aannemen:

OK: Er zijn geen fouten opgetreden

BADFMT: De LLC_IDU heeft niet het correcte formaat. Mogelijke oorzaken zijn:

- Ongeldig commando
- De buffers zijn te klein (De 586 verwacht buffers met een minimale grootte.)
- De totale buffergrootte komt niet overeen met de opgegeven waarde.

TOBUSY: Er zijn nu een maximaal aantal opdrachten in behandeling. Dit maximum is o.m. afhankelijk van het aantal messages dat in een mailbox past en moet vooraf worden gedefinieerd. (d.m.v. de konstante SIZE in de file MACSTRUC.H)

TIME_OUT: De opgegeven (of maximale) timeout waarde is verstreken.

TOSMALL: Deze melding kan alleen optreden als een DRECEIVE is gegeven en betekent dat er gedurende het time-out interval wel frames ontvangen zijn, maar dat t.g.v. onvoldoende buffer kapaciteit deze niet zijn gekopieerd.

CANCELLED: Het command is t.g.v. een DRESET vroegtijdig beeindigd.

& WEKSEMAPHORE.

Dit is een pointer naar een z.g. weksemaphore. Een v-operatie op deze semaphore betekent dat de LLC_IDU weer vrijgegeven is en het status-veld door de MAC laag is ingevuld.
Merk op dat deze v-operatie, afhankelijk van het commando, kan worden beschouwd als een MAC_response of als een MAC_indicate.

BUFFERINFO

Dit 16-bits woord geeft informatie over de totale buffer grootte en het aantal buffers dat een extern proces ter beschikking stelt. Het formaat van dit veld is:

bit 0 ..12 specificeert de totale buffer grootte in bytes
bit 13..15 geeft het aantal buffers. (maximaal 7)

VARIABLE PART

Dit veld is in principe variabel van grootte, met een maximum van 18 woorden. Afhankelijk van het commando is de indeling als volgt:

	DSEND	DRECEIVE (entry)	DRECEIVE (returned)
VAR[0]	Dest.address	-	Dest.address
VAR[1]	idem	-	idem
VAR[2]	idem	-	idem
VAR[3]	-	-	Source addr.
VAR[4]	BUFFERADDR1	BUFFERADDR 1	idem
VAR[5]	BUFFERSIZE1	BUFFERSIZE 1	idem
VAR[6]	BUFFERADDR2	BUFFERADDR 2	Type field
.			
.			
VAR[16]	BUFFERADDR7	BUFFERADDR 7	-
VAR[18]	BUFFERSIZE7	BUFFERSIZE 7	-

5.3 HET ZEND GEDEELTE

In deze paragraaf zal uitvoeriger worden ingegaan op de werking van het zend gedeelte.
Nadat een extern proces een LLC_IDU heeft aangemaakt, plaatst dit proces een pointer naar deze IDU in mailbox 1. Het DISPATCH proces leest deze mailbox en zal deze pointer doorgeven aan SSETUP.
Het SSETUP proces verricht nu de volgende handelingen:

- Controle of het aangeboden LLC_IDU het correcte formaat heeft.

- Een "check" of alle gespecificeerde zendbuffers groter zijn dan MINBSIZE.

Is niet aan deze voorwaarden voldaan, dan wordt in het status-veld van het LLC_IDU een foutmelding geplaatst en wordt er een v-operatie op de weksemaphore uitgevoerd.

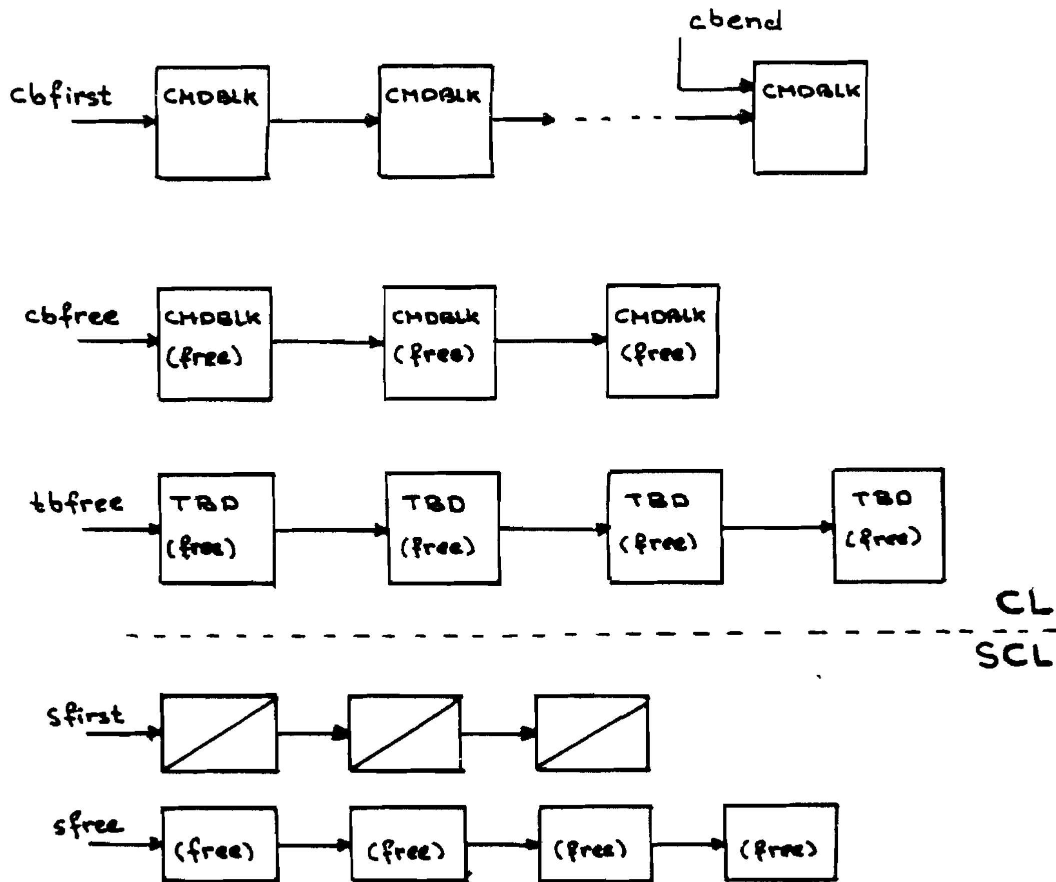
Is de LLC_IDU geaccepteerd dan wordt vervolgens gecontroleerd of:

- er nog plaats in de SCL lijst is om de aanvraag te noteren.
- Er nog command blokken zijn om in de CL te plaatsen
- er nog voldoende transmit buffer descriptors zijn.

Wordt aan deze voorwaarden niet voldaan, dan ontvangt het proces een TOBUSY response.

Figuur 5.2 geeft de organisatie van SCL, CL en TBD's aan. De SCL lijst bestaat uit een "pool" van lege elementen (aangegeven door de pointer sfree) en een werklijst. (aangegeven d.m.v. sfirst)
De CL voor de 586 wordt gemarkeerd door twee pointers, nl: cbfirst en cbend. Deze pointers wijzen resp. naar het begin en eind van de CL. Bij een lege lijst is cbfirst een NILL pointer.
De pointers cbfree en tbfree wijzen naar een lijst met niet gebruikte commandblocks en transmitbuffer descriptors. (TBD's)

Indien aan alle bovenstaande voorwaarden is voldaan, wordt de aanvraag in de SCL opgenomen, een transmit commando (+ TBD's) aangemaakt en in de CL geplaatst en wordt het TIMER proces gestart.
Hierna is het SSETUP proces weer beschikbaar om een volgende opdracht te accepteren.



figuur 5.2 - Datastructuren zend proces.

Het proces SDONE wordt geactiveerd zodra een command block verwerkt is of nadat een time_out is opgetreden.
Laten we eerst het geval beschouwen dat de 586 een commando heeft verwerkt. Na ieder commando genereert de 586 een interrupt. De interrupt handler zal vervolgens het afgehandelde commando block in mailbox 6 plaatsen. (in werkelijkheid alleen een pointer) Het proces SDONE kan hierdoor actief worden en het controleert dan allereerst of het commando zonder fouten is uitgevoerd.
Zijn er geen fouten opgetreden, dan wordt in de SCL lijst "opgezocht" welk extern proces dit commando heeft geplaatst. Vervolgens wordt de timer gestopt en wordt het proces gewekt met als status OK. Het afgewerkte commandblock + TBD's worden in de cbfree en tbfree lijst gezet.

