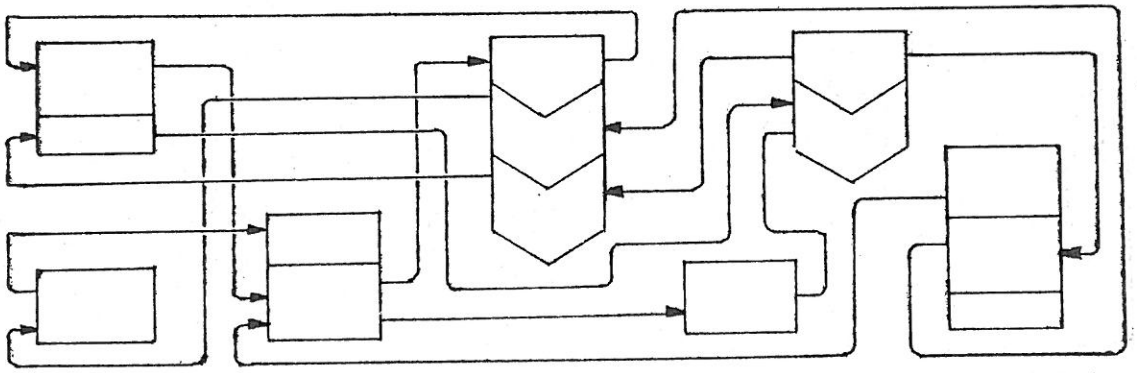
SYSTEM MODELLING



A methodology for describing the structure of complex Software, Firmware, and Hardware systems consisting of independent process components.

by
Ib Holm Sørensen

DAIMI PB-87
March 1978

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55

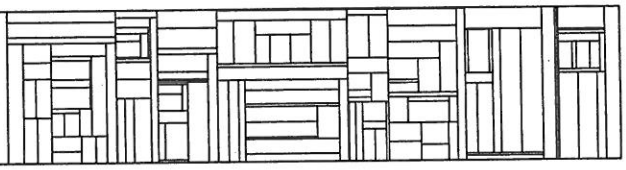


Table of Contents

Preface

This is the author's master's degree dissertation.

It is felt that a short outline of its history will prove useful to the reader.

Initially, the author chose to conduct a research project consisting of the design of a disk channel prosessor for the RIKKE experimental computer system. The actual implementation of the device was to be the subject of a different project. He subsequently realised that -short of implementing his design- there was no obvious way for him to document his design in sufficient comprehensible detail. He studied various known techniques, but found them unsatisfactory.

Additional research on his part resulted in the development of the Pictorial Representation presented hereby. Because of the newness of his notation, it itself needed extensive description. Thus the report turned out to be dominated by the notation which initially was considered as only peripheral to the documentation of the autor's disk channel prosessor design.

As sometimes happens in scientific research, what was first seen as a peripheral aspect was later recognised to be of dominating importance. Therefore the author chose to shift this dissertation's point of emphasis to the description tool, presenting the disk channel processor as a mere example in the use of the notation, where any other example could have done just as well. Thus his technical contribution in developing an original disk channel prosessor could have gone unnoticed, and needs to be pointed out in this preface.

Peter Kornerup
Mike Spier

Acknowledgements

I would like to thank my thesis supervisers Peter Kornerup and Michael J. Spier for their guidance in research, their editing, proofreading and general assistance during the preparation of this thesis.

It should also be mentioned, that this work is a consequence of Peter Kornerup's, Kurt Andersen's and Bruce Shriver's pioneering research leading to the RIKKE/MATHILDA system.

Thanks are due to Nigel Derrett for his constructive criticisms, to Svend-Erik Clausen for typing part of this paper, and to the members of the Technical Staff, who have introduced me to the world of Hardware.

Ib Holm Sørensen

1.INTRODUCTION

This dissertation presents a methodology for describing a complex system composed of processes with independent execution capabilities. The system to which the methodology is applied is a Disk Channel Processor for Aarhus University's experimental computer system, RIKKE/MATHILDA.

The present design is expected to be implemented in the future, and to be used as part of future research projects. It is impossible, indeed undesirable, at present to fix too strongly many implementation details which might have adverse effects upon future projects.

For example, a future project may be a design of a data base machine where the disk channel processor must know of data base details; or future research may consist of the measurement, evaluation and analysis of this disk channel processor, and of the optimisation of its performance, for example by moving certain services to hardware level or micro-processors.

In any case, there are elements of this design which will always be needed (e.g., the hardware double buffering device functions as the physical connection between disk drives and memory, or the coordination of read, write and seek), and others which may or may not be changed. The problem the author came up against while attempting to document the present design was how to present it with sufficient clarity so that the key concepts are specified with the greatest possible precision, while not overspecifying it in areas where technicalities would best be left to the actual implementors.

The solution chosen was to develop a System Modelling Language, and the Pictorial Representation. The former is used to describe important algorithmic components of the processor, the latter is used to describe how the various components interact with one another.

One of the first problems encountered when looking for a description tool was that by describing certain hardware-oriented pieces in terms of wiring diagrams, one practically excluded the possibility of implementing those pieces in microcode or software, because it was not clear whether the software structure presently in the author's mind would be recreatable in the future from such wiring diagrams. Similiarly, to use only a software notation would have made it unclear how to implement some parts in hardware.

Out of these considerations resulted the development of the Pictorial Representation, which lies somewhere on the borderline between wiring diagram and algorithm. Experience has shown it to be reasonably comprehensible to both programmer and hardware engineer, and experiments in translating it in either the hardware or the firm- software directions have until now proven to be relatively straightforward.

The description methodology that is being demonstrated makes use of the two notations, and has the effect that decisions concerning the implementation technology can be deferred without any essential loss of functional specificity.

1.1. Outline of Thesis.

Chapter 2 presents the language used for the description of various software, firmware and hardware mechanisms. The language - named the System Modelling Language - is a slightly extended version of the high level language BCPL ([6],[7]). Only three new concepts are added to the language definition; these provide facilities for the description of processes, mailboxes and indivisible operations.

The language is used for a detailed description of mechanisms in a system (mechanisms local to processes as well as mechanisms for process interaction). The Pictorial Representation, presented in chapter 3 , is used to describe inter-process relations in a system consisting of several independent process components.

With regard to interactions among component processes the two description tools are compatible; a description in the Pictorial Representation can be mapped into a description of the same system in the System Modelling Language and the

same description can be mapped back. This will be demonstrated through the example used to illustrate the properties and feasibility of the modelling tools. This example is a design for a Disk Channel Processor. The relevance of the Disk Channel Processor as an example is discussed in section 4.4.

Chapter 4 provides a introduction to the design of the Disk Channel Processor, which eventually will be a part of the Experimental Computer System (RIKKE/MATHILDA [2]) at the Computer Science Department of Aarhus University. The Disk Channel Processor is organised as a set of interacting processes each implementable in hardware, firmware or software.

The Disk Channel Processor design is divided into levels - Hardware, Firmware, Software.

The Hardware level - i.e. the part of the Disk Channel Processor to be implemented in hardware, is described in chapter 5 , using the Pictorial Representation and the System Modelling language (the System Modelling Language algorithms are given in Appendix A).

Chapter 6 provides a description of the microcoded part of the Disk Channel Processor. Additionally, this chapter gives a detailed programming scheme for an implementation (the algorithms for this part of the Disk Processor are given in appendix B).

Chapter 7 presents some conclusions and proposes topics for further study.

2. THE SYSTEM MODELLING LANGUAGE.

This paper uses BCPL ([6],[7]) algorithms for the description of various software, firmware and hardware functions.
The System Modelling Language is an extented version of BCPL. The extensions make it possible to describe synchronisation mechanisms for independent processes.
The language is used for descriptive purposes only, and the extensions were made informally. No claims are made concerning the formal properties of these additional constructs.
BCPL is chosen as the foundation for the System Modelling Language for the following reasons:

a) It is used as the system programming language for the experimental computer system at Aarhus University, with which the author is quite familiar.
b) It provides a wide set of control constructs.
c) It achieves simplicity and generality - by providing a single data type - the word.
d) The language does not employ facilities such as 'events','wait', 'cause' (B6700 algol [8]) or 'on condition' (PL/1). Thus defining such facilities within the present text, presents no risk of confusion.
e) It has been used for description purposes in [9] and [12].

The extensions to the definition of BCPL, necessary to describe interprocess communication, are very limited.
For description of static properties of a system consisting of independent, interacting processes, the declarations process and mailbox are introduced to describe processes and shared variables, respectively. For description of interprocess communication an indivisibility concept is introduced.
The three extensions, which are described in the following, are based on ideas in [9] and [12].

2.1. The Process Declaration.

The description in the System Modelling Language of the non-sequential properties in a system which consists of independent or concurrent activities, is provided through the process declaration.
A process declaration binds a name to an algorithm. The algorithm is a BCPL command ([7]).

example:

```
manifest { x = 1 }

static   { y = 3 }

mailbox  { CELL }


process P is
{
   let No = 0
   .
   .
   CELL := y + x
   .
}
```

Within P it is possible to reference the static, manifest and mailbox variable declared in the block where the declaration of P itself appears. (the mailbox declaration is described in the following section).
An independent process is by nature not a callable routine or function, i.e. it is not sequentially related to any statements in other algorithms. Mentioning the name of a process in other algorithms is therefore meaningless.

2.2. The Mailbox Declaration.

In order that two processes be capable of interacting, at least one shared memory component must exist and be accessible from both processes.
A named memory component which participates in the execution of more than one independent process, is called a **mailbox** .

In the system Modelling Language a mailbox is declared in a declaration list preceding the declaration of those processes which access the mailbox (2.1.).
It is obvious that the processes which share a mailbox must perform their operations upon the mailbox according to some prearranged conventions.
These (the conventions) are only described in the algorithms for the set of processes 'knowing' (through declaration) the mailbox and using it.
The minimum requirement for such conventions is, that simultaneous attempts to either read the value from, or write a value into a mailbox will cause the actual accesses to take place sequentially, one after another. All read- or write operations upon mailboxes are defined to satisfy this property ([9]).
Mailboxes are normally implemented as memory locations or communication wires used in process communication, or as state variables for the control of sharable resources.

2.3. Indivisibility.

Communication or sharing arrangements are set up either for a single named mailbox or for a set of logically connected mailboxes. For example signal, control information and data normaly go together when used in inter-process communication.
The description of the agreements according to which the processes in a computer system communicate, is independent of the implementation level (hardware, microprogram, software). Two emulated activities could share a memory location, or a hardware function could share a button on a printer device with the operator.

In order to describe these agreements the indivisibility concept is introduced.
A description of an operation or a set of operations either is encloced within indivisibility brackets ('<<' , '>>'), or is not.
Execution of operations within indivisibility brackets, behaves as if it is instantaneous, or uninterruptable.

For a single processor system the use of indivisibility brackets reflects the 'mode' of execution, which may be interruptable or uninterruptable. In such a system the bracketing can be implemented by using interrupt-disable and

interrupt-enable functions.

Example 1:

The following piece of code illustrates the behaviour of a 'wait' function using a mailbox EV as an 'event' variable.

```
.
until valof
{w
 << if EV = nothappened resultis false
    EV := nothappened >>
    resultis true
}w  loop
.   //continuation here means that event EV happened.
.
```

The value of the 'valof block' (an expression) is determined by executing the 'block' until a resultis command is executed, which causes the execution of the block to cease. The value of the expression is the value of the expression part of the resultis command. In the exemplified case, one of the resultis commands branches from an instruction enclosed within indivisibility brackets to an instruction not enclosed in indivisibility brackets.

A translation of this program into RIKKE 'machine-code' (hand compiled and optimized), illustrates the use of interrupt disable and interrupt enable instructions as a possible practical implementation of indivisibility. It is still assumed to be a single processor system.

The 'machine-code' program is given in the following:

```
t1: INTON           //t1 names an instruction not enclosed
                    //in indivisibility brackets; therefore
                    //interruptability is restored.
    INTOFF          //interrupt disable
    LM EV           //load mailbox EV onto stack
    EQ nothap       //top of stack gets true if EV=nothap
                    //else false
    JUMPt t1        //if top of stack is true, control is
                    //transferred to t1.
    LN nothap       //load the constant 'nothap'
    SM EV           //store top of stack in EV
    INTON           //restore interruptabilitity
```

Example 2:

With the System Modelling Language any complex communication can be described.

Let CON, LINE1, LINE2 be mailboxes.
Consider a conditional wait function, where a process P either waits for 'LINE1=free' or 'LINE2=free' depending on the contents of CON, whose value is changable, independently of P's execution, by another process. In other words P will wait for one of the conditions 'CON=read /\ LINE1=free' and 'CON=write /\ LINE2=free' to be satisfied. P is designed to act on which ever of the two compound conditions happens first.
Con is a local variable, which is used to remember the state of CON.

In the System Modelling Language the description would be:

```
mailbox { CON; LINE1; LINE2 }

manifest { free=1; read=1; write=2; busy=3 }

process P is
{
   let Con = 0
   .
   .
   until valof
   {cond
    <<switchon CON into
      {
      case read:  if LINE1=free do
                  {  Con := read
                     CON := busy
                     resultis true }
                  endcase
      case write: if LINE2=free do
                  {  Con := write
                     CON := busy
                     resultis true  }
                  endcase
      }>>
      resultis false
   }cond    loop
   .
```

3. THE PICTORIAL REPRESENTATION.

During the design of a system constructed from a set of independent cooperating processes it is convenient to have a pictorial tool with which these component processes and their interactions can be described.

The pictorial representation serves as a model of the system; each element of the real system which controls synchronization, sharing etc., has a pictorial counterpart in the model. If such a representation is to be of use in modelling or analyzing systems, it should be as simple as possible: it should represent only those components and connections which are essential to the understanding of the system. It should be complete. And, importantly, the notation must not add complexity or confusion to the system that is modelled.

3.1. The Properties of The Representation.

The pictorial representation should have the following properties:

3.1.1 Local mechanics should be left out.

Local variables - variables that are referenced only by a single process - do not influence the interaction between the processes, and have no effect upon the internal behaviour of any process other than the one to which they belong. Such variables, and all local mechanics related to them, should be eliminated from a description which only pictures inter-process relations.

3.1.2 All relevant mechanics should be described.

In a description of large, complex systems it is often tempting to simplify the description by eliminating some of the components or combining groups of components into single components in the

diagram. Some components are so integral to the whole systems behaviour that they cannot be removed from the description, as too many assumptions would have to be made about the abstracted pieces if the description were still to reflect the system's true behaviour. Such components have to remain in the description.

3.1.3 Independent groups of components should be isolated.

It is often impossible to model even a small system so compactly that the model clearly illustrates the total control-flow. A pictorial representation tends to grow in size. If a part of the system (e.g. a set of processes) is influenced by, or influences the remainder of the system according to simple communication arrangements, that part of the system can be isolated and temporarily removed, without elimination of relevant information. For example, in the case where a set of processes acts as a single subroutine or function, and only references resources shared with other processes during initialization and termination this technique is advantageous.

3.1.4 Strongly dependent components should be combined.

To further minimize the pictorial schema, a group of components in which the interaction between the components follows well-defined and comprehensible rules should be represented by a single construct, which contains the same information as the original separate components.

3.1.5 No redundancy should be introduced.

Single components in the real system should be represented by single components in the model. If this requirement is not met (for example, single real components are represented by a number of model components, each of which having no obvious counterpart in reality) the model will contain unnecessary information, and may become functionally misleading.

3.1.6 The Representation should be balanced.

To give a true overview of the real system, the relative complexity (or importance) of a component should be reflected by its pictorial representation. For example, if a very simple part misleadingly dominates the schema, then the representation is unbalanced.

3.2 The Elements of the Representation.

The schematic representation is a network describing processes, shared variables and the connections between the processes and the shared variables.
In a real system a shared variable can be a memory location, a communication wire, or any resource which can be accessed by more than one process. A process is an activity which may be realized by functions implemented in hardware, microprogram, software, or by direct operator action.
In the model neither the shared variables nor the processes are bound to a specific realization (i.e memory or processor).

3.2.1 Shared Variables.

A shared variable is represented by a line or a set of connected lines. As a memory location can have a specific content, a line or a set of lines can have a single value. The value of a line is determined by the processes which reference the shared variable which the line represents.

3.2.2 Processes and Connections.

A box represents a process. The process references shared variables only during activation and termination. Such processes are referred to as component processes or just as processes.
A component process - which reads a value from, or writes a value into a shared variable - is represented by a box connected to a line, which represents the shared variable.

Example:

let P1, P2, P3 be 3 independent processes; and
let M1, M2, M3 be 3 shared variables.
M1 is a shared component for P1 and P2.
M2 is shared by all processes.
M3 is shared by processes P2 and P3.

Schematic representations for this system are given in Fig. 3.1 and 3.2. The way of arranging lines and boxes, which is illustrated in Fig. 3.2 is to be preferred in more complex situations; it is used in all subsequent chapters.

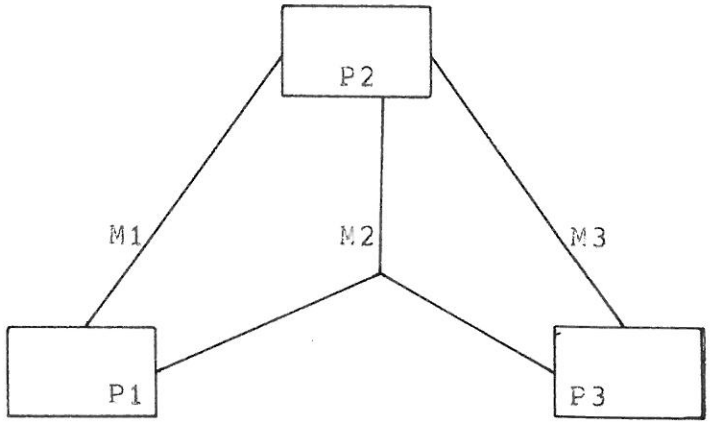


Fig. 3.1: A Schematic Representation.

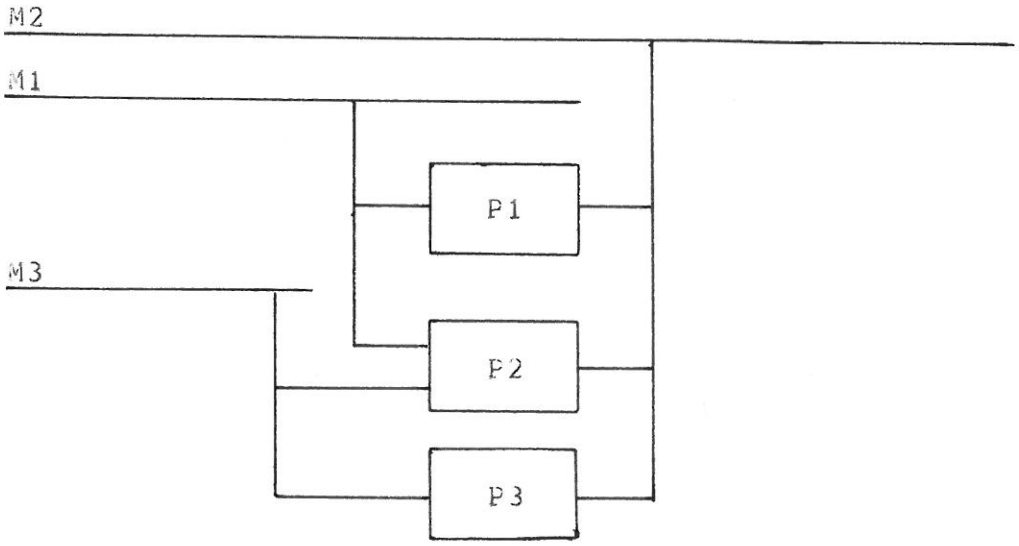


Fig. 3.2: An improved Representation.

So far the pictorial representation only contains a limited amount of information about the behaviour of the real system.
Essential properties, such as how sharing is controlled, how protection is guaranteed, and how communication protocols are arranged, have yet to be added to the description.

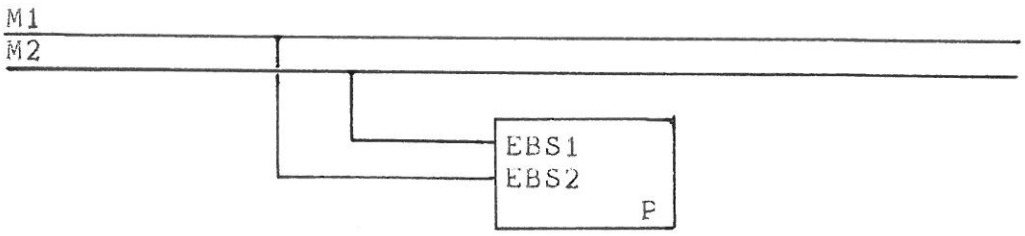
3.3 Elementary Behaviour Statements.

An important aspect of the behaviour of a system is how and when a process will inspect or influence other processes.
A process *inspects* another process by reading the value of a variable shared by the two processes.
A process *influences* another process by assigning a value to a variable shared by the two processes ([9]).
A connection point between a line and a process represents a potential for an inspection or an influence. The exact ef-

fect of such an inspection or influence is described by an Elementary Behaviour Statement (EBS). an EBS is an expression adjacent to the point where the line is connected to the box, specifying the action undertaken by the module (box), based on the value of the variable (line).

Example:

let M1 and M2 be shared variables and P be a process.



The statements (EBS1 and EBS2) specify upon which conditions process P will be enabled to execute and what changes it will make to the values of the shared variables during execution.

3.3.1 The Syntax of an EBS.

<EBS> ::= <C-list> <A> : <A>

<C-list>::= <relop><E>{,<relop><E>}* | empty

<A> ::= ⇐ <E> | empty

<E> ::= <P>{<binop><P>}*

<P> ::= <shared variable name> |
<local variable name> |
<constant name> | <integer> |
(<E>) | <unop><P>

<binop> ::= <relop> | <logop> | <arop>

<relop> ::= < | ≤ | = | ≥ | ≠

<logop> ::= ∧ | ∨ | ≡ | ≢

<arop> ::= + | -

<unop> ::= NOT | -

Alternative representations of the operators may be used.

An example of the syntax of EBS's is given in Fig.3.3.

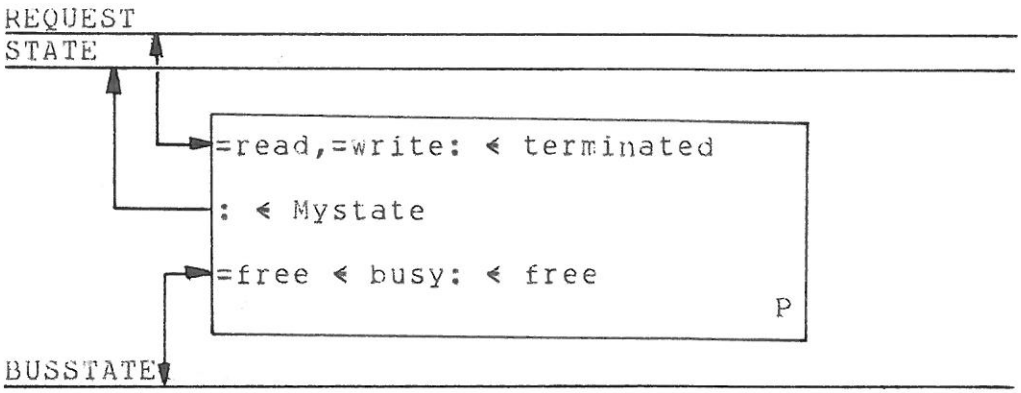


Fig 3.3: Elementary Behaviour Statements.

