MASTER

Designing a token ring LAN controller with IDaSS

Leermakers, H.P.

Award date:
1990

Link to publication

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

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

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

by H.P. Leermakers

Supervisor: Prof. ir. M.P.J. Stevens
Coach: ir. A.C. Verschueren

The department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student-project and graduation reports.
ABSTRACT

This report describes the designing of a part of a Token Ring LAN controller according to the ANSI/IEEE standard 802.5. The LAN controller is intended to be part of a Multi-processor Multitasking Co-processor (MMTCP) which is currently being developed at the Digital Systems group. The final LAN controller design is made with IDaSS, an interactive design environment for complex synchronous digital circuits.

As a summary of the preliminary work done so far, this report first introduces multitasking, the MMTCP basics and the choice of the LAN controller. Next, the ANSI/IEEE 802 standard family and their relations to the ISO/OSI Reference Model are explained. The Priority Operation protocol of the Token Ring is explained by means of an example.

While a set of sub-problems is gained to clearly define the actual design problem, a one bit time delay per attached LAN station is stated to be an important goal. The information given by the standard is filtered and transformed into a more useful form of information (some sort of data flow chart) to make the mapping onto an IDaSS design easier.

The basics of IDaSS are then described, and for the purpose of consistent designing with IDaSS, a set of design rules is drawn up, based on experience with IDaSS. These design rules are general purpose rules that can be used by any designer that uses IDaSS.

Finally, the IDaSS design of the partial Token Ring LAN controller is described. The partial design implements the Priority Operation protocol core, and it is well prepared to be extended as to fully implement the Operational Mode of the LAN controller, complete with error-handling. Monitor functions can also be added without any problems. The achieved delay per LAN-attached station is no more than one bit time, and merely depends on the hardware technology.

Cover: the illustration shows the screen view during a simulation session of the designed partial Token Ring LAN controller.
ACKNOWLEDGEMENTS

At this place I would like to thank my supervisors Prof. ir. M.P.J. Stevens and ir. A.C. Verschueren for guiding me through my master thesis project and for giving me the opportunity to enrich my knowledge and test my skills at such a challenging project.
# TABLE OF CONTENTS

**ILLUSTRATIONS AND TABLES** ................................................................. 5  
**LIST OF ABBREVIATIONS** ............................................................................ 7  
**PREFACE** .................................................................................................... 9  
**1 INTRODUCTION** .......................................................................................... 10  
**2 THE MULTI-PROCESSOR MULTITASKING CO-PROCESSOR** ......................... 11  
  2.1 Multitasking ............................................................................................... 11  
  2.2 Multi-processor architectures .................................................................... 12  
  2.3 MM TCP implementation ............................................................................ 14  
  2.4 Choice of a communication controller for the MM TCP ............................... 14  
**3 TOKEN RING** ............................................................................................. 16  
  3.1 ISO/OSI model and ANSI/IEEE 802 standard ............................................. 16  
  3.1.1 The OSI Reference Model ...................................................................... 16  
  3.1.2 The ANSI/IEEE 802 standard family ...................................................... 17  
  3.1.2.1 Sublayer relations ............................................................................. 17  
  3.1.2.3 Logical Link Control Sublayer ......................................................... 18  
  3.1.2.4 Medium Access Control Sublayer .................................................... 19  
  3.1.2.5 Physical Layer ................................................................................... 20  
  3.2 Token Ring Access Method ........................................................................ 21  
  3.2.1 Introduction ............................................................................................. 21  
  3.2.2 General description ............................................................................... 21  
  3.2.3 Token and Frame formats ...................................................................... 23  
  3.2.4 Priority operation .................................................................................. 24  
  3.2.5 Priority operation example .................................................................... 26  
  3.2.6 Considerations on the implementation of the protocol ............................ 32  
**4 DESIGN STEPS** .......................................................................................... 34  
  4.1 Problem definition ...................................................................................... 34  
  4.2 Problem tackling ......................................................................................... 35  
**5 IDaSS** .......................................................................................................... 37  
  5.1 General description ..................................................................................... 37  
  5.2 Limitations of IDaSS and their effect on the design .................................... 38  
  5.2.1 Synchronous machines ........................................................................ 38  
  5.2.2 Communication between levels ............................................................. 39  
  5.3 Choices within IDaSS ................................................................................ 41  
  5.3.1 Hardware versus software description .................................................. 41  
  5.3.2 Design rules ............................................................................................ 42  
**6 THE TOKEN RING DESIGN** ...................................................................... 46  
  6.1 Introduction .................................................................................................. 46  
  6.2 Signals .......................................................................................................... 46  
  6.3 Level 1 (TOP LEVEL): DATALINK ............................................................. 46  
  6.4 Level 2: TOKNRING .................................................................................. 49  
  6.5 Level 3: MAC2LLC ..................................................................................... 51  
  6.6 Level 3: PHY2MAC ..................................................................................... 52  
  6.7 Level 3: PHYSICAL .................................................................................... 62  
  6.8 Level 4: DETECTORS ............................................................................... 63  
  6.9 Level 4: ERROR .......................................................................................... 66
6.10 Level 4: PRIORITIES ......................................................... 69
6.11 Level 4: RECEIVE ............................................................. 69
6.12 Level 4: SEND ................................................................. 70
6.13 Level 4: STACKS ............................................................... 73
6.14 Level 4: TIMERS ............................................................... 75
6.15 Level 5: P ......................................................................... 78
6.16 Level 5: R ......................................................................... 79
6.17 Level 5: ED_DET ............................................................... 80
6.18 Level 5: SD_DET ............................................................... 81
6.19 Level 6: DATA/NDATA .................................................... 83

