Ontwikkeling van systeemsoftware voor de serie I/0 controller van THE KUNix machine

Deliege, R.J.H.

Award date:
1984

4881

ECB 941

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

Ontwikkeling van systeemsoftware voor de serie I/O controller van THE KUNix machine

door Rene Deliege.

Afstudeerverslag van : R.J.H. Deliege
Periode : december 1983 t/m december 1984
Afstudeerhoogleraar : Prof. Ir. A. Heetman
Coach : Ing. L.A. van Bokhoven

SAMENVATTING

In samenwerking met de Universiteit Nijmegen wordt aan de Technische Hogeschool Eindhoven gewerkt aan de ontwikkeling van een modulair geconstrueerde mini-computer. Een aantal I/O taken van het operating system worden hierin overgebracht naar aparte I/O controllers. Deze I/O controllers zijn zelfstandige micro-computers rond de Intel 80186 processor. Een van deze controllers zal de serie I/O (met terminals en aanverwante apparaten) verzorgen.

Dit verslag beschrijft de ontwikkeling van de software voor deze serie I/O controller. Aan de eisen aan deze software en een aantal implementatie details was reeds eerder gewerkt. Nu zijn vooral de structuur en de onderlinge interacties gerealiseerd. De software wordt namelijk opgebouwd uit een aantal parallelle processen.

Voor de I/O controllers is een eigen multi-tasking executive ontwikkeld. De implementatie hiervan was nog niet geheel afgerond en ze was ook nog niet getest. Aangezien deze executive de basis vormt voor de te ontwikkelen controller software, is een groot deel van de tijd besteed aan het afronden van deze executive.

Als ontwikkelgereedschap was ook een monitor/debugger voor de I/O controllers in ontwikkeling. Ook aan de ontwikkeling van deze monitor is meegewerkt.

Momenteel is een stuk functionerende controller software gereed. Hierin zijn echter nog niet alle gewenste functies gerealiseerd. Om flexibel genoeg te zijn moeten ook nog de nodige uitzonderings-gevallen (met name niet-terminal I/O) ondersteund worden.

Ik wil iedereen, werkende aan het "THE KUNix" project, bedanken voor de prettige samenwerking, met name mijn coach, dhr. van Bokhoven.

INHOUDSOPGAVE

I DE KUNIX MACHINE

In samenwerking met de Katholieke Universiteit Nijmegen (KUN, faculteit wiskunde en natuurwetenschappen, afdeling IVV) is de vakgroep EB van de afdeling Elektrotechniek van de Technische Hogeschool Eindhoven bezig met een project dat tot doel heeft het ontwerpen en bouwen van een modulair geconstrueerde minicomputer. In eerste instantie wordt Unix als operating system geimplementeerd, vandaar de informele naam "THE KUNix machine".

In het systeem zullen belangrijke taken van het operating system overgebracht worden naar I/O controllers om de hoofdprocessor te ontlasten en zo de performance van het systeem op te voeren. Deze I/O controllers zijn zelfstandige, complete (16 bit) microcomputers.

De koppeling tussen de diverse modulen vindt plaats met behulp van een VME bus. Ook wordt gewerkt aan de realisatie van een local area network (volgens ethernet standaard) als onderlinge verbinding.

Als modulen zijn op het moment ontwikkeld :

DE CENTRALE PROCESSOR

Als centrale processor is voor de Motorola 68010 gekozen. De centrale processor module bevat behalve deze processor twee Memory Management Units, 1 Mbyte lokaal geheugen (met error detection en correction), twee serie I/O poorten, een VME bus interface en een interface voor een lokale extensie bus.

GEHEUGEN MODULE

De geheugenmodule bevat maximaal 2 Mbyte geheugen met error detection en correction, een VME bus interface en een interface voor een lokale extensie bus.

DISK I/O CONTROLLER

Dit is een complete microcomputer rond de Intel 80186 processor met maximaal 512 Kbyte lokaal (dual ported) geheugen (met error detection en correction).
Er is voorzien in een interface naar een Winchester disk (SASI) en een floppy disk controller. Verder zijn twee serie I/O aansluitingen aanwezig en een VME bus interface.

SERIE I/O CONTROLLER

Een microcomputer identiek aan de disk I/O controller.
In plaats van de disk interfaces zijn nu echter 8 serie lijnen aanwezig (voor terminals en aanverwante apparaten).

GRAPHICS DISPLAY CONTROLLER

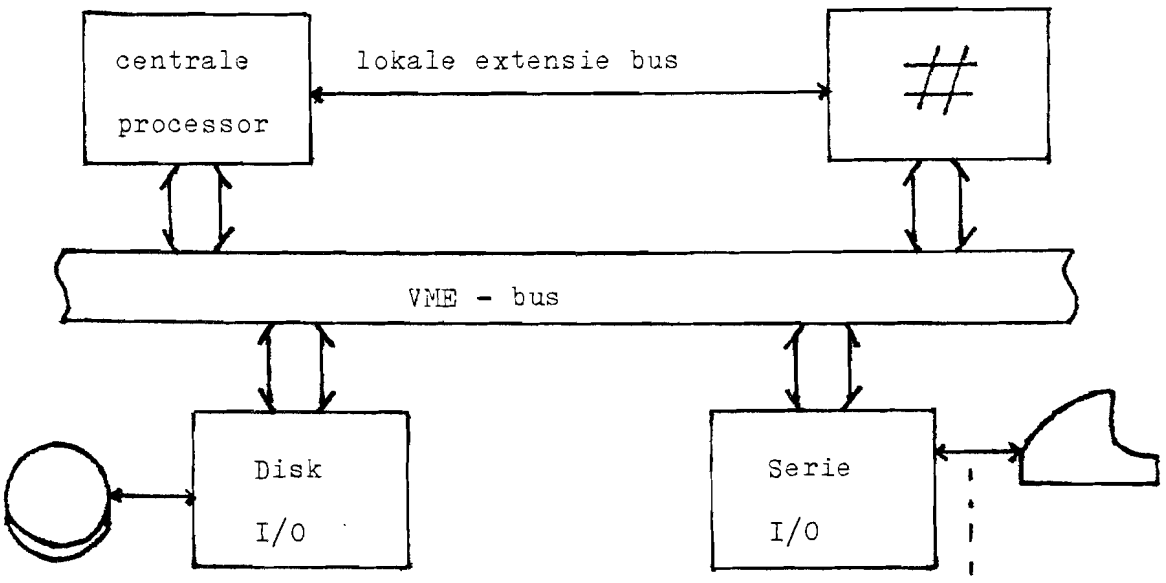
Een module gebaseerd op de NEC 7220 graphics display controller. Het oplossend vermogen is ongeveer 1024 x 1024 punten. Er zijn vier bitplanes aanwezig evenals een colour look up table.

II DE SERIE I/O CONTROLLER

2.1 INLEIDING

2.1.1 algemeen

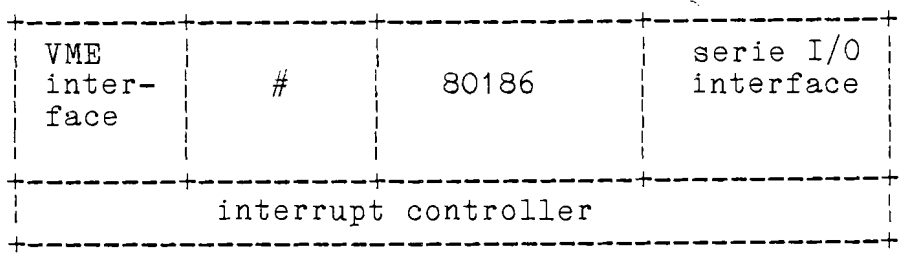
De serie I/O controller fungeert als een interface tussen een aantal terminals en een host. De host is in dit geval Unix op de centrale processor module, maar de toepassing is niet hiertoe beperkt.



Gezien vanuit de host fungeert de serie I/O controller als terminal server, dat wil zeggen ze accepteert opdrachten van de host, voert deze uit en zal een resultaat teruggeven. Opdrachten zijn bijvoorbeeld lees/schrijf regel of lees/schrijf karakter.
Doordat hier sprake is van een intelligente I/O controller kunnen lokaal een aantal functies worden uitgevoerd.
De toepassing is overigens niet alleen tot terminals beperkt, hoewel tot nu toe de aandacht vooral hierop gericht is geweest. Andere serie I/O apparaten zijn bijvoorbeeld printers, ponsband lezers/ponsers en snelle interfaces voor communicatie tussen computers.

2.1.2 hardware

De hardware configuratie van de serie I/O controller is schematisch als volgt :

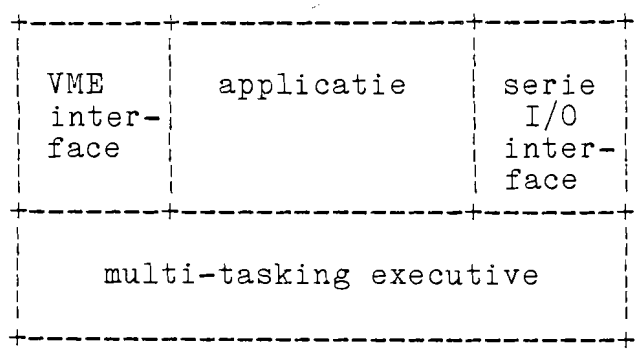


Voor een uitgebreide beschrijving van de hardware wordt verwezen naar [3].

De exacte uitvoering van de serie I/O controller is nog in ontwikkeling. Op het moment wordt voor het testen de disk I/O controller gebruikt, deze biedt twee serie aansluitingen via een 8274 Multi Protocol Serie I/O Controller chip.

2.1.3 software

De software voor de serie I/O controller is als volgt in te delen :



Het blok "applicatie" bevat de eigenlijke serie I/O controller software en is hardware onafhankelijk, de overige blokken zijn hardware afhankelijk.
De software voor de VME interface kan van de disk I/O controller software worden overgenomen. (zie [3])

De multi-tasking executive (LEX : Local EXecutive) is door een vorige afstudeerder ontwikkeld (zie [2]).
Deze LEX creert een multi-tasking omgeving voor de applicaties. De applicatie is afhankelijk van de funktie van de controller, de

LEX is gelijk voor alle controllers.
De LEX biedt mogelijkheden voor proces synchronisatie door middel van semaphoren en proces communicatie door middel van mailboxes. De LEX controleert de hardware van de controller, de verwerking van interrupts zal dan ook via de LEX plaatsvinden. Interrupts kunnen hiertoe aan een routine (interrupthandler) in de applicatie gekoppeld worden. Bij het optreden van een interrupt zal de LEX dan de betreffende routine activeren.

2.1.4 ontwikkel faciliteiten

Als ontwikkel-gereedschap is voor de controllers een monitor/debugger programma ontwikkeld. Deze monitor is in PROM in het systeem aanwezig en zal na een reset van de controller actief worden. Met behulp van deze monitor kan dan de te testen software in RAM geladen en getest worden.

Voor de ontwikkeling van de software is voornamelijk gebruik gemaakt van faciliteiten van de Universiteit Nijmegen. Het daar gebruikte systeem is een PDP 11/70 onder Unix. Op dit systeem is een komplete ontwikkel-omgeving voor zowel de 68000 als de 8086/80186 beschikbaar. Aanwezig zijn o.a. een C compiler, assembler, linker/loader en een disassembler.
De communicatie vond plaats via een modem (eerst 300, later 1200 baud) over een telefoonlijn. Aan de modem kan zowel een terminal als een Altos microcomputer aangesloten worden. In het laatste geval kan de Altos via een programma "transparant" gemaakt worden tussen console en seriepoort (verbonden met de modem).
De toepassing van de Altos heeft als voordeel dat met behulp van enige utilities files getransporteerd kunnen worden tussen de Altos en de PDP in Nijmegen. De te testen programma's kunnen dan lokaal op floppy disk opgeslagen worden. Bij het testen van het programma in de serie I/O controller is dan geen direkte verbinding met Nijmegen meer nodig. Het laden van de software in de controller kan dan bovendien op 9600 baud plaatsvinden.
De codefiles worden in Intel hex formaat overgezonden en opgeslagen.

Voor testwerk is ook een C compiler (BDS) onder CP/M beschikbaar, zodat ook op een Altos microcomputer het een en ander gedaan kan worden. Aangezien dit echter een Z80 machine is, is dit op assemblerniveau niet compatibel.

In het rekencentrum van de TH zijn enige IBM personal computers voor algemeen gebruik beschikbaar. Op deze machines kunnen we ook in 8086 assembler werken.

2.2 FUNCTIE

Vanwege de Unix implementatie zullen in ieder geval die functies geboden moeten worden die standaard in Unix aanwezig zijn. Deze functies vinden we in de meeste terminal drivers terug.
Dit zijn onder andere :
- expansie van tab karakters
- zichtbaar maken van controle karakters
- lower/upper case conversie
- eenvoudige protocols (xon/xoff, etx/ack)
- paginering uitvoer
- simpele line editing

Verdere wensen zijn :

- Uitgebreide line editing. Dit zijn bijvoorbeeld insert/delete acties midden in een regel. Dit is zeer gewenst.

- Ondersteuning van screen editing. Omdat screen editors een zware belasting voor het systeem vormen is een ondersteuning hiervan zinvol. De realisatie van een algemene ondersteuning is echter zeer complex en daarom momenteel achterwege gelaten. Ondersteuning voor een specifieke screen editor (bijvoorbeeld vi) is voorlopig ook nog niet gerealiseerd.

- Virtual Terminal Interface. De bedoeling hiervan is dat de serie I/O controller karakter conversies kan plegen. Terminalafhankelijke codes kunnen dan omgezet worden in VTI codes en omgekeerd.
  Omdat de serie I/O controller software (de uitgebreide line editing) zelf ook terminalafhankelijk is, is deze virtual terminal interface essentieel.

- Flexibele koppeling tussen functies en karakters. Het is zeer wenselijk dat de functie van een toets flexibel aan een bepaalde actie gekoppeld kan worden. Deze koppeling moet per terminal instelbaar zijn en door de gebruiker te wijzigen.
  Acties zijn bijvoorbeeld :
  - einde regel
  - wis laatste karakter
  - cursor besturing

De uitgebreide line editing is reeds (grotendeels) door een vorige afstudeerder (zie [1]) gerealiseerd. In de beginfase van mijn afstudeerwerk heb ik deze software bestudeerd en nog enigszins uitgebreid.
Door deze afstudeerder is ook een systeem van vertaaltabellen ontwikkeld. Zowel aan input- als aan outputzijde kunnen karakters worden vertaald. De vertalingen zijn niet beperkt tot alleen karakters, maar ook karakterreeksen zijn mogelijk. Dit is bijvoorbeeld nodig voor escape sequences die een actie tot gevolg hebben. Via deze tabellen kan ook worden bepaald welk karakter of reeks een bepaalde edit functie tot gevolg heeft.

Voor een uitgebreide beschrijving van deze tabellen en de implementatie wordt verwezen naar [1].