In het geval SDONE een timeout message leest, worden de volgende acties ondernomen:

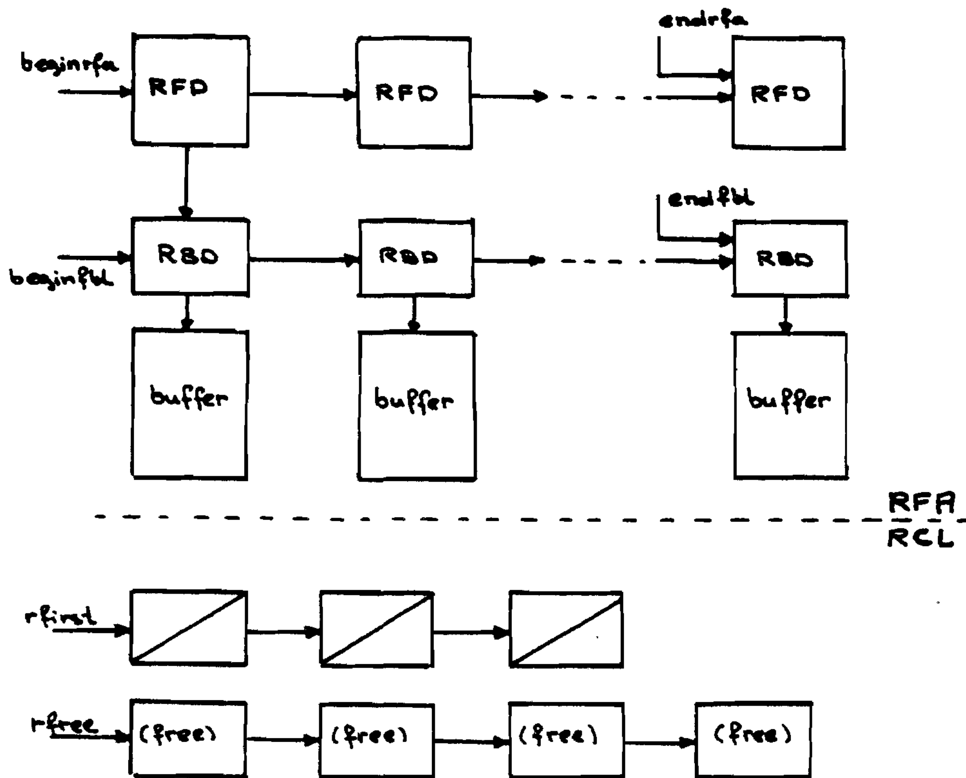
- In de SCL lijst wordt opgezocht op welk commando in de CL de timeout betrekking heeft.

- De CU van de 586 wordt tijdelijk gestopt (suspend). Vervolgens wordt in de CL het betreffende commando opgezocht. Er zijn nu twee mogelijkheden, t.w.
  a)
  Het commando is nog niet door de 586 uitgevoerd. In dit geval wordt het transmit commando gewijzigd in een NOP commando, tevens worden alle TBD's in de tbfree lijst gezet. Deze methode vereist minder handelingen dan die om het commando uit de CL te verwijderen en de 586 via het SCB opnieuw te starten.
  Vervolgens wordt het proces gewekt en uit de SCL verwijderd.

  b)
  Indien het transmit commando reeds door de 586 is uitgevoerd of in uitvoering is, wordt er aan de CL niets gewijzigd. Het proces wordt gewekt d.m.v. een timeout melding, maar blijft wel in de SCL staan. Het SCL element wordt pas verwijderd nadat de INT HANDLER het bij het proces behorende transmit commando aan SDONE heeft aangeboden.

5.4 HET RECEIVE GEDEELTE

Bekijken we vervolgens het complete receive proces. Figuur 5.3 geeft de datastructuren weer die hierbij gebruikt worden.
De pointer beginrfa en endrfa geven het begin resp. het einde aan van de lijst met frame descriptors. (RFD's)
beginfbl wijst naar de eerste vrije receive buffer descriptor. (RBD)
Bij het initialiseren van de MAC-laag (en bij een MRESET) worden beide structuren aangemaakt. Het aantal RFD's en RBD's kan in de file MACSTRUC.H worden opgegeven en deze waarden moeten zodanig gekozen worden dat er altijd voldoende "resources" voor de 586 beschikbaar zijn. (Deze waarden moeten t.z.t. experimenteel worden vastgesteld.)



figuur 5.3 - Datastructuren receive proces

Het proces RDONE verwerkt alle ontvangen frames en de timeout meldingen vanuit het TIMER proces. In het geval van een ontvangen frame controleert RDONE allereerst of het frame geen fouten bevat. Opgemerkt dient te worden dat dit in principe nooit mogelijk is, omdat in de default configuratie de 586 alle foutieve frames negeert. Mocht er t.g.v. een andere 586 mode toch foutieve frame's aan RDONE worden aangeboden, dan worden deze eveneens genegeerd en wordt de RFD + RBD's weer in de RFA gezet.
Indien echter het frame zonder fouten wordt aangeboden, wordt in de RCL gezocht welk extern proces in staat is dit frame te ontvangen. Dit gebeurt volgens een first-come-first-served mechanisme, waarbij het eerste proces dat voldoende bufferruimte beschikbaar heeft wordt geselecteerd.

RDONE geeft vervolgens via mailbox 8 deze informatie door aan FCOPY, waarna de ontvangen data gekopieerd kan worden.
Het FCOPY proces zal tevens de RFD en RBD's weer aan de RFA toevoegen.

Een timeout van het TIMER proces heeft tot gevolg dat het externe proces uit de RCL verwijderd wordt en een TIMEOUT ontvangt.

5.5 HET TIMER PROCES

Dit proces bestaat naast de eigenlijke timer() functie uit een interrupt handler voor het verwerken van de timer-interrupts. Als interruptbron kan een van de 186-timers worden gebruikt.
Alle opdrachten worden via mailbox 5 aangeboden. De message in mailbox 5 bevat de volgende informatie:

- Een pointer naar een SCL of RCL element. (Dit is een indirecte pointer naar een extern proces)

- De timeout waarde. (14 bits maximaal = 16.000 timer ticks)
  Een timeout waarde gelijk nul betekent het verwijderen van een eerdere aanvraag.

- Een proces identificatie. (2 bits) Dit veld wordt gebruikt om te bepalen of een timeout moet worden geplaatst bij het SDONE, dan wel het RDONE proces.

Alle aanvragen worden cummulatief gesorteerd in een queue.

Het TIMER proces is nog niet getest. De reden hiervoor is dat dit proces in feite deel uit moet maken van de LEX (een algemene LEX service) en het behoort geen onderdeel van de MAC-laag te zijn.

Wel is in appendix C een listing van het complete timer proces opgenomen. Deze listing zou eventueel gebruikt kunnen worden om een universeel LEX-TIMER proces te realiseren.

Bij de implementatie van de software is uiteraard wel rekening gehouden met een timer proces, d.w.z. alle vmail-operaties op mailbox 5 worden uitgevoerd. Ook de afhandeling van een timeout wordt door de processen SDONE en RDONE uitgevoerd.

Gedurende de testfase is gewerkt met een dummy (leeg) timer proces.

5.6 DE LLC LAAG

Deze laag is slechts voor een klein gedeelte gerealiseerd. De reden hiervan is het nog niet beschikken over de definitieve IEEE-802 standaardisatie voorstellen. (LLC en Internetworking)
Op het ogenblik verzorgt deze laag de LLC <-> MAC interface en ondersteund de LLC-klasse 1 service, echter zonder gebruik te maken van de Service Access Points.

De SAP's kunnen vrij eenvoudig gerealiseerd worden door in de LLC laag een lineaire lijst met SAP adressen op te nemen. Ieder element van deze lijst bevat dan specifieke informatie over een SAP. (Dezelfde methode is ook toegepast bij de RCL lijst. Het RDONE proces kan in deze lijst informatie over een LLC_IDU vinden.)
De LLC_SAP_ACTIVATE service primitieve creeert een SAP, hetgeen overeenkomt met het toevoegen van een nieuw element aan deze SAP_lijst. (LLC_SAP_DEACTIVATE verwijdert een element.)

Een definitieve keuze voor de LLC-laag kan pas gemaakt worden indien meer informatie over het internetworking model verkregen is. Pas dan is ook bekend welke service de LLC_laag moet aanbieden.

VI VOORTGANG

Op dit moment zijn er een aantal zaken gerealiseerd en getest. Het betreft hier dan de (Ethernet) hardware en de software voor de besturing van deze hardware. (t/m LLC_interface)
In dit hoofdstuk zal ik enige voorstellen doen op welke wijze het LAN projekt zou kunnen worden voortgezet.

HARDWARE

Hiermee zijn geen problemen meer te verwachten. De hardware is getest in zowel interne als externe "loopback"-mode bij snelheden tussen de 4Mbaud en 10Mbaud.

Een volgende fase is de realisatie van een point-to-point verbinding van twee ethernet stations. Hierdoor kan ook het transmit & receive gedeelte van de serieele controller worden getest, alsmede de generatie van het "collision" signaal.