The following spelling conventions for names are used:
Names of shared variables are in upper case.
Names of constants are in lower case, and
names of local variables have an initial upper case letter, followed by lower case letters.

To further add information to the model the lines representing shared variables have been equipped with arrows to indicate where the contents of the shared variables may be changed.

3.3.2 The Semantics of an EBS.

Consider the following example of an EBS:

relop1 E1, relop2 E2 : ⇐ E3

which is associated with a connection point between a process P and a shared variable M. It is assumed

that it is process P's only connection point.

If the condition

(M relop1 E1) $\vee$ (M relop2 E2)

is satisfied then process P is enabled, and P will 'execute'.

(If several EBS's are associated with a process, the process will be enabled only if the conditions described within all of these EBS's are satisfied, simultaneously.)

At the point in time when a process is activated, local copies are taken of shared variables accessed by the process.

The evaluation of all logical expressions appearing in a single box, and copying of shared variables into local copies is a single indivisible operation.

The ':' represents the execution of P's local statements, using its local variables and local copies of the shared variables (see 3.1.1).

At the termination of P the assignment

M := E3

will take place.

All assignments appearing after a ':' inside a box are assumed to take place as a single indivisible operation.

If expression E3 contains the name of a shared variable, then the value of the shared variable is the value of a copy taken when the process was initiated.

If an EBS has the format,

<C-list> ⇐ E :

the entire part to the left of the colon, is executed

as an indivisible operation (test-and-modify-if-true).

If the condition is satisfied the process will execute. The value of the local copy of the associated shared variable is the (only extant) value of the shared variable before modification.

If the condition list is empty the condition is always satisfied.

If no assignment is present in an EBS no change of the shared variable will take place. But a local copy will still be taken when the process is activated.

An EBS does not specify how a process may use the value of a shared variable once execution has begun.

3.3.3 Notation rules for EBS's.

Elementary Behaviour Statements are usualy very simple expressions. Some notation conventions can be used to further simplify them.

a) If the relational operator in the condition list (C-list) is omitted the equality operator will be assumed.

b) In the schematic representation a condition list may be pictured as a column rather than a list.

These notation rules for condition lists are shown in Fig.3.4, which illustrates a system which is seman-

tically equivalent to that shown in Fig.3.3.

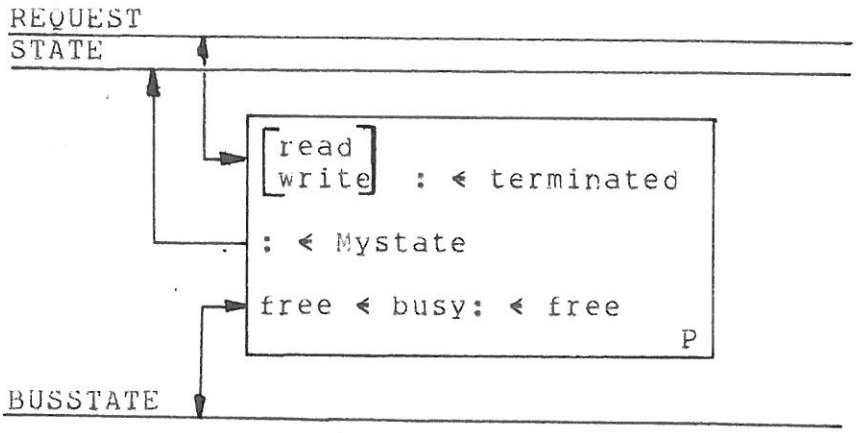


Fig 3.4: Notation Rules.

c) If the assignment part of an EBS, associated with line M, has the form,

   ⋖ M binop E

   and E does not contain the name M, then the '⋖ M' is optional - i.e the notation

   binop E.

   may be used.

d) A connection point between a box and a line may, for pictorial reasons, be on the right hand side of the box. A mirror notation

   E1 ⋗ : E2 ⋗ C

   for the associated EBS is more informative, and may be used. The interpretation of a mirrored EBS is obvious.

The notation rules for the assignment part of an EBS, and an example of a mirrored EBS, are illustrated in

Fig. 3.5 .

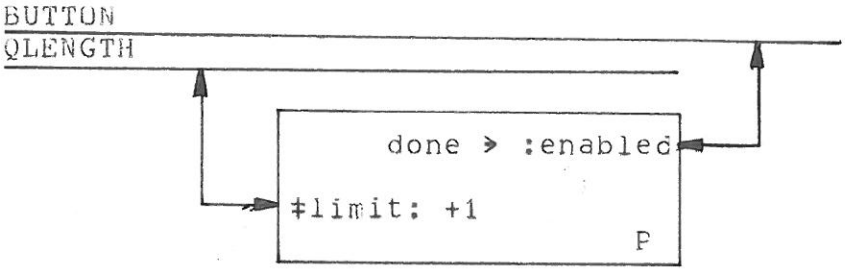


Fig 3.5: Notation Rules (continued).

3.3.4 Comparison of The Schematic Representation and the System Modelling Language.

The pictorial description of a process or a system can be directly mapped into the System Modelling Language of chapter 2.

Consider the example given in Fig. 3.4.

The Symbolic equivalent would be

```
mailbox { REQUEST ; STATE ; BUSSTATE }

manifest { read=1 ; write=2 ; terminated=3
           free=0 ; busy=1 }

process P is
{
   let Request = 0
   and Busstate = 0
   until valof
   {Pcond
   << if ( REQUEST=read \/ REQUEST=write ) /\
         BUSSTATE=free ) do
      {  Busstate := BUSSTATE
         Request := REQUEST
         BUSSTATE := busy
         resultis true  } >>
      or resultis false
   }Pcond loop

   {Paction
      let Mystate = 0
      S1
      .
      if Request=read do . .    // statements only
      .                         // involving local
      switchon Busstate into    // control flow
      .
      Sn
   << STATE := Mystate
      REQUEST := terminated
      BUSSTATE := free >>
   }Paction
} repeat
```

3.4 Sharing and Communication

In the previous section the emphasis was placed on the description of system behaviour seen from the point of view of a single independent process.

The set of EBS's inside a single box specifies how a process influences and inspects its environment.

The set of EBS's attached to a single line specifies the interaction between processes, sharing the line. These EBS's illustrate how independent processes influence each other, how sharing is controled, how communication protocols are arranged and how the protection in a system is maintained.

Six kinds of basic interactions can be described by means of an Elementary Behaviour Statement:

1) Conditional Influence (EBS ≡ <C-list>:≪<E>)

2) Influence (EBS ≡ :≪<E>)

3) Inspection (EBS ≡ :)

4) Conditional Inspection (EBS ≡ <C-list>:)

5) Singular Conditional Inspection (EBS ≡ <C-list>≪<E>:)

6) Singular Conditional Influence (EBS ≡ <C-list>≪<E>:≪<E>)

These are described in the following.

3.4.1 Conditional Influence.

If the EBS has the form,

C1 : ≪ C2

and the relation between a process P and a shared variable M is expressed by this EBS, then, if the condition M=C1 is satisfied P is capable of assigning a value to M. All other processes which share M with P may be influenced. As the influence is con-

ditional, communication protocols may be arranged.

Example:

The following is assumed:
a producer process P,
a comsumer process C, and
a control-line BUSSTATE, shared by P and C.
BUSSTATE is a state variable for a bus along which data transfer can proceed.
BUSSTATE has the values:
da ≡ data available on the bus.
sa ≡ space available on the bus - i.e previous data comsumed.

A model for this system is given in Fig. 3.6.

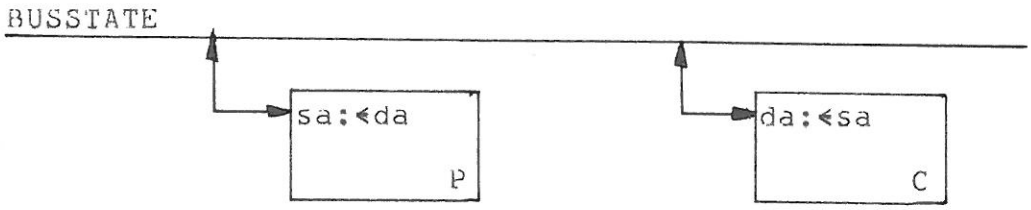


Fig. 3.6: A Producer-Consumer Model.

3.4.2 Influence.

If the connection between process P and variable M is defined by an EBS of the form,

: ⋖ C

then all processes, which are capable of reading M, are potential objects of process P's interference. As the influence is unconditional the sharing of M has to be controlled using other variables shared by the set of processes.

3.4.3 Inspection.

If an EBS (for a connection point between P and M) has the form,

:

then P is free to inspect all processes connected to M, unhindered. In addition the inspection is untraceable.

Example: (continued)

Consider the system illustrated in Fig. 3.7.
Let BUSSTATUS, BUSSTATE and BUS be lines shared by P and C.
BUSSTATE is the same as in Fig 3.6.
BUSSTATUS values:
byte ≡ data available on the bus has format byte.
word ≡ data available on the bus has format word.
error ≡ hard-error occurred, which forces the system to a halt.

The model of this system - given in Fig.3.7 illustrates influence and inspection.

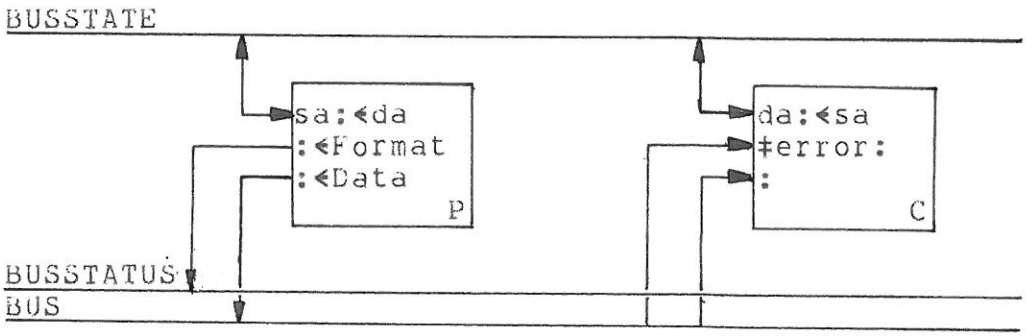


Fig. 3.7: A Producer-Consumer Model (continued).

3.4.4 Conditional Inspection.

Consider the consumer process in the previous example. The EBS '‡error' indicates that process C is able to execute only if BUS contains valid information - i.e C is able to inspect P only if the condition BUSSTATUS‡error holds - conditional inspection.

3.4.5 Singular Conditional Inspection.

If the same condition-list appears in more than one EBS associated with a single shared variable, then several processes are potentially able to execute upon the same single event.
If the event is the release of a resource shared by the processes, then the activation of a single process should prevent the remaining processes from executing. In the pictorial model this is described by means of an EBS (inside all processes) of the form,

C1 ⋖ C2 :

which indicates that the value of the associated variable is destroyed upon activation of the process.

This form provides an arbitration function to exclude all but one of a set of potentially mutually conflicting processes.

3.4.6 Singular Conditional Influence.

If the EBS has the format,

C1 ⋖ C2 : ⋖ C3

then the process can assign a value to M under condition of mutual exclution.

Example:

Consider a system with two producer processes P1 and P2,
a comsumer process C, and
a control line BUSSTATE, which may have the values:
sa ≡ space available on the bus.
da1 ≡ data produced by P1 available on the bus.
da2 ≡ data produced by P2 available on the bus.
busy ≡ the bus is temporarly unavailable.

The model for the system is given in Fig. 3.8.

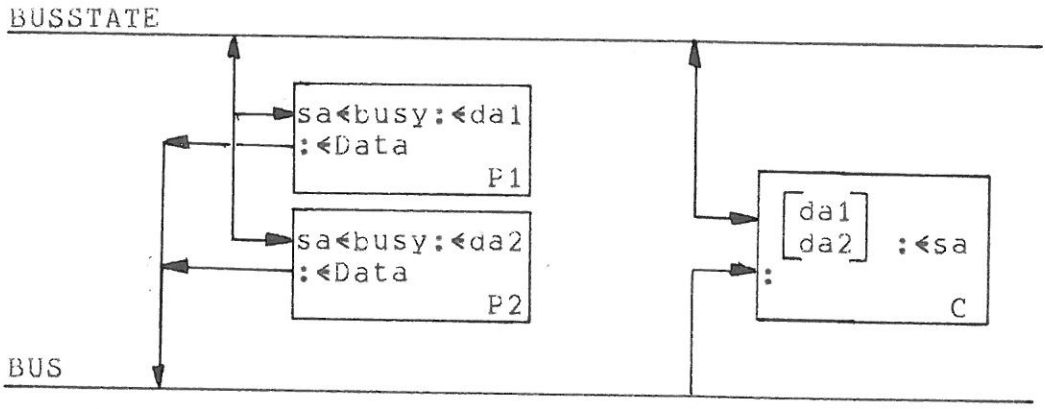


Fig. 3.8: Producer-Consumer Model (continued).

The previous examples have illustrated how special arrangements a set of EBS's associated with a single line guarantee proper sharing of resources (bus-systems, communication wires, sharable memory, buf-fers etc.).

A more complex example is as follows.

Consider an interrupt mechanism for a single proces-sor system.
An interrupt, together with the interrupt handling routine, can be regarded as a process 'stealing' a resource - the processor - from a running program.
The independent interrupt-system should support in-terrupt inhibition, according to a priority scheme.

For this the following is assumed:
INTL is a shared variable, which contains the value of the current interrupt level.
P<n> is a process which takes care of interrupts on level <n>.
P<n> monitors the signal line S<n>, which has the values:
int ≡ interrupt.
done ≡ previous interrupt handled.

Fig. 3.9 gives a model of such a system (three inter-rupt levels).
The inhibition of interrupts according to the priority rules is guaranteed by setting and restoring

the interrupt level, on process activation and process termination, respectively. The ':' represents the utilization of the processor, which executes the interrupt handler routine.
How the processor stores and restores the state of interrupted programs is not specified.

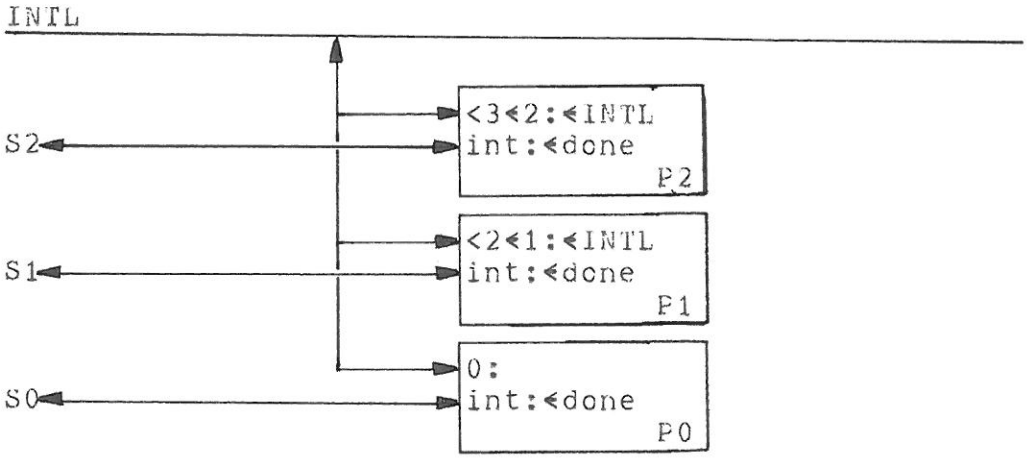


Fig. 3.9: Interrupt Mechanisme.

3.5. Parallelism and Concurency.

A potential for parallelism in a system exists if the model contains:

1) boxes which are not connected at all, or
2) boxes, one or more of which are related by means of influence or inspection only, or
3) connected boxes, where the EBS's associated with a single common line have the same condition list, or
4) two connected boxes, where the activation conditions for a process, represented by one of the boxes, will be satisfied upon activation of the process, represented by the other box.

1) and 4) will be illustrated in the following example:

Assume the following:
a) A Serial to Parallel Conversion Unit (StoP) which

receives serial formatted data, converts the data into its parallel equivalent and gates the converted data along a bus. Such a process may be time-critical, when it receives information from a synchronous device.

b) A transmitter process, which receives input from StoP and gates data along a bus to,

c) A comsumer process C.

d) Two state variables (STATE1,STATE2), one describing the state of the bus between StoP and T, the other the bus between T and C.

Fig. 3.10 gives a model of such a system. The communication arrangements illustrated in the model enable StoP to execute in parallel with C according to 1), and in parallel with T according to 4).

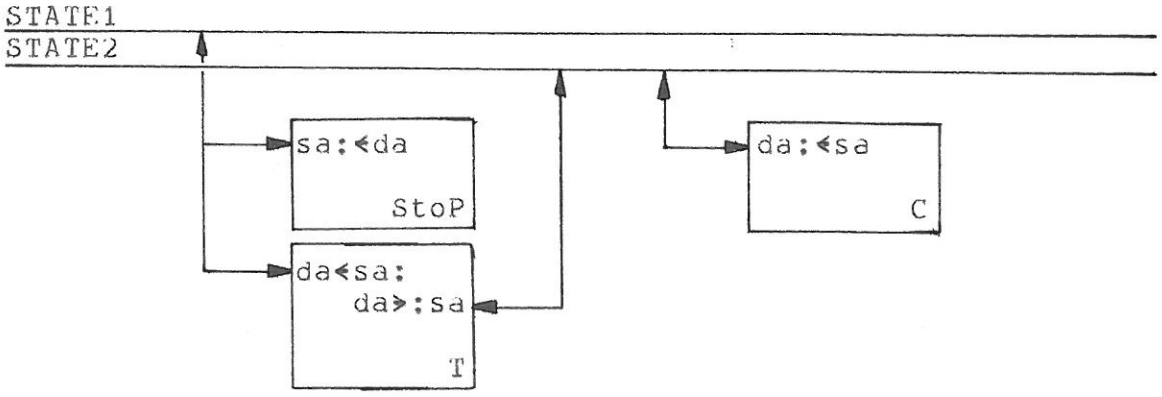


Fig. 3.10: Concurrent Processes.

3.6 Combined Constructs.

As mentioned in 3.1.4, strongly dependent components in a model should be combined into a single construct.
In a system, an example of a set of strongly dependent components is a set of interacting processes which are unable to execute in parallel, and where the mechanism which administers the flow of control between these processes is self-contained and provides a predictable flow of control (i.e. the set of processes may be influenced by its environment, but this does not disturb the logic of the internal control mechanism).

It is obvious that a single component may participate in several independent sequences of program execution (system behaviour). It is the responsibility of the designer to select the component processes to be mapped into a single construct in such a way that the model is aesthetically pleasing and comprehensible.
Once it is decided which components should be combined, the construct can be regarded as, and referred to as a single process.
The four combined constructs described in the following, form a sufficient set to describe the interaction between strongly dependent processes.

3.6.1. Sequence.

If the statements of a set of processes execute as a sequential program the combined construct (SEQ) will be used to describe the interaction between the processes.

Fig. 3.11. illustrates,
a) An example of a sequential control mechanism for 4 processes, P0,P1,P2 and P3.
b) The combined construct.
   The construct clarifies the behavior of the system. The notation 'T' indicates that the enabling condition for the process to which 'T' is associated (in-the-same-box) has to be satisfied and the process terminated, before control is transfered to the subsequent process. 'Control' flows top to bottom.

The component processes cannot be combined into a single box, as each process still is an independent component (see Fig. 3.12).

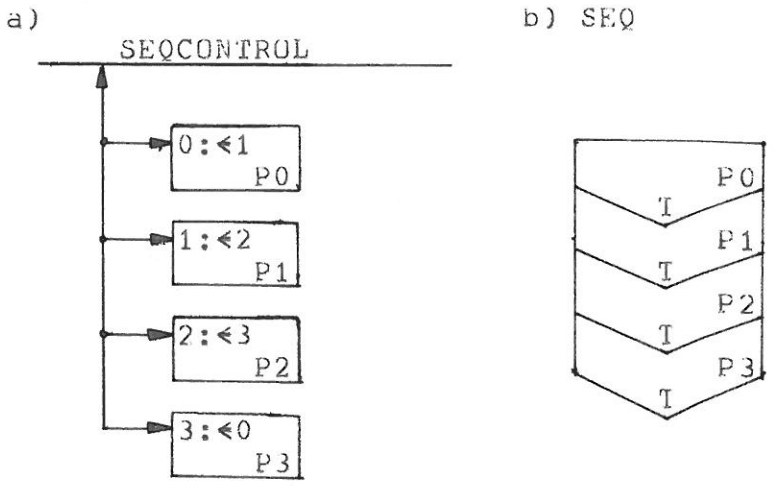


Fig 3.11: Sequence Control.

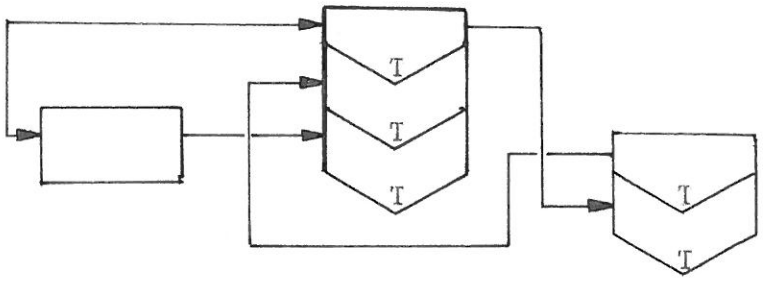


Fig. 3.12: Model utilising SEQ-constructs.

3.6.2 Priority.

Consider the system consisting of 2 producer processes (P1,P2), and 1 consumer process (C) illustrated in Fig. 3.8.
Both of the producer processes use the same resource during execution. The control mechanism described (i.e. exclusion) for interaction between the 2 producers does not provide a deterministic description of the behaviour of the system. From a given state of the system (e.g. BUSSTATE=idle) it is unpredictable which process (P1 or P2) may be activated next.

In a realization of such a system, a control mechanism which provide determinism may be used. In other words a control mechanism can be added to the producer-consumer system such that if both P1 and P2 have decided to execute and BUSSTATE=idle, then P1 will start; if P1 is not ready to excute and BUSSTATE=idle then P2 will execute (P1 has priority). If a set of strongly dependent processes are controlled by a priority mechanism the combined construct (PRI) will be used to describe the interaction between the processes.

Fig.3.13 illustrates,

a) model of a priority control mechanism for 3 processes : P0,P1 and P2. E<n> (n=0,1,2) is an enable inlet - for P<n> - with the values:
   t ≡ P<n> enabled;
   f ≡ P<n> terminated.

   The priority mechanisms are controlled through the use of the line PRICRITYCONTROL, which can have the values:
   <n> ≡ P<n> is allowed to start executing;
   b ≡ no processes are allowed to start executing.

   In this and the following examples P<n> is assumed to make its activation known to its environment using an EBS of the format 't≤f:'. P<n> could also be totally controlled by other processes (using 't:') or could make its termination known by means of an EBS in the format 't:≤f'.
   In general the enabling condition for P<n> (i.e. the 't') could be a combination of conditions associated with several inlets.
   To model such systems does not involve mechanisms of types other than those explained here.

b) The combined construct (PRI) is used to illustrate the priority-control mechanism. Component processes P0, P1 and P2 are illustrated separated by horizontal lines. Processes are ordered top to bottom with decreasing priority.
   At any time only one process can be active. If there is conflict between two or more processes concurrently able to activate, the higest priority

process is chosen.