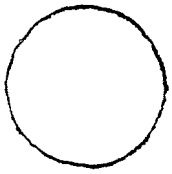
2.3 SOFTWARE

2.3.1 structuur

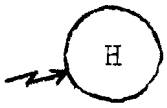
Zoals eerder werd vermeld zijn enige delen van de software reeds door een vorige afstudeerder geschreven. De totale structuur van de software waarbinnen dit zou moeten passen is ook enigszins aangegeven. Aangezien toendertijd de LEX en de hardware nog volop in ontwikkeling waren is de algemene software structuur toen niet verder uitgewerkt. Een groot deel van mijn afstudeerwerk bestond dan ook uit het definieren van een geschikte, flexibele software structuur. De toe te passen algoritmes werden hierbij min of meer vanzelf meeontwikkeld.

Eerst wordt een overzicht van de totale structuur gegeven, daarna volgt een beschrijving per proces.
De notatie is in C statements (vanwege de implementatie in C) en waar dit niet eenvoudig mogelijk was is een omschrijving gegeven. Er is ook gebruik gemaakt van de system calls zoals de LEX biedt.

Gebruikte symbolen :



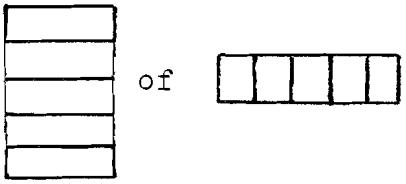
proces



handler
(routine geactiveerd door een interrupt)

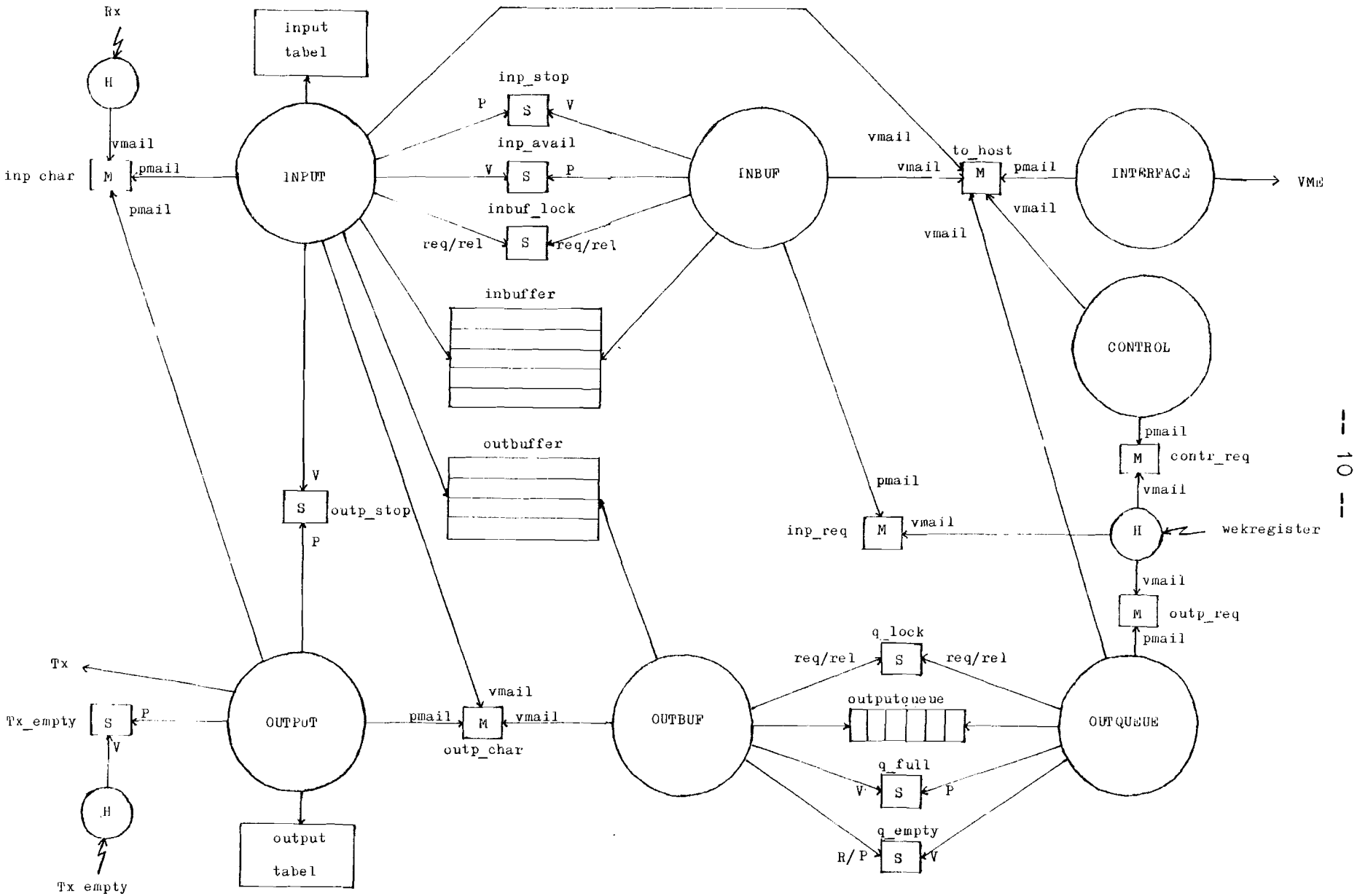
semaphoor
(synchronisatie tussen processen)

mailbox
(communicatie tussen processen)



datablok

De pijlen geven de toegang tot de betreffende objecten (semaphoren, mailboxes, datastructuren) door proces of handler weer (en geen dataflow richting !).


Rx
H
input tabel
inp_stop
P
S
V
vmail
inp char
M
pmail
INPUT
inp_avail
V
S
P
INBUF
vmail
to_host
M
pmail
INTERFACE
VME
pmail
vmail
inbuf_lock
req/rel
S
req/rel
vmail
inbuffer
CONTROL
outbuffer
pmail
M
contr_req
V
vmail
S
outp_stop
pmail
inp_req
M
vmail
H
wekregister
P
vmail
M
outp_req
pmail
q_lock
req/rel
S
req/rel
Tx
vmail
outputqueue
Tx_empty
S
P
OUTPUT
pmail
M
vmail
OUTBUF
OUTQUEUE
outp_char
V
q_full
V
S
P
H
output tabel
q_empty
R/P
S
V
Tx empty

INPUT

Input verwerkt de input karakters en plaatst ze in een buffer op regelbasis. Wat met het karakter gedaan wordt wordt via een tabel bepaald.
Eenvoudige protocols worden hier ook afgehandeld (in samenwerking met output).
Na een einde regel karakter wordt dit via een semaphoor aan het volgende proces (inbuf) gemeld.
Afhankelijk van de terminal mode worden de input karakters ook ge-echoed.

```
/* lees input character uit mailbox. dit wordt hierin geplaatst
 * door de interrupt handler, gekoppeld aan de input interrupt
 */
pmail(inp_char, &c, WAIT);

/* doe actie afhankelijk van karakter, zoals in vertaaltabel
 * gespecificeerd. hier is een switch i.p.v. de tabel gebruikt
 */
switch (c)
{ default :     /* normaal input karakter */
                if (echo) vmail(outp_char, &c, WAIT);
                plaats karakter in current line buffer

        /* CR is hier als einde regel karakter genomen */

  case CR :     if (echo) vmail(outp_char, &c, WAIT);
                request(inbuf_lock); /* lock input buffer */
                v(inp_avail);
                increment current line buffer
                als inputbuffer vol is en tandemmode :
                { vmail(outp_char, XOFF, WAIT);
                  set tandemstop
                  release(inbuf_lock); /* unlock input buffer */
                  p(inp_stop, WAIT);
                  reset tandemstop
                  vmail(outp_char, XON, WAIT);
                }
                else
                  release(inbuf_lock);

        /* xon/xoff protocol voor output */

  case XOFF :   if (xonxoffmode)  set xonxoffstop

  case XON :    if (xonxoffstop)
                { reset xonxoffstop
                  v(outp_stop);
                }

        /* etx/ack protocol voor output */

  case ACK :    if (etxstop)  v(outp_stop);

        /* ^C is hier als interrupt karakter genomen */

  case ^C :     maak interrupt message klaar
                vmail(to_host, int. msg, WAIT)
                /* in tegenstelling tot het normale verkeer wordt
                 * hier een message naar de host verzonden zonder
                 * dat daar door de host om gevraagd is.
                 */

        /* en dan zijn er natuurlijk nog alle
           uitgebreide line editing functies  */
}
```

INBUF

De functie van inbuf is de verwerking van de lees opdrachten van de host. Inqueue ontvangt deze opdrachten via een mailbox. Na een leesopdracht ontvangen te hebben zal inbuf wachten op een inputregel in de input buffer via een semaphoor, tenzij de vorige regel nog niet geheel gelezen was. Als er een regel is zal inbuf een resultaat bericht klaarmaken en deze via een mailbox aan het proces interface doorgeven die de verzending naar de host verzorgt.
Als de buffer vol was en het tandem (xon/xoff voor input) protocol actief zal het aantal verzonden regels bijgehouden worden. Als de input buffer half leeg is zal het wachtende proces input gewekt worden. De keuze om het input proces te wekken als de buffer half leeg is is vrij willekeurig en mischien niet altijd optimaal.

```
/* lees read opdracht van host */

pmail(inp_req, opdr, WAIT);

/* gebruikte variabelen :
 * lengte : lengte current line buffer
 *       (current line is niet dezelfde als in proces INPUT !!)
 * teller : hulpvariabele
 * constanten :
 * MAX : grootte inputbuffer (in regels)
 */
if (lengte == 0)  /* dan moeten we een nieuwe inputregel hebben */
{ p(inp_avail,WAIT);
  request(inbuf_lock);        /* lock input buffer */
  increment current line buffer
  lengte = lengte current line
  if (tandemstop)
  { if (++teller > MAX/2)  /* buffer is weer half leeg */
    { teller = 0;
      v(inp_stop);
    }
  }
  release(inbuf_lock);
}

prepareer result

update lengte

/* stuur result naar host */

vmail(to_host, result, WAIT);
```

OUTQUEUE

De functie van outqueue is de verwerking van schrijf opdrachten van de host. Outqueue ontvangt deze schrijfopdrachten via een mailbox. Na ontvangst van de opdracht zal de data in de output queue geplaatst worden als daar genoeg plaats is. Als er niet genoeg plaats is wordt een teller geinitialiseerd met het aantal benodigde plaatsen. Het proces (outbuf) dat de karakters uit de queue leest zal deze teller dan aflagen. Bij het nul worden van de teller wordt outqueue dan weer gewekt.
Dit betekent dat een datablok dat groter is dan de queue niet verwerkt kan worden. Het algoritme moet dan ook mischien aangepast worden om ook de data in gedeelten in de queue te kunnen plaatsen.

```
/* lees write opdracht van host */

pmail(outp_req, opdr, WAIT);

/* gebruikte variabelen :
 * lengte : lengte data in write opdracht
 * ruimte : aantal lege plaatsen in output queue
 * teller : hulpvariabele
 */
request(q_lock);          /* lock output queue */

if (lengte > ruimte)
{ teller = lengte - ruimte;
  release(q_lock);
  p(q_full, WAIT);  /* wacht tot er genoeg plaats is */
  request(q_lock);
}


plaats de data in de output queue

v(q_empty);           /* er staat nu weer iets in de queue */

release(q_lock);

prepareer een result

/* stuur result naar host */

vmail(to_host, result, WAIT);
```

OUTBUF

Outbuf leest karakters uit de output queue en plaatst ze in een mailbox ten behoeve van het output proces. Deze mailbox is nodig om de output karakters en de input echo samen te kunnen voegen. De karakters worden ook in een buffer op regelbasis geplaatst om ze eventueel als input te kunnen gebruiken (met behulp van de uitgebreide line editing).
Als de queue leeg is zal het proces wachten via een semaphoor. Het proces outqueue zal elke keer als het iets in de queue plaatst dit via deze semaphoor doorgeven.

```
/* test of de outputqueue leeg is
 * (mbv globale variabelen voor de queue administratie)
 */
request(q_lock);            /* lock output queue */
if (queue == empty)
{ r(q_empty,0);
  release(q_lock);
  p(q_empty, WAIT);    /* wacht tot er iets in queue geplaatst wordt */
  request(q_lock);
}

lees een karakter uit de queue

/* teller is dezelfde als in proces OUTQUEUE */

if (teller > 0)
  if(--teller == 0)  v(q_full);

release(q_lock);

/* geef karakter aan proces OUTPUT
 * en plaats het in een buffer tbv editing proces (INPUT)
 */
vmail(outp_char, karakter, WAIT);

plaats karakter in outp_buf

if (karakter == CR) increment current line buffer in outp_buf
```

OUTPUT

Output verwerkt de ouput karakters en zendt ze weg via de serie I/O interface. Wat met het karakter gedaan wordt wordt via een tabel bepaald. Eenvoudige protocols worden hier ook (in samenwerking met input) uitgevoerd. Ook worden hier de cursorpositie en dergelijke bijgehouden.

```
/* lees karakter */

pmail(outp_char, karakter, WAIT);

if (xonxoffstop)  p(outp_stop, WAIT);

doe diverse functies (tabs, contr. chars, etc.)

update cursor positie ed.

/* in paging mode moet bij een paginagrens op
 * een input karakter gewacht worden.
 * als de prioriteit van dit proces (tijdelijk)
 * hoger is dan die van proces INPUT kan dit
 * door een pmail op de input mailbox.
 */
if (paging && pagina full)  pmail(inp_char, kar, WAIT);

/* bij een etx/ack protocol wordt een output blok
 * afgesloten met een etx. hierna moet op een ack
 * van inputzijde gewacht worden.
 */
if (etxack && block full)
{ output ETX
  set etxstop
  p(outp_stop, WAIT);
}
```

INTERFACE

Lees een message uit de mailbox to_host en verstuur deze via de VME bus.
De ontvangst van berichten van de host geeft via het wekregister een interrupt. De aan deze interrupt gekoppelde handler zal de opdracht uit het opdrachtbuffer naar een vrij geheugensegment kopieren. Dit segment kan verkregen worden met een getseg call. De handler zal ook kijken wat voor type opdracht dit is (input, output of control). Het adres van het segment met de gekopieerde opdracht wordt dan via een mailbox aan het juiste proces doorgegeven.

CONTROL

De functie van control is de verwerking van controle opdrachten van de host. Control ontvangt deze opdrachten via een mailbox. Controle opdrachten zijn opdrachten die geen input of outputopdrachten zijn, bijvoorbeeld het veranderen van vertaaltabellen, het openen of sluiten van een verbinding.

2.4 VME VERKEER

De diverse modulen in "THE KUNix" machine zullen over de VME bus opdrachten en data moeten uitwisselen. Hiertoe is een mechanisme ontwikkeld dat gebruik maakt van een communicatiebuffer, een wekregister en een semaphoorregister. Dit mechanisme is beschreven in o.a. [3].
Voor dit verkeer is een standaard berichten formaat vastgelegd. Dit formaat bevat een aantal applicatie onafhankelijke en een aantal applicatie afhankelijke zaken.