7 CONCLUSIONS AND REMARKS ........................................... 84

REFERENCES ................................................................. 85

APPENDIX A : THE OPERATIONAL FINITE STATE MACHINE. .......... 86

APPENDIX B : DATA FLOW_CHARTS FOR THE OPERATIONAL FSM. 87
ILLUSTRATIONS AND TABLES

FIGURES

Figure 1: Multitasking: 'several jobs running on a single processor'. .......................... 12
Figure 2: Stations connected by a LAN route packets that are intended for other stations.  
These packets may or may not be originated by the routing (transmitting) stations  
(see station B). ........................................................................................................ 13
Figure 3: Structure of the MMTCP with the communication controller and its connection to  
the host system. ...................................................................................................... 15
Figure 4: The ISO/OSI Reference Model. ......................................................................... 16
Figure 5: Relations between the ISO/OSI Reference Model and the 802 standard  
family. ....................................................................................................................... 18
Figure 6: A Token Ring with 8 stations. Station B is physically disconnected from the ring.  
........................................................................................................................................ 22
Figure 7: Token and Frame formats. .................................................................................. 23
Figure 8: The fields in a token and in frames that are most important for the partial LAN  
controller design. ..................................................................................................... 24
Figure 9: Example - step 1. Token is at point shown. ....................................................... 26
Figure 10: Example - step 2. Station A captures token and begins transmission of a frame. 
........................................................................................................................................ 27
Figure 11: Example - step 3. Station B repeats frame and changes RRR to 4. ................. 27
Figure 12: Example - step 4. Station C repeats frame and changes RRR to 6. ................. 28
Figure 13: Example - step 5. Station A strips frame from the ring. ................................. 28
Figure 14: Example - step 6. Station A releases a new token. .......................................... 29
Figure 15: Example - step 7. Station B repeats token and changes RRR to 4. ................. 29
Figure 16: Example - step 8. Station C captures token and begins transmission of frame. 
........................................................................................................................................ 29
Figure 17: Example - step 9. Frame circulates with no modification. ............................... 30
Figure 18: Example - step 10. Station C begins stripping the frame from the ring. .......... 31
Figure 19: Example - step 11. Station C issues a new token of same priority. ................. 31
Figure 20: Example - step 12. Station A performs a REPLACE and transmits the token.  
........................................................................................................................................ 31
Figure 21: Example - step 13. Station B captures token and transmits frame. ............... 31
Figure 22: Example - step 14. Station B strips its frame from the ring. ............................ 33
Figure 23: Example - step 15. Station B issues a new token at the original priority. ....... 32
Figure 24: Example - step 16. Station A POPs its stacks and issues priority-free token. ... 33
Figure 25: Timing for data at 4 Mbit/s bitrate. Notice the decreased station delay with the  
16 MHz system clock compared to the 4 MHz system clock. The circles denote the  
events that are used but not simulated within IDaSS. ............................................. 39

Figure 26: Communication and timing in case of an error event:  
a) Two state controllers act upon a signal. ................................................................. 40
b) Two state controllers communicate by means of an operator. ................................. 40
c) An operator reports the error to a state controller. ................................................. 40
Figure 27: Top level schematic DATALINK. .................................................................. 47
Figure 28: Level 2 schematic TOKNRING. ................................................................. 49
Figure 29: Level 3 schematic MAC2LLC. ..................................................................... 51
Figure 30: Level 3 schematic PHY2MAC. ..................................................................... 53
Figure 31: Level 3 schematic PHYSICAL. ..................................................................... 62
Figure 32: Level 4 schematic DETECTORS. .............................................................. 65
Figure 33: Level 4 schematic ERROR. ........................................................................... 66
Figure 34: Level 4 schematic PRIORITIES. ............................................................... 69
Figure 35: Level 4 schematic RECEIVE. ...................................................................... 70
Figure 36: Level 4 schematic SEND. ............................................................................ 71
Figure 37: Level 4 schematic STACKS. ........................................................................ 73
Figure 38: Level 4 schematic TIMERS.