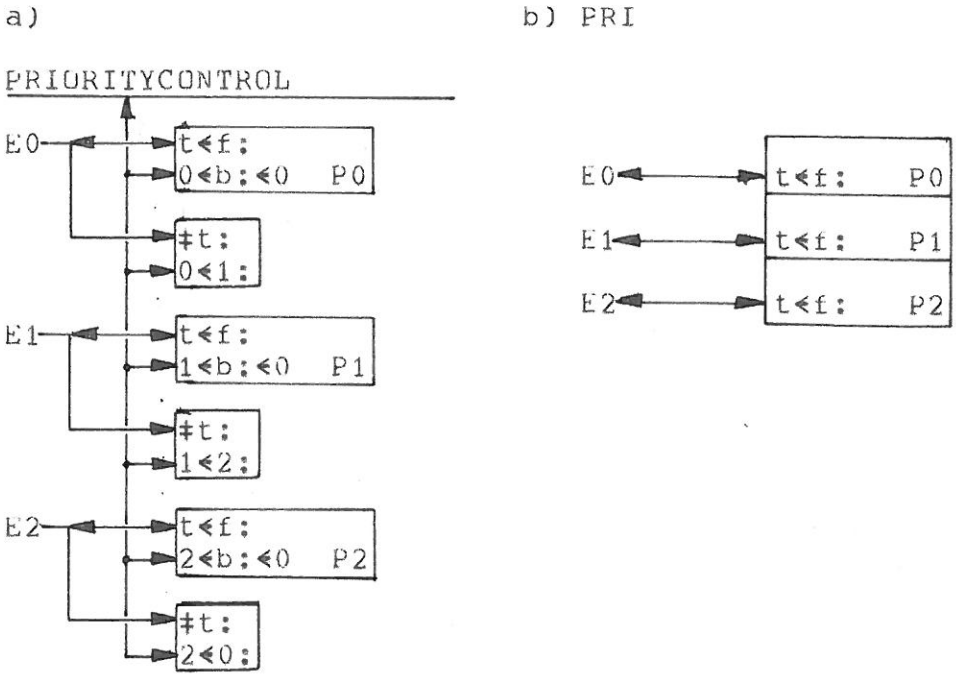


Fig. 3.13: Priority Control.

If a priority control mechanism is used in a two-producer, one-consumer system (analogous to the one illustrated in Fig. 3.8) then the model can utilize a combined construct. (see Fig. 3.14).
Note, in comparing Fig 3.8 and Fig.3.14 that the same interference problem was solved, in the former through a Singular Conditional Influence, and in the latter by use of priority.

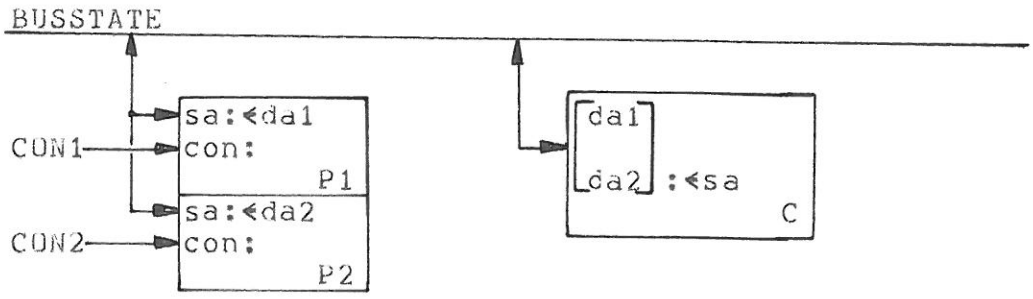


Fig. 3.14: Producer-Consumer Model.

Both processes make its termination known to process C.
Variables CON1 and CON2 have been added, in order to give the system a meaningful behaviour. Only in the case where the values of both CON1 and CON2 are simultaneously equal to 'con' will the priority scheme be used.

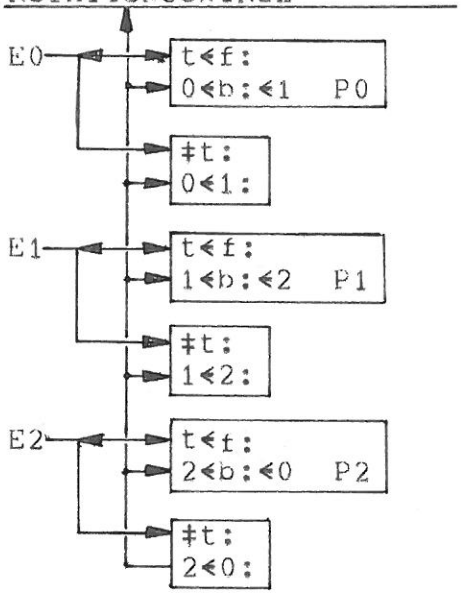
An example of a PRI construct described in terms of the system modelling language is given in Appendix A.1.

3.6.3 Rotation.

Due to properties of processes it may not be appropriate to use a priority scheduling mechanism, where a process may suppress the execution of other processes with lower priorities.
A control mechanism called Rotation Control may to be a better alternative for the control of a specific system. (i.e. round robin mechanism).

Fig. 3.15 illustrates,
a) A model of a system consisting of 3 processes, where none of the processes are capable of suppressing other processes.
b) The combined construct (ROT), used to describe such a behaviour.

a)



b) ROT

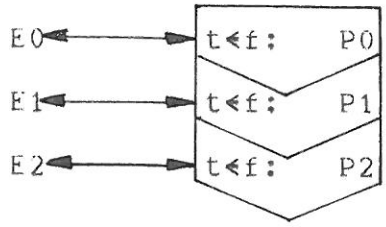


Fig. 3.15 Rotation Control.

The rotation control mechanism contains a priority scheme - the involved processes take turn in having priority. Futhermore it contains a rotation scheme.

3.6.4 Suppression.

If any process in a system consisting of a set of strongly dependent processes is capable of suppressing all other processes, the control mechanism is named suppression (SUP).

Fig 3.16 illustrates,

a) a model of a system of 3 proceses, where any process P<n> suppresses other processes, as long its enabling condition 'E<n>=t' remains satisfied.
b) the combined construct for a suppression control mechanism. The 'F' inside a box - which represents P<n> - indicates that transfer of control to P<n+1> is delayed as long as the enabling condition for P<n> remains satisfied.

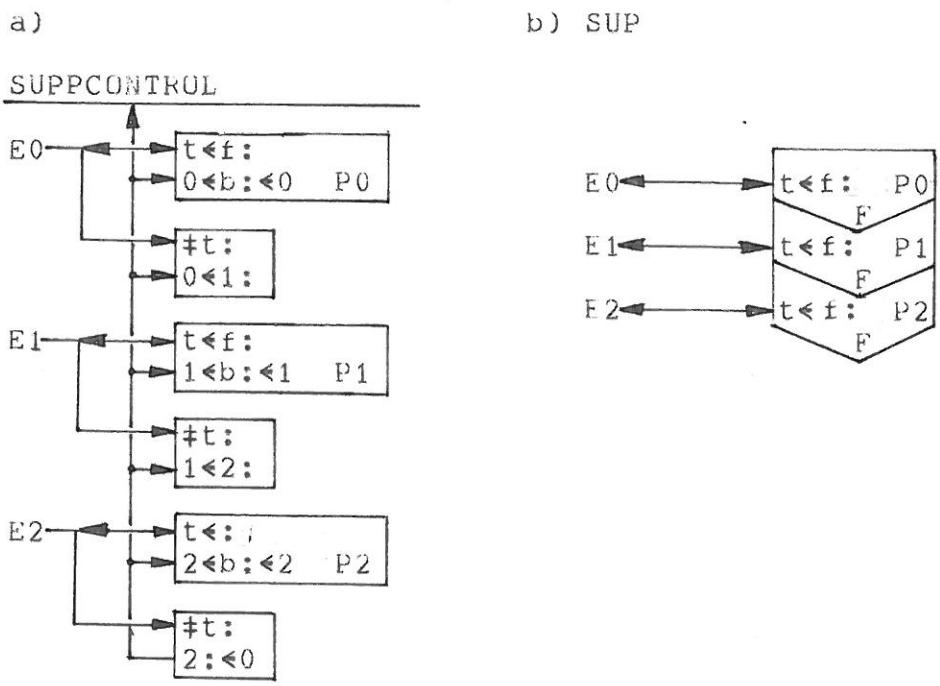


Fig. 3.16: Suppression Control.

The suppression control contains a priority scheme - any executing process will upon termination still have priority.
An example of a SUP construct described in terms of the system modelling language is given in Appendix A.2.

Example:

To illustrate the use of combined constructs in a concurent system consider an asynchronous pipeline process.

Such a process can be regarded as a set of minor processes, each performing the operations required on a specific stage on the pipeline.
All these processes should be able to proceed indepently and should only require access to sharable resources upon activation and termination. Upon activation, a process performing operations on stage n gets input from a line whose value is produced by the process performing operations on stage n-1.

Assume that the pipeline consists of,
a) 3 independent processes: P0,P1 and P2.
b) 3 lines: L0,L1 and L2 (tnese may be registers, memory locations or communication wires), where L<n> contains output produced by P<n-1>. P<n> takes input from L<n>.
c) 3 state variables: S0,S1 and S2, where S<n> specifies the current state of line L<n>, being
   da ≡ data available.
   sa ≡ space available.

Fig. 3.17 gives a model for such a system.

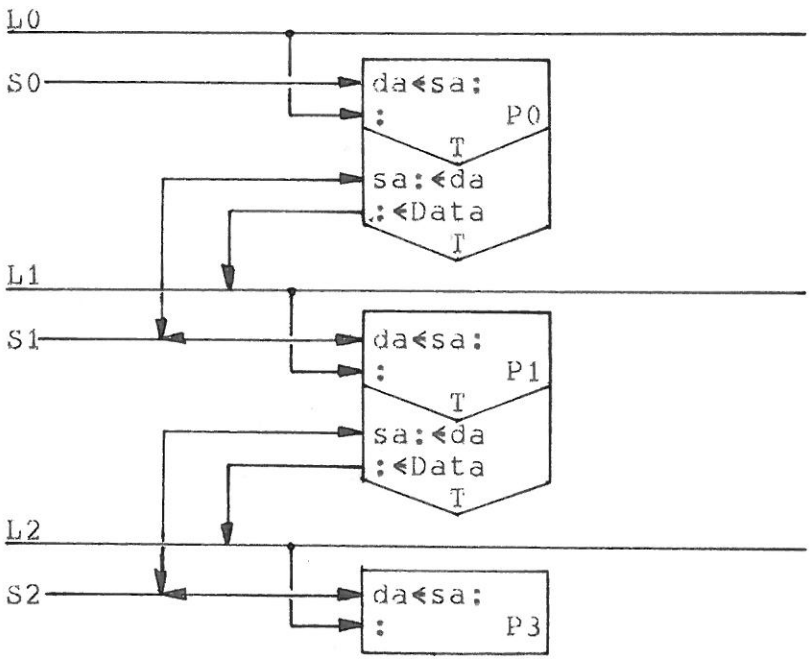


Fig. 3.17: A Pipeline System.

Each process in the pipeline consists of two component processes, organized in a SEQ construct. The first component starts processing data as soon as input is available. The second component process gates the result to a communication line as soon as the line is free.
According to the rules mentioned in section 3.5 all processes represented in this model by single constructs, are able to execute in parallel.

3.7 Remarks on Description Tools.

Other pictorial tools for describing and analysing system behaviour, such as flowcharts or Petri Nets [10], are available. Therefore the question could be raised why it should be necessary to develop yet another representation. The justification is that flowcharts are not adequate for modelling non sequential systems, and that Petri Nets are syntactically unacceptable in that they violate the requirements described in section 3.1.
Given that Petri Nets have been used for modelling of systems, the following remarks may be of interest:

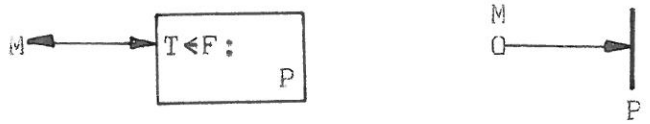
1) In terms of the semantics defined in this chapter, the only primitive operation allowed by Petri Nets is of The Singular Conditional Inspection EBS form.
The examples in this chapter have justified the existence of other EBS forms. Their lack in Petri Nets forces a deformation of the description: the designer is forced to add components to his model which have no counterpart in the system being modelled.

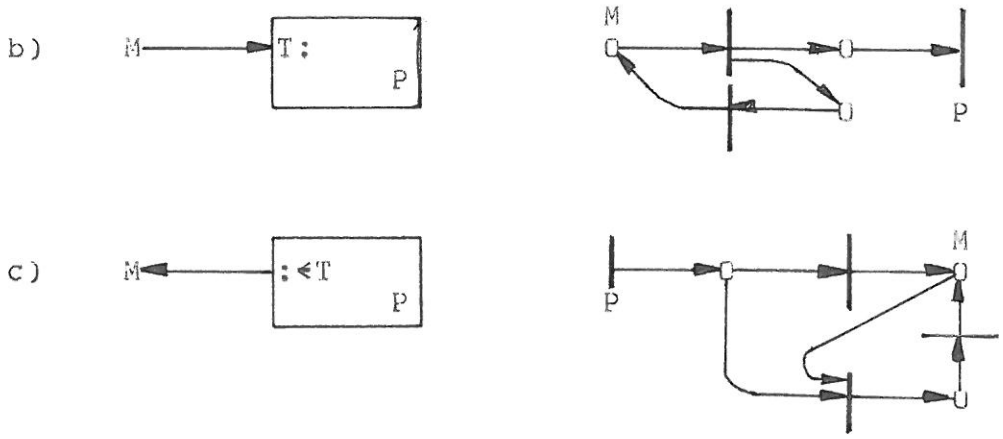
Example:

Assume
a) a shared variable M, which is represented by a 'place' in a Petri Net, and by a line in the Pictorial Representation,
b) a process P, which is represented by a 'bar' in Petri Nets, and by a box in the Pictorial Representation.
M has the values:
T ≡ true (in Petri Nets M is 'marked')
F ≡ false (in Petri Nets M is 'unmarked')

The following examples give 3 simple constructs, modelled using both the Pictorial Representation and Petri Nets.

a)





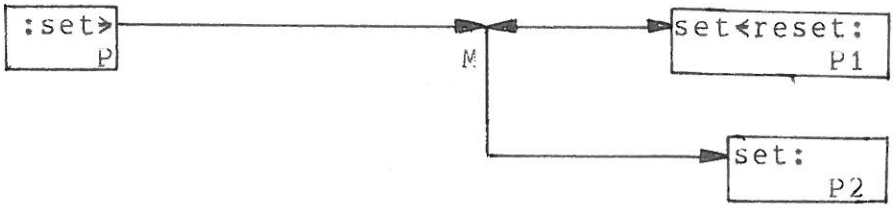
The examples illustrate how additional components (places,bars) have to be invented for the Petri Net model, leading to unnecessary syntactic 'noise'. In other words Petri Net models violate 3.1.5.
It is interesting to notice that the simplest EBS form is mapped into the most complex Petri Net construct, and vice versa.

2) The need for additional 'primitive components' is more than just for reasons of eliminating syntactic 'noise'.

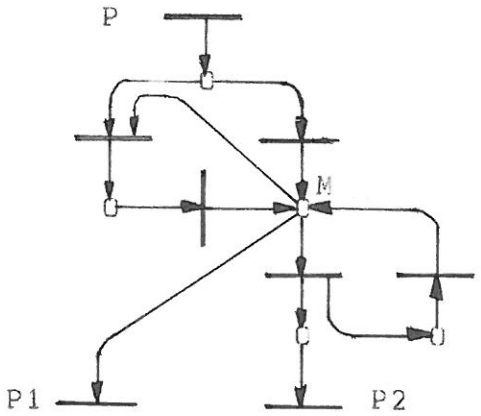
Consider a process which resets the state of a subsystem. In the Pictorial Representation this can be expressed as a pure influence. In Petri Nets it is certainly unclear how such a concept can be modelled, as a knowledge of the state of the subsystem has to be built into the model.

A simple example: Consider a process P, which upon a certain event (e.g. a clock) arbitrarily attemps to influence (EBS = '∈ set') two other processes P1 and P2 by writing to a line M. The influenced processes independently decide whether or not to actually accept the signal and act on it.

The model in The Pictorial Representation might look like:



and the model in Petri Net might be



which certainly is unclear. The concatenation of two simple and well defined subsystems, forces the designer to invent a complex 'interface', just to describe a clean communication.
As a consequence of this, Petri Net modelling does not always to comply with 3.1.3: it is not possible to isolate and temporarily remove components from the model.

3) The Petri Nets model does not provide any meaningful combined constructs such as those described in section 3.6. This forces Petri Nets Models for all but the most trivial systems to become of unmanageable size and utterly confusing.

4. AN ANALYSIS OF A DISK-CONTROLLER.

The usefulness of the new Pictorial Representation, presented in the previous chapter, is justified on the one hand by the inadequacy of existing tools for modelling concurrent systems and on the other by its demonstrated application to the description of a particular system. This system (chapter 4.- 5.) is rather extensive consisting of many interconnected independent processes. Obviously other systems have to be considered - and presumably further development of the Pictorial Representation is necessary. This is discussed in chapter 7.

The reason for developing the new pictorial representation for concurrent systems, was the need for a modelling tool, which could be used to describe, analyse and design a Disk Channel Processor for the Experimental Computer System ([2]) at Aarhus University.

4.1. The Configuration.

The Disk Channel Processor (referred to as the Disk-Controller) is realized as a set of interacting processes, each implemented by functions in hardware, microprogram or software.
To fully understand the analysis and the design of the disk controller a short introduction to the Experimental Computer System (RIKKE/MATHILDA) may be necessary.
The relevant system components are presented in the following sections.

4.1.1. The Physical Processor.

RIKKE - is a microprogrammable processor, with 4K of (64 bit) Control store.
The bus width is 16 bits.
The main data path architecture is a 16 bit 'version'

of that of the MATHILDA processor ([1],[2]). Instruction sequencing and control facilities for microprograms are equivalent to those of MATHILDA.
RIKKE is equipped with an internal I/O Data Bus, by means of which devices can be multiplexed under micoprogram control.
The microcoded part of the Computer System is referred to as the firmware level . The part of the Disk Controller, implemented on firmware level, is referred to as The Supporting Firmware , and is discussed in detail in chapter 6.

4.1.2. The Memory.

WIDESTORE - is a sharable memory, which can be accessed either as a 32K 64 bit wide memory, or as a 128K 16 bit wide memory.
WIDESTORE is equipped with a memory address- and data port multiplexer allowing for up to four physical processors to access the memory independently.
In addition, WIDESTORE provides Automatic Address Incrementing, which makes it possible for unintelligent devices to utilize direct memory access ([11],[13]).
WIDESTORE is considered to be the Main Memory of the Computer System ([5]), and will be referred to as such.

4.1.3. Virtual Processors.

The OCODE stack-machine (described in [3]) and,
The I/O Nucleus ([4]) are virtual processors realized by means of microcoded functions (i.e. firmware level).
The set of operations in the Computer System, which is executed by means of interpretation performed by virtual processors is referred to as The Software Level .
At the software level, lives the operating system.

4.1.4. The Operating System.

The RIKKE BCPL System ([5]) is an interactive single-user operating system, which runs on the OCODE

machine and uses the I/O-Nucleus to perform I/O.

The comminucation between the software level and the firmware level is administered by a microcoded Communication Module, which:
1) monitors a software level register, which contains eventual calls for firmware level functions;
2) transfers control to firmware level;
3) forwards results and completion status of firmware level functions to the calling software level routine, using micro-coded interrupt facilities.

4.1.5. The Supporting Hardware.

A hardwired logical unit (in the following referred to as The Supporting Hardware), the automatic address incrementing facilities of WIDESTORE and the disk-hardware, together form an unintelligent data channel between disk system and WIDESTORE.
The functions implemented in hardware will in the following be referred to as The Hardware Level .
The communication between the firmware level and the Supporting Hardware is established by means of 2 output data bus ports (32 bit), used by micro-coded functions for controlling the Supporting Hardware logic, and 1 input data bus port (16 bit) for status feed-back.
The Supporting Hardware is discussed in detail in chapter 5.

4.1.6. The Disk Hardware.

The disk system consist of 2 movable head disk units, which are made up of 2 recording disks each ([14]).
Each disk has two surfaces divided into twelwe sectors.
Each surface has 407 tracks (of which 400 are used).
Each track-sector contains one single recording block.
The Disk Hardware signals the supporting hardware when a sector mark or index mark is encountered.
The Disk Hardware acts on signals sent from The Supporting Hardware to,
1) record serially formatted data, sent from the sup-

porting hardware, on a specific disk-surface;
2) transmit serially formatted data, or
3) move the access arms to a specific cylinder.

In the model of the disk controller presented in this paper (Chapter 5. and 6.) it is assumed that only 1 disk unit is connected to the system.

The configuration of the Computer System is illustrated in Fig. 4.1..

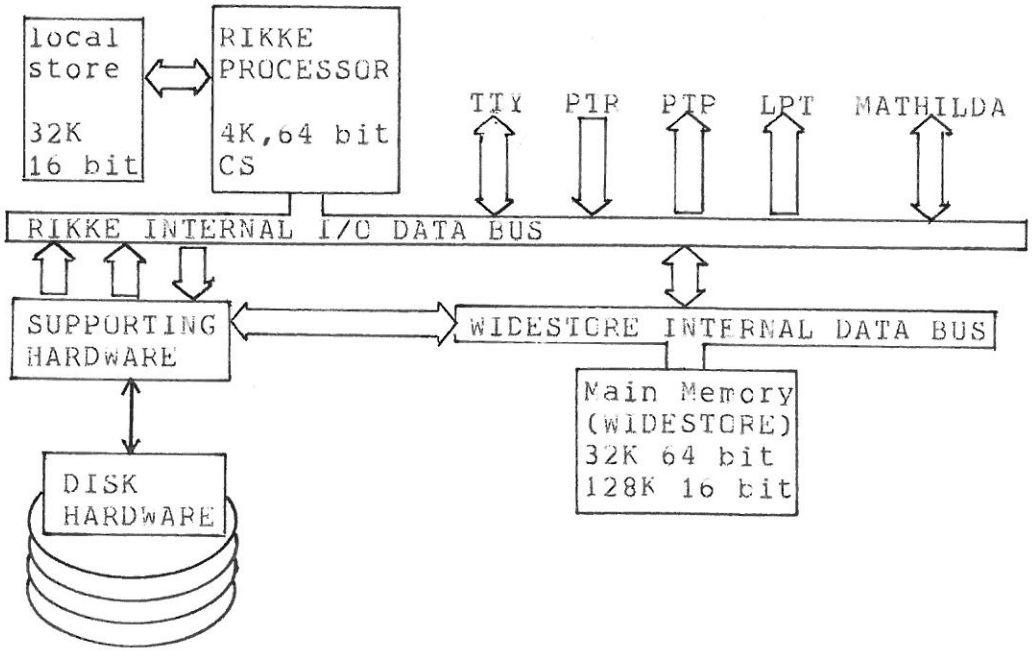


Fig.4.1: The Computer System.