De definitie van dit berichtenformaat is als volgt :

PACKET
Het totale bericht is een packet dat bestaat uit een of meerdere boodschappen.

fieldname	size	description
size	1 word	total size of packet in bytes
dest_mid	1 word	destination module identifier
trans_cntrl	1 word	transport control word
next_msg	1 word	ptr to next message (offset in buffer, 0 = no message)
msg_1	m words	first actual message
next_msg		
msg_2		
etc.		

MESSAGE
Een message is de logische boodschap die aan een bepaald proces wordt aangeboden.

fieldname	size	description
dest_sid	1 word	destination socket identifier
src_mid	1 word	source module identifier
src_sid	1 word	source socket identifier
cntrlsz	1 word	size of control information
datasz	1 word	size of data information
msg_id	2 words	message identification (high, low)
msg_type	1 word	message type (request,response,interrupt...)
resp_type	1 word	response type (wek, interrupt...)
cntrl_info	c words	control information
data_info	d words	data information

CONTROL_INFO
Het controle informatie veld bevat de applicatie afhankelijke informatie. Voor de serie I/O controller onderscheiden we twee gevallen : een opdracht aan een terminalserver en een terugmelding.

Opdracht :

fieldname	size	description
command	1 word	command
parameter	1 word	command parameter (optional)
mem_addr	3 words	memory address (optional) most significant bit of first word : 1 : local address (high_adr,med_adr,low_adr) 0 : vme address (am, high_adr, low_adr)

Terugmelding :

fieldname	size	description
status	1 word	response status

DATA_INFORMATION
Het data veld van de boodschap kan gebruikt worden om bij een opdracht of terugmelding meteen de data mee te geven. Een andere methode is dat in het controle informatie veld het adres van de data wordt meegegeven, waarna de aangesproken module de data zelf kan ophalen.

De definitie van mogelijke commando's is nog niet helemaal uitgewerkt. Een voorstel is :

Opdracht :

command	parameter	mem adr	data info
data_read	count	-	-
mem_read	count	*	-
data_write	count	-	*
mem_write	count	*	-
table_read	tabel identif.	*	-
table_write	tabel identif.	*	-
mode_read	-	-	-
mode_write	-	-	*
open	-	-	-
close	-	-	-
interrupt	interrupt id.	-	-

De laatste boodschap is in tegenstelling tot de vorigen een opdracht van de terminalserver aan de host !

Terugmelding (behorende bij een ontvangen en uitgevoerd commando) :

uitgevoerd commando	status	data info
data_read	count	*
mem_read	count	-
data_write	count	-
mem_write	count	-
table_read	count	-
table_write	error code	-
mode_read	error code	*
mode_write	error code	-
open	error code	-
close	error code	-

Opmerking :
Voor het transport op karakterbasis is het mischien zinvol een ander mechanisme te bedenken omdat zo de overhead per karakter wel erg groot is. Een mogelijke oplossing is een commando set_raw en reset_raw in te voeren. Na een commando set_raw zou dan met een verkort bericht of met direkte data overdracht gewerkt moeten kunnen worden.

III DE MONITOR

3.1 INLEIDING

Het monitor programma heeft twee functies :

1. Het initialiseren van de processor en de peripheral chips na een reset.

2. Het bieden van een aantal debugging faciliteiten.

De monitor is hiertoe altijd in PROM in het systeem aanwezig. Na een systeem of processor reset zal de monitor geactiveerd worden. Deze zal de processor en de peripheral chips, voorzover ze door de monitor gebruikt worden, initialiseren. Hierna wacht de monitor op commando's van de console. Voor een beschrijving van deze commando's wordt verwezen naar de gebruikers handleiding (Appendix A).

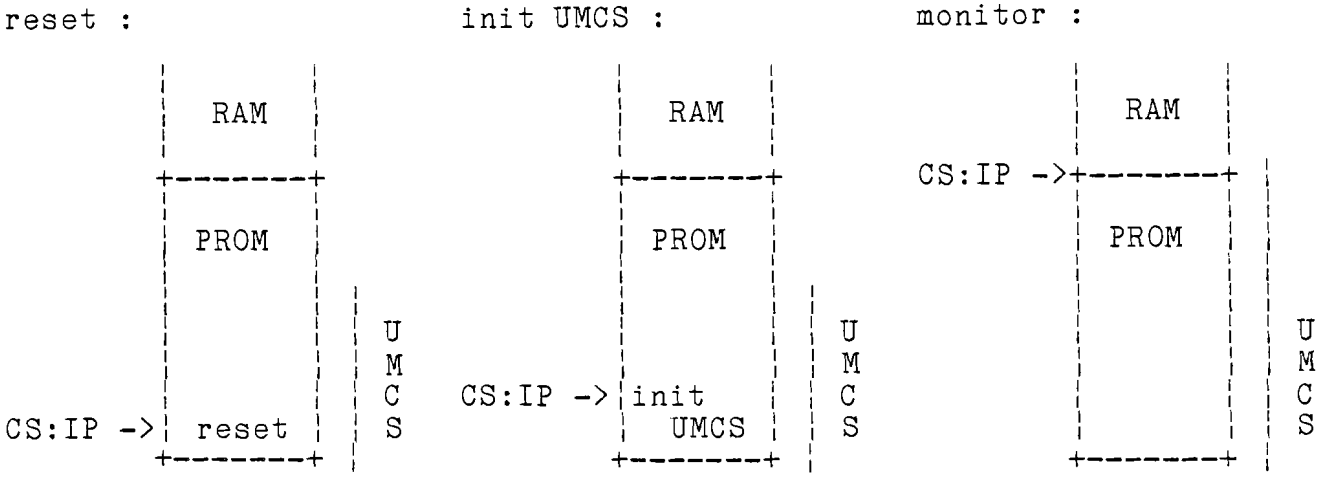
3.2 INITIALISATIES

Het initialisatie gebeuren is als volgt onder te verdelen :

```
monitorinit:
{       init_UMCS;
        goto text;
        text:   init_hardware;
                init_monitor_environment;
                init_application_environment;
                start_monitor_or_application;
}
```

3.2.1 UMCS

Na een reset begint de 80186 processor code te executeren op adres ffff0. Omdat dit adres vlak onder het hoogste adres (fffff) ligt, is er daar alleen maar plaats voor een jump naar het echte startadres. Omdat UMCS (Upper Memory Chip Select) dan nog niet geprogrammeerd is overeenkomstig de PROM grootte kan dan nog niet naar het begin van het PROM gebied gesprongen worden. Eerst zal UMCS geprogrammeerd moeten worden, dan volgt nogmaals een jump naar het begin van het PROM gebied. Dit ziet er als volgt uit :



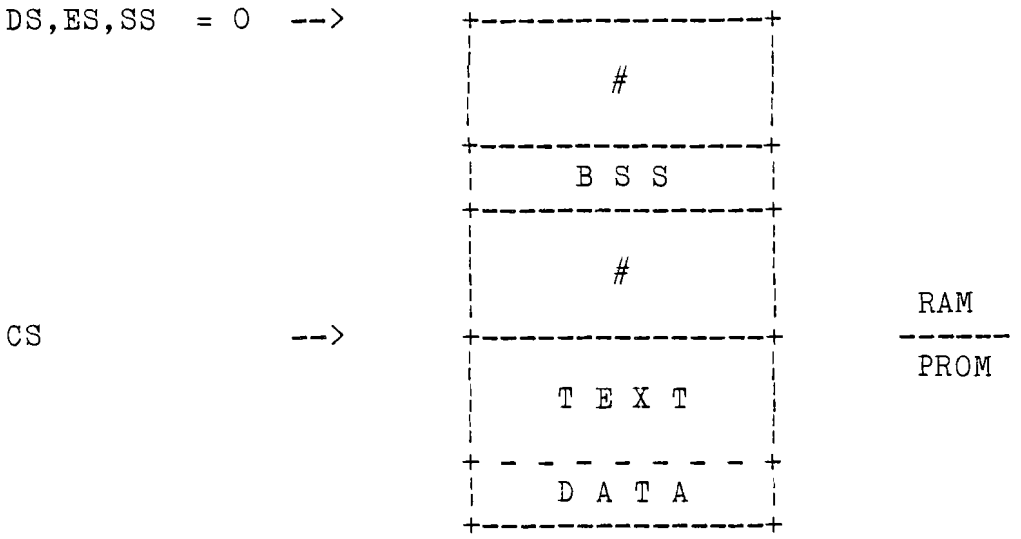
3.2.2 hardware

Wat de hardware betreft zullen eerst de overige chip select lijnen (MMCS, MPCS, PACS) geprogrammeerd moeten worden.
Dan zijn er de interne (in de 80186) peripherals. Hiervan worden door de monitor de interrupt controller en twee timers geinitialiseerd. Van de interrupt controller worden twee ingangen (INTO & INT2) als int/inta paar geprogrammeerd in verband met de externe 8259 interrupt controller.
Timers 0 en 1 worden gebruikt als baudrate generator voor de 8274 serie I/O controller. Beide maxcount registers worden gebruikt om een 50% duty cycle te krijgen.
De default baudrate is 9600 baud voor beide kanalen.
De timers moeten dus een clockfrequentie produceren van 16 * 9600 Hz.
De master clock frequentie is de kristal frequentie/2 = 4915200 Hz.
De timer input clock frequentie is de master clock frequentie/4 = 1228800 Hz.
De count registers moeten dus geladen worden met : 1/2 * timer_clock / (16*baudrate)

De externe te initialiseren peripherals zijn de 8259 Programmable Interrupt Controller en de 8274 Multi Protocol Serie I/O Controller. De laatste wordt door de monitor in "polled mode" gebruikt.

3.2.3 monitor environment

Omdat de monitor ook een aantal variabelen voor zijn administratie gebruikt zal hij ook de beschikking moeten hebben over een stuk RAM geheugen. (Geinitialiseerde) data zal zich natuurlijk in PROM moeten bevinden. Dit geeft de volgende geheugen lay-out :

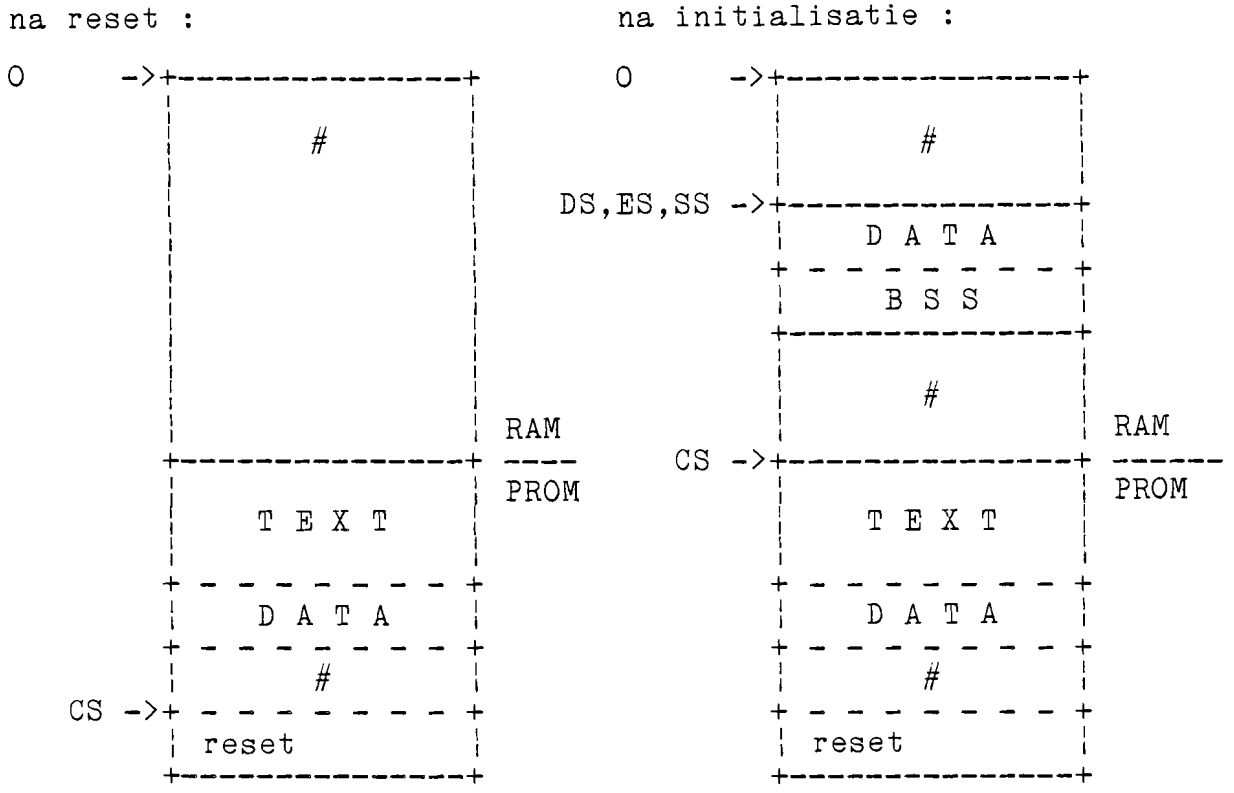


Opmerking :
TEXT is de programmacode
DATA is de geinitialiseerde data
BSS is niet geinitialiseerde data

overeenkomstig de Unix notatie.

In de praktijk was er geen apart data gebied maar was de data in de code opgenomen.
Omdat het data gebied (data en bss) nu verdeeld is over twee segmenten kan niet alle data zonder meer geadresseerd worden. Omdat de de monitor geheel in assembler geschreven was kon dit opgelost worden door voor referenties naar het (PROM) data gebied een CS prefix instructie te gebruiken.