De laatste stap vormt dan de koppeling van 3 of meer stations aan de ethernet kabel d.m.v. een transceiver. Het enige probleem hierbij is de hoge prijs van een transceiver. (f 1200,-)
Als alternatief voor deze dure transceiver kan t.z.t. ook worden gekozen voor de z.g. cheapernet transceiver chip, die momemteel door National Semiconductor wordt ontwikkeld.

SOFTWARE

Op software gebied zal er nog veel moeten gebeuren. Voor wat betreft de tot nu toe gerealiseerde software is het aan te bevelen om:

- Deze allereerst om te zetten naar de nieuwe (definitieve) versie van de LEX.

- Het timer proces uit de MAC-laag te halen en de LEX te voorzien van een universeel TIMER proces. Tevens verdient het aanbeveling om in de LEX enige "debug" faciliteiten op proces-niveau op te nemen. Hierbij kan gedacht worden aan het opvragen en modificeren van LEX objecten, zoals: Proces descriptors, handler tabellen, semaphoren en mailboxes. Hierdoor ontstaat de mogelijkheid om gedurende de testfase het programma verloop op eenvoudige wijze te beinvloeden.
  Ook de monitor op de 186 kaart zou moeten worden uitgebreid. Met name de mogelijkheid om meerdere "breakpoints" te specificeren moet zo snel mogelijk worden geimplementeerd.

Bij de verdere ontwikkeling van de software verdient het

aanbeveling allereerst voldoende informatie over het IEEE-802 project te verzamelen. Dit betreft dan de (nieuwe specificaties) van de LLC laag en het internetworking voorstel.
Uit eerdere publicaties van deze werkgroep (z.g. Draft Proposals) is reeds gebleken dat deze voorstellen zeer gedetailleerd zijn uitgewerkt en een complete beschrijving van alle te realiseren functies geven.

In dit verslag is een afleiding gegeven van de performance van ethernet. Deze afleiding heeft betrekking op het medium access protocol en de physische laag. Uit deze analyse blijkt dat van de 10Mbaud snelheid op het medium een effectieve snelheid van slechts 4Mbaud overblijft, zodra meerdere stations het netwerk gebruiken. Deze analyse zou uitgebreid kunnen worden. Hierbij wordt dan gedacht aan:

- De 80186 en de 586 communicatie. Omdat beide processoren dezelfde lokale bus gebruiken betekent dit dat op het moment dat de 586 een frame verzendt of ontvangt de performance van de 80186 meer dan gehalveerd wordt.

- Wat is de overhead van de LEX ?.
  Door toepassing van parallelle processen ontstaat weliswaar een zeer overzichtelijke programma structuur, maar dit heeft zeker nadelige konsekwenties op de totale performance. Een eenvoudige rekensom leert dat het ongeveer 5 ms duurt om een ethernet frame te verzenden of te ontvangen. Gedurende deze tijd kunnen ongeveer 5000 instructies door de 80186 worden uitgevoerd. (Mits de 586 niet actief is)
  Afgevraagd kan worden hoeveel hiervan gebruikt wordt door de LEX bij het opstarten van een volgend proces en uit hoeveel processen de totale netwerk software bestaat ??

Kortom, het verdient zeker aanbeveling om dit nader te bestuderen. De totale performance van het netwerk is direct bepalend voor het totaal aantal toegestane gebruikers en de toepassing (applicatie) ervan.

A. HET CSMA/CD BACKOFF -ALGORITME

Afgeleid kan worden dat bij CSMA/CD de kans dat slechts <u>een</u> station start met zenden gelijk is aan:

$$Q = n.p\,(1 - p)^{n-1}$$

Een optimum vinden we door:

$$\frac{\partial Q}{\partial p} = n\,(1 - p)^{n-1} - n.p\,(n - 1)(1 - p)^{n-2}$$

gelijk nul te stellen. Dit geeft:

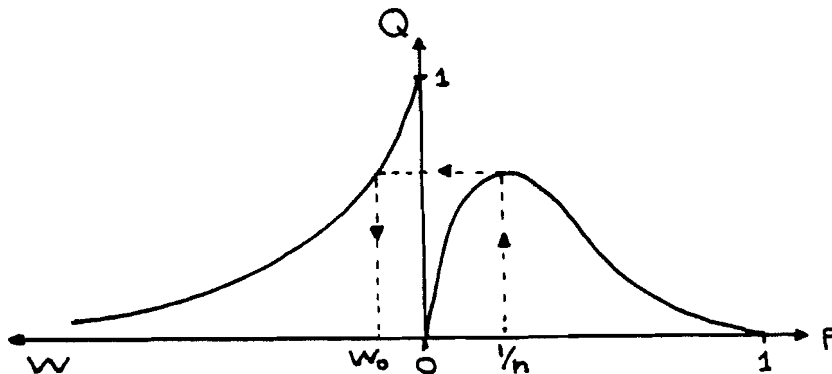
$$p = 1/n \qquad n = (1, \ldots, N)$$

In figuur A.1 is Q(p) uitgezet voor een bepaalde waarde van n.

Verder is afgeleid dat het "gemiddeld" aantal botsingen gelijk is aan:

$$W = \frac{1 - Q}{Q}$$

Deze functie is eveneens in figuur A.1 weergegeven.



figuur A.1 - De functies Q(p) en W(Q).

Ieder "back-off" algoritme zal de kans dat een station start met zenden (de individuele zendkans p) verkleinen nadat een botsing is geconstateerd. In Ethernet wordt p telkens met een factor 1/2 vermenigvuldigd en deze methode staat bekend onder de naam <u>Binary Exponential Backoff</u>
Meer algemeen geldt echter dat iedere aanpassing van p, afhankelijk van het aantal botsingen, moet leiden tot de optimale waarde van 1/n.

Uit figuur A.1 blijkt dit onmiddellijk.

- Als $p > 1/n$ neemt W toe t.g.v. de toenemende botsingskans. Het "backoff" algoritme zal in dat geval p verkleinen.

- Als $p < 1/n$ neemt W eveneens toe. Dit heeft tot gevolg dat Tcont groter wordt. De botsings kans neemt hierdoor af, ofwel Q neemt toe. In dat geval moet ook p toenemen. (Merk op dat deze situatie niet stabiel is. In de Ethernet standaard wordt telkens gestart met p=1)

Door toepassing van een "backoff" algoritme zal de zendkans p na verloop van tijd altijd de optimale waarde 1/n bereiken. De snelheid waarmee dit gebeurt is echter afhankelijk van het gekozen algoritme.

-0-0-0-0-0-0-

Q = de kans dat slechts een station uit de verzameling van n actieve stations start met zenden.

p = de kans dan een (individueel) station start met zenden.