The components TTY,PTR,PTP,LPT, local store and the MATHILDA microprogrammable processor are irrelevant to the modelling of the Disk Controller, and will be ignored in the following.

4.2. Disk Characteristics.

As mentioned in section 4.1.6 each track-sector contains only one recording block.
This track-sector organisation is actually established through combined functions in disk hardware and supporting hardware.
The recording block is not further divided into subblocks. Information normaly recorded in a header subblock such as physical track/sector identification, logical identification (fileid, pageid) is in this system stored in connexion with data. The processing of header information, which is performed by functions implemented in soft - and firmware, does not influence process synchronization mechanisms, and will therefore not be discussed in detail in this paper.
The logical sector numbering (i.e. the numbering used by software level programs) is equivalent to the physical sector numbering. This organisation enables the disk controller - if it is designed properly - to transfer physically consecutive track-sector blocks between disks and Main Memory, and such requests can actually be specified from software level programs.

4.3. The Levels of the Disk Controller.

As mentioned in section 4.1 the disk controller is realized by functions implemented in soft- firm- or hardware.
The description in the following will fix the border-lines between the three levels. However these fixed lines are only assumed for description purposes. They can gradually move upwards or downwards during development and implementation of the disk controller. This flexibility exists because of the architecture and configuration of the experimental Computer System (Fig.4.1.).
As an example it can be mentioned that all memory-side transfers (i.e. between Supporting Hardware and Main Memory), which in the following are modelled as if implemented as hardware functions, will actually, in the first 'version' of the disk controller, be 'simulated' in firmware. This simulation is done by using the connexions between the Supporting Hardware and the internal data bus of WIDESTORE on one hand, and the internal I/O data bus of RIKKE on the other.

4.3.1. The Software Level of the Disk Controller.

Five types of requests concerning data transfers can be submitted by application programs (OS filemanager, DB manager, user programs etc.) to the software part of the Disk Controller.

a) READ RANDOM.
Transfer a single sector, which constitutes a single block of a randomly structured file, from Disk to Main Memory.
All necessary control information (sectorno., memory address, cylinder, etc.) is stored in a request vector.

b) WRITE RANDOM.
Transfer a single block (sector) of a randomly organized file, from Main Memory to Disk.

c) READ CONSEC.
Transfer several contiguous sectors belonging to the same file from Disk to Main Memory. This request is used for multi-files, where each subfile is a consecutively structured file (i.e. data blocks occupy contiguous sectors on secondary storage). This allow for logical page sizes, which is a multiple of physical sector size.

d) WRITE CONSEC.
Transfer several blocks, which occupy contiguous memory positions, to contiguous sectors on disk surface.

e) FORMAT TRACK.
A complete track will be overwritten and each sector initialized to be considered as 'not-occupied'.
This request is not discussed in the succeding chapters describing firmware level and hardware level functions. The main logic of the request will be implemented as firmware level functions, which obviously occupy the disk resources in the disk controller for a complete revolution of the disks.

Disk Controller support for access methods for file structures, other than those supported by operations a)-d) (i.e. random file and consecutive multi-files), should be considered, and integrated in the disk controller. For example chained and indexed structured files. The present design does not provide for these more sophisticated services, but was made in anticipation of the need; adding these at this level should be a straightforward operation.

The software level part of the disk controller will 'unblock' CONSEC requests into simple requests, each for a single sector and store these new requests in a queue together with the RANDOM requests.
The organization of the Queue (Request Queue) is discussed in section 6.1.1.

4.3.2. The Firmware level.

This level will be described in detail in Chapter 6.
The firmware level functions process the Request Queue.
If the next request is for the current track, then a firmware readsector- or writesector function will be called; if not, a seek function will be called first.

The firmware level Read Sector function signals 2 supporting hardware functions: one to transfer disk-information to a local channel memory buffer; another to transfer data in a local channel memory buffer to Main Memory.
The firmware level Write sector function signals two hardware level functions: one to transfer a Main Memory buffer to a local channel buffer; another to transfer data in a local channel buffer to the disks.

4.3.3. The Hardware Level.

The Hardware Level functions of the disk controller are discussed in detail in Chapter 5.
As mentioned in the previous section, four transfer functions are implemented in supporting hardware. Two

of these control the disk-hardware and two others control the functions of WIDESTORE.

A complete picture of the levels in the Disk-Controller is given in Fig.4.2..

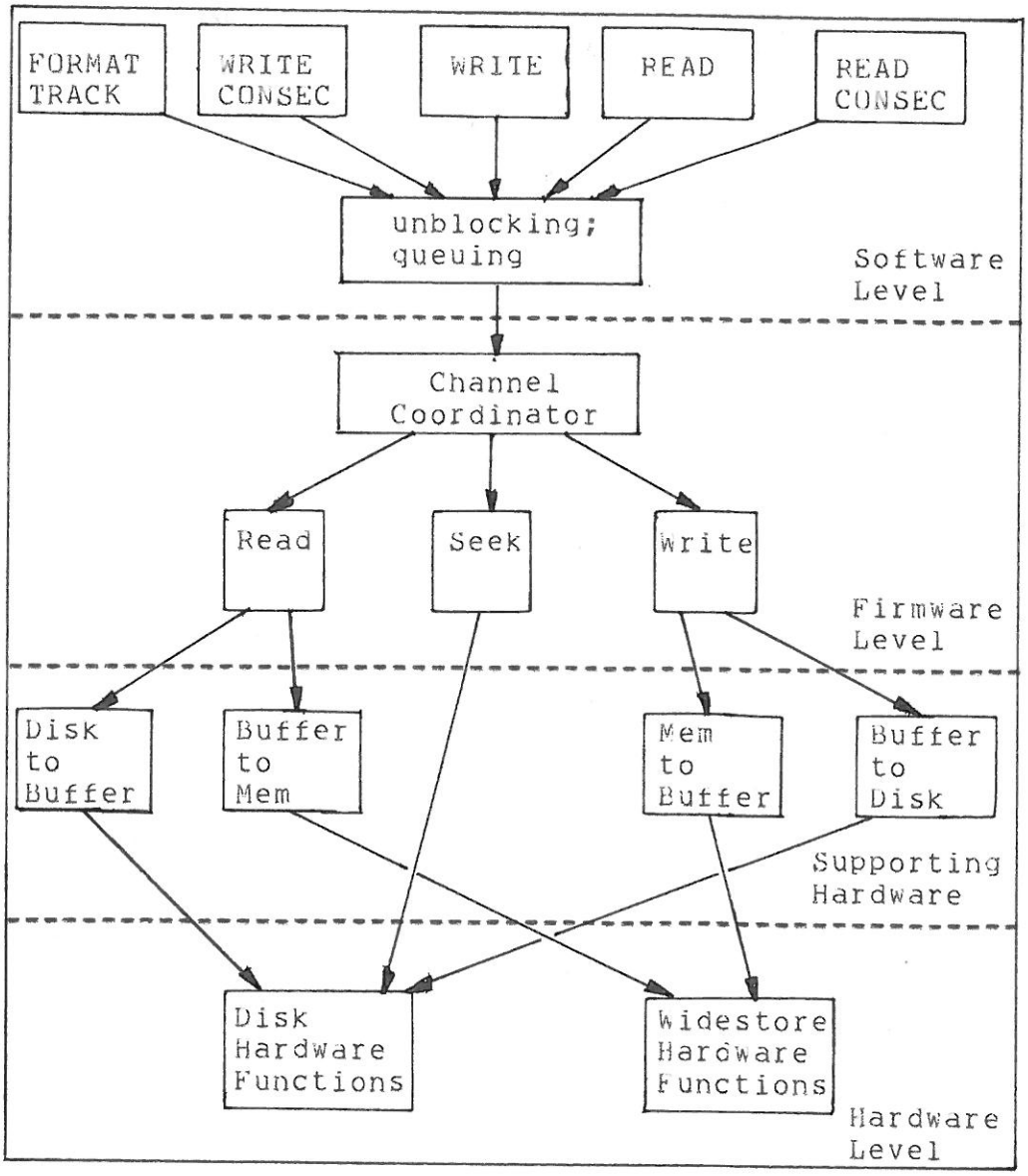


Fig.4.2: Disk-Controller (overview).

The model is incomplete. Its virtue lies in that it 'slices' the Disk Controller design into levels, making smaller and more managable parts, thus providing a good functional overview.
In the next two chapters the Supporting Hardware and the Supporting Firmware are shown in the true detail.

4.4. Relevance.

The idea of implementing device controllers or channel processors as a mix of software, firmware and hardware functions is certainly not new. The strategy has been used extensively both for experimental device controllers and for production machines - among others IBM System 370 ([15],[16]).

The design shown is original, although perhaps not entirely novel. It represent an acceptable solution to a set of peculiar problems:

a) RIKKE's need for a direct access mass storage device.
b) Aarhus University's need for a flexible storage system (attached to RIKKE) that is conducive to future development, research and experimentation (this is a reason for precluding an all-hardware device).

The author chose to 'kill two birds with one stone', and to develop the design needed for illustrative purposes (rather than use an existing one), in order to help further the RIKKE development process.

5. THE HARDWARE LEVEL.

The Disk Controller is designed to provide various degrees of service sophistication (i.e. it is not a fixed-purpose device) as well as to allow for future funtional evolution (being an experimental device). The main logic of the device is implemented in a combination of firmware and software, to gain this flexibility.
However, there are some specialized tasks which are more economically performed by supporting hardware.
In addition, other tasks have to be performed by hardware functions, because of timing requirements.

5.1. The Elements of the supporting Hardware.

The supporting hardware can logically be divided into the following groups.

5.1.1 Two buffers of 256 16-bit words each.

The buffers are connected to the main data bus system of the supporting hardware. The addressing latches for selecting a specific word in these local memory buffers will be incremented automatically during a block-read or block-write process. The two buffers will be refered to as Buf A and Buf B.

5.1.2 Input Data Bus System.

The input data bus system has 2 possible input sources, and two possible output destinations. The latter are local buffers, accesible through use of direct memory access (DMA). A 16-bit data word can be gated from any one source to one of the destinations.

5.1.3 Output Data Bus System.

Data words from one of the two local buffers can be gated onto the output data bus. Depending on the state of the control signals the information on the bus will be transfered to one of the destinations.

5.1.4. Memory Bus Control Logic.

A unit, realized as a hardwired program, which controls the flow of data from a local buffer to an output destination, and from an input source to any of the two local buffers.

NOTE 1:

So far the following objects for modelling have been identified inside the supporting hardware:

a) A Memory Bus Control Process (MBCP), which administers the usage of two local memory buffers, and two local bus systems.
b) Two input sources shared between the Memory Bus Control Process and two unspecified producer processes.
c) Two output destinations, which are shared between the Memory Bus Control Process and two consumer processes.

The Pictorial Representation for this system is shown in Fig.5.1. (it is intentionally incomplete, lacking control logic).

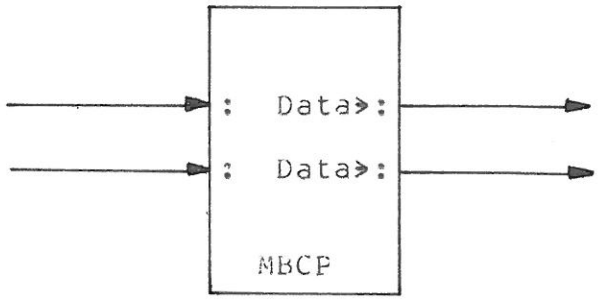


Fig. 5.1 The Memory Bus control process.

The local units administered by the Memory Bus Control Process are illustrated in Fig. 5.2., which shows how data may flow through the MBCP.

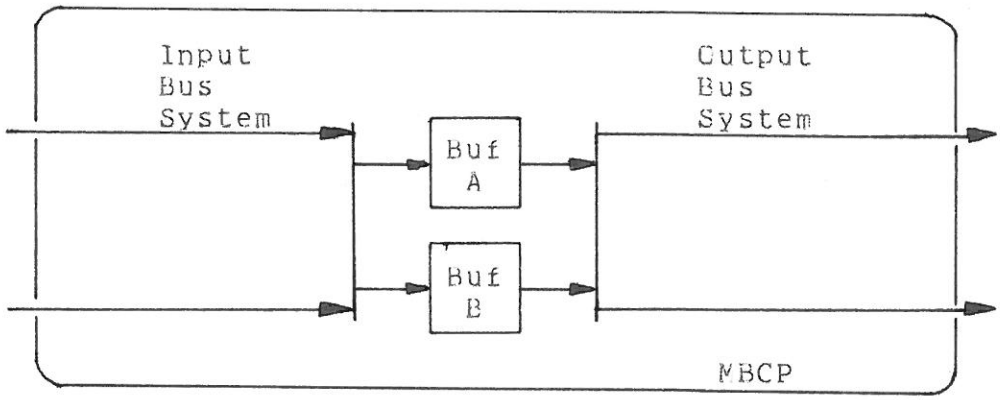


Fig. 5.2: Internal bus structures for MBCP.

5.1.5 Parallel to Serial Conversion, and Cyclic Redundancy Code Generation units.

During a disk-write operation the information in one of the local buffers is transfered using the output data bus system and DMA cycles to a parallel to serial conversion unit. Serial data is sent from the unit to a specific disk and as data words are being transfered a cyclic redundancy code (CRC) is generated and appended to the data stream to be recorded on the disk.

5.1.6 Serial to Parallel conversion and CRC Check Units.

During a disk to buffer transfer the bit stream from the disk is packed into 16 bit words and gated onto the input bus. A cyclic redundancy code is compiled and compared against the CRC check-word that was read in after the data-block. A status flag indicates the result of the comparison.

5.1.7. Input Port.

A 16-bit-wide input port connects the supporting hardware with the Main Memory of the computer system.

5.1.8 Output Port.

Similar to the input port a 16-bit-wide parallel output port connects the supporting hardware to Main Memory.

5.1.9 Parallel Input and Output Port Logic.

The supporting hardware is directly connected to the Main Memory of the computer system by means of an input, output and address port. The Main Memory is equipped with a data- and address port multiplexer ([11],[13]), which enables the supporting hardware to be in charge of block transfers between one of its own memory buffers and a buffer in the Main Memory of the computer system.
The Parallel Input Logic and the Parallel Output Logic control block transfers, and communicate with the multiplexer of the Main Memory.

NOTE 2:

The system modelled in Fig.5.1 can now be extended with the following independent processes:

a) Serial to Parallel Conversion Process (StoP), sending data to the Memory Bus Control Process, and receiving serial data from the disk hardware.

b) Parallel to Serial Conversion Process (PtoS), receiving data from the Memory Bus Control Process, and producing serial formatted data to the disk hardware.

c) Parallel Input Port Process (IP), sending data to the Memory Bus Control Process, and receiving data from Main Memory.

d) Parallel Output Port Process (OP), receiving data from the Memory Bus Control Process, and sending data to Main Memory.

The Pictorial Representation of these independent processes and the Main Memory Bus Control Process is given in Fig.5.3. (it is still incomplete). The figure also indicates how the supporting hardware communicates with its environment. Data flow and control flow not analysed on this level of description are pictured through use of dotted lines and heavy lines, respectively.

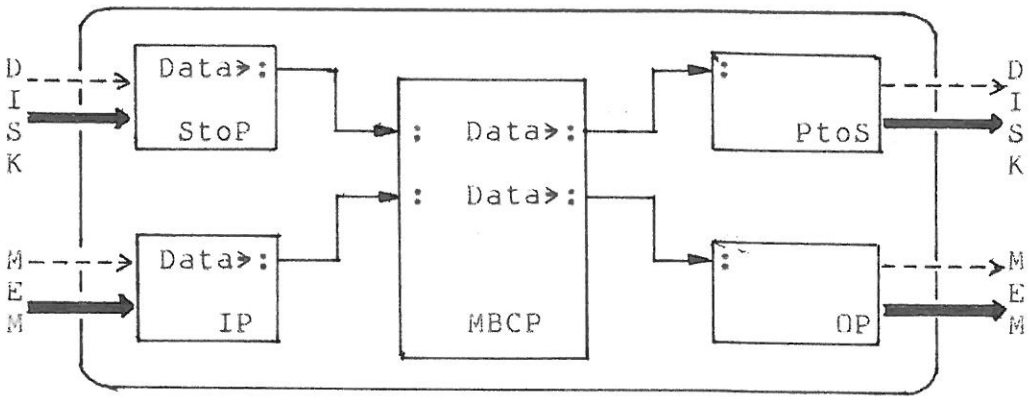


Fig. 5.3: The Memory Control and The I/O Units.

5.1.10 Sector Counter and Sector Match Units.

For each independent disk-unit administered by the supporting hardware there exists a sector address counter which is set appropriately when index or sector pulses, sent by the physical disk-hardware, are captured by the system. When requests for disk transfer are initiated, the actual disk-to-buffer channel process will be delayed until a match between the requested sector address and the internal sector counter for the specified disk-unit exists.
Upon the match all resources required for transfer of data between a local buffer and disk are reset by this unit. The disk I/O logic is signalled to start.

5.1.11. Parallel Port Control Unit.

When requests for memory to buffer or for buffer to memory transfers are initiated, all resources required to transfer data between Main Memory and local

buffer are reset by this unit, which signals the I/O processing units to start.

5.1.12 Direct Memory Access (DMA) Facilities.

A DMA facility is provided to effect whole block-transfers between the local buffers on one hand and either disk or Main Memory on the other. The decision was made to equip the supporting hardware logic with DMA facilities to relieve the disk-channel-controller of the time-consuming and monotonous job of incrementing address-pointers and monitoring I/O flags.
When a block transfer has been initiated, a status flag reflects the busy situation; It prevents the disk-channel-controller (if it is designed properly) from interfering with the 'uncompleted' channel-process. The data transfer process will terminate when an internal word-counter reaches its limit (256), i.e. when the local buffer is full. Also, the transfer will be forced to a completion if any new process that conflicts with the current process is initiated.

NOTE 3:

The system modelled in Fig.5.3 can now be extended with:

a) Disk I/O Request Process (Disk Admin), which receives control information from the Main Computer and from the disk hardware. This process signals on one hand either PtoS or StoP, and on the other hand the disk hardware, to start processing

b) Memory I/O Request Process (Mem Admin), which also receives control information from the computer system. It signals the parallel I/O processes as well as the Main Memory multiplexer to start a communication.

In these pictorial representations there is no box representing the DMA. It (i.e. the DMA) is an effect achieved through the cooperative behaviour of MBCP and the I/O processes, thus the logic needed to im-

plement it is distributed among these components; to show it in detail would destroy the modularity at the present level of description.

Fig.5.4 gives an overview of the channel-control system, incrementally developed through the previous sections. The model does not illustrate all communication signals necessary to control the system, but gives an overview of how the processes, realized in the supporting hardware, are connected and how they independently communicate with the environment.

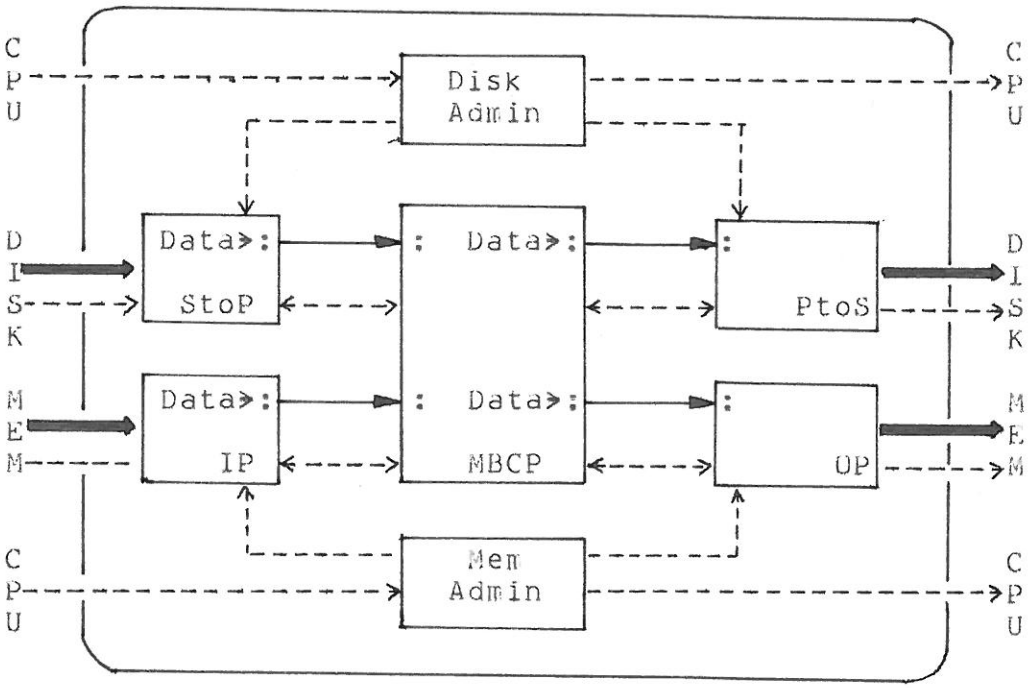


Fig. 5.4: Channel System.

5.1.13 Arm Positioning Control Units.

The task of positioning the read/write heads is simple compared to the task of administering data-transfers. Built-in logic inside the disk-hardware is capable of moving arms to a specific cylinder on request and of signalling to the outside world that the process has terminated. Therefore, further hardware

support is not needed because firm- and software mechanisms can easily take care of the arm positioning. However, because the supporting hardware has the additional purpose of providing a single uniform interface to all of the disk functions, requests to position arms as well as status information for termination, are both gated through the supporting hardware.
At the supporting hardware level the Arm Positioning Process is an independent process.

5.1.14 Disk-hardware Status Monitor Unit.

It is the responsibility of the higher-level firm- and software logic to drive the supporting hardware. To do so meaningfully (e.g., to avoide interference problems, and disable data transfers during seek operations), the higher level logic coordinates requests with disk status information.
The Disk Hardware Status Monitor is the interface unit, through which all feedback signals from the disk unit are routed to the higher level logic.

5.1.15 Suppporting hardware services.

In the previous sections, an overview of the collection of functions provided by the supporting hardware was given. With those, the following services can be performed:

1. Read a block of data from disk to a local buffer, involving the component processes:
   a) Disk I/O Request Process (Disk Admin).
   b) Serial to parallel conversion process (StoP).
   c) Memory Bus Control process (MBCP).

2. Write a block of data from a local buffer to disk, involving the component processes:
   a) Disk Admin.
   b) Parallel to serial conversion process (PtoS)
   c) MBCP.

3. Read a block of data from Main Memory to a local buffer, involving the component processes:

a) Memory I/O Request process (Mem Admin).
b) Parallel input port process (IP).
c) MBCP.

4. Write a block of data from a local buffer to Main Memory, involving the component processes:
   a) Mem Admin.
   b) Parallel output port process (OP).
   c) MBCP

5. Arm Positioning.
   Controlled entirely by the arm positioning process (APP).

6. Status Feed-back.
   Provided by the disk status monitor process DSMP.

In the next sections a complete model of the supporting hardware is given.

For each component process the following are specified:

a) The actions performed.
b) The enabling conditions.
c) The global resources required.