Figure 39: Level 5 schematic P.

Figure 40: Level 5 schematic R.

Figure 41: Level 5 schematic ED_DET.

Figure 42: Level 5 schematic SD_DET.

Figure 43: Level 6 schematic DATA/NDATA.

TABLES

Table I: Physical layer coding/decoding scheme.

Table II: DECODER/CODER (de-)coding scheme.
**LIST OF ABBREVIATIONS**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Address recognized bit</td>
</tr>
<tr>
<td>AC</td>
<td>Access Control</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic/Logic Unit</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>C</td>
<td>Frame Copied bit</td>
</tr>
<tr>
<td>CSMA/CD</td>
<td>Carrier Sense Multiple Access with Collision Detection</td>
</tr>
<tr>
<td>DA</td>
<td>Destination Address</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>E</td>
<td>Error detected bit</td>
</tr>
<tr>
<td>ED</td>
<td>Ending Delimiter</td>
</tr>
<tr>
<td>FC</td>
<td>Frame Control</td>
</tr>
<tr>
<td>FCS</td>
<td>Frame Check Sequence</td>
</tr>
<tr>
<td>FS</td>
<td>Frame Status</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite-State Machine</td>
</tr>
<tr>
<td>I</td>
<td>Intermediate frame bit</td>
</tr>
<tr>
<td>IDaSS</td>
<td>Interactive Design and Simulation System</td>
</tr>
<tr>
<td>IEEE</td>
<td>the Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organization</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LLC</td>
<td>Logical Link Control</td>
</tr>
<tr>
<td>MA</td>
<td>My Address</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MMTCP</td>
<td>Multi-processor Multitasking Co-processor</td>
</tr>
<tr>
<td>NMT</td>
<td>Network Management</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>P</td>
<td>Priority</td>
</tr>
<tr>
<td>Pm</td>
<td>PDU Priority</td>
</tr>
<tr>
<td>Pr</td>
<td>Received Priority</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Unit</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical layer</td>
</tr>
<tr>
<td>PLA</td>
<td>Programmable Logic Array</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase-Locked Loop</td>
</tr>
<tr>
<td>R</td>
<td>Reservation</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-Only Memory</td>
</tr>
<tr>
<td>Rr</td>
<td>Received Reservation</td>
</tr>
<tr>
<td>Sr</td>
<td>Highest Stacked Received Priority</td>
</tr>
<tr>
<td>Sx</td>
<td>Highest Stacked Transmitted Priority</td>
</tr>
<tr>
<td>SA</td>
<td>Source Address</td>
</tr>
<tr>
<td>SD</td>
<td>Starting Delimiter</td>
</tr>
<tr>
<td>SFS</td>
<td>Start-of-Frame Sequence</td>
</tr>
<tr>
<td>T</td>
<td>Token bit</td>
</tr>
<tr>
<td>TAM</td>
<td>Timer, Active Monitor</td>
</tr>
<tr>
<td>THT</td>
<td>Timer, Holding Token</td>
</tr>
<tr>
<td>TNT</td>
<td>Timer, No Token</td>
</tr>
<tr>
<td>TRR</td>
<td>Timer, Return to Repeat</td>
</tr>
<tr>
<td>TSM</td>
<td>Timer, Standby Monitor</td>
</tr>
<tr>
<td>TQP</td>
<td>Timer, Queue PDU</td>
</tr>
<tr>
<td>TVX</td>
<td>Timer, Valid Transmission</td>
</tr>
<tr>
<td>ULSI</td>
<td>Ultra Large Scale Integrated (circuits)</td>
</tr>
<tr>
<td>VCO</td>
<td>Voltage Controlled Oscillator</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integrated (circuits)</td>
</tr>
</tbody>
</table>
PREFACE

In 1986, Prof. ir. M.P.J. Stevens launched the idea of developing a multitasking operating system in hardware at the Digital Systems group. This resulted in the basic description of the Multi-processor Multitasking Co-processor (shorthand: 'MMTCP') by ir. A.C. Verschueren in 1987 ([1]). For interprocess communication between MMTCPs, an on-chip LAN controller of the Token Ring type was chosen (see reference [2]).