Naderhand is een stuk C code toegevoegd (de disassembler), dit vereist wel data, bss en stack in een segment. Bss en stack moeten zich natuurlijk in RAM bevinden, data natuurlijk in PROM. De oplossing voor deze tegenstrijdige eisen is de data in PROM te plaatsen en bij de start van de monitor naar RAM te copieren. Het volgende probleem is dan dat tijdens de executie code en data beide op adres 0 starten (in CS resp. DS segment) maar bij het maken van de PROM willen we de data achter de code hebben. Om dit op te lossen is een utility prm86 ontwikkeld die een PROM image aanmaakt aan de hand van twee inputfiles. Een van deze files bevat de applicatie (hier de monitor), de ander de eerder beschreven jump voor de reset en de UMCS initialisatie. De header van de gemaakte object file wordt zo aangepast dat het startadres van de data van de applicatie (was 0) gelijk wordt aan einde text. De data wordt dan aangevuld tot het einde van de PROM gebied met net voor dit einde de beschreven UMCS initialisatie en jump's. Een applicatie kan dus gelinkt worden zodat code en data beide op 0 starten (ld86 -i ; de -i vlag geeft aan dat we aparte text en dataruimte wensen).
De geheugen lay-out is dan als volgt :



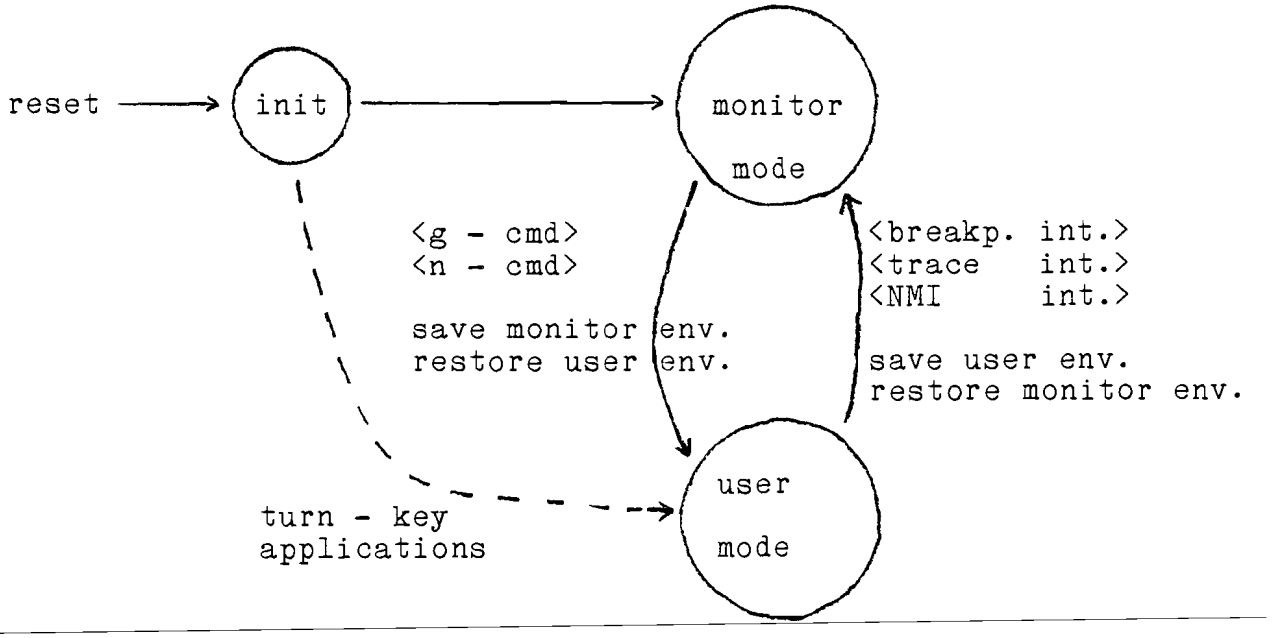
3.2.4 application environment

De monitor biedt de mogelijkheid onder zijn supervisie een stuk (applicatie) code te executeren. Deze executie wordt gestart met een "go" commando. Met een interrupt (bijvoorbeeld NMI) kan deze executie onderbroken worden en teruggekeerd worden naar de monitor. Bij een go commando kan ook een (en maximaal een) breakpoint geplaatst worden. Op het opgegeven adres wordt dan een breakpoint instructie (INT 3) geplaatst. De oorspronkelijke inhoud van deze plaats wordt gered en bij terugkeer naar de monitor (hoe dan ook) hersteld.
Een applicatie kan ook instructie voor instructie uitgevoerd worden met behulp van het "next" commando. Er kan dan een count en een eindadres opgegeven worden (beide optioneel). Er worden dan instructies uitgevoerd totdat er count instructies uitgevoerd zijn of totdat eindadres bereikt is.
Op monitor niveau kunnen alle registers voor de applicatie bekeken en gewijzigd worden. Het zal duidelijk zijn dat dit niet de actuele processor registers zijn omdat op dat moment de monitor actief is. Dit betekent dat de registerinhouden voor het gebruikers programma ergens in het RAM geheugen staan. Voor het starten van dit gebruikers programma worden de processor registers met deze waarden geladen. Hierna mag niet een monitor instructie meer uitgevoerd worden !! Het omgekeerde geldt bij het terugkeren naar de monitor.

Tijdens de initialisatie worden de applicatie segment registers geladen met het adres van het eerste vrije geheugensegment na het monitor data gebied. De overige registers zijn initieel nul.

3.2.5 start monitor or application

Tijdens de ontwikkeling van een applicatie zal deze telkens met behulp van de monitor in RAM geladen worden (via een seriepoort) en vervolgens geexecuteerd worden. Bij het testen van een wat duurzamere versie is het mogelijk de applicatie samen met de monitor in PROM te plaatsen (mits het geheel niet groter is dan het PROM gebied).
Door nu de applicatie registers (CS, DS, ES, SS, IP) te presetten op de start van de applicatie en in plaats van op een console commando te wachten meteen met een "go" te beginnen zal na een reset meteen de applicatie gestart worden. Door een (breakpoint of NMI) interrupt kan naar de monitor gegaan worden voor debugging faciliteiten.
Het samenspel tussen monitor en applicatie kan als volgt weergegeven worden :



IV DE LOCAL EXECUTIVE

4.1 BEGINSTATUS

De lexfiles zoals afgeleverd door Lex Borger bevatten alle system calls en de daarvoor benodigde declaraties.
Om tot een werkende LEX te komen moeten hier nog twee zaken aan toegevoegd worden :

1. een initialisatiegedeelte of proces om de hardware, de systeemobjecten en de proces-stacks te initialiseren.

2. een combine utility om van de LEX en proces object files tot een memory image te komen. Referenties tussen de LEX en processen moeten door deze utility opgelost worden.

Bijna onontbeerlijk is ook een debuggerfaciliteit.

Voorstellen voor een combine utility en een debugger zijn in het verslag van Lex Borger aangegeven, maar nauwelijks uitgewerkt.

Tijdens het teststadium is eerst geprobeerd om zo eenvoudig mogelijk tot een werkende (testbare) LEX te komen, zonder tegelijk allerlei nieuwe utilities mee te moeten testen.
Dit is als volgt gerealiseerd :

- er is geen aparte debuggerfaciliteit voor de LEX ontwikkeld. Voor de 186 kaarten is reeds een eenvoudige monitor/debugger ontwikkeld (zie hfdst. 3) met als faciliteiten o.a. :
  geheugen laden & lezen
  software downloaden over seriepoort
  uitvoeren programma (met breakpoint)
  single steppen programma
  samen met een disassembly listing en een symbol-table is dit voldoende om te kunnen testen.

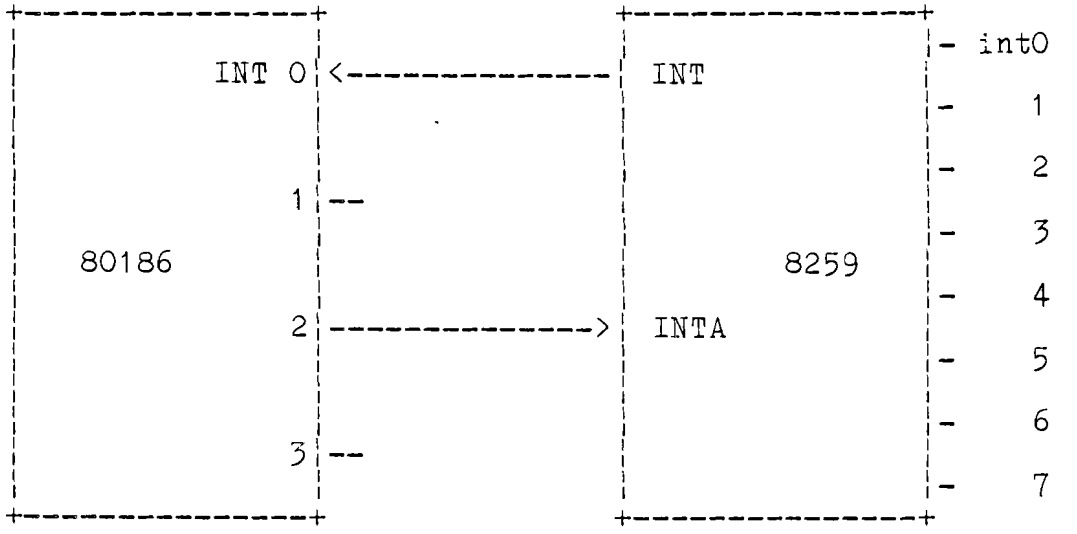
- de combine utility is tijdelijk achterwege gelaten.
  door de (test)processen als functions aan de main (LEX) toe te voegen ontstaat een geheel dat in een keer gelinkt kan worden. Dit vermijdt het probleem van de referenties tussen de LEX en processen. Ld86 kan nu namelijk alle referenties oplossen.

4.2 INITIALISATIES

Er zijn drie categorien van zaken die geinitialiseerd moeten zijn voordat de LEX kan werken :
1. de hardware
2. de systeemtabellen in de LEX
3. de process stacks

Hardware

Het is duidelijk dat een correct geinitialiseerde hardware een voorwaarde is voor een correct functionerende LEX.
De monitor initialiseert op het moment echter praktisch alle essentiele hardware zaken, zodat dit niet nog eens in de LEX hoeft te gebeuren. Gebruikte peripheral chips vallen onder verantwoordelijkheid van de applicatie die ze gebruikt en moeten dus door die applicatie geinitialiseerd worden. De enige uitzondering is de 8259 interrupt controller omdat de LEX de interrupt afhandeling controleert. Deze interrupt controller is als volgt met de processor verbonden :



De INT0 aansluiting van de 80186 moet dus als INT/INTA ingang geprogrammeerd en ge-demaskeerd worden. (De INT/INTA programmering wordt door de monitor gedaan). De 8259 moet zo geprogrammeerd worden dat deze interrupt vectoren 128-135 aflevert (dit verwacht de LEX). Alle ingangen moeten gemaskeerd zijn.

Systeem tabellen

In het LEX data segment staan een aantal tabellen voor de diverse systeemobjecten. Al deze objecten zullen een juiste beginwaarde moeten krijgen.

Per object volgt nu wat de initiele waarde van elk onderdeel van het object is.

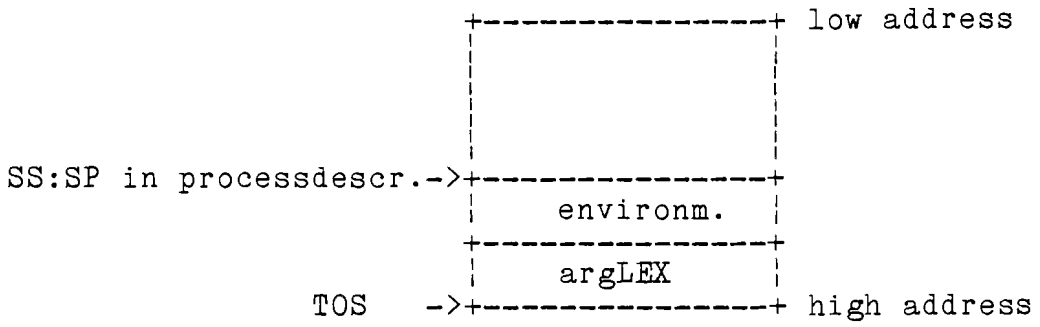
```
semaphoren
      waarde          0
      queuepointer    NILL

mailboxes
      aantal messages         0
      queuepointer            NILL
      freepointer             eerste box
      readpointer             eerste box

handlers
      detached & disabled

segmenttable
      eerste segment start op eerste vrije geheugenplaats
                      grootte tot einde RAM
      andere segmenten grootte 0

ry_list
      pointer naar eerste proces

running
      pointer naar eerste proces

processdescriptors
      prioriteit      64
      status          READY
      queuepointer    ry_list
      stackpointer    process stack
      susp. semaphore waarde          0
                      queuepointer    NILL
```

Process stacks

De process stacks moeten twee parameters bevatten : argLEX en env. De eerste is de layout structure zoals die in het LEX data gedeelte voorkomt, de tweede is de process environment (alle registers, incl. segment registers, IP en flags, excl. SS en SP). Initieel zijn alleen van belang :

```
CS              : begin code segment
IP              : offset in code segment
DS,ES           : begin data segment
FLAGS           : 0x200         (interrupts enabled)
```

Voor een beschrijving van de argLEX parameter zie de LEX handleiding (appendix B).
De geinitialiseerde process stack ziet er als volgt uit :



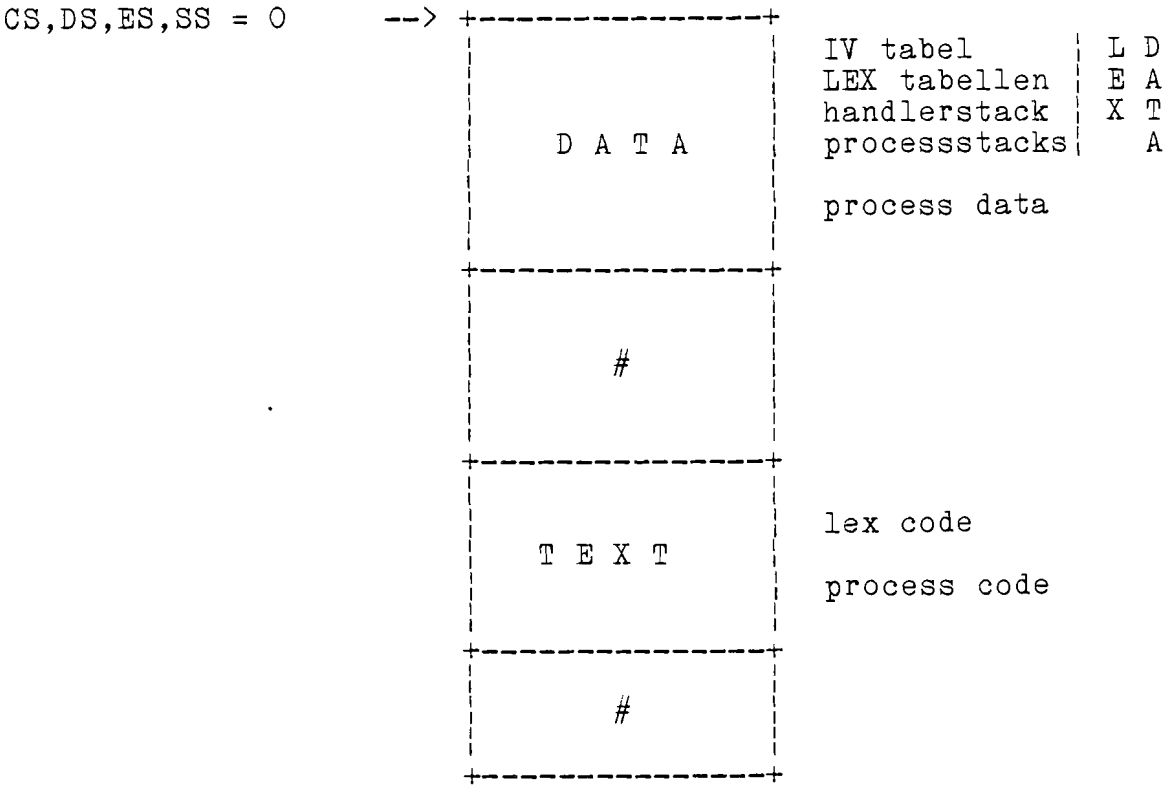
In de definitieve oplossing (zie volgende paragraaf) worden de process stacks door de combine utility gemaakt en geinitialiseerd. Hierbij wordt de argLEX structure uit de LEX data ruimte gecopieerd.

4.3 LINKING & GEHEUGENINDELING

Op een gegeven moment hebben we de beschikking over een verzameling source files, zowel van de LEX als van de applicaties. Uit deze source files zal een object file gemaakt moeten worden.
Zoals al vermeld is eerst een tijdelijke oplossing gezocht door de LEX en de applicatie als een geheel te zien.