W = is het gemiddeld aantal botsingen.

```
#include "structs.h"

#define INT586 15
#define T2INT 19
#define NILL 0xFFFF
#define MINBSIZE 32      /* minimum buffersize */
#define NOFRECB  10      /* number of receive buffers */
#define RBSIZE   128     /* receive buffer size */

/* COMMANDS FOR THIS 82586 DRIVER */
#define MSEND 1          /* send a frame */
#define MRECEIVE 2       /* receive a frame */
#define MRESET   127     /* reset */

#define VARSIZE 18       /* size in words of varpart in command descriptor */
#define OFFSET  4        /* start of sendbuffer specification in cmd descr. */

/* process identification */
#define SENDID 0
#define RECID 1
#define INTERID 2
#define EXTRAID 3

/* commands for the 82586 */
#define NOP 0
#define START 1
#define RESUME 2
#define SUSPEND 3
#define ABORT 4

/* status of the 82596 */
#define IDLE 0
#define SUSPENDED 1
#define CREADY 2
#define NORESRC 2
#define RREADY 4

/* action commands for the 82586 */
#define AXNOP  0
#define AXADRS 1
#define AXCONF 2
#define AXMCST 3
#define AXTRMT 4
#define AXTDR  5
#define AXDMP  6
#define AXDGN  7
```

```
/* data for configure command (12 bytes) */
#define CONFDAT0 0xe80c
#define CONFDAT1 0x66bf /*internal loopback + save bad frames */
#define CONFDAT2 0x6008
#define CONFDAT3 0xfa00
#define CONFDAT4 0x8801 /*collision and carrier sensed internal */
                                /* also set promiscuous mode */
#define CONFDAT5 0xff40

/* constants used by timer proces */
#define SIZE 8       /* max numbers of processes that the driver can handle */
#define MAXTIME 0x3fff

/* status return */
#define ERROR -1
#define OK 1
#define BADFMT 1          /* BAD command descriptor format */
#define TOBUSY 2          /* NO RESOURCES available for handling this call */
#define TIMEOUT 3         /* TIME OUT */
#define TOSMALL 4         /* Your receive buffers are too small */
#define CANCELLED 5       /* command is canceled, due to a reset command */

/****************************************************************************
 *                                                                          *
 *     SYSTEM CONTROL BLOCK                                                 *
 *                                                                          *
 ****************************************************************************/

struct sysctrlbl
{ unsigned s_xxxx1   : 4;
  unsigned s_rus         : 3;
  unsigned s_xxxx2   : 1;
  unsigned s_cus         : 3;
  unsigned s_xxxx3   : 1;
  unsigned s_status  : 4;
  union { unsigned s_command;
          struct indiv
          { unsigned s_xxxx4   : 4;
            unsigned s_ruc     : 3;
            unsigned s_reset   : 1;
            unsigned s_cuc     : 3;
            unsigned s_xxxx5   : 1;
            unsigned s_ack     : 4;
          } rcc;
        } s_cmd;
 struct cmdb *s_cblp;
 struct rframed *s_rfrp;
 unsigned s_crcerrs;
 unsigned s_alnerrs;
 unsigned s_rscerrs;
 unsigned s_ovrnerrs;
 } scb;
```

```
struct mailbox *mb1, *mb2, *mb3, *mb4, *mb5, *mb6, *mb7, *mb8;
struct semaphore *sem1, *sem2;

long ds;        /* data segment register */
struct message intmsg;  /* message used by interrupt handler */
long make_adr();

/******************************************************************************
 *                                                                            *
 *   HANDLER COMMAND LIST  Gives some extra info for send/receive processes   *
 *                                                                            *
 ******************************************************************************/

struct rcl              /* receive command list */
{ struct rcl *rcl_next;
  struct cmdd *rcl_cdp;
} *rfirst, *rfree, reccl[SIZE];

struct scl              /* send command list */
{ struct scl *scl_next;
  struct cmdd *scl_cdp;
  struct cmdb *scl_cbp;
} *sfirst, *sfree,  sendcl[SIZE];

/******************************************************************************
 *                                                                            *
 *  COMMAND DESCRIPTOR  Note that parm[1] gives the startadress of varpart    *
 *                                                                            *
 ******************************************************************************/

struct cmdd
{ unsigned cd_command;
  unsigned cd_stat_out;
  struct semaphore *cd_weksem;
  unsigned cd_bufsize : 13;
  unsigned cd_nof_buf : 3;
  int cd_parm[18];
};
```

```
/*************************************************************************
 *                                                                       *
 *    RECEIVE FRAME AREA   DATA STRUCTURES                               *
 *                                                                       *
 *************************************************************************/

struct rframed          /* receive frame descriptor */
{ unsigned rf_status : 13;
  unsigned rf_rokflag: 1;
  unsigned rf_busy       : 1;
  unsigned rf_complete : 1;
  int rf_xxxx1      : 14;
  unsigned rf_suspend : 1;
  unsigned rf_endlst : 1;
  struct rframed *rf_next;
  struct rbufd *rf_rbdp;
  unsigned rf_dstaddr[3];
  unsigned rf_srcaddr[3];
  unsigned rf_type;
} *beginrfa, *endrfa, rfdes[SIZE/2];

struct rbufd            /* receive buffer descriptor */
{  unsigned rb_count : 14;
   unsigned rb_filled : 1;
   unsigned rb_eof        : 1;
   struct rbufd *rb_next;
   long rb_address;
   unsigned rb_size : 14;
   int rb_xxxx1  : 1;
   unsigned rb_endlst : 1;
} *beginfbl, *endfbl, rbdes[NOFRECB];

char recbuf[NOFRECB][RBSIZE];

int noffd, nofbd;
```

```
/*************************************************************************
 *                                                                       *
 *   COMMAND LIST AREA                                                   *
 *                                                                       *
 *************************************************************************/

struct cmdb           /* 586 command block */
{ unsigned cb_status : 12;
  unsigned cb_aborted : 1;
  unsigned cb_ok         : 1;
  unsigned cb_busy       : 1;
  unsigned cb_complete : 1;
  unsigned cb_command : 3;
  int cb_xxxx1     : 10;
  unsigned cb_e_int   : 1;
  unsigned cb_suspend : 1;
  unsigned cb_endlst  : 1;
  struct cmdb *cb_next;
  unsigned cb_var[7];
} *cbfirst, *cbfree, *cbend, cmdblk[SIZE/2];

/*************************************************************************
 *                                                                       *
 *    TRANSMIT BUFFER DESCRIPTORS Note that transmitbuffer are located   *
 *                                    in data area of process.           *
 *************************************************************************/

struct tbufd           /* transmit buffer descriptor */
{ unsigned tb_count : 14;
  int tb_xxxx1   : 1;
  unsigned tb_eof       : 1;
  struct tbufd *tb_next;
  long tb_address;
  unsigned tb_size      : 14;
  unsigned tb_xxxx2 : 1;
  unsigned tb_el        : 1;
} *tbfree, txbufd[SIZE*3];
```

```
/*****************************************************************************
 *                                                                           *
 *   TIMER PROCESS COMMON DATA                                             *
 *                                                                           *
 *****************************************************************************/

struct timerq
  { struct timerq *t_next;
    int t_interval;
    unsigned t_procid;
    int *t_help;
  } twq[SIZE*2], *tfirst, *tfree;

union timermsg
{ int imsg;
  struct sepmsg
  { unsigned timeout : 14;
    unsigned procid  : 2;
  } smsg,
} tmsg;

/* data used by external process */
char exbuf1[64], exbuf2[64], exbuf3[160];
struct cmdd excmd[2];
```

```
# include "macstruc.h"
# include "sysdata.h"
# include "lexsupp.h"

/**** DISPATCHER ******/
/* Function of this proces is to receive commands from external processes
   and send this command to INTERNAL, RECEIVE or SEND process
   There are also some special functions within this dispatcher, like:
   - RESET CU
   - RESET RU
   - and even more to come. */


dispatch()
{ struct message msg;
  int inthdlr();

  rmail(mb1 = sys_lo.mbox0 + 0);
  rmail(mb2 = sys_lo.mbox0 + 1);
  rmail(mb3 = sys_lo.mbox0 + 2);
  rmail(mb4 = sys_lo.mbox0 + 3);
  rmail(mb5 = sys_lo.mbox0 + 4);
  rmail(mb6 = sys_lo.mbox0 + 5);
  rmail(mb7 = sys_lo.mbox0 + 6);
  rmail(mb8 = sys_lo.mbox0 + 7);

  r(sem1 = sys_lo.sem0, 1);
  r(sem2 = sys_lo.sem0 + 1, 1);

  newds(ds = newds(01));        ds = ds << 4;

  rfirst = (struct rcl *)NILL; sfirst = (struct scl *)NILL;
  beginrfa = endrfa = (struct rframed *)NILL;
  cbfirst = cbend = (struct cmdb *)NILL;

  attach(INT586, inthdlr, INDIRECT);

  setscb();

 /* now do a diagnose and a configure command (internal loopback) */

  if(!diagnose(cmdblk)) writeln("diagnose failed !");
  if(!configure(cmdblk)) writeln("configure command failed");

  scb.s_cmd.s_command = NULL;
  scb.s_crcerrs = NULL;
  scb.s_alnerrs = NULL;
  scb.s_rscerrs = NULL;
  scb.s_ovrnerrs = NULL;
```

```
  greset();

/* at this place we should executed the following commands:
        - diagnose
        - configure
        - set individual adddress
        - set multicast address
*/

  myprioty(80); /* start other processes */

do { pmail(mb1, &msg, WAIT);
    msg.lo = NULL; /*  in this version i don't use long pointers */
       switch(((struct cmdd *)msg.hi)->cd_command) {

case MRESET  : greset(); break;
case MRECEIVE: vmail(mb3, &msg, WAIT); break;
case MSEND   : vmail(mb2, &msg, WAIT); break;
default:       wake_up((struct cmdd *)msg.hi, BADFMT); };
  }while(TRUE);
}

greset ()
{ curst(); rurst();}

/******************************************************************************/
/*                                                                            */
/*              SPECIAL FUNCTIONS                                             */
/*                                                                            */
/******************************************************************************/