Meanwhile, ir. A.C. Verschueren has been working on IDaSS, which is shorthand for 'Interactive Design and Simulation System'. IDaSS is an environment in which synchronous systems can be developed in a top-down fashion, and it's therefore targeted towards complex systems like Ultra Large Scale Integrated circuits (ULSI).

Both the MMTCP and IDaSS lead to my project: designing a Token Ring LAN controller with IDaSS. Because designing the total controller would be a vast job in measures of time, focusing was done primarily on the possibilities of IDaSS (giving feedback to the IDaSS project), as well as giving a context for the design of the LAN controller. This context should be interpreted as a basic part of the LAN controller, combined with a lot of hints about how the design could (or should) be completed. It should be quite clear that extendibility was a major item in designing that basic part of the LAN controller: it forced the design steps to be optimized and optimized again, because new extendibility demands arose as each design step evolved. Because this optimizing has been proven to take a huge part of the total design time, the reader who is interested in further extending the LAN controller design is strongly recommended to study reference [3] thoroughly. The latter reference is the ANSI standard for the Token Ring LAN controller. Studying this standard should reduce the number of adaptations and optimizations to be made to the design at later stages.

Novice users of IDaSS will find this report to be an example of problem tackling with IDaSS. The reader that has already had experience with IDaSS will notice that the complexity of the accomplished partial design lies more in the (diversity of) communication and synchronization between design blocks than in the number of involved blocks. Anyhow, this report was not meant to be a study on IDaSS, but it may very well serve as a guideline for consistent designing with IDaSS.
1 INTRODUCTION

At the Digital Systems group, a Multi-processor Multitasking Co-processor (shorthand: 'MMTCP') is being developed currently. The MMTCP should be a hardware form of a multitasking operating system, enhanced with some multi-processor supporting features. As a part of the MMTCP, a Token Ring LAN controller should be developed according to the ANSI/IEEE standard 802.5.

Also, an Interactive Design and Simulation System (shorthand: 'IDaSS') for digital circuits is being developed at the Digital Systems group. IDaSS is an environment that is specifically targeted towards VLSI and ULSI designs of complex data processing synchronous circuits.

As a replenishment to both projects mentioned above, this report describes the development of (a part of) a Token Ring LAN controller with IDaSS.

First we will take a look at preliminary work done on the MMTCP and the choice of the LAN controller. This will give us the context of the system we should design.

Second we will take a glance at the Token Ring basics, introducing the ISO/OSI model, the OSI layers and the Token Ring sublayers. Also a brief description of the principles of the Token Ring Access Method will be given.

The next topics that will be discussed are the general design steps: problem definition (including design goals) and problem tackling (involving several descriptions of the required design and the choice of an appropriate starting point).

At that point we need to know the possibilities and limitations of IDaSS, so we will take a look at IDaSS from a designer's point of view to see if we can extract a useful set of design rules that can be used consistently. Such a set will more or less eliminate choices in the future regarding the (often many) different ways a design can be implemented with IDaSS.

The final chapter will then describe the partial design of the Token Ring controller made with IDaSS. An attempt is made to explain the relations between the different levels and blocks of the design. Also the decisions that were made will be justified. This chapter will also reach some suggestions on how the design could (or should) be completed. In some cases, special attention will be given to the interface with the MMTCP (which after all takes account for some high-level communication functions).

Conclusions and remarks will close the report.
2 THE MULTI-PROCESSOR MULTITASKING CO-PROCESSOR

2.1 Multitasking

High level programming languages provide means to divide a program into smaller subprograms that each handle part of the problem. These subprograms, called 'routines', are by themselves divided into smaller routines, and so on. The lowest level routines do the actual job.

A program generally consists of a main routine at the highest level that starts routines at lower levels (therefore called 'subroutines'), who in their turn use routines at levels beneath their's. Instead of speaking of a routine that 'uses' another routine, routines are said to be 'called' by other routines. This calling can be seen as a command that initiates the subroutine and that sometimes is accompanied with specific parameters that tell the called routine what data should be altered or what specific action(s) should be taken.

The benefits of the use of routines are legion: computer programs often contain a number of repeated actions that are essentially identical. In order to reduce duplicate program code, such an action can be done by a specific routine, which by definition can be called more than once. Also, big programs can be developed by several people, each writing a different part of the program. Each part will then be a specific routine that is responsible for a part of the main program. Only the interface of each routine has to be known by the other programmers. Furthermore, a program can be made easier to understand and to adapt, because the inner working of every subroutine can be kept hidden to higher level routines. In that way, if a program is to be adapted, it is likely to be that only one of the routines has to be adapted instead of the whole program. Last but not least, in some programs the structure of the problem to be solved is such that several sub-solutions can -or even should- be delivered at the same time. A single big program that is sequentially executed will not do the job here. On the other hand, if we could map the sub-solutions on routines and we give these routines means to run at the same time, the problem is solved.