At this level of description the functions of Disk-hardware and Main Memory Multiplexer are irrelevant, since the mechanisms to control these functions are local to each process.
In The Pictorial Representation these mechanisms are eliminated (3.1.1), and in The System Language Description they are illustrated as function or routine calls (3.1.1, 3.1.3).

The model of the disc controller presented in the next chapters is for the special case where a single physical disk drive is connected to the system.
The model can easily be upgraded to a more general case - by duplicating all relevant disk-side variables and adjusting logic trivially.

5.2 The Memory Bus Control Process.

The data path architecture of the MBCP (Fig. 5.2.) is intended to support a double buffering strategy in transfering a sequence of physical consecutive sectors from the disk surface to Main Memory or vice versa. The strategy will minimize the transfer time for multi-sector block transfers.

To illustrate this consider:

A sector read operation, which involve the following steps:
a) transfer a sector to one of the two local buffers, and check header information.
b) depending on the result of the previous operation, then transfer the data in the buffer to Main Memory.

With a suitable design of the MBCP, which is tne only process participating in both disk-side and memory-side operations, step b) can proceed independently with a new disk-side transfer, now involving the remaining buffer.
This strategy can be used for a series of requests, assuming that tranmission rates are higher for memory transfers than for disk transfers.
This use of the supporting hardware is reflected in the design of the bus gate and memory gate control of MBCP. As the system is only equipped with 2 local buffers, and as any transfer along a bus involves a buffer, only two data paths can be active at any point in time.
Therefore, without losing flexibility, the following can be assumed:
a) at any point in time one buffer (Bufa or Bufb) is dedicated for disk transfers.
b) and the remaining buffer is dedicated for memory transfers.

These restrictions simplify the information needed to control the MBCP.

5.2.1. The component processes of MBCP.

the following activities provided by the MBCP can be identified:

a) Clear a buffer pointer for Bufa or Bufb. Upon initialization of a new disk-side transfer the

BMCP is signaled to clear a pointer (Bufaptr or Bufbptr).

b) Transfer a single word from any of the two sources to a buffer location. Whenever one of the two 'source' processes signals 'data available', the MBCP will act.

c) Transfer the contents of a buffer location to any of the two destinations. Whenever one of the two 'destination' processes signals 'space available' the MBCP will act.

To be able to model the MBCP the following definitions are necessary:

DEFINITIONS 1:

BUF ::= Control information, sent to the supporting hardware, specifying buffer allocation for disk- and memory-side data transfers. The information is,
a ≡ Bufa allocated for disk-side transfers, and (5.2.) Bufb allocated for memory-side transfers;
b ≡ Bufb allocated for disk-side transfers.

BTODPORT ::= Repository for a single word produced by MBCP, to be comsumed by the Parallel to Serial Conversion Process.

BTODSTATE::= State variable for BTODPORT, being
sa ≡ space available, i.e. signal to MBCP to produce a new word;
da ≡ data available, i.e. previous word not consumed.

DTOBPORT ::= Repository for a single word produced by the Serial to Parallel Conversion Process (i.e. from the disk).

DTOBSTATE::= State variable for DTOBSTATE, being
da ≡ data available;
sa ≡ previous word consumed.

BTOMPORT ::= Repository for a single word produced by MBCP, to be comsumed by the Parallel Output Process.

BTOMSTATE::= State variable for BTOMPORT, being
sa ≡ space available.
da ≡ data available.

MTOBPORT ::= Repository for a single word produced by the Parallel Input Process.

MTOBSTATE::= State variable for MTOBPORT, being
da ≡ data available;
sa ≡ space available.

PTRCLEAR ::= Signal to clear local buffer pointer, being
dclear ≡ clear bufferpointer for the buffer allocated for disk-side transfers.
mclear ≡ clear buffer pointer for the buffer allocated for memory-side transfers.

The internal logic of the Memory Bus Control Unit can only handle a single activity at a time, i.e. it is realized as a sequential program containing a central condition polling and control dispatching facilities. A priority is enforced by the system, which gives the synchronous disk-side operations priority.
As a consequence of this, the PRI combined construct of section 3.6.2 is used to model the MBCP (see Fig. 5.5).
The only relevant information to be illustrated in the pictorial representation is the control information (BUF) and the control signals (BTODSTATE etc.).

The manifest-list on each line indicates which values the environment is able to write to the variable, represented by the line.

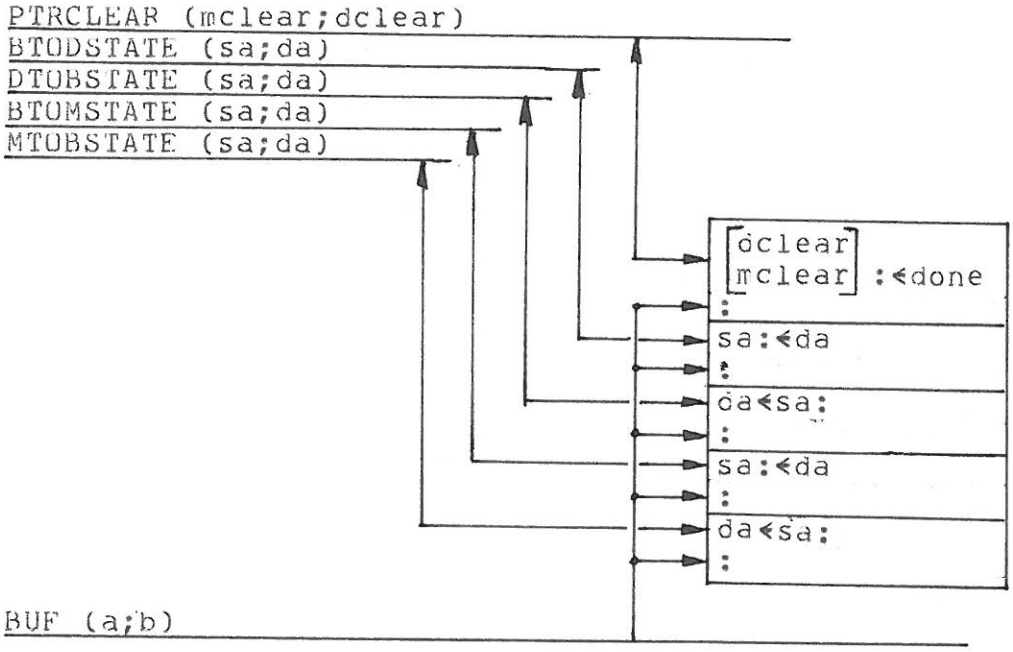


Fig. 5.5: The Memory Bus Control Process.

A complete description using the System Modelling Language, is given in Appendix A.1..

5.3 The Disk Read/Write Control Process.

The Serial to Parallel and The Parallel to Serial Conversion processes were described as two independent components in section 5.2.
However, as the two processes share the same communication wire for serial transfer and the same shift register for conversion, they are not able to run in parallel. In the model this is reflected by combining the two component processes into a single construct. In the following this construct is refered to as The Disk Read/Write Control Process (DRWC).

5.3.1 The component processes of DRWC.

In terms of the definitions in section 5.2.1 the following activities are provided by the DRWC process:

a) Consume a word in BTODPORT, signal the MBCP that BTODPORT is emptied, write the word to the disk surface and increment a word counter.
b) Write a single word (16 bit) from disk surface to DTOBPORT, increment a word counter and signal MBCP that data is available.
c) Upon termination of a sector transfer (i.e. when the word counter reaches its limit - 256) write the completion status to a state variable (in the case of a disk read operation the cyclic redundancy code check may fail).

To be able to model the DRWC process the following definitions are necessary (in addition to those of section 5.2.1).

DEFINITION 2:

DOP ::= State variable for the DRWC process, being
read ≡ disk read in progress;
write ≡ disk write in progress;
done ≡ previous operation terminated.

DWC ::= word count for current disk transfer operation.

DISKOP ::= State variable for current disk request, being
read ≡ A disk to buffer request initiated;
write ≡ A buffer to disk request initiated;
ready ≡ previous disk requests succesfully terminated.
crcerr ≡ crc error occured during disk read operation.

The distinction between DOP and DISKOP is essential, as will become obvious in section 5.5.
A model picturing all relevant control and status information for the DRWC process is given in Fig 5.6.
As the DRWC is an integral part of the supporting

hardware it is impossible to give the isolated construct any meaning. The interaction between DRWC and MBCP is shown in Fig.5.7.

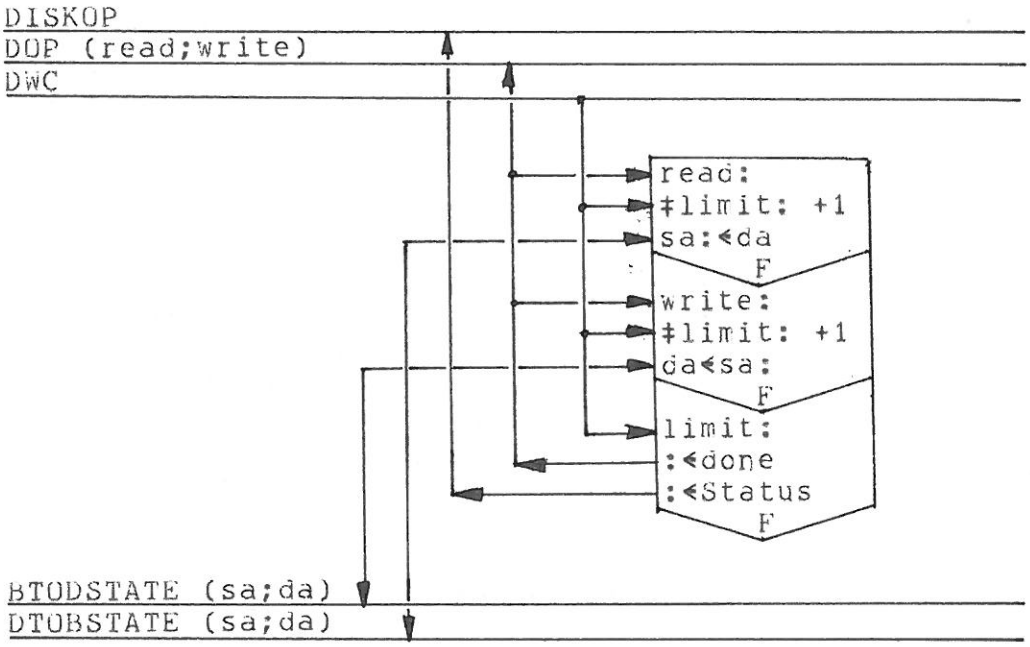


Fig. 5.6: The Disk Read/Write Control Process.

A detailed description in the System Modelling Language of the process is given in Appendix A.2..

5.4. The Memory Read/write Control Process.

The logic of Main Memory is only capable of controlling a single block transfer at a time ([13]). Therefore the parallel I/O processes (described in section 5.1.9) are unable to execute simultaneously.
As the difference between serial and parallel data transfer is irrelevant at this level of description (and therefore abstracted away) the behaviour of the parallel I/O processes and the serial I/O processes are comparable. Referring to the description in the previous section, the construct, representing the MRWC, shown in Fig. 5.7 should be self-explanatory.

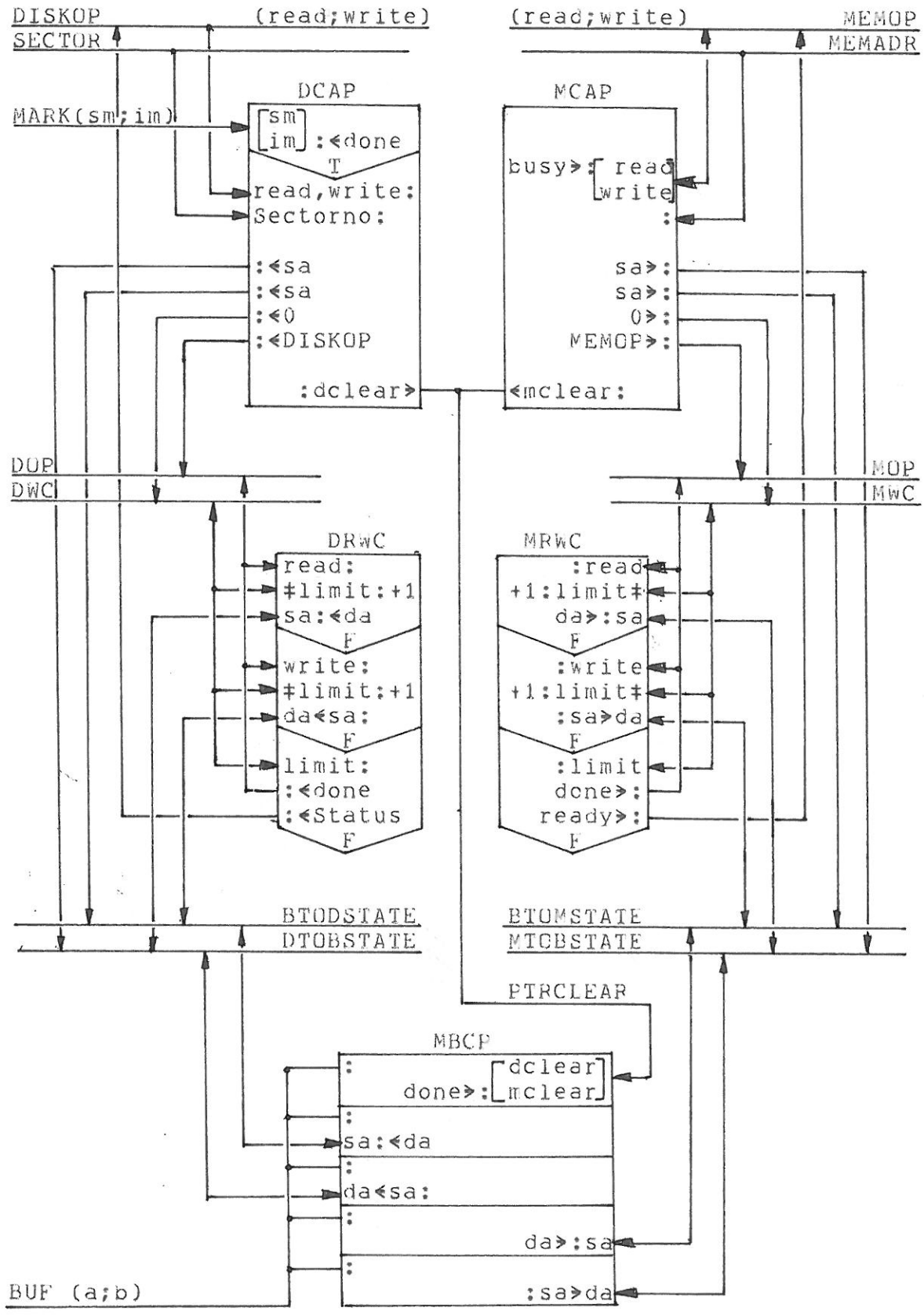


Fig. 5.7: Channel System.

5.6. The Disk and Memory Transfer Processes.

The Supporting hardware sevices described in section 5.1.15. (pt 1.-pt 4.) can now be reformulated in terms of the introduced notation.
The channel system (pictured in Fig.5.7.) can be described as two independent processes (Fig.5.8.) relieved of all local control mechnisms irrelevant for processes living outside the supporting hardware. Such processes only care about the top of the hierarchy, i.e. the processes DCAP and MCAP.

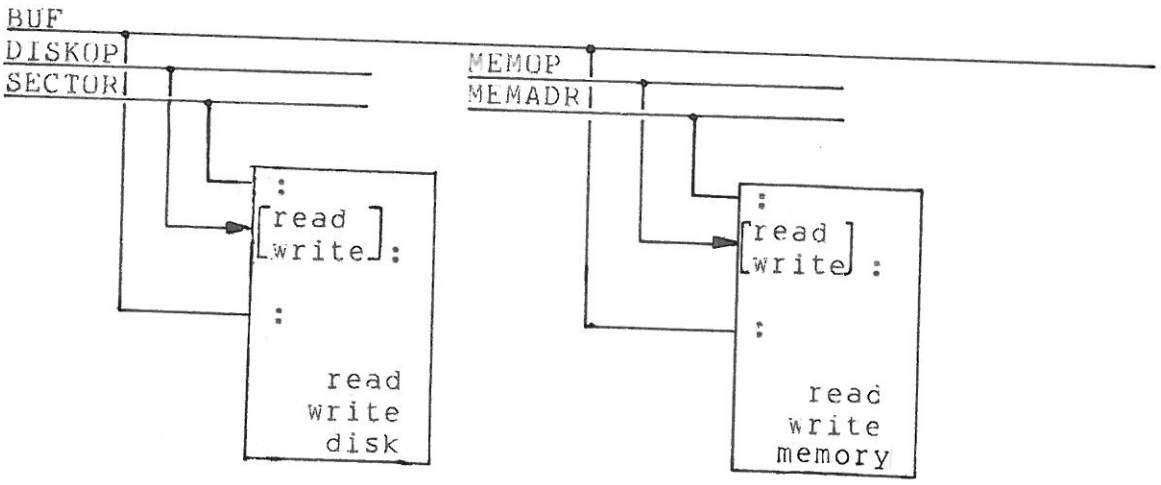


Fig.5.8: Channel System (reduced).

5.7. Arm Positioning Process.

As described in section 5.1.13 the Arm Positioning Process (APP) is an independent process.
To model the APP the following definitions are necessary.

DEFINITION 6:

CYL ::= Disk cylinder address for next seek request.

SEEKOP ::= State variable for seek operation, being
seek ≡ seek request initiated;
att ≡ head movement in progress, i.e. disk-hardware accepted seek command;

ready ≡ drive not in the process of executing a seek operation;
illadr ≡ illegal cylinder address;
incompl ≡ malfunctioning of seek operation.

The single component process APP is shown in Fig.5.9., and a description in The System Modelling Language is given in Appendix A.6.

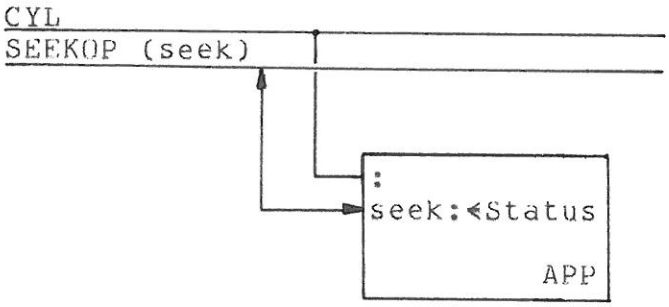


Fig.5.9: The Arm Positioning Process.

5.8. The Disk Status Monitor Process.

The higher level logic of the disk controller coordinates request to seek, read/write disk and read/write memory with the status of the suppporting hardware and status of the disk hardware.
This higher level, which is realized in firmware in the computer system, is not capable of reading the value of the state variables : SEEKOP, MEMOP and DISKOP ,since these variables are realized as computer system output buffers.
The status of these variables together with the status of the disk hardware are continuously monitored by The Disk Status Monitor Process (DSMP). The DSMP assembles all relevant status information for the hardwired part of the disk controller into a single word (each bit indicates a state) and sends it to a buffer, readable by the computer system.
In the model of the DSMP in Appendix A.7. an incomplete set of available status information is illustrated. The set is, however, sufficient to fully understand the synchronization aspects of the disk controller.

DEFINITION 7:

STATUS ::= Assembled status information, each bit reflects a state of a variable, as
bit 0 ≡ (SEEKOP ≡ ready);
bit 1 ≡ (DISKOP ≡ ready);
bit 2 ≡ (MEMOP ≡ ready);
etc. see Apendix A.7.

In the higher level of the disk controller, STATUS(<n>) will be used to refer to the n'th bit of the status word.

The component process DSMP is shown in Fig.5.10.

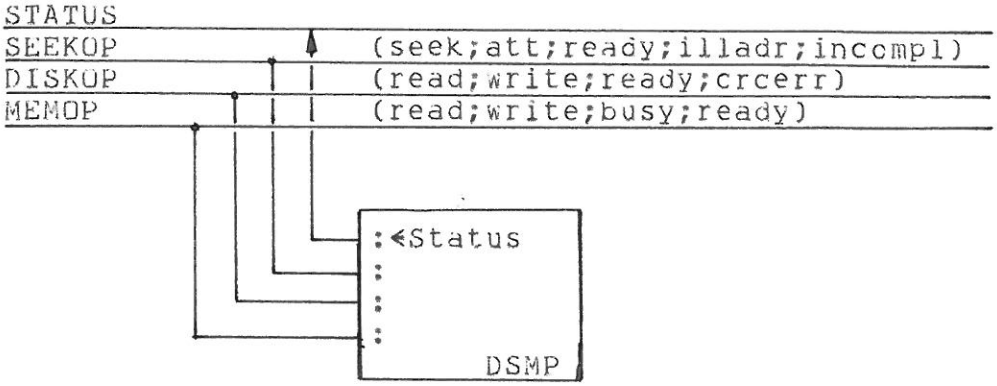


Fig.5.10: The Disk Status Monitor Process.

6. THE FIRMWARE LEVEL.

In the previous chapter a detailed description was given of an independent hardware processor, which provides a single uniform interface between the primitive disk-hardware and the main computer.
The most important property of the supporting hardware is, that it relieves the disk I/O service routines (implemented in firmware) of the monotonous and time consuming jobs of monitoring disk status flags and transfering single words between disk and main memory.

The supporting hardware provides:

a) Block transfer.
   At any time there is a hardware data bus connection between the disk drive and a buffer; at any time there is a hardware data bus connection between a buffer and main memory.
   Data transfer along the 2 busses can proceed independently and concurrently (section 5.2).
   The disk-side buffer can be exchanged (flipped), essentially instantaneously, with the memory-side buffer (section 5.2. and definition 1.).
b) Seek operations.
c) Status feed-back.

The logical interfaces between firmware disk service routines and the transfer- , seek- and status processes realized in hardware were illustrated in Fig. 5.6., 5.7. and 5.8., respectively.

The supporting hardware is potentially capable of overlapping independent data transfers. However, it is the responsibility of intelligent firmware routines to coordinate hardware functions in such a way that as much independent data-transfer activity as possible is actually overlapping, thereby maximizing disk I/O throughput.

The design of the firmware level disk control mechanisms should satisfy the following requirements:

a) As many system components as meaningfully possible should be inplementable as independent processes.
b) These processes should be connected as loosly as possible, and each should synchronize itself knowing little or nothing about the other processes.
c) the combined behaviour of these processes should spontaneously take advantage of all possibilities for overlapping I/O, including pipelining of disk-read sequences and disk-write sequences.

The basic idea, behind the system-model presented in this chapter as a solution satisfying these requirements, is the obvious existence of two independent asynchronous pipelines. One for disk-read operations (utilizing a hardware disk-side function and a hardware memory-side function, in sequence); and one for disk-write operations (utilizing a memory-side operation followed by a disk-side operation). Such asynchronous pipelines will automatically - if they are designed properly - overlap the execution of sequences of disk-read operations and sequences of disk-write operations. Additionally, the system is designed to optimize the execution of a sequence of mixed read- , write- and seek request.

In the following presentation, the model of the system is broken into modules, which are described in turn. However, this does not reflect the actual process of the system's development.
First, the well defined modules in the pipeline system - such as memory-to-buffer, buffer-to-disk etc. - were isolated, analyzed and modelled independently, and then interfaced together.
During the interfacing stage, the existence of new component processes appeared and caused redefinition of the interfacing specification for other components.
The following sections present the final state, of this iterative design process.

6.1.The Channel Traffic Coordinator.