rurst() /* reset all receive operations */
        /* first abort RU
           release all rcl elements and signal waiting processes with
           a BREAK.
           Move all receive- frame and buffer descriptors to the free lists.
           reset mailbox 3, 8 and 7. */
{
struct rframed *q;
struct rbufd *r;
struct rcl *s;
int i;

disable(INT586);        /* must be done, otherwise the int handler will
                           start the receive unit again */

scb.s_cmd.rcc.s_ruc = ABORT;
ca(); while(scb.s_cmd.s_command != NULL) ;
```

```
/* setup frame descriptor list */
beginrfa = q = &rfdes[0]; i = 0;
while(i < SIZE/2) {     q->rf_status = 0;
                        q->rf_rokflag = q->rf_busy = q->rf_complete = 0;
                        q->rf_suspend = q->rf_endlst = 0;
                        q = q->rf_next = &rfdes[i++] ;};
(endrfa = q)->rf_next = (struct rframed *)NILL;
endrfa->rf_endlst = 1;
noffd = SIZE/2;

/* setup receive buffer descriptor list */
beginfbl = r = &rbdes[0]; i = 0;
while(i < NOFRECB) {    r->rb_count=0;
                        r->rb_filled = r->rb_eof = 0;
                        r->rb_address = make_adr(&recbuf[i][0]);
                        r->rb_endlst = 0;
                        r->rb_size = RBSIZE;
                        r = r->rb_next = &rbdes[i++]; };
(endfbl = r)->rb_next = (struct rbufd *)NILL;
endfbl->rb_endlst = 1;
nofbd = NOFRECB;

/* setup link to receive buffer descriptor */
beginrfa->rf_rbdp = beginfbl;

/* wake up all waiting processes */
while(rfirst != (struct rcl *)NILL) { wake_up(rfirst->rcl_cdp, CANCELLED);
                        settmr(0, RECID, (int)rfirst);
                        rfirst = rfirst->rcl_next; };

/* now interrupt can be enabled, because RU won't be started */
enable(INT586);

/* setup a rcl list */
rfirst = (struct rcl *)NILL;
rfree = s = &reccl[0]; i = 1;
do s = s->rcl_next = &reccl[i++]; while(i < SIZE);
s->rcl_next = (struct rcl *)NILL;

/* reset mailboxes */
rmail(mb3); rmail(mb7); rmail(mb8);
} /* rurst */

curst()         /* reset CU */
{
struct cmdb *p;
struct tbufd *r;
struct scl *s;
int i;

scb.s_cmd.rcc.s_cuc = ABORT;
ca(); while(scb.s_cmd.s_command != NULL);
```

```
/* first wakeup all waiting processes */
while(sfirst != (struct scl *)NILL) { settmr(0, SENDID, (int)sfirst);
                        wake_up(sfirst->scl_cdp, CANCELLED);
                        sfirst = sfirst->scl_next; };

/* setup a clean scl list */
sfirst = (struct scl *)NILL;
sfree = s = &sendcl[0]; i = 1;
do s = s->scl_next = &sendcl[i++]; while(i < SIZE);
s->scl_next = (struct scl *)NILL;

/* move action command blocks and transmit buffer descriptors to free list */
cbfirst = cbend = (struct cmdb *)NILL;
cbfree = p = &cmdblk[0]; i = 1;
do p = p->cb_next = &cmdblk[i++]; while(i < SIZE/2);
p->cb_next = (struct cmdb *)NILL;

tbfree = r = &txbufd[0]; i = 1;
do r = r->tb_next = &txbufd[i++]; while(i < SIZE*3);
r->tb_next = (struct tbufd *)NILL;

rmail(mb6); rmail(mb2);
}

long make_adr(i)        /* this functions returns a long address */
unsigned i;
{ return (ds + (long)i);
}

setscb()        /* setup scb */
{ struct intpntr        /* intermediate system pointer */
  { int ip_busy;
    int ip_scboffs;
    long ip_scbbase;
  } *iscp;
  long svds;

        svds = newds(0x10001);
        iscp = (struct intpntr *)0xfff0;

        iscp->ip_busy=0x0001;   /* busy flag */
        iscp->ip_scboffs=(int)&scb;
        iscp->ip_scbbase=(long)ds<<4;

        ca();

        if( iscp->ip_busy) writeln("586 didn't answer");
        newds(svds);
        return;
}
```

```
/**** SEND SETUP ******/
/*
 * first do some sanity checks, like:
        - is bufsize equal to the individual sum of the sendbuffer size.
        - are all sendbuffer sizes >= the minimum size
        - are there enough transmitbuffer descriptors available
        - is a action commandblock available
        - is an element in the scl list available
 * then fill in action command, transmitbuffer descriptors and scl element.
 * start timer and add action command to command list */


ssetup()        /* main function of send_setup process */
{ struct message msg;
  struct scl *sss, *sclp;
  struct cmdd *ssd;     /* pointer to external command */
  struct cmdb *ssc;     /* pointer to 586 transmit command */
  struct tbufd *sstf, *sstl; /* pointer to first & last transmitbuffer descr. */
  int nbuf, i;

do {
ssl: pmail(mb2, &msg, WAIT);
     nbuf = (ssd = (struct cmdd *)msg.hi)->cd_nof_buf;
     if( !scmdchk(ssd) ) wake_up(ssd, BADFMT); else

        /* check if resources are available */

        if(cbfree == (struct cmdb *)NILL || sfree == (struct scl *)NILL)
                /* no room to handle this command */
                wake_up(ssd, TOBUSY);
        else /* now look if there are transmitbuffers descriptors */

                if((sstl=sstf=tbfree) == (struct tbufd *)NILL) wake_up(ssd, TOBUSY);
              else { /* check for enough tx buf descriptors */
              i = 0;
              while(++i <nbuf) { sstl = sstl->tb_next;
                 if(sstl == (struct tbufd *)NILL) { wake_up(ssd, TOBUSY); goto ssl; };
                             };
              cbfree = (ssc = cbfree)->cb_next;
              sfree = (sss = sfree)->scl_next;
              tbfree = sstl->tb_next;

       /* ssc points to action command block
          sss  "     "  element of scl list
          ssd  "     "  command descriptor
          sstl "     "  last transmit buffer descriptor
          sstf "     "  first "         "       "           */
        ssc->cb_var[0] = (unsigned)sstf; /* set link to txbufdescr. */
        ssc->cb_command = AXTRMT;        /* command is transmit     */
        ssc->cb_suspend = 0;      /* disable suspend option  */
        copy((char *)&(ssd->cd_para[0]), (char *)&(ssc->cb_var[1]), 8); /* copy dest. addr and field */
```

```
      /* action command block is filled now, continue with
         transmit buffer descriptors */
       /* note that status field and the busy,complete endlist
          and interrupt flags are filled in by the function add() */


      i = 0;
      while(i < nbuf) { sstf->tb_eof = 0;
                        sstf->tb_count = ssd->cd_parm[OFFSET   + 1 + 2*i];
                        sstf->tb_address = make_adr(ssd->cd_parm[OFFSET + 2*i++]);
                        sstf = sstf->tb_next; };
      sstl->tb_eof = 1; sstl->tb_next = (struct tbufd *)NILL;

      /* fill in scl element and place it at end of the list */
      sss->scl_next = (struct scl *)NILL;
      sss->scl_cdp     = ssd;
      sss->scl_cbp     = ssc;
      sclp = sfirst;
      if(sclp == (struct scl *)NILL)   sfirst = sss;
       else {   while(sclp->scl_next != (struct scl *)NILL) sclp = sclp->scl_next;
            sclp->scl_next = sss; };

      /* start timer */
      if((ssd->cd_stat_out > MAXTIME) || (ssd->cd_stat_out == 0))
           settmr(MAXTIME, SENDID, (int)sss);
      else settmr(ssd->cd_stat_out, SENDID, (int)sss);

      add(ssc);
      }; /*else*/
    }while(TRUE);
}

scmdchk(p)      /* check command descriptor for correct syntax */
struct cmdd *p;
{ int i, size;
  register int j;
  i = size = 0;
  while(i < p->cd_nof_buf) {
                        j = p->cd_parm[OFFSET + 1 + 2*i++];
                        if(j < MINBSIZE) return FALSE; else size+=j; };
       return size == p->cd_bufsize;
}
```

```
/***** SEND DONE *****/
/* if a time_out is received, then cancel current command.
   If this is impossible, because the 82586 is currently executing this
   command, then don't remove the process from the scl list and set
   *scl_cdp     to NILL. If 82586 is not executing the send command, then
   remove transmit buffer descriptors from command block and change
   transmit command to a NOP. (This is done because there's no other way
   to remove an command from a dynamic list.)
   If a Command block was executed, then the following actions are taken.
   if *scl_cdp==NILL then the process is already timed_out,     so move commandblock
   and tbuffer descriptors to the free list.
   if an error occurred then add commandblok at end of the list, so it will be
   executed again.
   if the send was errorfree, then the timer is stopped, the commandblok
   and the transmitbuffers are moved to the free list, and the proces is
   signaled. */


sdone()
{
struct message msg;
do{ pmail(mb6, &msg, WAIT);
    if (msg.lo  == NULL) s_time_out((struct scl *)msg.hi);
    else txdone((struct cmdb *)msg.hi);
  } while(TRUE);
}/*sdone*/

s_time_out(p)   /* received a timeout */
struct scl *p;
{ struct scl *q;
  while((q = sfirst) != (struct scl *)NILL) {   if(q == p) goto match ;
                                q = q->scl_next; };      return;

  match:
  /* try to cancel outstanding command and signal process */

  if(cancel(p->scl_cbp) ) sdeq(p); /* remove process from sfirst list */
    else /* command is under execution, so don't remove element */
         p->scl_cdp     = (struct cmdd *)NILL;
  v(p->scl_cdp->cd_weksem);
}

txdone(p)       /* a transmit commandblock is executed */
struct cmdb *p;

{ struct scl *q;        /* first search for the process that is waiting */
if((q = sfirst) == (struct scl *)NILL) { /* no processes are waiting */
                         cdeq(p); return; };
while(q->scl_cbp != p) {q = q->scl_next;
                   if(q == (struct scl *)NILL) { cdeq(p); return; };
                  };
```

```
/* found matching element in scl-list, first    look if there was a timeout */
if(q->scl_cdp == (struct cmdd *)NILL) { /* timeout      */
                    cdeq(p); sdeq(q); return; };

/* o.k. there's a frame send, the timer is still running and
   there's a process waiting */
if(p->cb_ok != OK)       { /* error while sending frame */
                           /* do a retransmission */
                 add(p); q->scl_cdp->cd_stat_out = ERROR; return; };

/* the frame is send errorfree, so stop timer and signal process */
settmr(0, SENDID, (int)q); cdeq(p); sdeq(q); wake_up(q->scl_cdp, OK);
}

cdeq(p)         /* place commandblock and transmitbuffer descriptors to
                   cbfree and tbfree list */
struct cmdb *p;
{
if(p->cb_command == AXTRMT) freetxb((struct tbufd *)p->cb_var[0]);