The latter situation is often encountered in 'real world' processes (for instance in a chemical plant, where external changes have to be processed at 'real time', while other processes should be controlled and guarded at the same time). One could of course use a separate computer for each external process, but this would impose other problems in cases where these computers should communicate with each other (at real time speed!), thus shifting the problem. Instead, a single central computer is used which runs a control program that simulates the running of several programs at the same time (one for each process) and offers facilities for these programs to 'communicate' with each other. Such a program is referred to as a 'realtime multitasking executive'.

In practice, computer systems are also used to run several programs (possibly for different human users) at the same time. Some computers also communicate with each other using a (local area) network in order to use collective, expensive pieces of equipment like fast printers, large background stores and dedicated fast computers. In the latter two cases, the demand for 'real time' handling is replaced for protection of the computer system against mistakes and misuse of the human users. The principle of processor-sharing,however, stays.

Fig 1: The hardware of most processors is incapable of running several programs (or 'tasks') at the same time, so the multitasking executives must switch the processor between those tasks that want to make use of it. This 'task switching' thus is an algorithm that divides the processor time amongst the

1 Note: The text in this chapter is for the greater part based on reference [1]. The reader who is interested in more details about multitasking and about the MMTCP in particular, is encouraged to read reference [1]. Furthermore, references [5] and [6] give an insight into one of the most widely used multitasking operating systems nowadays; UNIX. In this chapter we will only give the basic description of multitasking and focus on the basics of the MMTCP which lead to the necessity of a communication controller.
tasks. The decision about which task may run at what time is one that is based on two algorithm types: priority driven algorithms and time slicing algorithms.

Figure 1: Multitasking: 'several jobs running on a single processor'.

With priority driven algorithms, tasks are given a priority, indicating the importance of the task. The task with the highest priority is allowed to run first, and only stops running if it has finished completely, if it has to wait for another task to finish, or if it has to wait for an 'event' in the external world.

With time slicing algorithms, tasks may run one at a time in a circular fashion. All tasks will run shortly within a certain fixed time period. Each task is given a specific fraction of the time period, after which a task switch is forced. The fraction of time given to each task depends on the total amount of processor time the task needs, and on how fast a task should complete its job.

Modern multitasking operating systems (like UNIX for example) are dedicated software programs that use a combination of both principles mentioned above.

2.2 Multi-processor architectures

With the advent of microprocessors, increasing the system throughput of complex data processing multitasking systems by boosting the speed of the processor has become more expensive than building a system comprised of several processors. In a multitasking environment, the tasks can now be distributed over several processors, approaching the ideal of running all tasks at the same time (one task per processor, as mentioned in paragraph 2.1 with the real time multitasking executive).

In a multi-processor environment, one can differentiate the processors so that each of them excels in a specific form of computation (for instance, some processors can be equipped with numeric coprocessors, others with array processors, et cetera). If the processors have a way to communicate with each other, then parts of programs that run on one specific processor that may benefit from using such expensive hardware at other processors, can be run on one of these extended processors. As the reader may have guessed, such communication facilities have indeed been developed. We distinct two forms of interprocessor communication: 'Bus-ed' processors and LAN (Local Area Network) interconnected processors.

Bus-ed processors are processors whose system buses are connected in such a way that they have access to a common area of working memory. This memory serves as a communication channel, as it is used to exchange messages between the processors. This way of communication can be made very fast. The only problem, however, is that the distance between the processors is severely limited.
With LAN interconnected systems, data transfers are done with specialized input/output devices that can deal with greater distances (up to a few kilometers). Data therefore must be transferred between the working memories and these devices. The data is transferred in so-called 'packets' that contain control, address and data fields. The control fields are used to tell the receiver what the packet contains, and as the name indicates, the control fields also contain information to control and safeguard the data flow. The address fields are used to direct a packet of data to the correct receiver(s), and the data field (also called the 'information field') contains the actual data.

Depending on the LAN structure, a packet may pass several processor systems before arriving at the destination system(s), and a packet may also take different routes in the network (see Fig 2). The hardware that handles the data transfer can generate an interrupt when a packet has been received and transferred into working memory, so that the processor can inspect the packet and act upon its contents.

![Figure 2: Stations connected by a LAN route packets that are intended for other stations. These packets may or may not be originated by the routing (transmitting) stations (see station B).](image)

Depending upon the amount of intelligence present in the LAN controllers, the attached processor (called the 'host-processor') is more or less involved in the data transfers. In some cases, the host processor must handle each byte received with an interrupt routine. At the other extreme, the LAN controller is capable of handling multiple buffers, check addresses, and handling the lowest levels of the link management and error recovery procedures.