4.3.1 Tijdelijke oplossing

De LEX sources bevatten een main : het initialisatie gedeelte. De applicaties zijn alle als functions geschreven.
Dit geheel kan dus gecompileerd en in een keer gelinkt worden (met aparte text en data ruimten). Dit gebeurt met het commando :
ld86 -i -R textstart//0 -o <objectfile> <lexfiles> <appl. files>
Dit geeft de volgende geheugenindeling :



Bij het starten van de LEX zal het initialisatie gedeelte de process stacks initialiseren, referenties naar de proces code zitten vast ingeprogrammeerd via de namen van de functions.
De process stacks zijn onderdeel van de LEX data en alle even groot.
Om zo eenvoudig mogelijk tot een werkende LEX te komen was dit een zeer bevredigende oplossing.

Opm. : doordat alle segment registers nu nul zijn zullen fouten door het veranderen van segment registers (wat nogal eens gebeurt in de LEX) niet aan het licht komen. De segment registers moeten nul zijn omdat het LEX data gebied altijd op nul start (in verband met de interrupt vector tabel). Ze zijn voor LEX en applicatie gelijk omdat er maar een text en een data segment is.

4.3.2 Definitieve oplossing

Ook bij de definitieve oplossing bevat de LEX een main : de initialisatie. Processen, handlers en process stacks zijn nu echter compleet onafhankelijk van de LEX.
Er wordt nu gewerkt met een aantal onafhankelijke load modules (een gecompileerd en gelinkt programma). Een van deze modules is

de LEX. De applicatie modules bevatten een of meerdere processen. Dit heeft tot gevolg dat :

- processen in een load module kunnen onderling gebruik maken van in die module gedefinieerde globale data.

- code en data gedeelte van een load module zijn ieder maximaal 64 K. dit is eventueel een beperking voor het aantal processen in een module.

De mogelijkheid van globale data is erg wenselijk omdat anders alle communicatie tussen processen via mailboxes plaats moet vinden. Voor processen die een hoeveelheid administratie gemeenschappelijk hebben is dit erg bezwaarlijk en soms onmogelijk. Een mailbox kan namelijk pas gelezen worden als er iets in gezet is, een variabele kan altijd gelezen worden.
Het gebruik van globale data geeft echter wel problemen als we processen re-entrant gaan gebruiken, dat wil zeggen er zijn meerdere processen die dezelfde code gebruiken. De combine utility zal in dat geval een exemplaar van de code en data in de object file plaatsen en voor ieder proces apart een stackframe en een process descriptor creeren. Dit ene exemplaar van de code is juist wat we willen hebben, maar het ene exemplaar van data geeft problemen. Iedere nieuwe variant van een proces behoort zijn eigen variabelen te hebben. Voor de auto variabelen is dit altijd zo omdat ze op de stack geplaatst worden (die per proces verschillend is). Er zijn twee remedies :

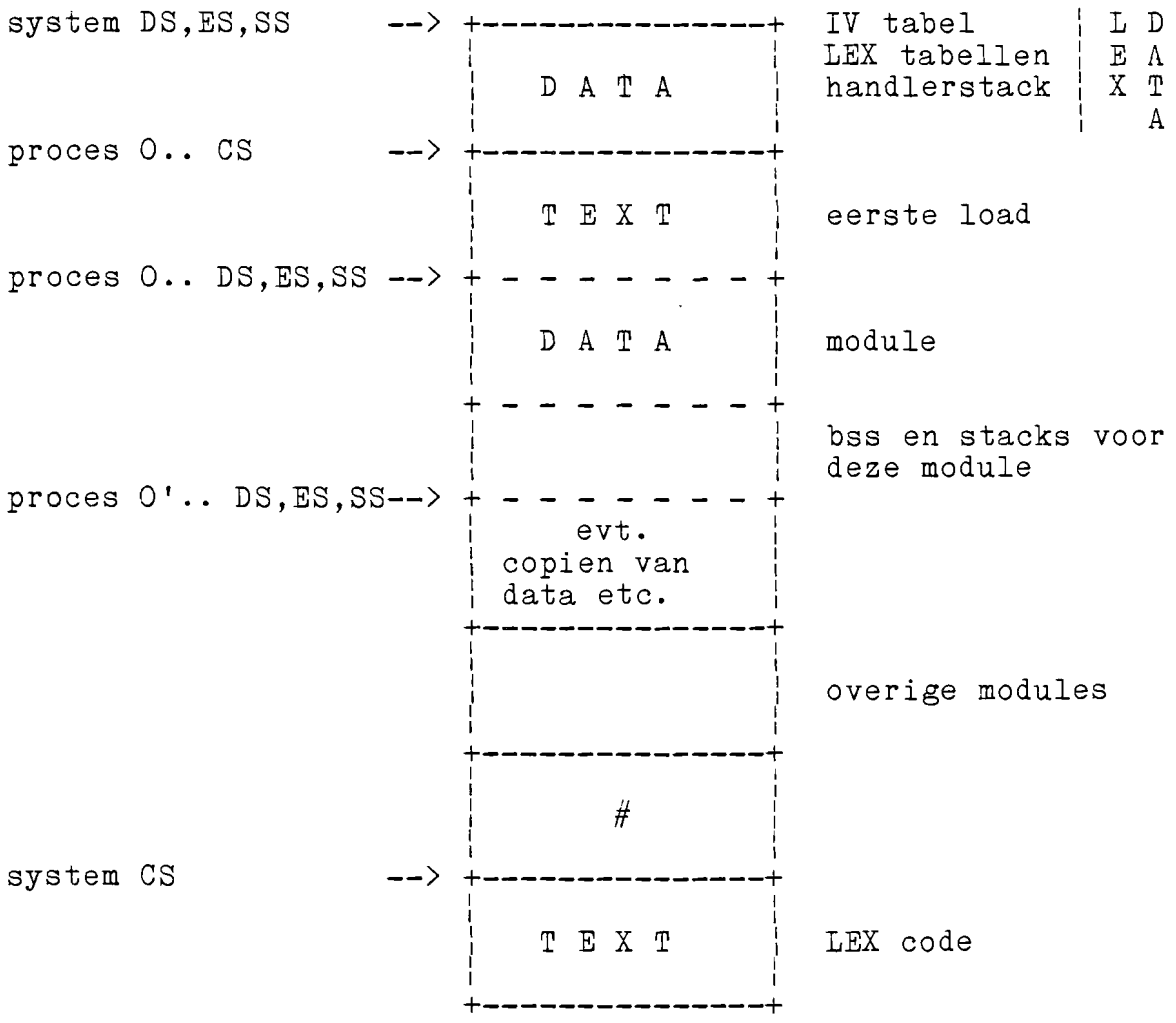
1. Laat de combine utility elke maal dat de processen in deze load module re-entrant gebruikt worden een data gebied in de object file creeren. Elk proces vindt in zijn process descriptor de start van zijn data segment.
   Voordelen : elke set processen (samen in een load module) heeft gewoon een eigen data segment en kan dus normaal aan de globale variabelen refereren. Het hele dupliceer gebeuren is onzichtbaar voor de gebruiker. Voor elke set processen is een datasegment van maximaal 64K beschikbaar.
   Nadelen : ook data die niet als variabele gebruikt wordt (bijvoorbeeld stringconstanten) wordt gedupliceerd. Net als de code is hier maar een exemplaar van nodig.

2. Plaats alle globale variabelen die we per load module gedupliceerd willen zien in een structure. Declareer een array van deze structures met zoveel elementen als we exemplaren van de betreffende processen willen hebben. Omdat elke nieuwe variant van een proces via de argLEX parameter een volgnummer heeft, kan dit volgnummer gebruikt worden als een index in dit array. Elk proces kan dan bijvoorbeeld een pointer initialiseren naar zijn eigen globale structure.
   Voordelen : we beslissen zelf welke data gedupliceerd wordt en welke niet.
   Nadelen : alle referenties aan globale variabelen moeten nu via pointers. De gebruiker moet nu zelf een array declareren, pointers initialiseren, kortom hij

moet precies weten wat hij doet. Er is nu slechts een datasegment met alle data (evt. gedupliceerd) en alle proces stacks. Dit geheel is maximaal 64K. Dit kan een probleem worden als we veel processen of veel exemplaren van hetzelfde proces hebben.

Aangezien beide methoden een verschillende combine utility vereisen zullen we nu een keuze moeten maken. Gezien de geheugen grootte beperking en de problemen voor de gebruiker is de tweede methode het minst geschikt.

De geheugenindeling is nu als volgt :



Er is een combine utility ontwikkeld die uitgaande van de LEX en applicatie object files twee files creeert : een met de LEX code (evt. in ROM te plaatsen), en een met de LEX data en alle processen (altijd in RAM).
Kennis over de LEX object file (structures ed.) wordt verkregen door het includen van de file structs.h, kennis over de applicatie object files wordt via een door de gebruiker te maken

configuration file geleverd. Hierin wordt ook aangegeven hoe groot de stack voor elk proces moet zijn. Deze stacks worden door de utility tijdens het combineren tussengevoegd en geinitialiseerd. De process descriptors in de LEX data worden ook gedeeltelijk (stackpointer) tijdens het combineren geinitialiseerd. Het is mogelijk meerdere processen met dezelfde code op te nemen, er is dan maar een exemplaar van de code van het betreffende proces, maar er is voor elk proces een aparte plaats in de process descriptor tabel en elk proces heeft een eigen datasegment met daarin een eigen stack. Ter onderscheid tussen de verschillende processen met dezelfde code bevat de argLEX parameter een veld (procid) dat een volgnummer (0..) per proces met identieke code bevat.

4.4 UTILITIES

4.4.1 aplcomp

Aplcomp dient om uitgaande van een of meerdere C of assembler source files een load module te creeren zonder kennis van compiler vlaggen, mee te linken support file etc.
Aplcomp is geimplementeerd als een Unix shell script.
De input (source) files worden zonder subscript opgegeven. Aplcomp test dan of file.c of file.a86 bestaat en zal deze dan compileren dan wel assembleren.
Tenslotte worden alle files gelinkt samen met lexsupp.b .

4.4.2 comb

Comb voegt de LEX en applicatie load modules samen.
Voor het samenvoegen moet een configuration file gemaakt worden waarin de informatie over de applicatie staat. In de LEX handleiding (appendix B) wordt deze configuration file beschreven.

Als uitvoer ontstaan twee files:
lex.rom : code van de LEX
lex.ram : data van de LEX en code en data van de applicaties
Deze files zijn standaard Unix object files. Alle informatie is hier in het datasegment opgeslagen. In de file lex.ram is ook een symbol-table aanwezig, dit zijn de beginadressen van de procescodes.
Door middel van een vlag is het ook mogelijk slechts een van beide files aan te maken.

Comb copieert text en data gedeelten van de invoer (object) files naar de uitvoer (memory image) files, voor de bss en stack

gedeelten worden nullen in de uitvoerfile geplaatst.
Bij het begin van een text of data segment worden ook telkens zoveel nullen tussengevoegd tot het adres een veelvoud van 16 is. Deze adressen moeten namelijk als segment adres kunnen fungeren.

De file lex.ram bevat altijd data startend vanaf adres 0. De file is van adres 0 tot aan START (offset startadres RAM gedeelte ten opzichte van 0) gevuld met nullen. Dit is helaas niet op een eenvoudige manier te vermijden. Als we de data namelijk op START laten beginnen moet deze offset op drie plaatsen bekend zijn :

1. Bij het linken omdat de linker de juiste adressen moet invullen en dus de offset vanaf 0 moet weten.
2. Bij het laden in het geheugen omdat de data uiteraard op de juiste plaats terecht moet komen. Dit kan bijvoorbeeld door de offset bij het maken van een hex file op te geven.
3. In de combine utility omdat de utility variabelen uit de file leest. De plaats van deze variabelen in de file vindt de utility met behulp van de symbol-table. De symbol-table bevat echter de adressen van de variabelen (inclusief eventule offset). In de file staat nu echter alleen de data. Om de plaats in de file te vinden moeten we dus de offset van het adres aftrekken.

Omdat het voorkomen van een constante op drie plaatsen de zaak niet flexibel maakt is voor de oplossing gekozen dat de data altijd op 0 start. Dit is als volgt gerealiseerd : bij ld86 wordt de optie -i gebruikt (aparte text en data ruimte, beide startend op 0). De declaratie van de data (in de file lexdecl.as) start nu met .=START (location pointer is START). Het gebied van 0 tot START wordt dan vanzelf met nullen gevuld.
Bij de andere oplossing wordt niets met de location pointer gedaan. Ld86 wordt nu gebruikt met de optie -R 0//START : text start op 0, data op START.

De symbol-table voor lex.ram wordt eerst in twee array's opgebouwd en na verwerking van alle applicatie files naar de file lex.ram geschreven.

In de configuration file moet het aantal semaphoren, mailboxes en processen worden opgegeven. Deze staan ook al als constanten in de LEX. Dit heeft de volgende reden : we willen liefst een LEX gebruiken voor verschillende applicaties of verschillende stadia van een applicatie. De LEX kan dan eventueel in ROM geplaatst worden. Om toch dezelfde LEX te kunnen gebruiken voor toepassingen met verschillende aantallen semaphoren, mailboxes of processen kiezen we voor de constanten in de LEX het maximale aantal van elk dat kan voorkomen. In de LEX dataruimte zal dan genoeg ruimte gereserveerd worden voor het maximale aantal.
In de configuration file geven we dan het werkelijke aantal op dat gebruikt gaat worden. Dit aantal wordt dan geinitialiseerd. Deze werkelijke aantallen worden ook in de LEX ingevuld, zodat het maximale en het geinitialiseerde aantal in de LEX bekend is.
Het aantal opgegeven werkelijke processen moet exact gelijk zijn aan het aantal aanwezige processen. Met de process descriptors wordt namelijk een linked list opgebouwd. Hierin mogen natuurlijk alleen process descriptors voorkomen die naar een aanwezig proces wijzen. De overgebleven process descriptors kunnen wel later

ingevuld worden en aan de linked list (ry_list) toegevoegd. Zo kan een proces gecreeerd worden. Het deleten van een proces is ook mogelijk, alleen krijgen we dan een lege plaats midden in de process descriptor tabel. Dit zou dan geadministreerd moeten worden.
Semaphoren en mailboxes staan ieder op zichzelf. Daarom is het niet echt nodig het werkelijk gebruikte aantal te initialiseren, ze zouden even goed allemaal geinitialiseerd mogen worden. Bij herhaaldelijk gebruik van dezelfde LEX object file is het echter mischien niet meer duidelijk hoeveel semaphoren of mailboxes maximaal gebruikt mogen worden. Daarom moet het werkelijke aantal in de configuration file opgegeven worden. Het is zo onmogelijk ongemerkt het maximum te overschrijden. Bovendien moet de utility dit aantal kennen als meerdere (onafhankelijke) applicaties gecombineerd moeten worden. Het adres van het eerste object in de argLEX parameter wordt dan per applicatie verhoogd met het aantal gebruikte objecten door die applicatie zodat de volgende applicatie niet dezelfde objecten kan refereren.