p->cb_next = cbfree;
cbfree = p;
}

cancel(p)       /* try to cancel action command, pointed by p */
struct cmdb *p;
{ int errcode;
  disable(INT586);
  suspnd();
  if(p->cb_busy == 0 && p->cb_complete == 0) {
    p->cb_command = AXNOP;
    freetxb((struct tbufd *)p->cb_var[0]);/* remove transmit buffers descr. */
    errcode = TRUE; } else errcode = FALSE;
  resume(); enable(INT586); return errcode;
}

freetxb(p)      /* move tbufd's to tbfree list */
struct tbufd *p;
{ struct tbufd *tl;
  if(p == (struct tbufd *)NILL) return; else tl = p;
  while(tl->tb_next != (struct tbufd *)NILL) tl = tl->tb_next;
  tl->tb_next = tbfree; tbfree = p;
}

suspnd()
{
scb.s_cmd.rcc.s_cuc     = SUSPEND;
ca();
}
```

```
resume()
{
scb.s_cmd.rcc.s_cuc     = RESUME;
ca();
}

add(p)          /* add a action command block at the end of
                   command list. ( command and tbufdescriptor
                   must be set ) */
struct cmdb *p;
{
unsigned *up;
up = (unsigned *)p; *up = 0; /* clear status field */
p->cb_endlst = p->cb_e_int = 1;
p->cb_busy = p->cb_complete = 0;
p->cb_next = (struct cmdb *)NILL;
disable(INT586);
if(cbfirst == (struct cmdb *)NILL) { /* setup a new list */
                     cbfirst = cbend = p;
                     scb.s_cblp = p;
                     cu_start(); }
 else { cbend->cb_next = p;
        cbend->cb_endlst = 0; /* dynamic list */
        cbend = p; };
enable(INT586);
}

sdeq(p)         /* deque element pointed by p from sfirst list
                   and add this element at the end of sfree list */
struct scl *p;
{
struct scl *q, **r;
q = *(r = &sfirst);
while((q!=p) && (q!=(struct scl *)NILL)) q = *(r = &q->scl_next);
if(p!=q) return;        /* end of list */

if(q != (struct scl *)NILL) *r = q->scl_next; else *r = (struct scl *)NILL;

if((q = sfree) != (struct scl *)NILL)   {
  while(q->scl_next != (struct scl *)NILL) q = q->scl_next;
  q->scl_next = p; }
else sfree = p;
p->scl_next = (struct scl *)NILL;
}
```

```
/****   RECEIVE SETUP ****/

/* - first look if there's an empty rcl element. If not then cancel
     and signal process with a TOBUSY.
   - check for proper command format.
   - setup an rcl element and start TIMER process.
   - start the Receive Unit */


rset()
{
struct cmdd *cmdp;
struct message msg;
struct rcl *rclp, *p;

do { pmail(mb3, &msg, WAIT);
     cmdp = (struct cmdd *)msg.hi;
     if(rfree == (struct rcl *)NILL) /* no entrys on rcl available, signal process */
                    wake_up(cmdp, TOBUSY);

     else { if ( !cmdcheck(cmdp) ) /* bad command format */
                                   wake_up(cmdp, BADFMT);
     else { /* command format is ok and there's an rcl element available */
            rclp = rfree;       /* take an element */
            rfree = rfree->rcl_next;
            rclp->rcl_next = (struct rcl *)NILL;
            rclp->rcl_cdp = cmdp;

            /* add rclp to end of rcl list */
            if(rfirst == (struct rcl *)NILL) { /* setup rfirst list and start   RU */
                                 rfirst = rclp; ru_start();}
            else { p = rfirst;
                   while(p->rcl_next != (struct rcl *)NILL) p = p->rcl_next;
                   p->rcl_next = rclp;
                 }
            /* start the timer and then it's done */
            if((cmdp->cd_stat_out > MAXTIME) || (cmdp->cd_stat_out == 0))
                 settmr(MAXTIME, RECID, (int)rclp);
              else settmr(cmdp->cd_stat_out, RECID, (int)rclp);
              }
        }
   }while(TRUE);
}/*rset*/
```

```
wake_up(p, cause)        /* signal waiting proces */
unsigned cause;
struct cmdd *p;
{
p->cd_stat_out = cause;
v(p->cd_weksem);
}

cmdcheck(p)      /* check command descriptor for valid buffer size */
struct cmdd *p;
{ int size, i;
  i = size = 0;
  while(i < p->cd_nof_buf) { size += p->cd_parm[OFFSET + 1 + 2*i++]; };
        return size == p->cd_bufsize;
}

settmr(value, id, hclp) /* set timer */
unsigned value; /* timeout value or 0 (=cancel) */
unsigned id;     /* process id */
int hclp;        /* pointer value to scl or rcl element */
{
union timermsg tmsg;
struct message msg;

tmsg.smsg.timeout = value;
tmsg.smsg.procid = id;
msg.hi  = tmsg.imsg;
msg.lo  = hclp;
vmail(mb5, &msg, WAIT);
}
```

```
/*** RECEIVE DONE ***/

/* read mailbox 7 and if a timeout message arrives then remove waiting process
   from rcl list and place empty rcl element at the end of rcl-free list.
   If a frame receive message arrives, then check if an error occurred.
   (In ethernet configuration this is impossible, because bad frames
   aren't saved !) But if there's an error then start copy process with
   an NILL pointer to command descriptor. If the frame was good, then start
   the copy proces and place in mailbox 8 a pointer to the frame descriptor
   and a pointer to the command descriptor. Remove the process from the
   rcl list and add rcl element at the end of rfree list. */


rdone()
{
struct message msg;
int framesize;
struct rcl *p;
do{ pmail(mb7, &msg, WAIT);
    if(msg.lo == NULL)  /* we received a timeout from TIMER */
                        r_time_out((struct rcl *)msg.hi);
    else /* we received a frame ! */
        if (((struct rframed *)msg.hi)->rf_rokflag == 0) { /* frame has an error */
                                msg.lo = NILL;
                                vmail(mb8, &msg, WAIT);}
        else /* the frame is error free */
             if((p = rfirst) == (struct rcl *)NILL) {   /* but nobody is waiting for it */
                                     msg.lo  = NILL;
                                     vmail(mb8, &msg, WAIT);}
             else { /* ok now search for a buffersize match */
                    framesize = compsz((struct rframed *)msg.hi);
                    while((p->rcl_cdp->cd_bufsize < framesize)  && (p != (struct rcl *)NILL))
                                    {p->rcl_cdp->cd_stat_out = TOSMALL;
                                     p = p->rcl_next;};
                    if(p == (struct rcl*)NILL) { /* none of the processes has enough
                                  buffer space */
                                  msg.lo = NILL;
                                  vmail(mb8, &msg, WAIT);}
                    else { /* we found a process that can accept this frame */
                           settmr(0, RECID, (int)p);    /* stop timer */
                           msg.lo = (int)p->rcl_cdp;
                           p->rcl_cdp->cd_bufsize = framesize;
                           rdeq(p);
                           vmail(mb8, &msg, WAIT);};
                  };/*else*/
  }while(TRUE);
 }/*rdone*/
```

```
r_time_out(p)
struct rcl *p;
{
rdeq(p);
v(p->rcl_cdp->cd_weksem);
}

rdeq(p)         /* deque element pointed by p from rfirst list
                   and add this element at the end of rfree list */
struct rcl *p;
{
struct rcl *q, **r;
q = *(r = &rfirst);
while((q!=p) && (q!=(struct rcl *)NILL)) q = *(r = &q->rcl_next);
if(p != q) return; /* we reached end of list without a match */

if(q != (struct rcl *)NILL) *r = q->rcl_next; else *r = (struct rcl *)NILL;

if((q = rfree) != (struct rcl *)NILL)   {
  while(q->rcl_next != (struct rcl *)NILL) q = q->rcl_next;
  q->rcl_next = p; }
else rfree = p;
p->rcl_next = (struct rcl *)NILL;
}

compsz(p)       /* compute framsize. p points to a framdescriptor.
                   also reset all filled and eof flags. */
struct rframed *p;
{
int size;
struct rbufd *q;
q = p->rf_rbdp; size = 0;
while(q->rb_eof == 0) {
  q->rb_filled = 0; size += q->rb_count;
  q = q->rb_next; };
q->rb_eof = q->rb_filled = 0;
return size += q->rb_count;
}
```

```
/*******     FCOPY PROCESS *************/

/* this process copies data from receive buffer to buffers of waiting
   process.
   If a NILL cmdp is received then no data will be copied.
   The receive done process has computed already the framesize
   and placed it in location bufsize of the command descriptor.
   All receive buffer are added to RFA and free buffer list is updated */



fcopy()
{

struct rframed *framep;
struct cmdd *cmdp;
struct message msg;
struct rbufd *rbufp, *lrbufp;
unsigned framesize;
unsigned rbcount;        /* counts nof receive buffers */
char *srcp, *dstp;
struct buffer_info
{ unsigned bf_address;
  unsigned bf_size;
} d_buf_info[7], *dbi;
int src_cnt, dst_cnt;
do {
    pmail(mb8, &msg, WAIT);
    lrbufp = rbufp = (framep = (struct rframed *)msg.hi)->rf_rbdp;
    cmdp = (struct cmdd *)msg.lo;

    if(cmdp != (struct cmdd *)NILL) { /* copy received  frame */
    /* first fill d_buf_info with adresses and size of destination buffers
       after that use space in command descriptor for frame's src and dst adress
       and frame type field */

    framesize = cmdp->cd_bufsize;       /* filled in by rdone process */
    dbi = d_buf_info;
    copy((char *)&(cmdp->cd_parm[OFFSET]), (char *)dbi, 28);
    copy((char *)&(framep->rf_dstaddr[0]), (char *)&(cmdp->cd_parm[0]), 14);
    /* now start copy proces */
              rbcount = 1;
           srcp = (char *) get_offset(&(lrbufp ->rb_address));
           dstp = (char *) dbi -> bf_address;
           dst_cnt = dbi -> bf_size;
           src_cnt = lrbufp -> rb_count;
```

```
        while(framesize) {

        if(!src_cnt) { lrbufp=lrbufp->rb_next; src_cnt=lrbufp->rb_count;
                       srcp=(char *)get_offset(&(lrbufp->rb_address));
                       rbcount++; };
        if(!dst_cnt) { dstp=(char *)++dbi->bf_address;
                       dst_cnt=dbi->bf_size; };
        if(src_cnt<dst_cnt) { copy(srcp,dstp,src_cnt); dstp+=src_cnt;
                              dst_cnt-=src_cnt; framesize-=src_cnt; src_cnt=0; }
         else {copy(srcp,dstp,dst_cnt); srcp+=dst_cnt;
               src_cnt-=dst_cnt; framesize-=dst_cnt; dst_cnt=0; };
                                 };
    };/* end if of copy process */

    /* ok now add framedescriptor to RFA and add bufferdescriptor(s)
       to FBL and start receive unit. (only if processes are waiting) */

    /* adding a framedescriptor */
    framep -> rf_endlst = 1; /* it      becomes be the last on the list */
    framep -> rf_busy = framep -> rf_complete = 0;