The Channel Traffic Coordinator Process (CTC) is the most complex and intellegent process in the firmware part of the disk-controller. It administers the initialization of the read and write pipeline processes to start Disk/Main Memory transfer operations, and the initialization of disk arm movements. It coordinates current operations with requests for new activities, in such a way that data transfers are overlapped (while the security of the system is maintained).

6.1.1. The Queue Structure.

Requests for Disk to Main Memory transfers, Main Memory to Disk transfers and for arm movements, are assumed to be queued in a strucrure, initialized by the Operating System in such a way that requests to single tracks will be extracted by the CTC one after another with increasing sector numbers. The Disk Drive Queue is the conglomerate of several Track Queues as illustrated in Fig.6.1..

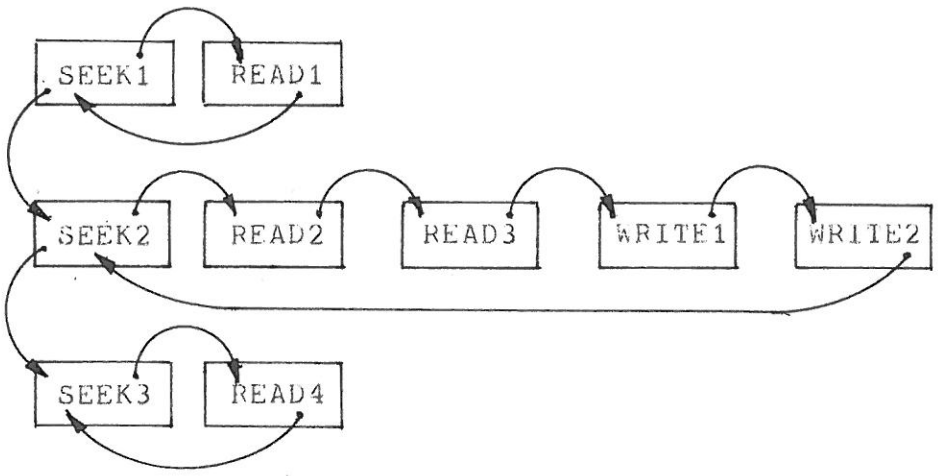


Fig.6.1: The Disk Drive Queue.

Information about the current physical sector-number, sent to the CTC from the supporting hardware, enables CTC to start emptying the Track-Queue at the most optimal point, thereby reducing latency time

(this function of the CTC is not considered in this paper).

6.1.2 Timing for a sequence of read requests.

The design of The Channel Traffic Coordinator mainly rests on the timing properties of transfer operations, which determine the possibilities for overlapping data transfers.
The following example illustrates the overlapping of the operations required to excute a sequence of Disk Read Requests, for transfering a series of physical consecutive sectors from disk to main memory.

The following is assumed:

a) The disk-arms are in position, i.e. the sectors to be transfered are all located on a track belonging to current cylinder.
b) The asynchronous Read Pipeline is empty.
c) The next 4 requests in the Request-Queue are:
   R1 ≡ request to transfer sector 1 to memory.
   R2 ≡ - - - - 2 - - .
   R3 ≡ - - - - 3 - - .
   SEEK ≡ request to move disk drive arms to a new cylinder.
c) There exists a communication buffer RWORK - a line for request information - whose value is produced by the CTC and consumed by a process executing the first stage in the pipeline (i.e.the process of transfering a sector from disk to one of the two memory buffers located in the supporting hard-ware).
d) There exists a communication interface, for now referred to as FLIP, which, on termination of a disk-side operation, connects current disk-side buffer to Main Memory and connects the remaining buffer to the Disk. The FLIP process initiates the next step in the Read pipeline - the memory-side operation.

Fig.6.2. gives a diagram of the timing aspects in

processing the 4 requests.

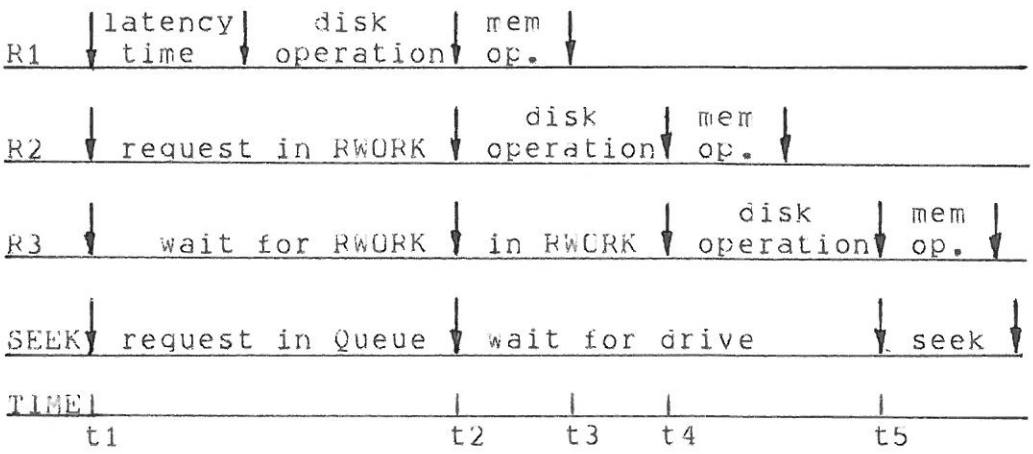


Fig.6.2: Timing for sequence of read requests.

NOTE:

a) R1 enters the pipeline at t1 and R2 at t1+(the time CTC takes to remove a request from the Queue).
b) R3 enters the pipeline at t2.
c) During t2-t3 three requests are in the pipeline and CTC is waiting for disk resource to initiate a seek.
d) Under the asumption that no seek operation is in progress the only resource required to initiate a new read is the line RWORK.
e) The FLIP interface is in operation at t2, t4 and t5.
f) Transmission rates between disk and buffer are substantially lower than between high-speed main memory and buffer (in the real system, there is a 1-20 ratio). Overlapping may therefore seem unnecessary.
   However, the header-checking - performed by the disk controller before memory operations (chapter 4.) - and the capability of transfering consequetive sectors, together require overlapping to take place.

6.1.3. Timing for mixed read and write requests.

The control mechanisms required for overlapping mixed read- and write requests are more complex than the overlapping mechanisms for a sequence of requests of the same type.
The control cannot be performed by the mechanisms provided by any of the two asynchronous interacting pipelines.

The overlapping has to be controlled by the Channel Traffic Coordinator, in the following way:

a) If a read-request (for sector <n+i>) follows a write-request (for sector <n>), then the CTC must delay the initialization of the read pipeline until the disk-side operation process involved in performing the write-request, has terminated.
If this protocol is not followed, the new read-request may lock the disk-side resource which should be used in the final stage for the write-request pipeline, i.e. the request for sector <n+i> will be processed before the request for sector <n>, which certainly was not the intention, as it causes the write request to wait in the pipeline until the read request releases the disk, and sector <n> actually has been passed once.
The CTC administers this situation by monitoring a disk-workload variable, which is incremented by CTC when a new request is send to any of the two pipelines, and decremented upon termination of a disk-side operation.

b) If a write-request follows a read-request the initialization of the write-request must be delayed until the previous read-request has entered the disk-side stage of the pipeline.
The CTC administers this situation by delaying the new write-request until the RWORK resource is released by the previous read-request.

These situations are illustrated in detail in the description of the CTC in Apendix B.1.

6.1.4. The Model of The Channel Traffic Coordinator.

To understand the model of the CTC (Fig.6.3.) the following definitions are necessary.

DEFINITION 1:

QUEUESTATE ::= State variable for the Queue, which is a shared resource of the CTC and the Operating System. Values,
lock ≡ the Queue is locked.
unlock ≡ the Queue is unlocked.

DRIVE ::= State variable for physical disk drive, being
seeking ≡ the disk is in progress of moving arms.
<n> ≡ count of jobs in the pipelines still pending for, or in progress of a disk transfer.

READSIGNAL ::= Signal to begin a disk to main memory transfer; the disk arms are positioned; all necessary control information is in RWORK. Values,
start ≡ signal to start.
free ≡ previous signal consumed, and RWORK free.

RWORK ::= Disk sector and memory destination address for next read operation.

WRITESIGNAL::= Signal to begin a main memory to disk transfer; the disk arm is positioned; all necessary information is in WWORK. Values,
start ≡ signal to start.
free ≡ previous signal consumed, and WWORK buffer free.

WWORK ::= Memory source address and destination disk sector for next write request.

SEEKSIGNAL ::= Signal to begin a disk drive arm positioning. All data-transfers to disk and all current disk-to-buffer operations have been finished (i.e.

DRIVE=0), and no new ones will be issued (i.e. DRIVE set to seeking) until seek is finished. All necessary control information is in SWORK. Values,

start ≡ signal to start.

free ≡ previous signal consumed and SWORK free.

SWORK ::= Cylinder address for next seek request.

In the symbolic description of the processes in the firmware level disk controller in Appendix B., the variables RWORK, WWORK and other variables which are used to contain transfer control information, are pointers to request vectors containing disk-side control information, memory-side control information, and both temporary and completion status information.

The model of The Channel Traffic Controller, which is shown in Fig.6.3., is a PRI construct (section 5.6.2), where each component describes the activities performed by the CTC to start a specific job (read; write; seek; newread; newwrite).
The signal, which selects a specific box, is local to the CTC; it is determined by the types of the next and the previous request, only.

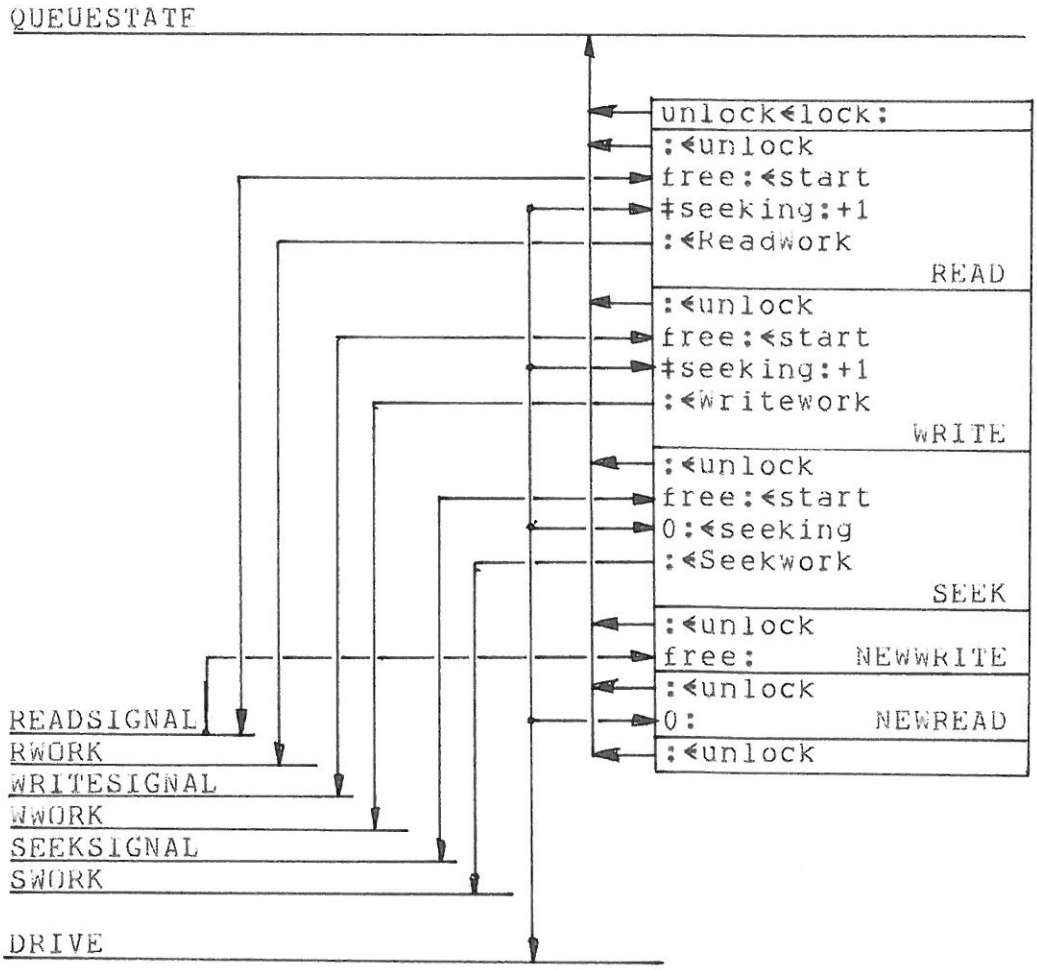


Fig. 6.3: The Asynchronous Channel Traffic Coordinator.

The local control logic of CTC is described in detail in Appendix B.1.
The behaviour of the CTC in interacting with other processes in the firmware part of the disk-controller is illustrated i Fig.6.9., and it might be advantageous to consult this complete model in order to see the functions of the CTC with their full meaning.

6.2. The Drive Arm-positioning Handler.

The Drive Arm-positioning Handler (DAPH) is a simple process.
The guarantee that arm-positioning does not interfere with other processes is held by the CTC process which on the one hand delays signals to the DAPH until all pending disk-side transfer operations have terminated, and on the other hand delays new transfer operations until the DAPH signals its termination (by changing DRIVE from 'seeking' to 0).

The shared variables accessed by DAPH are given in Definition 1.

Two models of the DAPH are given (Fig.6.4. and Fig.6.5).

In Fig.6.4. the communication with the seek process realized in hardware is illustrated (see section 5.7) in addition to the communication with the CTC.

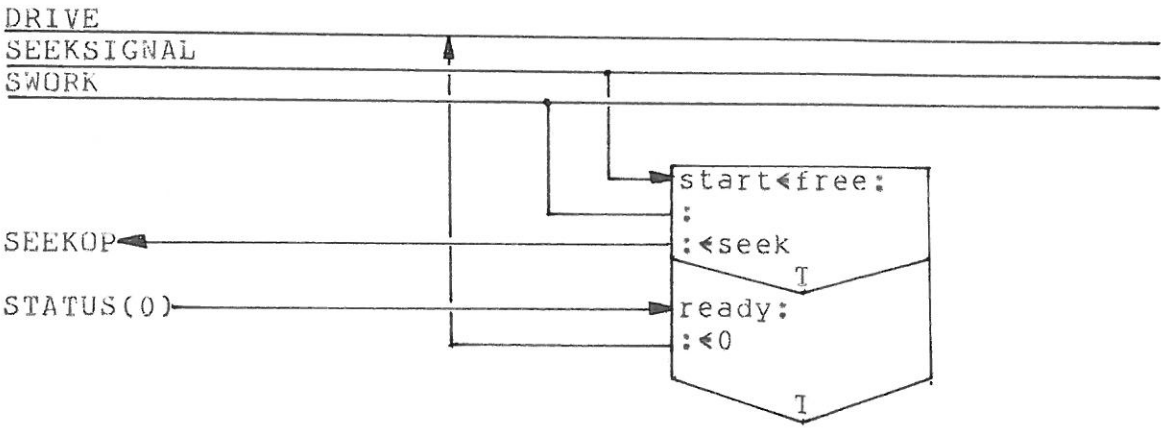


Fig. 6.4: The Drive Arm-positioning Handler.

The components describing the communication with the hardware do not add any information to the total picture of the combined behaviour of the firmware implemented part of the disk controller. These components provide a description of

local control mechanisms, only - i.e. no process, described on this level, other than DAPH uses the resources (SEEKOP and STATUS(0)).
Such components are best be eliminated from the model, thereby relieving the model of 'noise'.
Furthermore the DAPH SEQ-construct can be collapsed into a single box (Fig.6.5.).

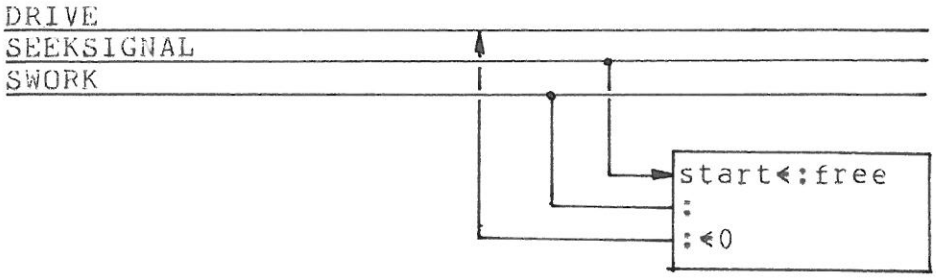


Fig. 6.5: The Drive Arm-positioning Handler (reduced).

6.3. The Disk Read/Write Controller.

The Disk Read/Write Controller (DRWC) administers the initialization and termination of disk sector transfers between the disk and the local memory buffers of the supporting hardware.
The DRWC process is divided into two minor processes; one which performs disk-to-buffer activity as the operation on the first stage of the Read pipeline; another which performs buffer-to-disk activity in the Write request pipeline.
Due to the architecture of the supporting hardware, these independent processes are not capable of executing simultaneously.

In addition to those of DEFINITION 1, the following components are referenced in the model of the Disk Read/Write Controller (Fig.6.6.).

DEFINITION 2:

DISKSTATE ::= State variable for disk-side channel, being
idle ≡ ready for either disk-to-buffer operation or buffer-to-disk operation.
busy ≡ lock to prevent more than one process from acting on the condition DISKSTATE=idle. As two independent processes - the Read pipeline and the Write pipeline - may simultaneously wait for this condition, the condition must be singular (section 3.4.5).
dtom ≡ Disk-to buffer activity terminated; waiting for buffer-to-Memory activity. All necessary control information is in DPARAM. (Both the information and the signal are used to control the Read pipeline process).
mtod ≡ Memory-to-buffer activity terminated; waiting for buffer-to-Disk activity. All necessary control information is in DPARAM. (This signal and the information enclosed are used to control the Write pipeline process).

DPARAM ::= Disk-side repository of parameters and temporary status for either the Disk Read pipeline process waiting for Main Memory activity or the Disk Write pipeline process waiting for Disk activity.

EOREQUEST ::= Signal to the Operating System, that a Read or Write transfer is completed. Values,
term ≡ request terminated; completion status information is in EORSTATUS.
free ≡ previous signal consumed.
busy ≡ lock to prevent more than one of the pipeline processes from terminating with the condition 'EOREQUEST=free'.
Both the Read pipeline and the Write pipeline signal their completion by means of EOREQUEST signals.

EORSTATUS ::= Transfer completion status information.

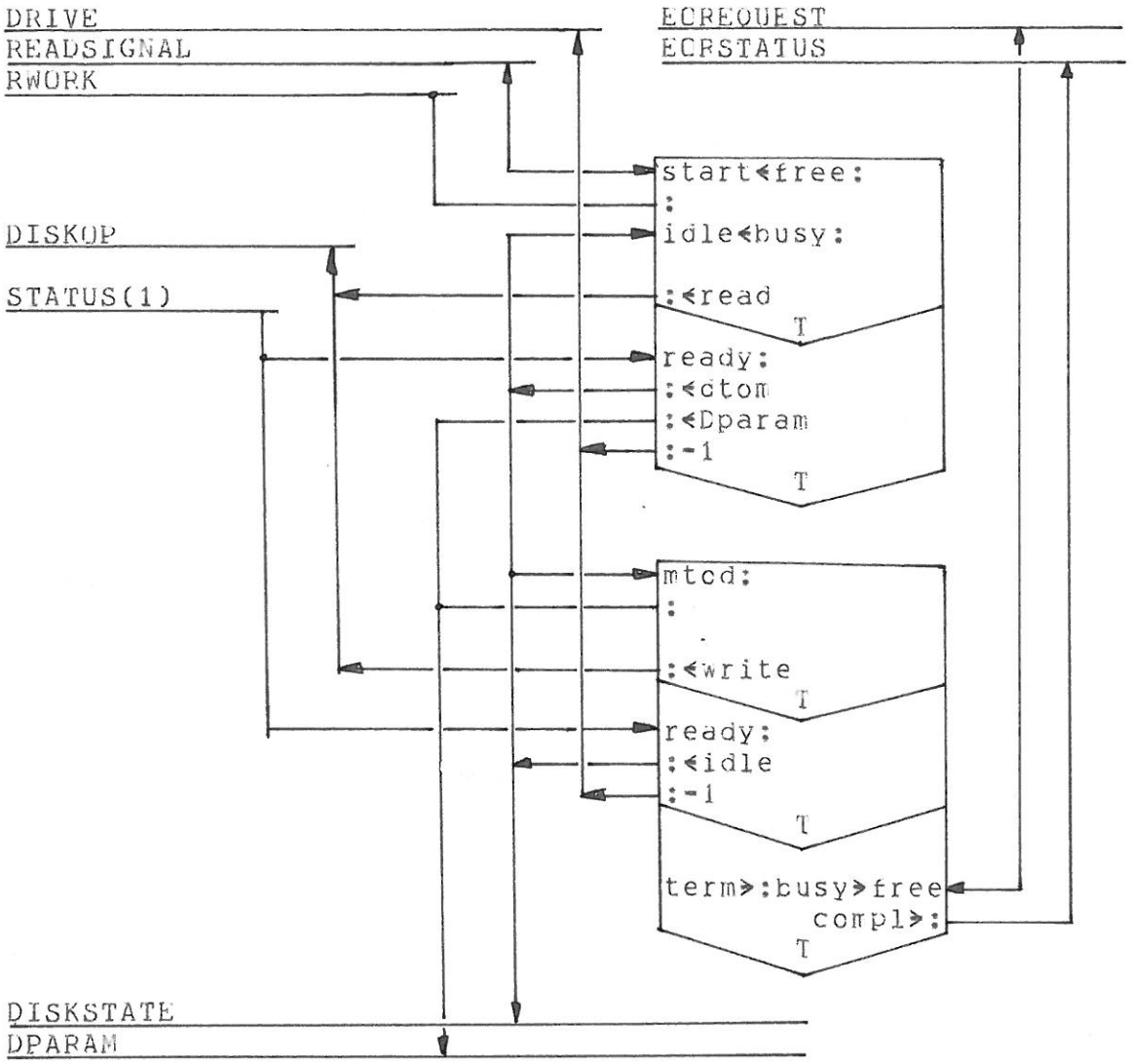


Fig. 6.6: The Asynchronous Disk Read/Write Controller.

Additional remarks on the model of The Disk Read/Write controller:

a) upon termination of a disk-to-buffer or a buffer-to-disk operation the work-load counter (DRIVE) is decremented.
b) As the two processes which perform disk-side operations, exclude each other through arrangements held by the DISKSTATE variable, the components describing the com-

munication with the supporting hardware (DISKOP and STATUS(A)) can be eliminated and regarded as internal function calls. The reduced model is illustrated as a component in the model of the complete system in Fig.6.9.

c) The buffer-to-disk process notifies the Operating System of the termination of a Main Memory to Disk request. Completion status is written into EORSTATUS, which the write pipeline shares with the Read pipeline. The sharing is controlled by using an EBS of the Singular Conditional form for the EOREQUEST variable. The notification is the final stage in the Disk Write pipeline.

6.4. The Memory Read/Write Controller.

The Memory Read/Write Controller (MRWC) resembles the Disk Read/Write Controller. Only points of difference between them are noted.