Van de functions die als proces optreden hoeft alleen de naam opgegeven te worden, comb zal het bijbehorende adres opzoeken met behulp van de symbol-table in de object file.

De grootte van lex.ram wordt in de LEX data ingevuld, zodat bij het initialiseren bekend is hoeveel RAM LEX plus applicatie gebruiken en welk stuk RAM nog vrij is.

4.4.3 hex

Hex zet files zoals door comb gemaakt om naar Intel hex formaat.

Hex is een variant op de bestaande utility ihex86. Ihex86 is echter beperkt tot segmenten van 64K. Te zijner tijd moet dit aangepast worden, maar omdat in ons geval (lex.rom en lex.ram) alle informatie in het datasegment staat, is even een tussenoplossing gemaakt.

De werkwijze van hex is :

De door de gebruiker gegeven offset wordt als base adress record gegenereerd. Vervolgens wordt de inhoud van de invoerfile als data records gegenereerd. Is de lengte van de file groter als 64K dan wordt na 64K een nieuw base adress record tussengevoegd.
De zaak wordt afgesloten met een end of file record.

4.5 TESTPROCEDURE & TESTRESULTATEN

Als eerste is de LEX getest met slechts een proces, dit om de initialisatie te testen en te komen tot proces niveau.
Vervolgens is telkens met behulp van twee processen een functie van de LEX (system call) getest.

Bij het testen zijn nog geen structurele fouten in het ontwerp naar voren gekomen, maar wel een aanzienlijk aantal programmerings fouten, varierend van typefouten tot stackmisbruik.

De interne namen van de system calls zijn tijdens het testen gewijzigd (dit is definitief). Een aanroep van bijvoorbeeld een p system call ging als volgt :

p (applicatie) --> softw. interrupt --> a_p (LEX) --> p (LEX)

Er bestaan dus twee (verschillende) routines met de naam p. Als de LEX en applicatie apart gelinkt worden kan dit. Omdat bij het testen alles samen gelinkt werd en deze dubbele naam toch verwarrend is, is dit gewijzigd. De system calls intern in de LEX hebben een s_ prefix gekregen.
De p routine heet nu bijvoorbeeld in de LEX s_p.

test 0 : initialisatie

```
proc()
{ while(1) proceed();
}
```

test 1 : process switching

```
proc0()
{ while(1)
  { print('0');
    proceed();
  }
}
proc1()
{ while(1)
  { print('1');
    proceed();
  }
}
```

opm. : print is een (tijdelijke) routine in de LEX die een karakter op de console print.

test 2a : prioriteiten, semaphoren (WAIT mode), p & v

```
proc0()
{ struct semaphore *s;
  s = sys_lo.sem0;        /* rechtstreeks uit LEX
                             i.p.v. argLEX parameter */
  myprioty(16);           /* verhoog prioriteit (default is 64) */
  while(1)
  { print('0');
    p(s,WAIT);
  }
}
proc1()
{ struct semaphore *s;
  s = sys_lo.sem0;
  while(1)
  { print('1');
    v(s);
  }
}
```

test 2b : semaphoren reset functie (waarde geven)

als test 2a, maar in proc0 :

```
  myprioty(16);
  r(s,5);                 /* preset s op 5 */
```

test 2c : semaphoren reset functie (deblokkering)

als test 2a, maar in proc1 :

  r(s,2) in plaats van v(s)

test 3a : mailboxes pmail & vmail

```
proc0()
{ struct mailbox *m;
  struct message msg;
  char c;
  m = sys_lo.mbox0;
  myprioty(16);
  while(1)
  { print('0');
    pmail(m,&msg,WAIT);
    c = (char) msg.lo;
    print(c);
  }
}
proc1()
```

```
{ struct mailbox *m;
  struct message msg;
  m = sys_lo.mbox0;
  msg.hi = 0;
  msg.lo = (int) 'x';
  while(1)
  { print('1');
    vmail(m,&msg,WAIT);
  }
}
```

test 3b : pmail op volle mailbox

als test 3a, maar in proc0 :

```
  myprioty(128);
```

test 3c : rmail

als test 3a, maar in proc1 :

```
  rmail(m) in plaats van vmail(m,&msg,WAIT)
```

test 4 : getseg & dispseg

```
proc0()
{ struct segptr s1, s2, s3;
  while(1)
  { getseg(&s1,1);
    getseg(&s2,1);
    getseg(&s3,2);
    dispseg(&s1);
    dispseg(&s2);
    dispseg(&s3);
    getseg(&s1,2);
    getseg(&s2,3);
    dispseg(&s2);
    dispseg(&s1);
  }
}
proc1()
{ while(1) proceed();
}
```

opm : de juiste werking is onderzocht door het programma in stappen uit te voeren (breakpoint) en na elke getseg of dispseg de segment tabel te bekijken.

test 5a : interrupt handler indirect attached

```
proc0()
{ int handler();
  attach(96,handler,INDIRECT);
  enable(96);
  while(1)
  { print('0');
    proceed();
  }
}
handler()
{ print('h');
  intret(96);
}
proc1()
{ extern int96();
  while(1)
  { print('1');
    int96();
    proceed();
  }
}
```

opm : int96 is een tijdelijke routine in de LEX die een software interrupt (vector 96) genereert.

test 5b : interrupt handler direct attached

als test 5a, maar in proc0 :

attach(96,handler,DIRECT) in plaats van attach(96,handler,INDIRECT)

en handler :

```
_handler:
        push    ax
        push    dx
        push    *'h'
        call    _print
        add     sp,*2
        push    *96
        call    _intend
        add     sp,*2
        pop     dx
        pop     ax
        iret
```

test 5c : hardware interrupt en end of interrupt

als test 5a, maar nu vector 134 in plaats van 96, en geen int96

meer. de interrupts worden nu met de hand gegenereerd op de PIC.

4.6 OPMERKINGEN EN SUGGESTIES

Mogelijke deadlock door de system semaphore.

Bij kritische acties in de systeem tabellen wordt de zaak afgeschermd met behulp van lock/unlock. Lock zet de system semaphore, unlock geeft hem vrij. Wordt er nu voor een unlock nog een lock gedaan (kan dit voorkomen ?) dan wordt actief op de system semaphore gewacht. Omdat er nu niets anders gedaan wordt dan wachten (interrupts zijn gedisabled) wordt de system semaphore niet vrijgegeven.
Het lijkt zinvol de system semaphore gewoon overboord te gooien.

Attach van software interrupts.

Momenteel kunnen handlers in de LEX alleen maar aan hardware interrupts gekoppeld worden. Als we het mogelijk maken dat handlers ook aan software interrupts gekoppeld kunnen worden, kunnen we als het ware system calls toevoegen. Via een software interrupt zijn deze routines dan namelijk vanuit elk proces te activeren.
Omdat software interrupts niet gedisabled kunnen worden moeten we ervoor zorgen dat er geen interrupt optreed voordat de betreffende handler attached is. Mischien is een aparte attach routine geschikter. Die kan dan wanneer de interrupt disabled zou moeten zijn een nullsys call in plaats van de handler neerzetten.

Wijzigingen ten opzichte van verslag Lex Borger.

Er is in de loop der tijd het een en ander gewijzigd aan de LEX. De achtergronden en structuur, zoals beschreven in het verslag van Lex Borger [2], gelden echter nog steeds. Hier volgt een lijst van zaken die gewijzigd zijn en bij bestudering van het oorspronkelijke verslag handig zijn om te weten.

- De system calls hebben intern (file lexcall.c) een s_ prefix gekregen.

- De environment structure is precies andersom gedeclareerd. Bij het redden van de process environment worden alle registers door een combinatie van acties op de stack gered. Om aan deze registers vanuit een C programma te kunnen refereren is in C een structure gedeclareerd die dezelfde layout heeft als de geredde environment op de stack. De volgorde van de registers in deze structure was echter net andersom.

- De argLEX parameter is aangepast. Een aantal parameters was namelijk alleen voor de initialisatie nodig. De initialisatie is echter geen proces tussen de processen geworden, maar is een stuk LEX code dat rechtstreeks aan alle LEX data kan refereren. Ook is de argLEX parameter nu niet meer voor alle applicaties gelijk. De startadressen voor de objecten (semaphoren en mailboxes) worden nu per applicatie opgehoogd, zodat elke applicatie een eigen set objecten heeft.

- Een aantal constanten zijn in een include file geplaatst (config.h). Eerst waren er namelijk constanten in verschillende files met dezelfde betekenis. Bovendien moest dan voor het wijzigen van deze constanten in de LEX sources ingegrepen worden.
  De C pre-processor wordt gebruikt om de constanten in assembler files toe te kunnen passen. Alle assembler files hebben nu een .as suffix.

- Het argument van de dispseg system call is nu een pointer naar een segmentpointer overeenkomstig de getseg system call. Dit is namelijk logischer voor de gebruiker.

- In sysleave is een test toegevoegd zodat gedetecteerd wordt als er geen ready proces meer is. Er wordt dan in een lus in sysleave gewacht.
  Een andere oplossing is het toepassen van een (dummy) altijd ready zijnd proces. De extra test is echter zeer eenvoudig en klein en geeft de zekerheid dat het systeem in dit opzicht altijd veilig is.

- De routine newsp redt nu ook het bp register. Bij het wisselen van stack in een functie (bijv. system call) moet bp gered worden omdat bp bij het beeindigen van de functie (door middel van de leave instructie) gebruikt wordt.

- De interrupt vector tabel is nu geen LEX data meer maar wordt dynamisch opgebouwd tijdens de initialisatie van de LEX. Dit heeft als voordeel dat de waarde van het code segment register niet meer van te voren vastgelegd. hoeft te zijn. Nu kan namelijk de actuele CS waarde gebruikt worden voor de interruptvectoren. Vroeger was dit gewoon een constante in de LEX. Bovendien worden nu alleen maar die vectoren geinitialiseerd die door de LEX gebruikt worden, zodat andere (bijvoorbeeld door de monitor gebruikte) vectoren onveranderd blijven.

- Er zijn drie files toegevoegd :

config.h	dit is een include file die de applicatie afhankelijke constanten voor de LEX bevat.
lexstart.as	deze file bevat de main, deze initialiseert de segment registers en een stackpointer. als stack wordt de handler stack gebruikt. deze stack is toch alleen maar voor de initialisatie nodig. ook bevat deze file de routine vectorinit die de gegeven interrupt vector plaatst in de vectortabel.

lexinit.c deze file bevat de overige initialisaties (zie par. 4.2). Na de initialisaties wordt met behulp van een sysleave call het eerste proces (in de process-descriptor tabel) gestart. Vanaf dat moment is het multi-tasking systeem operationeel.

V VOORTGANG

De serie I/O controller software is nog (lang) niet af, onder andere door het werk verricht aan de monitor en vooral de LEX. In dit hoofdstuk zullen een aantal zaken aangegeven worden die nog uitgewerkt moeten worden.

5.1 VERTAALTABELLEN

Zoals aangegeven in hoofdstuk 2.2 is een systeem van vertaaltabellen ontwikkeld. Per terminal is een set van twee tabellen (een ingangs en een uitgangstabel) nodig. Vragen die hierbij optreden zijn onder andere :

- Waar staan deze tabellen : in de dataruimte voor de processen voor die terminal of ergens apart in het geheugen ? Bedenk dat de tabellen geen vaste grootte hebben. Aangezien alle processen (inclusief dataruimtes) aaneengesloten in het geheugen liggen moet in het eerste geval een vaste (maximale) ruimte voor de tabel gereserveerd worden.

- Gebruiken meerdere identieke terminals dezelfde tabel of ieder een eigen kopie van dezelfde tabel ? Dit hangt uiteraard samen met de vraag waar de tabellen geplaatst worden.

- Hoe kan een gebruiker een nieuwe tabel laden ? Gaat dit via een (Unix) utility of rechtstreeks door de controller software ?

- Hoe wordt het Unix stty commando geimplementeerd ? Sommige opties zouden eigenlijk een verandering in de tabel moeten veroorzaken. Dit is echter niet zo eenvoudig gezien de opbouw van de tabel (linked lists).

- Waar staan de tabellen waaruit gekozen kan worden ? Staan deze op disk of zijn alle mogelijke tabellen ergens in het geheugen van de controller aanwezig ?

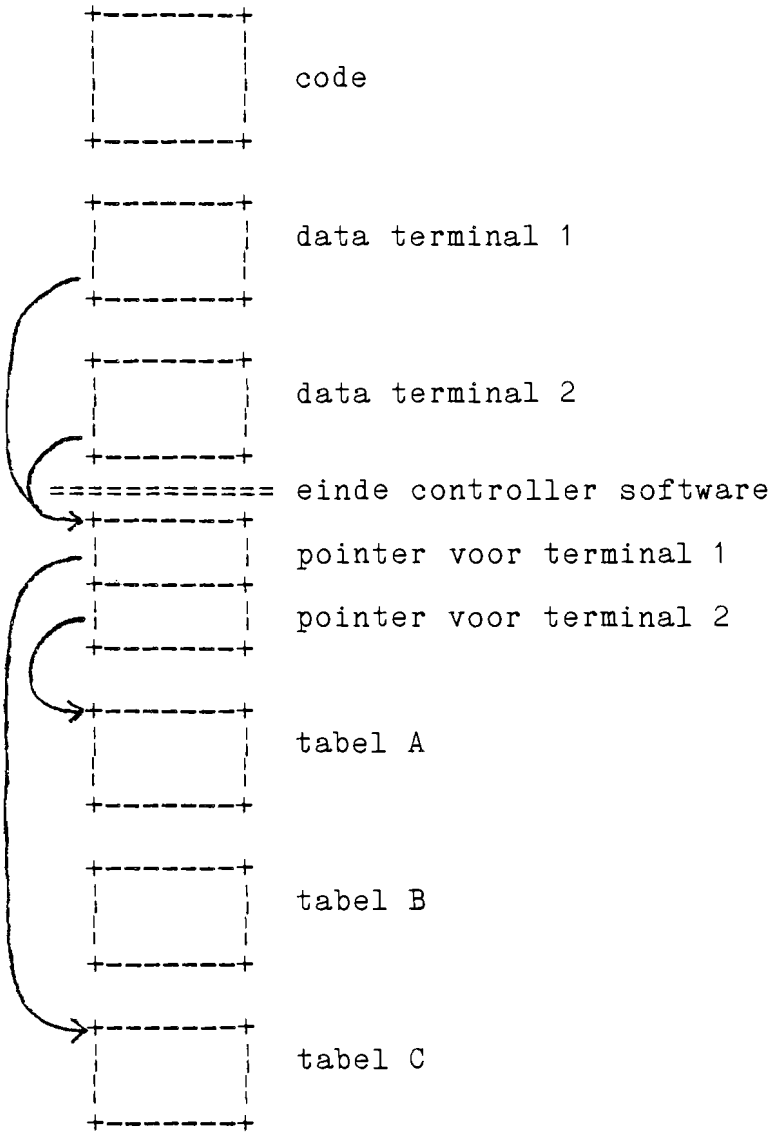
- Hoe wordt per terminal een default tabel ingesteld ?

- Hoe kan de verzameling beschikbare tabellen uitgebreid of gewijzigd worden ?