    framep -> rf_next = (struct rframed *)NILL;
    framep -> rf_rbdp = (struct rbufd *)NILL;
    disable(INT586);
    if(noffd++) {endrfa->rf_next = framep;
                endrfa->rf_endlst = 0;
                endrfa=framep; }
     else beginrfa=endrfa=framep;

    /* adding a recieve buffer descriptor */
    lrbufp -> rb_endlst = 1;
    lrbufp -> rb_next = (struct rbufd *)NILL;
    if(nofbd) {endfbl->rb_next=rbufp;
               endfbl->rb_endlst = 0;
               endfbl=rbufp;  }
     else beginfbl=endfbl=rbufp;
    nofbd+=rbcount; ru_start();
    enable(INT586);

    /* now signal waiting process */
    if(cmdp != (struct cmdd *)NILL) wake_up(cmdp, OK);
  }while(TRUE);
}/*fcopy*/

copy(s, d, i)   /* copy i bytes from s to d */
char *s, *d;
int i;
{ while(i-- != 0) *d++ = *s++;
}

get_offset(p)   /* return offset of the address pointed by p */
long *p;
{ return (int)(*p - ds);
}
```

```
ru_start()      /* start receive unit of 82586 */
                /* The RU is started if there is at least:
                    - one process waiting for a frame to recieve.
                    - two ore more framedescriptors
                    - two ore more receive buffer descritors
                    - the RU must not be ready. */
{
if(scb.s_rus == RREADY || noffd < 2 || nofbd < 2 || rfirst == (struct rcl *)NILL) return;

/* now start RU */
beginrfa->rf_rbdp = beginfbl;
while(scb.s_cmd.s_command != NULL);
scb.s_rfrp = beginrfa;
scb.s_cmd.rcc.s_ruc = START;
ca();
}



/*********** TIMER PROCESS ***************/
/* This process reads an message from mailbox 5. The format of this message
   is: msg.lo : bit 0-13 time out value
              : bit 14 and 15 a proces indification
       msg.hi : a pointer to an element of a rcl list.

    The structure timerq is organised cumulative (based on intervals) */


timer()
{
  struct message msg;

  rmail(mb5);

  while (TRUE)         /* the operational loop */
  { pmail(mb5, &msg, WAIT);
  }
}/* timer */
```

```
/***** INTERRUPT HANDLER FOR 82586 INTERRUPTS *****/
/* All interrupts from the 82586 are acknowledged by this handler, but
   only CX and FR interrupts are serviced !
   CNA and RNR interrupts can only occur when the end of CBL or RFA is
   reached. In that case the RU and CU are automatically started by other
   processes.
   Note that CX interrupt service routine also starts the CU after scanning
   the command list and finding the CU in a idle state. */


inthdlr()
{
  unsigned status;
  while(scb.s_cmd.s_command != 0) ;
  scb.s_cmd.rcc.s_ack = scb.s_status; /* acknowledge interrupt */
  status = (unsigned)scb.s_cmd.rcc.s_ack;
  ca(); while(scb.s_cmd.s_command != 0) ;

        switch(status & 0x000C) {  /* only FR and CX !!! */
case 12 : fr_int(); cx_int(); break;
case  8 : cx_int(); break;
case  4 : fr_int(); break;
                                }; /* switch */
intret(INT586);
}

fr_int() /* we recieved a frame ! */
{

while(beginrfa != (struct rframed *)NILL) { /* do receiveframe processing */
                        /* remove framdescriptors with complete flag set */
        if(beginrfa->rf_complete == 0) { /* sorry for the interrupt */
                                       ru_start(); break; };
        /* found a framedescriptor */
        while(beginfbl->rb_eof != 1) { beginfbl=beginfbl->rb_next; nofbd-- ;};
        beginfbl=beginfbl->rb_next; noffd--; nofbd--;

        intmsg.hi = (int)beginrfa;
        beginrfa = beginrfa->rf_next;
        intmsg.lo = NILL;
        vmail(mb7, &intmsg, WAIT); };
}
```

```
cx_int()
{

while(cbfirst != (struct cmdb *)NILL) { /* do command block processing  */
if(cbfirst->cb_complete == 0) { cu_start(); break; };
      intmsg.hi = (int)cbfirst; intmsg.lo = NILL;
cbfirst = cbfirst->cb_next; vmail(mb6, &intmsg, WAIT); };
}

cu_start()      /* start command unit */
{ if(scb.s_cus == CREADY || cbfirst == (struct cmdb *)NILL) return;

  /* now the CU can be started */

  while(scb.s_cmd.s_command != 0) ;
  scb.s_cmd.rcc.s_cuc = START;
  scb.s_cblp = cbfirst; ca();
}


proc0()         /* external tranmitting proces */
{ struct semaphore *mysem;
  struct message msg;
  unsigned i;

r(mysem = sys_lo.sem0 + 2, 0);
while(TRUE) {   excmd[0].cd_command = MSEND;    /* send command */
                excmd[0].cd_stat_out = 0;       /* no time_out */
                excmd[0].cd_weksem = mysem;
                excmd[0].cd_bufsize = 128;      /* transmit buffer size */
                excmd[0].cd_nof_buf = 2;        /* two buffers */
                excmd[0].cd_parm[OFFSET] = (int)&exbuf1[0];
                excmd[0].cd_parm[OFFSET+1] = 64;
                excmd[0].cd_parm[OFFSET+2] = (int)&exbuf2[0];
                excmd[0].cd_parm[OFFSET+3] = 64;

                msg.hi = (int)&excmd[0];
                writeln("Tx - setup");
                vmail(mb1, &msg, WAIT);
                /*
                 * wait for send *
                                 */
                p(mysem, WAIT);
        if(excmd[0].cd_stat_out == OK) writeln("Tx - completed");
        else writeln("Tx - error");
        /* wait a while */ for(i=0;i<65000;i++) ;
              };
 }
```

```
procl()          /* external receiving proces */
{ struct semaphore *mysem;
  struct message msg;
  unsigned i;

r(mysem = sys_lo.sem0+ 3, 0);
while(TRUE) {   excmd[1].cd_command = MRECEIVE; /* send command */
                excmd[1].cd_stat_out = 0;       /* no time_out */
                excmd[1].cd_weksem = mysem;
                excmd[1].cd_bufsize = 160;      /* transmit buffer size */
                excmd[1].cd_nof_buf = 1;        /* two buffers */
                excmd[1].cd_parm[OFFSET] = (int)&exbuf3[0];
                excmd[1].cd_parm[OFFSET+1] = 160;

                msg.hi = (int)&excmd[1];
                writeln("Rx - setup");
                vmail(mb1, &msg, WAIT);
                /*
                 * wait for receive *
                                   */
                p(mysem, WAIT);
        if(excmd[1].cd_stat_out == OK) writeln("Rx - completed");
        else writeln("Rx - error");
        /* wait */ for(i=0;i<65000;i++) ;
             };
}


diagnose(p)
struct cmdb *p;
{ disable(INT586);
  p->cb_complete = p->cb_ok = 0;
  p->cb_command = AXDGN; p->cb_endlst = 1;
  scb.s_cblp=p; scb.s_cmd.rcc.s_cuc=START;
  ca();
  while(!p->cb_complete);
  scb.s_cmd.rcc.s_ack = scb.s_status; /* ack status */ ca();
  return p->cb_ok;
}

configure(p)
struct cmdb *p;
{ disable(INT586);
  p->cb_complete = p->cb_ok = 0;
  p->cb_command = AXCONF; p->cb_endlst = 1;
  p->cb_var[0]=CONFDAT0;        p->cb_var[1]=CONFDAT1;
  p->cb_var[2]=CONFDAT2;        p->cb_var[3]=CONFDAT3;
  p->cb_var[4]=CONFDAT4;        p->cb_var[5]=CONFDAT5;
  scb.s_cblp=p; scb.s_cmd.rcc.s_cuc=START;
  ca();
  while(!p->cb_complete);
  scb.s_cmd.rcc.s_ack = scb.s_status; /* ack status */ ca();
  return p->cb_ok;
}
```

```
/*********** TIMER PROCESS ***************/
/* This process reads an message from mailbox 5. The format of this message
   is: msg->lo : bit 0-13 time out value
               : bit 14 and 15 a proces indification
       msg->hi : a pointer to an element of a rcl list.

    The structure timerq is organised cumulative (based on intervals) */

#include "llcstruc.h"

timer()
{
  struct message *msg;
  struct timerq *p;
  int timer2();          /* handler */
  int i;

  tstop(); rmail(mb5);
  tfirst = (struct timerq *)NILL;          /* initialize waiting queue */
  tfree = p = &twq[0]; i = 1;
  do { p = p->next = &twq[i++]; } while(i<SIZE *2);
  p->next = (struct timerq *)NILL;
  attach(T2INT, timer2, INDIRECT);
  timerstart(); myprioty(10);

  while (TRUE)          /* the operational loop */
  { pmail(mb5, msg, WAIT);
    disable(T2INT);
    tmsg.imsg = msg->lo;
    if (tmsg.smsg.timeout == 0) getq(msg->hi); else putq(msg->hi, tmsg.imsg);
    enable(T2INT);
  }
}

timer2()        /* handler for timer ticks */
{
 struct timerq *p;
 struct message *m;
 p = tfirst;
 m->lo = NULL;  /* timer id */
 while(p != NILL) {
   if(--(p->interval) == 0) { /* signal process */
     switch(p->procid) {

     case SENDID : m->hi = p->hclp; vmail(mb6, m); break;
     case RECID  : m->hi = p->hclp; vmail(mb7, m); break;
     case INTERID: ; /* not used ready */
     case EXTRAID: ; /* dummy element */ }
     deq();
     p = p->next;}
   else break; };
  intret(T2INT);
 }
```

```
deq()
/* dequeue first element from *tfirst queue and place
   it in *tfree queue */
{
  register struct timerq *p;
  if((p = tfirst) == NILL) return;
  tfirst = tfirst->next; p->next = tfree;
  tfree = p;
} /*deq*/

getq(e)
/* move one element of the *tfirst to the tfree queue. for the element to be
   moved, hclp must equal e. */
  int *e;
{
  register struct timerq **p, *q;
  if (*(p = &tfirst) == NILL) return;
  /* now search for matching pointer */
  q = tfirst;
  while (q->hclp != e) { /* search */
    p = &(q->next);
    if ((q = q->next) == NILL) return; };      /* not found */

  /* update wait interval of next element */
  if(q->next == NILL) /* no next element */ *p == NILL;
    else (*p = q->next)->interval += q->interval;
  q->next = tfree; tfree = q;
}
```

```
putq(e, i)
/* transfer an element of the *tfree queue to de *first
   queue, sorted by argument i. *tfirst queue is organized
   cumulative. */
int e;
int i;          /* bit 0..13 is timeout value */
                /* bit 14..15 is procid */
{ struct timerq **p, *q, *r;
  union timermsg s;
  unsigned id;  /* process id */
  int to;       /* timeout value */
  s.imsg = i;
  to = s.smsg.timeout;
  id = s.smsg.procid;
  if (*(p = &tfirst) != NILL) { /* search for the first element with an
                                   interval equal or greater then to */

  q = tfirst;
  while (to > q->interval) {
    to -= q->interval;
    p = &(q->next);
    if ((q =q->next) == NILL) break; }; /* while */
                              }; /* if */
/* now add an element before q (which also may be a NILL pointer
   *p is a pointer pointing to q (or *p == NILL) */

  tfree = (r = tfree)->next;    /* r = new element */
  r->next = q;
  r->interval = to; r->procid = id;
  r->hclp = e;
  *p = r;
  if(q != NILL) q->interval -= to;
}
```