If two tasks need to communicate with each other while they are running on systems that are connected to different LANs, then there are two ways to transfer packets from the sending system's LAN to the receiving system's LAN. The first is that the LANs have a computer system in common that can communicate with both of them. The second is that a computer system in one of the LANs is equipped to communicate with a similar system in the other LAN. Such extended computer systems are called 'bridges' or 'gateways', depending on whether or not they connect LANs of the same type. The term 'bridge' will be used to indicate both of them.

The reader may have noticed that addressing is a major key in LANs, and that addressing is most complicated at the bridges. The detection of packets intended for other LANs and the checking whether they can be forwarded or not should be done in the LAN controller hardware, because otherwise all packets should be received and passed to the host processor first for checking in software.
2.3 MMTCP implementation

At the Digital Systems group, the possibilities of building a multitasking operating system in hardware have been studied, leading to the basic description of the Multi-processor Multitasking C0-processor (MMTCP) ([1]).

The MMTCP is intended to be the hardware form of a multitasking operating system, coupled with a LAN controller to exchange messages between peer MMTCPs. These messages will make it possible to let tasks communicate as if they were running on the same processor while they may in fact be located in computer systems far apart.

The MMTCP will also be equipped with a local working memory, used to store the necessary data structures that go with each task for identification and such (task descriptors, semaphores, mailboxes and so on). This memory is also used to buffer LAN data packets and is inaccessible to the host processor.

An MMTCP can be instructed to act as a bridge. This is done by keeping a list of accessible LAN numbers in the working memory, and transferring any packet intended for such an accessible LAN to the local host’s memory using DMA (Direct Memory Access). The host will then be instructed to forward this packet to the correct remote LAN. At the receiving end, a bridge host uses DMA to transfer a received packet into a buffer in the memory of its MMTCP, after which this packet will be handled as if it was sent by a local MMTCP.

To be able to transfer large blocks of data efficiently using the MMTCP network, the MMTCPs should be equipped with hardware that connects to DMA controllers in the host system, making it possible to do 'end-to-end' DMA transfers of data blocks. Within the MMTCP, there should also be functions available to connect and disconnect a MMTCP and its LAN, to determine the location of a LAN failure and to check out which stations are present within the local LAN.

Chapters 2 and 3 of reference [1] summarize the functions provided by the MMTCP as well as the functional blocks which the MMTCP consists of.

2.4 Choice of a communication controller for the MMTCP

In reference [2], the considerations are described regarding the choice of a communication controller for the MMTCP.

In reference [2], it is decided that the required communication controller should have no direct connection to the host processor’s system bus (see Fig 3). The main reason for this is the increased data processing speed and the saving of processor time at the host processor, which results from the absence of such a direct connection. This is because with normal communication operations, interactions with the host processor are not necessary.

Furthermore, the communication controller should be on the MMTCP chip on account of costs for the total system hardware. Also, although only the highest (most abstract) layer of the communication controller will actually communicate with the MMTCP, all layers of the communication controller should be accessible by the MMTCP (see Fig 3). This is because all layers can then use some features of the MMTCP (memory, for instance) and can be tested separately by the MMTCP.

Last but not least, the MMTCP should contain one or more special building blocks to interface between every layer and the host processor to provide the network management functions (reporting errors, for instance). The reader will agree that most of the network management however should be done at the MMTCP, and if possible, at the communication controller itself. In that way, communication becomes more and more transparent to the host processor.
According to reference [2], a LAN controller of the Token Ring type is preferred to a Token Bus LAN controller or a CSMA/CD system ('Carrier Sense Multiple Access with Collision Detection'). All three systems use a standard, provide a priority scheme and offer flexibility in designing for a specific system capacity. The Token Ring controller however, in spite of being more complex than the other two controller mechanisms, offers the best overall performance, while the Token Ring controller's access time is acceptable.

The data signalling rate of the Token Ring LAN controller shall be 4 Mbit/s, while 16 Mbit/s should also be possible for future use.

The next chapter will introduce the Token Ring LAN controller standard and its relation to the ISO/OSI model.
3 TOKEN RING

3.1 ISO/OSI model and ANSI/IEEE 802 standard

3.1.1 The OSI Reference Model

The Reference Model of Open Systems Interconnection (OSI) is a layered network structure proposed by the International Standards Organization (ISO) ([7]). The purpose of the OSI model is to promote the development of worldwide data-communications standardization. Networks are partitioned into a series of layers to reduce their design complexity.