DEFINITION 3:

MEMSTATE ::= State variable for memory-side channel, being
- idle ≡ ready for either memory-to-buffer operation or buffer-to-memory operation.
- busy ≡ lock to prevent more than one process from acting on the condition MEMSTATE=idle. As two independent processes - the Read pipeline and the Write pipeline - may simultaneously wait for this condition, the condition must be singular.
- mtod ≡ Memory-to-buffer operation terminated; waiting for buffer-to-Disk operation. All necessary control information is in MPARAM.
- dtom ≡ Disk-to-buffer operation terminated; waiting for buffer-to-Memory operation. All necessary information is in MPARAM.

MPARAM ::= Memory-side repository of parameters and temporary status for either Disk Write pipeline process waiting for disk-side, or Disk Read pipeline process waiting for memory-side resource.

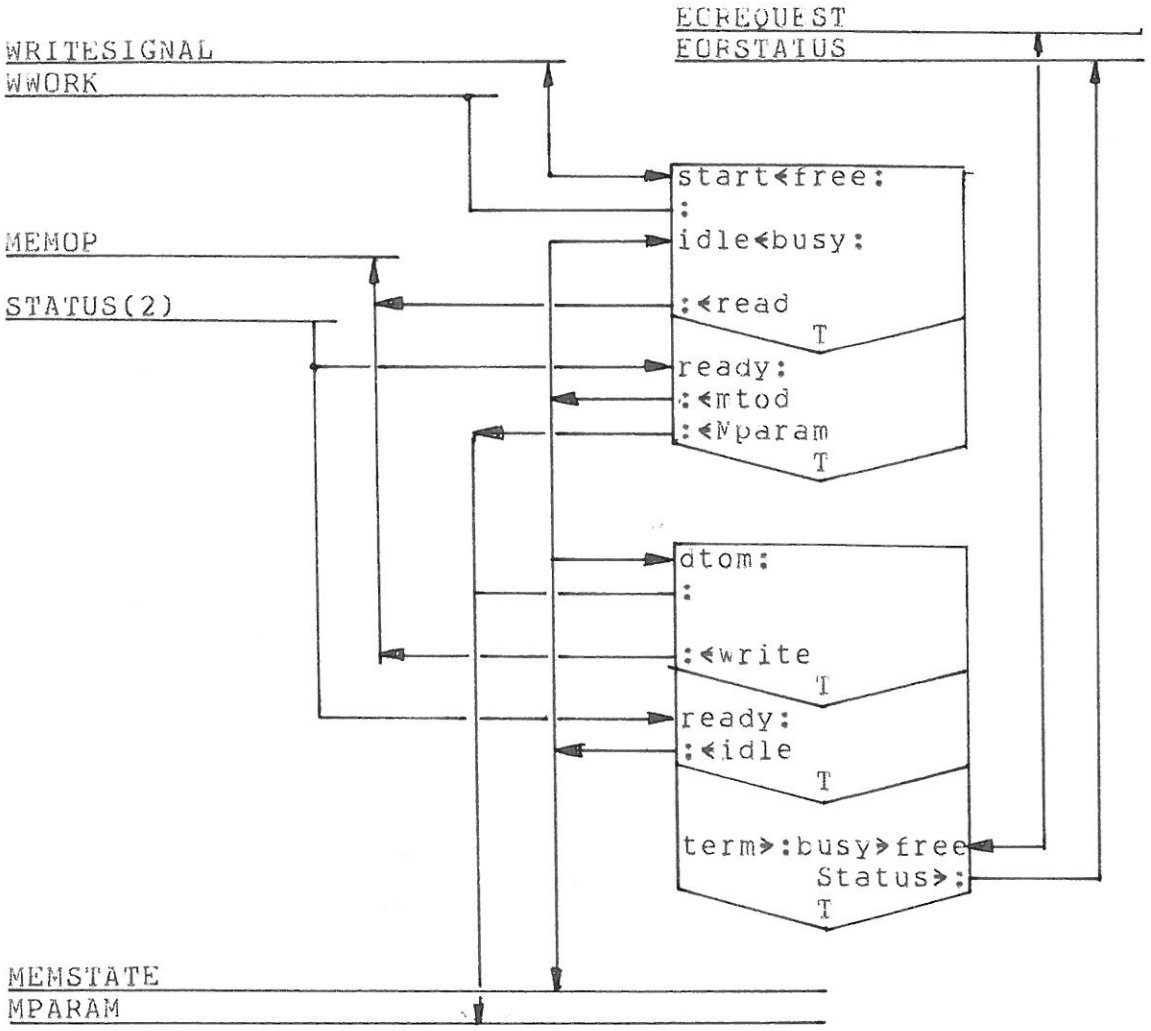


Fig. 6.7: The Asynchronous Memory Read/write Controller.

6.5. The Channel Traffic Governor.

There is one point in the two pipelines, which is extremly complex and critical, namely the conjunction point between the pipelines.
This is the point where The Disk Read process potentially waits for memory resources, and The Disk Write process potentially waits for disk resources.
A mutual interdependency may arise between the two pipeline

processes at this point.

The 'interface' process between the disk-side transfer process and the memory-side transfer process should provide the following:

a) conventional communication between a single process and its successor for both the Read and the Write pipeline. This controls the situation, where the two pipelines are executing independently.
b) a 'swap' of resources if the pipelines are executing concurrently, and have reached the point of mutual interdependency.

This leaves 3 situations, for the interface process - The Channel Traffic Governor (CTG) - to handle :
a) DISKSTATE=dtom and MEMSTATE=idle;
   the Read pipeline is active only, in which case the signal DISKSTATE=dtom should be forwarded to the memory-side process.
b) DISKSTATE=idle and MEMSTATE=mtod;
   is the inverse situation, now for the Write pipeline.
c) DISKSTATE=dtom and MEMSTATE=mtod;
   both pipelines are in operation and a swap of signals as well as control information has to take place.

As illustrated in the pictorial representation (Fig.6.8.) the CTG is designed to consist of 2 component processes.
The model of the CTG in Appendix B.5 consists of a single component process, which will act upon a complex state signal.

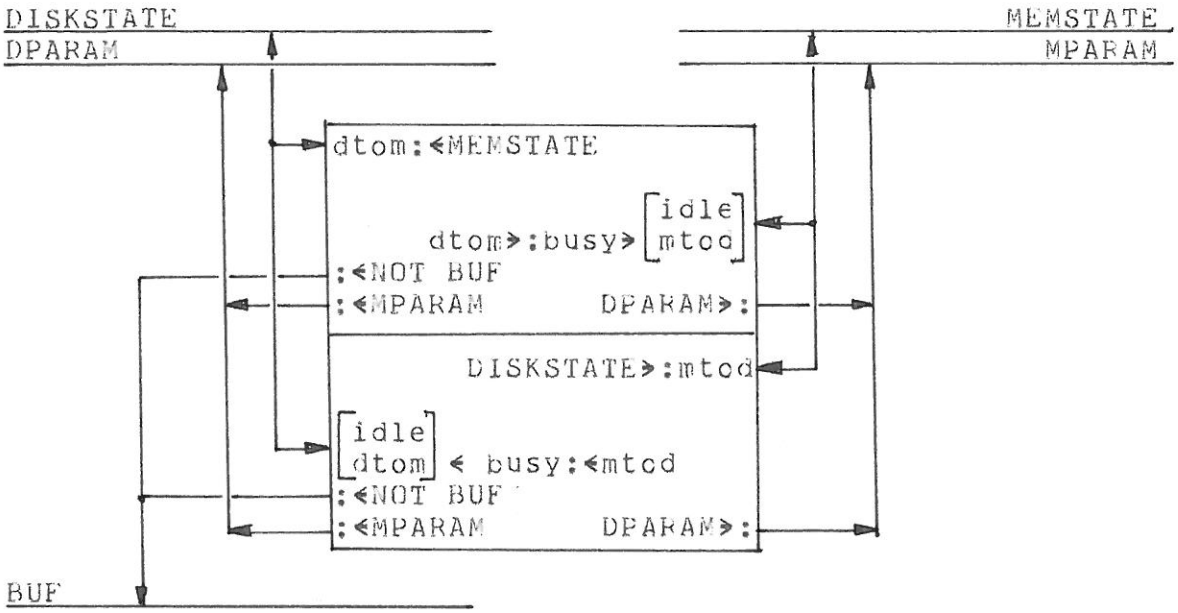


Fig. 6.8: The Asynchronous Channel Traffic Governor.

6.6. The Model of The Supporting Firmware.

A complete model of all components - processes and shared variables - which is involved in the firmware part of the disk-controller is illustrated in Fig.6.9..
In addition to being a model for an implementation of the system (not necessaryly a firmware implementation) the complete model can be used to verify the protectedness of the system.

The set of EBS's associated with a resource, should guarantee that the resource is properly shared; i.e. these EBS's reflect the agreements among the set of processes sharing the resource.
Several ways of sharing resources are illustrated in the model. For example:

a) Sharing of the disk-resource is controlled, since the component processes using the disk do not act upon the same event. The disk-to-buffer operation is enabled if 'DISKSTATE=idle' and buffer-to-disk operation is enabled

if 'DISKSTATE=mtod'.

b) The 'register' for request completion information (EORSTATUS) is shared by two component processes acting upon the same event. As stated earlier, the sharing must be controlled by using Singular conditions Statements. In this case, the EBS's associated with EOREQUEST have the form: 'free≤busy:≤term'.

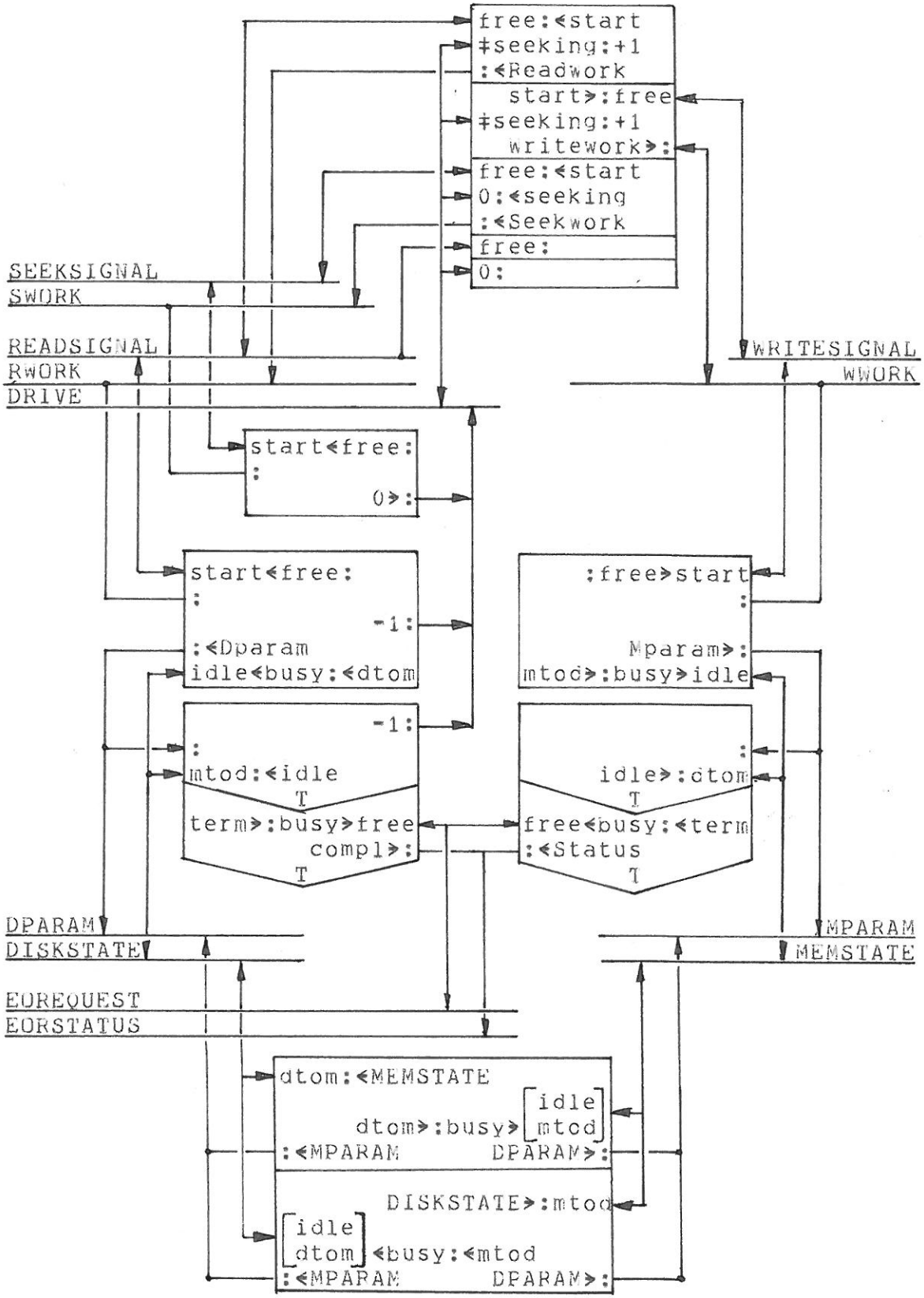


Fig. 6.9: The Model of the Supporting Firmware.

6.7. Implementation in Firmware.

The set of processes described in this chapter are realized by functions implemented in firmware.

The reasons for implementing this set of processes at the firmware level - instead of the hardware or software level - are as follows.

a) Flexibility.
The Disk Controller is an experimental processor, which should be left sufficiently malleable or plastic to allow for future functional development.
The physical device configuration may change. By implementing the 'interface' to external devices in flexible firmware, no alterations in or additions to existing applications (file-managers, user-programs etc.) may be necessary .
The users' requirements may change. By implementing the 'interface' (seen from the physical configuration point of view) to user applications, in flexible firmware, the interface may be changed or be extended to support these new requirements without affecting hardwired components. One area where extended interface requirements may arise is that of large-scale on-line data bases. 'Low-level' retrieval facilities could profitably be implemented as intelligent secondary storage interfaces.

A hardware implementation would not provide this flexibility, but it would be provided by a software level realization.

b) Timing Requirements.
the Supporting Hardware provides an asynchronous interface to the disk hardware; However, interrupts caused by the Supporting hardware (disk-side-operation terminated, etc.) can in practice not be inhibited, e.g. if requests for transfers of consecutive sectors reside in the request queue or in the pipeline, the interrupt sent from the Supporting Hardware upon termination of one disk-side-operation should indirectly bring forward a signal to start the next transfer, within a certain time frame.

The operations performed in this case, by various interrupt handlers and I/O service routines, have to execute within the time span of a sector-gap, in order to maintain a continuous flow of data from disk to main memory, i.e. if the service routines break the time frame the disk will idle for at least one revolution.

The software level, in the specific system to which the disk drives are connected ([2]), is based on a virtual CPU ([5]), and all I/O performed by software level routines cause one or several context switches between CPU environment and I/O environment ([4]).
Due to high virtual CPU cycle-time (10µs-40µs) and time-consuming context switches (40µs), interrupt-handlers and I/O service routines realized by functions in software would not be able to satisfy the time requirements described above (sector gap is 300µs).

6.7.1. Polling and Control Dispatch Module.

Having decided to realize a set of interacting processes by functions implemented on a single level, where multiprosessing is not supported -(in this case the firmware level), the following strategy is recommended:

a) Use the process description (Fig.6.9. or Appendix B) to design a single, central polling and control dispatching module.
b) Realize the module by a highly efficient firmware function.
c) Realize the remaining part of the processes as normal sequential firmware interrupt handlers.

The virtual machine, realized by a set of microcoded functions, shares the physical processor with the firmware part of the disk controller (and other logical channel controllers). A central monitor administers the sharing.
The Polling and Dispatch routine for the disk-controller occupy the processor resource for at least as long as it takes to execute the statements of the routine once.

Therefore b) is a vital requirement.

A description of the Central Signal Polling and Control Dispatch Process is given in Fig.6.10. and a description of the microcoded interrupt handlers are given in Apendix B.6..
Both the Polling/Dispatch process and the interrupt-handlers are normal, sequential, and rather simple, programs - their combined behaviour is however quite incomprehensible.

In addition to the definitions given in the preceding sections the following remarks on the descriptions are necessary:

a) In addition to showing the interaction between the processes on the firmware level, the description also illustrates the communication with the supporting hardware (see Fig.6.4., Fig.6.6.-6.8).
New states for DISKSTATE and MEMSTATE (reading, writing) have been introduced to enable the system to administer this communication. These states function as enable-interrupt masks.
b) The variable NEXTREQUEST mirrors the internal control mechanism of The Traffic Channel Coordinator (section 6.1.4.).
Its use should be self-explanatory.
c) TERMREAD/TERMWRITE indicates whether or not a request has terminated.
The handler for the interrupt-event
TERMREAD=pending $\wedge$ EOREQUEST=free
will notify the Operating System of the termination.
d) DPARAM and MPARAM are repositories for current disk-side and mem-side operations, respectively.
e) The priority schema used in the polling and dispatch process, gives the functions on the final stages of any pipeline priority.
f) No indivisibility is described, the disk controller engages the processor until none of the conditions in the polling process are satisfied.

```
process POLL is
{poll
  goto valof
  {
    if TERMREAD=pending /\ EOREQUEST=free
       resultis eorreadaction
    if TERMWRITE=pending /\ EOREQUEST=free
       resultis eorwriteaction
    if MEMSTATE=writing /\ STATUS(2)=ready
       resultis termmemwrite
    if DISKSTATE=writing /\ STATUS(1)=ready
       resultis termdiskwrite
    if MEMSTATE=dtom
       resultis startmemwrite
    if DISKSTATE=mtod
       resultis startdiskwrite
    if (DISKSTATE=idle /\ MEMSTATE=mtod) \/
       (DISKSTATE=dtom /\ MEMSTATE=mtod)\/
       (DISKSTATE=dtom /\ MEMSTATE=idle)
       resultis flipaction
    if MEMSTATE=reading /\ STATUS(2)=ready
       resultis termmemread
    if DISKSTATE=reading /\ STATUS(1)=ready
       resultis termdiskread
    if MEMSTATE=idle /\ WRITESIGNAL=start
       resultis startmemread
    if DISKSTATE=idle /\ READSIGNAL=start
       resultis startdiskread
    if NEXTREQUEST=read /\ READSIGNAL=free /\
       DRIVE#seeking
       resultis startDisktoMem
    if NEXTREQUEST=write /\ WRITESIGNAL=free /\
       DRIVE#seeking
       resultis startMemtoDisk
    if NEXTREQUEST=seek /\ SEEKSIGNAL=free /\
       DRIVE=0
       resultis startseek
    if NEXTREQUEST=newwrite /\ READSIGNAL=free
       resultis startMemtoDisk
    if NEXTREQUEST=newread /\ DRIVE=0
       resultis startDisktoMem
    if DRIVE=seeking /\ STATUS(0)=ready
       resultis termseek
  }
}poll
```

Fig. 6.10: Signal Polling and Control Dispatch.

In mapping the descriptions into microcode the only non-trivial algoritm is the algoritm for The Signal Polling and Control Dispatching Process, which must be coded as a highly efficient micro-coded function.

It is, however, outside the scope of this paper to present and explain the micro-program for the polling and dispatch process (5 instructions : 1.5µs). It can be mentioned that the basic idea behind the 'solution' is to implement the signal polling as priority encoding, in the following way:

1) to represent each relevant condition in the firm-ware level / part of the disk controller as a single bit (positioned according to the schema given in the symbolic description of the polling / dispatching process).
2) use the bus masking facility provided by the microprogrammable processor to perform full-word logical operations.
3) use bus bit-encoding facilities to select and enable the one 'active' interrupt-handler which has priority.

7. CONCLUSION.

The goal of the work which has been presented in this paper, is to develop a modelling tool for analyzing, describing and designing the structure of a soft-, firm- and hardware system consisting of independent process components.
A Pictorial Representation has been proposed (Chapter 3) to accomplish this goal, and the tool has been applied to an example system to illustrate its feasibility.

The description provided through The Pictorial Representation and The System Modelling Language should be sufficient to allow for the described system to be implemented as functions at any level - be it software, firmware, hardware or a mix of these levels.

To this end, the Pictorial Representation is intentionally designed to be comprehensible to both software - and hardware engineers.
Additionally, all 'level' dependent requirements are eliminated from the model. For example a line in the representation, may be implemented as a communication wire, register, memory location, variable etc., depending on the level to which this part of the system is 'assigned'.
Design criteria or constraints, such as timing requirements, existence of hardware, software facilities available, required flexibility, cost, etc., which would influence the decision of how to implement a particular function, are abstracted from the model representation.
Such design constraints are only considered at the final stage of a development of a system - the implementation.

To illustrate this 'implementation' flexibility attention is drawn to section 4.3 of this paper, where a firmware 'solution' to parts of the low level memory-side operation was mentioned. Additionnally, a hardware design utilising priority encoding logic and Read Only Memory for the part of the Disk-Controller, which was assigned to the firmware level in this paper, is proposed in [13].

7.1. Future work.

Due to the short period of time The Pictorial Representation has been under development, the modelling tool cannot be declared complete.

The tool must be studied further.

7.1.1. Extentions to The Elementary Behaviour Statements.

Seen from a component process (box) point of view, an EBS was associated with a single shared variable (line). In the condition-list of an EBS it was possible to specify relations between the value of the line on one hand, and a constant or the value of a local variable on the other.
It might be advantageous if it were possible to specify relations between values of two or several lines.
For example:
consider the Channel Traffic Governor described in section 6.5.; the model was designed to consist of two component processes (Fig 6.8).
An alternative model - now allowing for a new 'line-to-line' relation form of EBS could be,

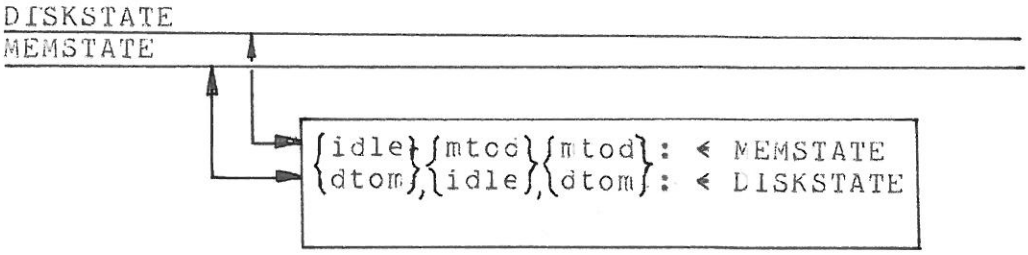


Fig.7.1: The Channel Traffic Governor (modified).

The exact interpretation is given in Appendix B.5.

7.1.2. Verification.

Attempts to verify the design by using information contained in the description have been made, for example in section 6.6..
Further investigations have to be made in this area, and systematic rules for examination of protectedness, sharing etc., should, if possible, be provided. Existing theoretical tools (for example Petri Nets) may prove to be usable for verification in connexion with the pictorial representation.

List of References.

[1] P. Kornerup, B. Shriver:
"A Description of the MATHILDA Processor".
DAIMI PB-52, Computer Science Department,
Aarhus University, Denmark, September 1975.

[2] P. Kornerup, B. Shriver:
"An Overview of the MATHILDA System".
SIGMICRO, January 1975.

[3] I. Holm Sorensen, E. Kressel:
"A Proposal for a Multi-Programming BCPL System
On RIKKE-1".
DAIMI MD-19, Computer Science Department,
Aarhus University, Denmark, October 1975.

[4] E. Kressel, E. Lynning: "The I/O Nucleus on RIKKE-1".
DAIMI PB-21, Computer Science Department,
Aarhus University, Denmark, October 1975.