LITERATUURLIJST

[lit1]
Flint D.C. The data ring main New York: John Wiley, 1983

[lit 2]
Gee K.C.E. Introduction to Local Area Computer Networks
London: Macmillan Press, 1983

[lit3]
Local Area Networks State of the art Report
Pergamon Infotech Limited, 1983

[lit4]
The Ethernet Data link layer and physical layer specification
Xerox, Intel and Digital - September 30, 1980

[lit5]
L.Li et. al. Evaluation of token passing scheme's in LAN's
IEEE Computer Networks, 1982

[lit6]
Arthurs E. et al. IEEE Project 802 LAN standards Traffic-
Handling Characteristics Committee Report Working Draft
Silver Spring Md, june 1983

[lit7]
Stuck B.W. Calculating the maximum mean datarate in LAN
Computer, may 1983

[lit8]
Cushman R.H Hands-
on network design project gives insight into LAN features
EDN, march 22, 1984

[lit9]
Hindin H.J. Dual chip sets forge vital link for ethernet local-
network scheme Electronics, October 6, 1982

[lit10]
Sinha A.N. An Ethernet LAN prototype for THE KUNix machine
THE report, august 1984

[lit11]
82586 Reference Manual Intel, 1983

[lit12]
Intel LAN Components User's Manual Intel, 1984

[lit13]
IEEE 802.2 Logical Link Control Draft C - 17 May, 1982

[lit14]
IEEE 802.3 CSMA/CD Draft C - 17 MAy, 1982

[lit15]
ECMA TR/14 Local Area Networks, Layers 1 to 4
Architecture & Protocols ECMA, september 1982

[lit16]
Borger L. A Local EXecutive for THE KUNix Machine
Afstudeer verslag THE, 1 dec 1983

[lit17]
Deliege R. Ontwikkeling van systeem software voor de serie I/O-controller van THE KUNix machine Afstudeerverslag THE, dec 1984