Meer als illustratie dan als een gefundeerd voorstel volgt hier een mogelijke oplossing :

In de vrije geheugenruimte (buiten LEX en controller software en data) zijn alle mogelijke tabellen opgeslagen, met natuurlijk wat administratie (welke tabellen staan er en waar staan ze). Hierbij staat ook een array (met zoveel elementen als er terminals zijn)

van pointers naar een tabel.
Bij het logisch openen van een terminal-verbinding ("inloggen") zal dan via dit array de default tabel voor die terminalaansluiting gevonden worden en naar de dataruimte voor die terminal gekopieerd worden. Eventueel kan ook een lijst van mogelijke tabellen getoond worden met de vraag welke geladen moet worden. Dit bijvoorbeeld voor modemlijnen waarbij het type terminal vooraf niet vaststaat.



Dit array van default tabellen en de verzameling tabellen is dus gescheiden van de controller software en kan dus aangepast worden zonder in de eigenlijke software in te grijpen.
Het voorstel is om elke keer bij het inloggen een default tabel te laden omdat men zo onafhankelijk is van wat een vorige gebruiker gedaan heeft. Een andere mogelijkheid zou zijn alleen bij het starten van het systeem een set default tabellen te laden.

Voor het inloggen moet er natuurlijk wel een (zeer eenvoudige) tabel aanwezig zijn om te kunnen inloggen.

5.2 NIET-TERMINAL I/O

De ontwikkelde software is vooral ontworpen voor data transport op regelbasis. Bij bijvoorbeeld screeneditors vindt data transport op karakterbasis plaats. Als we dan toch het hele opdracht/terugmelding verkeer handhaven wordt de overhead per karakter wel erg groot. Er zou bijvoorbeeld een "set raw" commando kunnen zijn waarna met verkorte opdrachten/terugmeldingen gewerkt kan worden, of zelfs met direkte data overdracht (naar een afgesproken plaats).
Een ander uiterste is het transport van zeer grote blokken data zonder een duidelijke onderverdeling in regels of dergelijke. Denk hierbij bijvoorbeeld aan ponsband lezers/ponsers of communicatie tussen computers onderling. Hierbij stellen we meestal geen prijs op regel editing of karakter conversies. Er zou dan een stuk software uitgeschakeld moeten worden.
Beide gevallen zijn eigenlijk moeilijk in de ontworpen structuur onder te brengen.

5.3 BOOTEN

De applicatiesoftware zal (gedeeltelijk) in RAM moeten staan. De code zou eventueel (samen met de LEX code) in PROM geplaatst kunnen worden, maar de data (LEX en applicatie) moet in RAM staan. De totale hoeveelheid data en code zal waarschijnlijk wel te groot zijn om beide in PROM te plaatsen en na starten de data naar RAM te kopieren. Bovendien zullen we bepaalde data (tabellen e.d.) niet in PROM willen hebben in verband met de flexibiliteit. Er zijn dus een aantal mogelijkheden, bijvoorbeeld :

- Alles staat in ROM, dat wil zeggen LEX, controller software en data. Dit bijvoorbeeld voor vaste, niet te grote applicaties.
- De LEX plus controller software staat in ROM, data wordt geladen.
- Alleen de LEX staat in ROM, de hele applicatie wordt geladen.
- Er is alleen een monitor of iets dergelijks in ROM, alles (LEX plus applicatie) wordt geladen.

Na het starten van het systeem zal dus eventueel een hoeveelheid data en/of software geladen moeten worden. Hoe dit gedaan wordt en waar dit vandaan komt is nog de vraag. De software zou via Unix geladen kunnen worden (file op disk) of rechtstreeks van de disk controller.

5.4 UITBREIDINGEN LEX

Er zijn nog tenminste drie zaken die eigenlijk aan de LEX toegevoegd zouden moeten worden :

1. De mogelijkheid om dynamisch een proces toe te voegen of te verwijderen. Zie ook de opmerkingen hierover in par. 4.4.2.

2. De mogelijkheid om processen een verschillende default prioriteit te geven. De prioriteit voor een proces zou dan in de configuration file opgegeven moeten worden. Deze prioriteit kan dan door de combine utility in de process-descriptors ingevuld worden. Het probleem is dan echter dat de linked list van process-descriptors niet op prioriteit gesorteerd is (maar op volgorde van voorkomen in de configuration file).

3. De mogelijkheid om mailboxes van variabele grootte te hebben, dat wil zeggen dat we bijvoorbeeld in de configuration file per mailbox de grootte kunnen opgeven. De vereist een andere declaratie van de mailbox structure. De mailbox zelf is nu een onderdeel van deze structure en daarom noodgedwongen voor alle boxes even groot. We zouden ergens een aparte verzameling boxes willen hebben en in de mailbox structure een pointer naar zo'n box. Dit betekent wel dat alle system calls werkende op mailboxes aangepast moeten worden.

A. HANDLEIDING MONITOR

A.1 INLEIDING

De monitor is een systeem programma dat altijd in PROM aanwezig is. De functies van de monitor zijn :

1. het initialiseren van de hardware voorzover nodig voor het functioneren van de monitor.

2. het bieden van een aantal debugging faciliteiten.

De volgende geheugengebieden zijn voor de monitor gereserveerd :

in PROM : fe000 t/m fffff (2 * 2732)
: fc000 t/m fffff (2 * 2764)
:
: e0000 t/m fffff (2 * 27512)

in RAM : dit hangt van de actule monitor versie af. Verifieer !

De interrupt vector tabel wordt ook door de monitor geinitialiseerd.

A.2 COMMANDO BESCHRIJVING

Een commando wordt afgesloten met een return, een backspace wist het laatst getypte karakter.
Alle letters (ook in hex getallen) moeten in lower case gegeven worden.
Alle adressen, counts en data zijn in hex en hoeven niet met nullen aangevuld te worden (1 voldoet i.p.v. 0001).

Een adres wordt als volgt gespecificeerd :

<adres> ::= [<segm>:]<offset>

<segm> ::= cs|ds|es|ss|<word>

<offset>::= <word>

<word> ::= 0..ffff

als <segm> niet gespecificeerd wordt, wordt DS genomen, met uitzondering van de g, n en u commando's die CS als default hebben.

b : baud rate selection

b<3|12|48|96|192|384>

Kies de baudrate voor poort A

c : calculate

c<word1><word2>

Geeft som en verschil van word1 en word2 (in hex)

d : dump memory

d[b|w] <adres>

Dumpt het geheugen vanaf opgegeven adres in bytes of words.
Default is words.
Bij dump in bytes wordt ook het overeenkomstige ASCII karakter getoond.
Er wordt telkens een regel (16 bytes) getoond, een spatie geeft de volgende regel, een return keert terug naar commando niveau.

f : fill

f[b|w] <adres> <data> <count>

Vul count bytes of words (default) vanaf adres met data.

g : go

g[<adres>][-<eindadres>]

Start een gebruikersprogramma vanaf adres (default is CS:IP).
Als eindadres opgegeven wordt wordt op dit adres een breakpoint geplaatst.

h : help

h

Geeft een overzicht van de beschikbare commando's
en hun syntax.

i : input

i<b|w> <port>

Input een byte of word van opgegeven inputport.

lt2 : load

lt2[:<string>]

Laad data (in Intel hex formaat) via seriepoort A.
De eventueel opgegeven string wordt eerst (afgesloten
met een return) naar poort A verzonden.

m : move

m <from_adres> <eind_adres> <to_adres>

Copieer bytes van from_adres t/m eind_adres naar to_adres
en verder.
Voor eind_adres mag alleen maar een offset opgegeven worden,
geen segment specificatie.

n : next instruction

n[<count>][-<eind_adres>]

Voer count instructies (default is 1) uit en toon
alle registers en de volgende instructie.
Als eind_adres opgegeven is gaat dit door totdat count instructies
zijn uitgevoerd of eind_adres bereikt is (afhankelijk van aan welke
voorwaarde het eerst wordt voldaan).

o : output

o<b|w> <port> <data>

Output de gegeven databyte of word naar port.

r : display registers and flags

r[f<id> | <rid>]
<id> ::= flagname

<rid> ::= registername

r toont de inhoud van alle registers en flags zoals
gebruikt in een gebruikersprogramma.
r gevolgd door een register of flag identificatie toont
het gegeven register of flag. Hierna kan een nieuwe waarde
gegeven worden (een return laat het ongewijzigd).

s : substitute memory

s[b|w] <adres>

Toon byte of word (default) vanaf adres.
Hierna kan een nieuwe waarde gegeven worden of een
return (laat hem ongewijzigd).
Een punt (.) als nieuwe waarde keert terug naar
commando niveau.

t : transparant

t

Verbind de console met poort A.
^G keert terug naar commando niveau.

u : unassemble

u <adres>

Disassembleer de instructie op het gegeven adres.
De notatie is als de Unix 8086/80186 assembler.
Een spatie geeft de volgende instructie, een return keert
terug naar commando niveau.

B. HANDLEIDING LOCAL EXECUTIVE

B.1 INLEIDING

De local executive (LEX) is een real-time multi-tasking operating system.
De applicatie wordt gedefinieerd als een aantal processen en handlers, dit zijn de actieve systeemobjecten. Andere objecten zijn semaphoren, mailboxes en geheugensegmenten.

Processen zijn "oneindig" (eindeloze lus). De LEX kent ze middels een tabel met process descriptors. Hierin staan de geredde waarden van stackpointer en stacksegment register. Bovenaan de stack van een onderbroken proces staat de environment (alle processor registers) nodig om het proces weer te starten.
Er is een ready-list van processen. Nadat een proces ready gemaakt is (bijvoorbeeld door een v operatie op een semaphoor) wordt het proces hier ingesorteerd naar gelang zijn prioriteit. Ook het wachten op een semaphoor of mailbox gebeurt met een queue geassocieerd aan die semaphoor of mailbox.
De dispatching, het wisselen van actief proces, gebeurt telkens bij beeindiging van een system call of interrupthandler. Dat betekent dat een running proces running blijft tot tot het zelf een system call doet (en daarin wacht of een proces met hogere prioriteit ready maakt) of tot een interrupt handler een proces met hogere prioriteit ready gemaakt heeft.

Interrupts worden via de LEX afgehandeld door interrupt handlers. Een handler hoort wel bij een bepaald proces, deelt daarmee segmenten en globale data, maar leidt verder een eigen leven. Ook als het proces wachtende is komt bij een interrupt de handler meteen in actie, alleen opgehouden door eventuele interrupts van hogere prioriteit. Bij beeindiging wordt weer een dispatch uitgevoerd, waardoor het onderbroken proces (dat in het algemeen niets met de interrupt te doen had) eventueel wachtend wordt en een door de handler gewekt proces met hogere prioriteit actief wordt.

Er zij twee typen handlers : indirect en direct.
Bij een indirecte handler wordt de hele proces omgeving door de LEX gered voordat de handler geactiveerd wordt. Door een speciale exit routine wordt de handler verlaten.
Een directe handler wordt meteen na een interrupt geactiveerd, dit verhoogt de reactie snelheid, maar de programmeur is verantwoordelijk voor het redden van alle gebruikte registers. Directe handlers kunnen ook alleen maar in assembler geschreven worden. Zij kunnen wel C functions aanroepen mits zij geen pointer referenties gebruiken die aan stack variabelen refereren. Er wordt namelijk gewerkt op een aparte handlerstack.

Semaphoren zijn van het tellende type. De p en v operaties zijn beschikbaar evenals een r operatie om een semaphoor op een waarde te initialiseren.

Mailboxes dienen om messages door te geven. Een message is 32 bit breed, voor grotere messages kan dit gebruikt worden als een pointer.

Wachten op een semaphoor of mailbox vindt plaats in een wait of no-wait mode. In de wait mode is het wachtende proces geblokkeerd tot de semaphoor of mailbox door een ander proces (of handler) vrijgegeven wordt. In de no-wait mode geeft de system call een error code terug als niet aan de gevraagde operatie voldaan kan worden. Mailboxes hebben een vaste grootte zodat het zenden van een message ook tot blokkering kan leiden.

Geheugensegmenten kunnen in veelvouden van 16 byte aangevraagd worden. De LEX administreert alleen het nog vrije geheugen, niet meer gebruikte segmenten kunnen weer teruggegeven worden. Omdat de segment tabel een vaste grootte heeft kan deze vol raken. De LEX meldt dit alleen via een error code maar neemt zelf geen actie. De tabel moet daarom zo groot gemaakt worden dat de kans op vol raken te verwaarlozen is. Verkregen segmenten mogen door meerdere processen gebruikt worden.

Voor een uitgebreide beschrijving van de LEX wordt verwezen naar [2]. Deze handleiding is voor een groot deel een uittreksel van dit verslag.

B.2 SYSTEM CALLS

Dit hoofdstuk geeft een beschrijving van de beschikbare system calls in de LEX. Error en parameter constanten zoals in deze beschrijving gebruikt zijn gedefinieerd in de file structs.h. Tenzij anders vermeld is de return waarde van een system call 0. De vector zoals bij elke call vermeld is de interne interrupt vector gebruikt voor die system call en kan gebruikt worden om de system calls vanuit assembler te activeren.
Alle system call namen zijn in lower case, met uitzondering van P, V, R.

NULLSYS () [vector=32]
De nullsys call doet niets, er wordt zelfs geen proces omgeving gered. Het doel van deze call is het hebben van een routine waarheen niet gebruikte interrupt vectors kunnen wijzen.

PROCEED () [vector=39]
De proceed call implementeert een "round robin" mechanisme in de LEX. Een proceed call zal het volgende proces in de ready lijst met dezelfde prioriteit actief maken. Een proces met een

lagere prioriteit kan nooit actief worden zolang er processen met een hogere prioriteit ready zijn.
Aangeroepen vanuit een handler is het effect alsof het geinterrumpeerde proces proceed aanriep. Een toepassing is bijvoorbeeld een timer interrupt.

MYPRIOTY (pr) [vector=40]
int pr;
Myprioty dient om de prioriteit van het proces op te vragen of te veranderen.
Als pr ongelijk aan nul is wordt pr de nieuwe prioriteit. In alle gevallen is de returnwaarde de prioriteit voor de aanroep.
Aangeroepen vanuit een handler is het effect alsof het geinterumpeerde proces myprioty aanriep.

SUSPEND () [vector=50]
Een aanroep van suspend is hetzelfde als een aanroep van p(&proc->su_sem, WAIT). Het proces wordt hierdoor geblokkeerd. Ready maken kan via v(&proc->su_sem).
Een aanroep van suspend vanuit een handler is een efficiente manier om het geinterrumpeerde proces te blokkeren. Een toepassing is bijvoorbeeld een debugger, die zo het actieve proces kan blokkeren.

P (sem, md) [vector=33]
struct semaphore *sem;
int md;
Op de (tellende) semaphoor wordt een p operatie uitgevoerd. Er zijn drie mogelijkheden :
1. De waarde van de semaphoor is positief. De waarde wordt dan met een verlaagd en het proces of de handler gaat verder.
2. De waarde van de semaphoor is nul en de md parameter is WAIT. Het proces wordt dan geblokkeerd tot een v operatie op deze semaphoor wordt uitgevoerd.
3. De waarde van de semaphoor is nul en de md parameter is NO_WAIT. Het proces of de handler gaat verder maar de return waarde is nu NO_UNITS.