[5] E. Kressel, I. Holm Sorensen:
"The First BCPL System on RIKKE-1".
DAIMI PB-17, Computer Science Department,
Aarhus University, Denmark, August 1975.

[6] M. Richards: "BCPL: A tool for compiler writers
and system programming".
Spring Joint Computer Conference, 1969.

[7] M. Richards: "The BCPL Reference Manual".
Technical Memorandum No.69/1-2,The Computer Laboratory,
Corn Exchange Street, Cambridge, England.

[8] E.I. Organick: "Computer System Organization".
Academic Press 1973.

[9] M.J. Spier: "Lecture Notes on Operating Systems".
Computer Science Department,
Aarhus University, Denmark, December 1977.

[10] J.L. Peterson: "Petri Nets".
Computing Surveys, Vol.9, No.3, September 1977.

[11] P. Kornerup:
"The Wide Store on the RIKKE/MATHILDA System".
(DAIMI internal document), November 1974.

[12] M.J. Spier, I. Holm Sorensen:
"Advanced Process Synchronization".
Lecture notes, Computer Science Department,
Aarhus University, Denmark, November 1977.

[13] I. Holm Sorensen, P. Kornerup:
"WIDESTORE, a sharable Memory for RIKKE/MATHILDA".
(DAIMI internal document), to be published.

[14] "DIABLO 4000, Data Manual".
Data Recording Instrument Company Limited, England.

[15] "Microprogramming in I/O"
Infotech State of the Art Lectures.
Microproramming and System Architecture,
3-5 april, 1973, London.

[16] H. Katzan, Jr.:
"Computer Organization and the System/370"

glossary

asynchronous : characteristic of processes that operate at arbitrary times, possibly determined by the action of other processes.

bus : group of wires that carry information.

channel : complex process that performs input/output transfers, usually for several peripheral devices.

clock : device that generates timing signals.

cyclic redundancy code : a checksum compiled from a sequence of data words.

direct memory access (DMA) : transfer of a block of information directly between an external device and memory without using a CPU service routine.

firmware : microprogram.

gate control : control of a logic device with one or more input and one output.

latency time : the time spent waiting for a specific disk sector to arrive under the heads.

index mark : pulse indicating that the next sector accessable on a disk device is sector 0.

multiplexor : a device that connects one of many inputs to an output.

parallel transfer : transfer of several bits of data at one time; transfer of all bits of one word at once.

sector mark : pulse indicating that the read/write head for a disk device is passing a sector gap.

serial transfer : transfer of data bits one at a time.

synchronous : characteristic of processes that operate at fixed time intervals, generally determined by a clock.

APPENDIX A.

The appendix contains a description of the processes realized in the Supporting Hardware, using the System Modelling Language.

A.1. Memory Bus Control Process (MBCP)

```
mailbox { BUF; PTRCLEAR; BTODSTATE; BTODPORT;
          DTOBSTATE; DTOBPORT; BTOMSTATE;
          BTOMPORT; MTOBSTATE; MTOBPORT }

manifest { sa=0; da=1; dclear=0; mclear=1;
           done=2; a=0; b=1 }

process MBCP is
{mbcp
   let Buf = 0
   let Ptrclear = 0
   let Bufa = vec 255
   and Bufaptr = 0
   let Bufb = vec 255
   and Bufbptr = 0
   let Data = 0

priority:
   if valof
   {clearcond
   << if PTRCLEAR=dclear \/ PTRCLEAR=mclear do
      { Buf := BUF
      Ptrclear := PTRCLEAR
      resultis true } >>
      or resultis false
   }Clearcond  goto clearaction

   if valof
   {BtoDcond
   << if BTODSTATE=sa do
      { Buf := BUF
        resultis true } >>
      or resultis false
   }BtoDcond goto btodaction
```

```
        if valof
        {DtoBcond
        << if DTOBSTATE=da do
           { Buf := BUF
             DTOBSTATE := sa
             Data := DTOBPORT
             resultis true } >>
           or resultis false
        }BtoDcond goto dtobaction
        if valof
        {BtoMcond
        << if BTOMSTATE=sa do
           { Buf := BUF
             resultis true } >>
           or resultis false
        }BtoMcond goto btomaction

        if valof
        {MtoBcond
        << if MTOBSTATE=da do
           { Buf := BUF
             Data := MTOBPORT
             MTOBSTATE := sa
             resultis true } >>
           or resultis false
        }MtoBcond goto mtobaction
        goto priority

    clearaction:
        switchon Ptrclear into
        {
          case mclear: (Buf=a -> Bufbptr, Bufaptr) := 0
          case dclear: (Buf=a -> Bufaptr, Bufbptr) := 0
        }
        << PTRCLEAR := done >>
        goto priority

    btodaction:
        if Buf=a do
        { Data := Bufa!Bufaptr
          Bufaptr := Bufaptr+1
        }  or
        { Data := Bufb!Bufbptr
          Bufbptr := Bufbptr+1
        }
        << BTODPORT := Data
           BTODSTATE := da >>
        goto priority
```

```
dtobaction:
    if Buf=a do
    { Bufa!Bufaptr := Data
      Bufaptr := Bufaptr+1
    }  or
    { Bufb!Bufbptr := Data
      Bufbptr := Bufbptr+1
    }
    goto priority
btomaction:
    if Buf=b do
    { Data := Bufa!Bufaptr
      Bufaptr := Bufaptr+1
    }  or
    { Data := Bufb!Bufbptr
      Bufbptr := Bufbptr+1
    }
    << BTOMPORT := Data
       BTOMSTATE := da >>
    goto priority

mtobaction:
    if buf=b do
    { Bufa!Bufaptr := Data
      Bufaptr := Bufaptr+1
    }  or
    { Bufb!Bufbptr := Data
      Bufbptr := Bufbptr+1
    }
    goto priority
}mbcp
```

A.2. Disk Read Write Controller (DRWC)

```
mailbox{ DOP; DWC; DISKOP; BTODSTATE;
           BTODPORT; DTOBSTATE; DTOBSTATE }

manifest { sa=0; da=1; read=0; write=1; done=2
           ready=2; crcerr=6; limit=256 }

Process DRWC is
{drwc
   let Dop = 0
   let Data = 0
   let Crcword = 0
   let Crcgenerate(a,b)= { }
       //local function generating a new
       //cyclic redundancy word
   let Next16bit() = valof { }
       //local function returning the next 16 bits
       //from disk surface
   let Out16bit() be { }
       //local routine to record 16 bits
       //on disk surface
   let Checkcrc(a,b) = valof { }
       //local function returning false if the cyclic
       //redundancy ckeck failed else true

   while valof
   {readcond
   << if (DOP=read /\ DWC#limit /\ DTOBSTATE=sa)
      resultis true >>
      or resultis false
   }readcond  do
   {readaction
     Data := Next16bit()
     Crcword := Crcgenerate(Data,Crcword)
     << DTOBPORT := Data
        DTOBSTATE := da
        DWC := DwC+1 >>
   }readaction
```

```
while valof
{writecond
<< if (DOP=write /\ DWC≠limit /\ BTODSTATE=da) do
   { Data := BTODPORT
     BTODSTATE := sa
     resultis true  } >>
   or resultis false
}writecond   do
{writeaction
  Out16bit(Data)
  Crcword := Crcgenerate(Crcword,Data)
  << DWC := DWC+1 >>
}writeaction

while valof
{termcond
<<if DWC=limit /\ DOP≠done do
  Dop := DOP
  resultis true  } >>
  or resultis false
}termcond  do
{termaction
  let status=ready

  if DOP=write do Out16bit(Crcword)
  or
    if CheckCrc(Crcword,Next16bit()) do
       Status := ready
    or Status :=crcerr
  << DOP := done
     DISKOP := Status >>
}termaction
```

A.3. Memory Read Write Control (MRWC)

```
mailbox { MOP; MWC; MEMOP; BTOMSTATE;
          BTOMPORT; MTOBSTATE; MTOBPORT }

manifest { sa=0; da=1; read=0; write=1;
           done=2; ready=3; limit=256 }

process MRWC is
{mrwc
  let Mop=0
  let Data=0
  let Nextword() = valof {}
      //local function returning next word
      //from main memory
  let Outword(w) be {}
      //local routine sending word w
      //to main memory


  while valof
  {readcond
  << if (MOP=read /\ MWC#limit /\ MTOBSTATE=sa)
     resultis true >>
     or resultis false
  }readcond  do
  {readaction
    Data := Nextword()
    << MTOBPORT := Data
       MTOBSTATE := da
       MWC := MWC+1 >>
  }readaction
```

```
  while valof
  {writecond
  << if (MOP=write /\ MWC#limit /\ BTOMSTATE=da) do
     { Data := BTOMPORT
       MTOBPORT := sa
       resultis true } >>
     or resultis false
  }writecond  do
  {writeaction
    Outword(Data)
    << MWC := MWC+1 >>
  }writeaction

  while valof
  {termcond
  << if MWC=limit /\ MOP#done do
     { MOP := done
       MEMOP := ready
       resultis true } >>
     or resultis false
  }termcond
}mrwc  repeat
```

A.4. Disk Channel Administration Process (DCAP)

```
mailbox { DISKOP; SECTOR; MARK; CLEARPTR;
          DOP; DWC; BTODSTATE; DTOBSTATE }

manifest { sectormark=0; indexmark=1; done=2;
           read=0; write=1; sa=0; dclear=0 }

process DIAD is
{diad
  let Mark=0
  let Sectorno=0
  let Diskop=0

  until valof
  {markcond
  << if MARK=sectormark \/ MARK=indexmark do
     { Mark := MARK
       resultis true } >>
     resultis false
  }markcond loop
  {markaction
    if Mark=sectormark do Sectorno := Sectorno+1
    or sectorno := 0
    << MARK := done >>
  }markaction

  if valof
  {matchcond
  << if (DISKOP=read /\ SECTOR=Sectorno) \/
        DISKOP=write /\ SECTOR=Sectorno) do
     { CLEARPTR := dclear
       Diskop := DISKOP
       resultis true } >>
     or resultis false
  }matchcond  do
  {matchaction
    << BTODSTATE := sa
       DTOBSTATE := sa
       DWC := 0
       DOP := Diskop >>
  }matchaction
}diad  repeat
```

A.5. Memory Channel Administration Process (MCAP)

```
mailbox { MEMOP; MEMADR; CLEARPTR; MCP;
          MWC; MTOBSTATE; BTOMSTATE }

manifest { read=0; write=1; busy=2; sa=0; mclear=1 }

process MEAD is
{mead
  let Memop=0
  let Memadr=0

  if valof
  {memcond
  << if MEMOP=read \/ MEMOP=write do
     { Memop := MEMOP
       Memadr := MEMADR
       CLEARPTR := mclear
       resultis true } >>
     or resultis false
  }memcond do
  {memaction
    let Startblockread(a) be { }
    let Startblockwrite(a) be { }
        //local routine which starts the block
        //processing unit inside Main Memory

    if Memop=read do Startblockread(Memadr)
    or Startblockwrite(Memadr)
    << MTOBSTATE := sa
       BTOMSTATE := sa
       MWC := 0
       MCP := Memop >>
  }memaction
}mead  repeat
```

A.6. Arm Positioning Process (APP)

```
mailbox { SEEKOP; CYL }

manifest{ seek=1; att=2; illadr=5;
          incompl=4; ready=3 }

process APP is
{app
  let Cyl=0

  if valof
  {seekop
  << if SEEKOP=seek do
     { Cyl := CYL
       SEEKOP := attention
       resultis true } >>
    or resultis false
  }seekcond do
  {seekaction
    let Status=0
    let Seek(c) = valof  { }
        //local routine to position arms.
        //returns status for Disk-hardware
        //on termination

    Status := Sek(Cyl)
    << SEEKOP := Status >>
  }seekaction
}app  repeat
```

A.7. Disk Status Monitor Process (DSMP)

```
mailbox { STATUS; MEMOP; DISKOP; SEEKOP }

manifest { ready=3; incompl=4;
           illadr=5; crcerr=6 }

process DSMP is
{dsmp
  let Seekop,Memop,Diskop=0,0,0

  << Seekop := SEEKOP
     Memop := MEMOP
     Diskop := DISKOP >>
  {statusaction
    let status=0
    let Diskstatus() = valof { }
        //local function returning Disk-status
        //ready or not ready
    let Putcond(cond,word,pos) be { }
        //local routine to put condition as
        //a bit into word at position pos

    Putcond(Seekop=ready,lv status,0)
    Putcond(Diskop=ready,lv status,1)
    Putcond(Memop=ready,lv status,2)
    Putcond(Diskop=crcerr,lv status,3)
    Putcond(Seekop=incompl,lv status,4)
    Putcond(Seekop=illadr,lv status,5)
    Putcond(Diskstatus(),lv status,6)
    << STATUS := Status >>
  }statusaction
}dsmp  repeat
```

APPENDIX B.

This appendix contains a description of the processes realized on the firmware level in the Disk Controller.

B.1. Channel Traffic Coordinator (CTC)

```
mailbox{ QUEUESTATE; DRIVE; READSIGNAL;
         RWORK; WRITESIGNAL; WWORK;
         SEEKSIGNAL; SWORK }

manifest{ unlock=1; lock=0; seeking=-1;
          read=0; write=1; seek=2;
          nothing=3; free=0; start=1
          newread=4; newwrite=5    }

process CTC is
{ctc
  let Readwork=0
  let writework=0
  let Seekwork=0
  let ChoosefromQueue() = valof { }
      //local function returning next kind of
      //job from queue, being:
      //read; write; seek; nothing; newread; newwrite.
      //Newread is returned if a new read request is
      //following a write request; if a write comes
      //after a read newwrite is returned.

  let ResetNFQ() be {}
      // Resets Queue information preventing
      // ChoosefromQueue from returning the values
      // newwrite or newread

  let RemovefromQueue() = valof { }
      //local function returning the pointer to
      //the next request block (equivalent to the
      //block accesed in previous call of
      //ChoosefromQueue; in addition the request
      //block is removed from the queue chain.
```

```
    <<QUEUESTATE := lock>>
    switchon ChoosefromQueue() into
    {
      case read: <<if READSIGNAL=free /\
                       DRIVE#seeking do>>
                 {
                   Readwork := RemovefromQueue()
                   << Rwork := Readwork
                      DRIVE := DRIVE+1
                      QUEUESTATE := unlock
                      READSIGNAL := start >> }
                 endcase

      case write: <<if WRITESIGNAL=free /\
                       DRIVE#seeking do>>
                  { writework := RemovefromQueue
                    << WWORK := Writework
                       Drive := DRIVE+1
                       QUEUESTATE := unlock
                       WRITESIGNAL := start >> }

      case seek: <<if SEEKSIGNAL=free /\ DRIVE=0 do>>
                 { Seekwork := RemovefromQueue()
                   << SWORK := Seekwork
                      DRIVE := seeking
                      QUEUESTATE := unlock
                      SEEKSIGNAL := start >> }
                 endcase

      case newread:
                 << if DRIVE=0 do >> ResetNFQ()
                 endcase
      case newwrite:
                 << if READSIGNAL=free do >> ResetNFQ()
                 endcase

      case nothing:
    }
    <<QUEUESTATE := unlock>>
  }ctc  repeat
```

B.2. The Drive Arm-positioning Handler (DAPH)

```
mailbox{ DRIVE; SEEKSIGNAL; SWORK }

manifest{ free=0; start=1 }

process DAPH is
{daph
  let Swork=0
  let Moveheads(cyl) be { }
      //local function realized
      //in the supporting hardware; access is made
      //to SEEKOP and STATUS(0)

  if valof
  {seekcond
  <<if SEEKSIGNAL=start do
    { Swork := SWORK
      SEEKSIGNAL := free
      resultis true>> }
    or resultis false
  }seekcond  do
  {seekaction
    Moveheads(Swork)
    << DRIVE := 0 >>
  }seekaction
}daph  repeat
```

B.3. The Disk Read/Write Controller (DRWC)

```
mailbox{ DRIVE; READSIGNAL; RWORK; ECREQUEST;
         EORSTATUS; DISKKSTATE; DPARAM }

manifest{ free=0; start=1; term=1; idle=0;
          busy=1; dtom=2; mtod=3; diskpart=0;
          statuspart=2; compl=1 }

process DRWC1 is
{drwc1
  let Dparam=0
  let DiskReadHandler(sector) = valof { }
      //local function which utilize the access
      //possibilities (DISKOP; STATUS(1) and STATUS) to
      //the supporting hardware to perform disk-
      //operations. The result of a read (crcerr etc.)
      //is returned.

  if valof
  {readcond
  << if READSIGNAL=start /\ DISKSTATE=idle do
     { Dparam := RWORK
       READSIGNAL := free
       DISKSTATE := busy
       resultis true >> }
     or resultis false
  }readcond  do
  {readaction
    Dparam!statuspart:=DiskReadHandler(Dparam!diskpart)
    << DRIVE := DRIVE-1
       DPARAM := Dparam
       DISKSTATUS := dtom >>
  }readaction
}drwc1  repeat
```

```
process DRWC2 is
{drwc2
  let Dparam=0
  let DiskWriteHandler(sector) be { }
      //local function, which utilize the access
      //possibilities (DISKCP; STATUS(1) and STATUS) to
      //the supporting hardware to perform
      //disk-operations.

  until valof
  {writecond
    << if DISKSTATE=mtod do
       { Dparam := DPARAM
         resultis true >> }
       or resultis false
  }writecond loop
  {writeaction
    DiskWriteHandler(Dparam!diskpart)
    << DRIVE := DRIVE-1
       DISKSTATE := idle >>
  }writeaction

  until valof
  {termcond
    << if EOREQUEST=free do
          EOREQUEST := busy
          resultis true >>
       or resultis false
  }termcond loop

  {termaction
    << EORSTATUS := compl
       EOREQUEST := term >>
  }termaction
}drwc2  repeat
```

B.4. The Memory Read/Write Controller (MRWC)

```
mailbox{ WRITESIGNAL; WWORK; MPARAM;
         MEMSTATE; EORequest; EORstatus }

manifest{ free=0; start=1; term=1; idle=0;
          busy=1; dtom=2; mtod=3; statuspart=2;
          mempart=1 }

process MRWC1 is
{mrwc
  let Mparam=0
  let MemReadHandler(addr) be { }
      //local function, utilizing access possibilities
      //(MEMOP; STATUS(2)) to the supporting hardware.

  if valof
  {readcond
  << if WRITESIGNAL=start /\ MEMSTATE=idle do
  { Mparam := WWORK
    MEMSTATE := busy
    WRITESIGNAL := free
    resultis true >> }
  or resultis false
  }readcond do
  {readaction
    MemReadHandler(Mparam!mempart)
    << MPARAM := Mparam
       MEMSTATE := mtod >>
  }readaction
}mrwc1  repeat
```

```
process MRWC2 is
{mrwc2
  let Mparam=0
  let MemWriteHandler(addr) = valof { }
      //local function, utilizing access possibilities
      //(MEMOP; STATUS(2)) to the supporting hadware.
      //additionally MemWriteHandler performs
      //header-check to ensure that the requested
      //sector is the one received from
      //disk- hardware.

  until valof
  {writecond
  << if MEMSTATE=dtom do
     { Mparam := MPARAM
       resultis true >> }
    or resultis false
  }writecond loop

  {writeaction
    Status := MemWriteHandler(Mparam!mempart)
    << DISKSTATE := idle >>
  }writeaction

  until valof
  {termcond
    << if EOREQUEST=free do
          EOREQUEST := busy
          resultis true >>
       or resultis false
  }termcond loop
  {termaction
    Status,Mparam := Status \/ Mparam!statuspart
    << EORSTATUS := Status
       EOREQUEST := term >>
  }termaction
}mrwc2  repeat
```

B.5. The Channel Traffic Govenor. (CTG)

```
mailbox{ DISKSTATE; MEMSTATE;
         DPARAM; MPARAM }

manifest{ idle=0; busy=1; dtom=2; mtod=3 }

process CTG is
{ctg
  let Mparam,Dparam=0,0
  let Memstat,Diskstat=0,0
  let Flipbuf() be { }
      //local routine to "swap" the local
      //buffers of the supporting hardware,
      //i.e. the buffer allocated for memory side
      //operations will be allocated for the next
      //disk side operation and vice versa.

  if valof
  {flipcond
    if << (DISKSTATE=idle /\ MEMSIDE=mtod) \/
          (DISKSTATE=dtom /\ MEMSIDE=mtoc) \/
          (DISKSTATE=dtom /\ MEMSIDE=idle) do
    { Mparam := MPARAM
      Dparam := DPARAM
      Diskstate := DISKSTATE
      Memstate := MEMSTATE
      DISKSTATE := busy
      MEMSTATE := busy
      resultis true >> }
    or resultis false
  }flipcond  do
  {flipaction
    Flipbuf()
    << MPARAM := Dparam
       DPARAM := Mparam
       MEMSTATE := Diskstate
       DISKSTATE := Memstate
  }flipaction
}ctg  repeat
```

B.6 Interrupthandlers.

```
eorreadaction:
   EORSTATUS := STATUS!statuspart
   EOREQUEST := term  //Notify signal to user (OS)
   goto Poll

eorwriteaction
   EORSTATUS := compl
   EOREQUEST := term  //Notify signal to user (OS)
   goto Poll

termmemwrite:
   MEMSTATE := idle
   STATUS := MPARAM
   TERMREAD := pending
   goto Poll

termdiskwrite:
   DISKSTATE := idle
   TERMWRITE := pending
   DRIVE := DRIVE-1
   goto Poll

startmemwrite:
   Checkheader()
   MEMADR := MPARAM!mempart  //signal the supporting
   MEMOP := write            //hardware.
   MEMSTATE := writing
   goto Poll

startdiskwrite:
   SECTOR := DPARAM!diskpart  //signal the supporting
   DISKOP := write            //hardware.
   DISKSTATE := writing
   goto Poll

flipaction:
   swap(DISKSTATE,MEMSTATE)
   swap(DPARAM,MPARAM)
   swapbuffer()

termmemread:
   MEMSTATE := mtod
   goto Poll

termdiskread:
   DISKSTATE := dtom
   DRIVE := DRIVE-1
   goto Poll
```

```
startmemread:
   MEMADR := WWORK!mempart     //signal the supporting
   MEMOP := read               //hardware.
   MPARAM := WWORK
   WRITESIGNAL := free
   MEMSTATE := reading
   goto Poll

startdiskread:
   SECTOR := RWORK!diskpart    //signal the supporting
   DISKOP := read              //hardware.
   DPARAM := RWORK
   READSIGNAL := free
   DISKSTATE := reading
   goto Poll

startDisktoMem:
   RWORK := RemovefromQ()
   NEXTREQUEST := ChoosefromQ() //can also be set by
                                //means of other
                                //functions, e.g.
                                //upstarts an empty
                                //Request Queue.
   READSIGNAL := start
   DRIVE := DRIVE+1
   goto Poll

startMemtoDisk:
   Wwork := RemovefromQ()
   NEXTREQUEST := ChoosefromQ()
   WRITESIGNAL := start
   DRIVE := DRIVE+1
   goto Poll

startseek:
   CYL := RemovefromQ()        //signal the supporting
   SEEKOP := seek              //hardware.
   DRIVE := seeking
   NEXTREQUEST := ChoosefromQ() //first request in
                                //new Track-Queue.

termseek:
   DRIVE := 0
```