![Figure 4: The ISO/OSI Reference Model.](image)

The OSI model is illustrated in Fig 4. It has seven layers that each perform a well defined function, described below:

- **Physical layer**: Concerned with transmitting raw bits over a communication channel. The design issues here deal with mechanical, electrical and procedural interfacing with the subnet, in order to establish, maintain, and release physical connections between data link entries.

- **Data link layer**: The task of the data link layer is to transform the raw transmission facility into a line that appears free from transmission errors to the network layer. This is done by breaking the data that is to be transmitted into data frames (officially called 'protocol data units'; PDUs). In order to create and recognize frame boundaries, special bit patterns are attached to the beginning and the end of each frame. Thus, more specifically, the data link layer must also process the acknowledgement frames sent back by the receiver.

- **Network layer**: Controls the operation of the subnet: it accepts messages from the source host, convert them to packets, and determines how the packets are routed within the subnet.

- **Transport layer**: Accepts data from the session layer, splits it up into smaller units if necessary, passes these to the network layer, and ensures that the pieces all arrive correctly at the other end. Connections are established and deleted, message streams are multiplexed onto one physical channel and so on, to make efficient use of the network.

- **Session layer**: Is the user's interface to the network. Once the connection has been established with the destination address provided by the user, the session layer may manage the dialog, if the user has requested that service.
The presentation layer may provide the user's process with services as cryptographic transformations, text compression, terminal handling, and file transfer.

The content of the application layer is up to the individual user. It defines how processes communicate with one another and how they make use of the network-based services provided by the application layer.

The required communication controller has to implement the bottom two layers: the physical layer and the data link layer. The MMTCR will implement the network layer and the transport layer. The top three layers define how the network is used and is therefore implemented by the host system.

3.1.2 The ANSI/IEEE 802 standard family

3.1.2.1 Sublayer relations

As mentioned in paragraph 2.4, a Token Ring LAN controller was chosen to serve the MMTCR with interprocess communication services. The Institute of Electrical and Electronics Engineers (IEEE) has developed a family of standards for LANs, which are standardized by the American National Standards Institute (ANSI). This family of standards deals with the physical and data link layers as defined by the ISO/OSI Reference Model. The Data Link layer is separated into the Logical Link Control sublayer and the Medium Access Control sublayer, which will be described in the next paragraphs. In addition to these sublayers, a Network Management is defined that interfaces with each sublayer. The Token Ring LAN controller is described by the following ANSI/IEEE standard numbers:

802.1, which describes the relations between the LAN standards and The ISO/OSI Reference Model, 802.2, which describes the Logical Link Control (see reference [4]), and 802.5, which describes the Medium Access Control and the Physical layer (see reference [3]).

Fig 5 shows the partitioning of the bottom two layers of the OSI model in relation to the standards mentioned above. The next paragraphs will describe the elements of Fig 5.

---

![Relations between the ISO/OSI Reference Model and the 802 standard family.](image)

---

Note: all information given by reference [2] is identical to that given by reference [3], except for the Active Monitor FSM. Because Monitor functions will not be implemented yet, we will still use the common available reference [3].
3.1.2.2 Network Management

The Network Management (NMT) is the conceptual control element of a station which interfaces with all the layers of the station and is responsible for the setting and resetting of control parameters, obtaining reports of error conditions, and determining if the station should be connected or disconnected from the medium.

Because the NMT should interface with all the layers (See Fig 5), we should bear in mind that in our hardware implementation, the NMT will probably be nested throughout the different building blocks. In a proper design however, we should, whenever possible, center some management functions of the NMT in separate blocks. One could think of error-indicating bits that can be packed into an error word or in a special status word which can then easily be examined as a whole by other sublayers. And of course, because several management functions (like error reporting for instance) require a connection with the host system, the NMT should have an interface with the internal MMTCP buses. This interface will also serve to communicate with the higher level network management functions in the MMTCP that can’t or shouldn’t be implemented within the LAN controller.

Another possibility is to let the MMTCP do all the Network Management (through access via its bus, see paragraph 2.4). This will, however, give quite an overhead to the MMTCP, not even mentioning the design problems one could encounter at getting entrance into the numerous Token Ring LAN controller elements with the MMTCP bus. We therefore decide to do as much management as possible in the hardware of the Token Ring LAN controller itself.

Furthermore, because the LAN controller should be able to be programmed and tested by software in the host system, some other controller elements should also be accessible via the MMTCP. If possible, these elements therefore also should be grouped together.