Aangeroepen vanuit een handler is de md parameter niet van belang, altijd wordt NO_WAIT genomen.

V (sem) [vector=34]
struct semaphore *sem;
Op de semaphoor wordt een v operatie uitgevoerd. Als een proces op de semaphoor wacht wordt het wachtende proces met de hoogste prioriteit ready gemaakt, anders wordt de waarde van de semaphoor met een verhoogd.

R (sem, val) [vector=48]
struct semaphore *sem;
int val;
De waarde van de semaphoor wordt gelijk aan de waarde van de val parameter. Processen die wachten op deze semaphoor worden ready gemaakt.

PMAIL (mbox, msg, md) [vector=35]

```
struct mailbox *mbox;
struct message *msg;
int md;
```

De pmail call is analoog aan de p operatie op een semaphoor. In tegenstelling tot een semaphoor kunnen nu echter messages doorgegeven worden. Als er een message beschikbaar is zal deze aan het proces doorgegeven worden op de plaats *msg in zijn data segment.
Als de mailbox vol was zal na de pmail call het proces met de hoogste prioriteit wachtende met een vmail call door kunnen gaan.
De betekenis van de md parameter is als bij de p call. De error code als er geen message beschikbaar is is NO_MSG.

VMAIL (mbox, msg, md) [vector=36]

```
struct mailbox *mbox;
struct message *msg;
int md;
```

De vmail call is analoog aan de v operatie op een semaphoor. Als de mailbox nog niet vol is (een mailbox heeft een vaste grootte, gelijk voor alle mailboxes in het systeem) wordt de message van de plaats *msg in het data segment van het proces gecopieerd naar de mailbox queue. Als er een proces wachtende is met een pmail call wordt het wachtende proces met de hoogste prioriteit ready gemaakt.
Als de mailbox vol is kan op plaats gewacht worden of een error code (NO_MSG) als return waarde gegeven worden (afhankelijk van de md parameter). Zie de p system call voor details betreffende de md parameter.

RMAIL (mbox) [vector=49]

```
struct mailbox *mbox;
```

De rmail call reset de opgegeven mailbox. De mailbox queue wordt leeggemaakt en alle wachtende processen worden ready gemaakt en ontvangen een message bestaande uit nullen.

GETSEG (seg, siz) [vector=37]

```
struct segptr *seg;
int siz;
```

Met de getseg call kan een geheugensegment aangevraagd worden. Geheugen wordt geadministreerd in partities van 16 byte.
De siz parameter specificeert het gewenste aantal partities. Als er een segment van de gewenste grootte vrij is zal de segment *seg in het data segment van het proces hiernaar wijzen, anders wordt de error code NO_MEM als return code gegeven.

DISPSEG (seg) [vector=38]

```
struct segptr *seg;
```

Het geheugensegment, aangegeven door de seg parameter, wordt weer bij het vrije geheugen gevoegd. Omdat de segment table een vaste grootte heeft kan deze vol raken als het geheugen te zeer versnipperd raakt (aansluitende vrije segmenten worden tot een segment gemaakt). In dit geval wordt de error code NO_TABLE als return waarde gegeven. Dit is een systeem fout

maar er wordt door de LEX geen actie ondernomen. De segment table moet daarom groot genoeg gemaakt worden.

ATTACH (vect, addr, md) [vector=41]
int vect, (*addr)(), md;
Met behulp van deze system call wordt een handler gekoppeld aan een interrupt. De routine *addr wordt nu elke keer als de interrupt met vector vect optreed uitgevoerd (als deze interrupt ook enabled is).
Tijdens de executie van de handler zijn code- en data segment gelijk aan die van het proces dat de attach call deed.
Na een attach call is de betreffende interrupt disabled.
De md parameter is DIRECT of INDIRECT en bepaalt de methode van koppeling (zie inleiding).
Als vect geen te koppelen vector is is de return waarde ATT_UNABLE, en als vect al gekoppeld is is de return waarde ATT_ERROR.

DETACH (vect) [vector=42]
int vect;
Een eventuele koppeling tussen een handler en de interrupt met vector vect wordt verbroken. Als vect geen te koppelen vector is is de return waarde ATT_UNABLE.

ENABLE (vect) [vector=43]
int vect;
Na een enable system call kunnen interrupts met vector vect de processor interrumperen.
Als vect geen te koppelen vector is is de return waarde ATT_UNABLE en als vect niet gekoppeld is is de return waarde ATT_ERROR.

DISABLE (vect) [vector=44]
int vect;
Na een disable system call kunnen interrupts met vector vect de processor niet interrumperen. Eventule error codes zijn als bij de enable call.

INTEND (vect) [vector=45]
int vect;
Om de hardware een end-of-interrupt melding te geven moet door een direct gekoppelde handler de intend call aangeroepen worden.

INTRET (vect) [vector=46]
int vect;
Intret combineert het werk van intend en de speciale return acties voor een indirect gekoppelde handler. Deze routine moet gebruikt worden om een indirect gekoppelde handler te verlaten.

INTV (sem) [vector=51]
struct semaphore *sem;
Deze routine doet hetzelfde als de v system call, maar mag in tegenstelling tot deze v call door een direct gekoppelde handler aangeroepen worden.

INTDS (vect) [vector=47]
    int vect;
    Om in een direct gekoppelde handler gebruik te kunnen maken van het data segment van het proces dat de handler gekoppeld heeft moet intds aangeroepen worden.

B.3 SYSTEM UTILITIES

Met elke applicatie module moet de file lexsupp.b meegelinkt worden. Deze file bevat naast de system call interfaces ook een aantal utilities die functies in een C programma bieden die anders alleen met behulp van assembler gerealiseerd zouden kunnen worden. Deze utilities zijn :

newds(r) unsigned r;
    Plaats r in het DS register. De returnwaarde is de DS waarde voor de call.

newes(r) unsigned r;
    Als newds, maar nu voor het ES register.

io_write(p,r) int p,r;
    Schrijf woord r naar outputpoort p. De returnwaarde is r;

io_read(p) int p;
    De returnwaarde is het woord gelezen van inputpoort p;

B.4 HET SCHRIJVEN VAN DE APPLICATIE

B.4.1 inleiding

Een applicatie zal bestaan uit een of meerdere processen en nul of meerdere handlers. Beide kunnen zowel in C als in assembler geschreven zijn (m.u.v. directe handlers).

Een applicatie kan verdeeld worden over verschillende load modules (een compleet gecompileerd en gelinkt programma), maar een handler moet zich in dezelfde module bevinden als het proces dat hem controleert.
Elke module bevat minstens een proces. Functions kunnen of als functions gebruikt worden, of als proces fungeren. Doordat deze

processen samen gelinkt worden is het gebruik van globale data mogelijk.
Processen zullen in aparte modules geplaatst worden als :

- ze onafhankelijk van elkaar zijn (geen gemeenschappelijke data). door het apart linken is er dan geen onderlinge beinvloeding mogelijk door bijvoorbeeld het gebruik van variabelen met dezelfde naam.

- ze erg groot zijn. een module kan maximaal een text en data segment van ieder 64K bevatten.

B.4.2 system interface

Elke load module moet met behulp van een #include de volgende twee files opnemen :

structs.h : declaratie van systeemobjecten en (return)codes
lexsupp.h : declaratie van de system calls

Om te kunnen refereren aan systeem objecten heeft elk proces een parameter : argLEX.
Deze bevat per object type het aantal aanwezige objecten en het adres van de eerste. Hierdoor kan elk proces een pointer initialiseren naar elk gewenst object.
Het adres van het eerste object wordt per applicatie opgehoogd, zodat verschillende applicaties altijd verschillende objecten gebruiken. Tussen twee processen in een verschillende applicatie is dus geen synchronisatie of communicatie mogelijk.
Tevens bevat de argLEX parameter een veld (procid) dat per entry van hetzelfde proces een volgnummer (0..) bevat.
De declaratie van de argLEX parameter :

```
struct layout
{
        int procnr;             /* cumulative process number */
        int procid;             /* proces image number */
        struct semaphore *sem0; /* pointer to semaphore 0 */
        int nsem;               /* number of semaphores */
        struct mailbox  *mbox0; /* pointer to mailbox 0 */
        int nmbox;              /* number of mailboxes */
} argLEX;
```

B.4.3 processen en handlers

Processen :

- mogen niet eindigen
- kunnen alle system calls gebruiken, behalve intend, intds en intv
- mogen geen p operatie doen op de suspension semaphore van een ander proces

Indirectly attached handlers :

- mogen alle system calls gebruiken
- mogen geen pointer expressies gebruiken die naar stack variabelen refereren, maar alleen via namen. stack en data segment zijn nml. verschillend tijdens de executie van een handler.
- moeten eindigen met de intret routine in plaats van een return statement
- mogen ook direct door een proces aangeroepen worden (intret doet dan niets)

Directly attached handlers :

- moeten elk register redden voor gebruik
- geredde registers terugplaatsen
- moeten interrupts gedisabled houden
- mogen alleen de intend, intds en intv system calls gebruiken
- moeten eenmaal intend aanroepen
- moeten zo kort mogelijk zijn en zo weinig mogelijk stack gebruiken
- moeten eindigen met een iret instructie

Attachbare interrupts

```
1   : single step
2   : non-maskable interrupt
3   : breakpoint
8   : timer 0
10  : dma 0
11  : dma 1
13  : INT 1 (80186)
15  : INT 3 (80186)
18  : timer 1
19  : timer 2
128-135 : externe interrupt (8259) 0-7
```

Opm. : De hardware biedt ook andere mogelijkheden voor externe interrupts, maar het gebruik hiervan vereist een aanpassing in de LEX.

Voorbeeld van een load module :

```
#include "structs.h"
#include "lexsupp.h"

/* declaration of global variables */

proc1(argLEX)              /* first proces */
struct layout argLEX;
{ /* declaration of local variables */
  initialise();
  while(TRUE)
  {  /* repetitive process actions */
  }
}

initialise()
{ int handler1(), handler2();
  attach(132,handler1,INDIRECT);
  attach(133,handler2,DIRECT);
  /* other initialisations */
  enable(132);
  enable(133);
}

handler1()
{ /* declaration of local variables */
  /* interrupt service actions */
  intret(132);
}

_handler2:
        ¦ save used registers
        push    ax
        push    ds
```

```
        push    bx
        ¦ intds(133)
        push    *133
        call    _intds
        add     sp,*2
        ¦ interrupt service actions,
        ¦ using global data, ax & bx
        ...
        ...
        ¦ intend(133)
        push    *133
        call    _intend
        add     sp,*2
        pop     bx
        pop     ds
        pop     ax
        iret

proc2(argLEX)           /* second proces */
struct layout argLEX;
{ /* declaration of local variables */
  struct semaphore *s;
  s = argLEX.sem0 + argLEX.procid;
    /* if this process is used reentrant, then the first
       entry of this proces uses the first semaphore in sem.
       table, second entry uses second semaphore etc. */
  while(TRUE)
  { /* repetitive process actions */
  }
}
```

B.5 COMPILING EN LINKING

B.5.1 Applicaties

Elke load module wordt gevormd uit een of meerdere source files (in C of assembler). Deze files moeten gecompileerd of geassembleerd worden en gelinkt worden samen met lexsupp.b.

Dit alles kan gebeuren met het commando :

aplcomp outputfile inputfiles

met outputfile : de naam van de te creeren load module
inputfiles : de namen van de source files zonder suffixes
C sources moeten suffix .c hebben,
assembler sources suffix .a86

B.5.2 De LEX

Afhankelijk van de applicatie moet een aantal constanten in de LEX aangepast worden.
Deze constanten staan in de file config.h en hebben de volgende betekenis :

NPROC : maximaal aantal processen
NSEM : maximaal aantal semaphoren
NMBOX : maximaal aantal mailboxes
NMSG : grootte mailboxes (max. aantal messages)
NSEG : grootte segment table
HSTSZ : handler stack size (words)
START : startadres RAM gedeelte (LEX data + processen)

Een make lex.out commando zal nu een nieuwe LEX load module aanmaken.

B.6 SAMENVOEGEN LEX EN APPLICATIE

Voor het samenvoegen moet een configuration file gemaakt worden waarin de informatie over de applicatie staat.
De layout van deze file is :

```
applications
per application: { nprocs  nsem  nmbox  nfiles
                  per file:  { filenaam  procs  entries
                               per proc:   procnaam  stacksize
```

Verklaring symbolen :

applications	: aantal applicaties
nprocs	: totaal aantal processen (incl. meerdere entries van hetzelfde proces)
nsem	: aantal gebruikte semaphoren
nmbox	: aantal gebruikte mailboxes
nfiles	: aantal loadmodules voor deze applicatie
filenaam	: naam van een load module
procs	: aantal verschillende processen in deze load module
entries	: aantal malen dat deze processen gestart worden
procnaam	: naam van de function die als proces optreed
stacksize	: stacksize (words) voor dit proces

Een voorbeeld van een configuration file is :

```
2
6 3 2 1
file1 2 3
proc1 100
proc2 50
2 1 1 2
file.a 1 1
proc 50
file.b 1 1
proc 50
```

Het samenvoegen van de LEX en applicatie gaat dan met het commando :

```
comb [-rom|-ram]  configuration-file
```

Als uitvoer ontstaan twee files :
lex.rom : code van de LEX
lex.ram : data van de LEX en code en data van de applicaties

Zonder vlag worden beide files aangemaakt, met -rom of -ram wordt alleen lex.rom of lex.ram aangemaakt.

Dit zijn files volgens het Unix objectfile formaat en kunnen dus met pr86, nm86, sz86 etc. behandeld worden. Alle code en data is in het datasegment van de file opgeslagen.
De file lex.ram bevat altijd data startend op adres 0. De file is van adres 0 tot aan START (startadres RAM gedeelte) gevuld met nullen.
De startadressen van de processen zijn in de symbol-table van de file lex.ram opgenomen en kunnen dus bijvoorbeeld met nm86

opgevraagd worden.

Voor het omzetten van deze files naar Intel hex formaat is de utility hex beschikbaar :

hex offset [inputfile [outputfile]]

Bij het laden van de hex file zal de inhoud geplaatst worden startend op adres offset.

Als we een eventueel bestaande interrupt vector tabel bij het laden van lex.ram niet willen overschrijven moet het gedeelte met nullen tot aan START verwijderd worden. Dit kan bijvoorbeeld door grep op de Intel hex file. Er is een entry in de makefile die dit alles verzorgt :

make apl

zal van lex.rom en lex.ram de hex files rom.x en ram.x maken.

De rom file mag overal in het geheugen geplaatst worden, de ram file alleen vanaf adres START.

LITERATUUR

[1] Ontwikkeling van ondersteunende systeemsoftware voor een serie I/O controller,
door P. Bregman.
Afstudeerverslag, mei 1983

[2] A local executive for THE KUNix machine,
by L. Borger.
Afstudeerverslag, december 1983

[3] Een disk subsystem voor THE KUNix machine,
door W. Quaden.
Afstudeerverslag, oktober 1984