At this point, we conclude that we in fact have to design a Token Ring LAN controller that is not fully integrated within the MMTCP, but one that is more or less a stand-alone version that has an interface with the MMTCP through the internal MMTCP memory access and message buses and maybe through additional control buses too. As far as the NMT is concerned, as much management as possible should be done in hardware in the Token Ring LAN controller itself.

3.1.2.3 Logical Link Control Sublayer

The Logical Link Control (LLC) sublayer constitutes the top sublayer in the Data Link layer and is common to the various medium Access Methods that are defined and supported by the IEEE/Std 802 activity. Together with the Token Ring Medium Access Control (MAC) sublayer which will be described in the next paragraph, it completes the functionality of the Data Link layer.

The LLC is that part of a data station that supports the logical link control functions of one or more logical links. The LLC generates command PDUs and response PDUs for transmission, and interprets received command PDUs and response PDUs.

Specific responsibilities assigned to a LLC include:
1) Initiation of control signal interchange
2) Organization of data flow
3) Interpretation of received command PDUs and generation of appropriate response PDUs
4) Actions regarding error control and error recovery functions in the LLC sublayer.

The LLC sublayer provides interface services to the Network Layer (layer 3), to the MAC sublayer, and to the [LLC Sublayer Management function].

The interface service specification to the Network Layer provides a description of the various services that the LLC, plus underlying layers and sublayers, offer to the Network Layer, as viewed from the Network Layer. These services will probably end up as a form of pure hardware communication
between the LAN controller and the MMTCP (it should be clear now that the MMTCP constitutes the Network Layer and maybe a part of the Network Management Layer). One could, for instance, think of interrupts to the MMTCP in case of the correct reception of a PDU (as mentioned in paragraph 2.2). This interrupt will be accompanied with additional data to tell the MMTCP where to find the PDU that is received. The information and address data can then be extracted by the MMTCP via its bus, preferably by DMA methods.

The interface service specification to the MAC sublayer provides a description of the services that the LLC sublayer requires of the MAC sublayer, independent of the form of the medium Access Methodology, and of the nature of the medium itself. Let us keep in mind that we are supposed to design both the LLC and MAC sublayers, so we will not be restricted by a sharp distinction between the two sublayers. Whenever appropriate, we might choose to join parts of both sublayers.

The interface specification to the LLC Sublayer Management function provides a description of the management services that are provided to the LLC sublayer. Because this interface specification is the subject of further ongoing study and resolution ([4]), we will integrate the LLC Sublayer Management functions with the NMT.

All of the above Interface Service Specifications are given by reference [4] in the form of primitives that represent the logical exchange of information and control in an abstract way. They do not specify or constrain the implementation of entities or interfaces. In order to get familiar with the LLC sublayer, the reader is encouraged to read reference [4].

3.1.2.4 Medium Access Control Sublayer

The Medium Access Control (MAC) sublayer is that part of a data station that supports the medium access control functions that reside just below the Logical Link Control sublayer. The MAC procedures include framing/deframing of data units, performing error checking, and acquiring the right to use the underlying physical medium.

The MAC sublayer provides interface services to the LLC sublayer, to the Physical (PHY) layer, and to the Network Management (NMT) (see Fig 5).

The MAC to LLC Service specifies the services required of the MAC sublayer by the LLC to allow the local LLC sublayer entity to exchange LLC data units with peer LLC sublayer entities.

The PHY to MAC Service specifies the services provided by the PHY layer that allow the local MAC sublayer entity to exchange MAC data units with peer MAC sublayer entities.

The MAC to NMT Service specifies the services provided at the boundary between the network management and the MAC sublayer. This interface is used to monitor and control the operations of the MAC sublayer. We see that our wish to center network management functions (see paragraph 3.1.2.2) is supported by reference [3] in the form of this MAC to NMT Service.

All services mentioned above are requested through primitives that are defined by chapter 5 of reference [3] in an abstract way: each service names the particular primitive and the required information that is passed between a sublayer and the MAC sublayer. The exemplary form in which the primitives are specified resemble procedure headers that are encountered at high-level programming languages: the name of the procedure that is called, together with the variables. Therefore we will call this a 'procedural description of the service specifications'. We recognize here that the MAC standard gives us a hint on how to group several functional elements together! Later on, as we will study the Token Ring Access Method in more detail in order to map the method on hardware, we may profit from this hint (and in fact, we will).
3.1.2.5 Physical Layer

The Physical (PHY) Layer is responsible for interfacing with the medium, detecting and generating signals on the medium, and converting and processing signals received from the medium access control layer.

The PHY layer encodes and transmits the four symbols presented to it at its MAC interface by the MAC sublayer. The symbols exchanged between the MAC and PHY layers are: 0 = binary zero, 1 = binary one, J = non-data-J, K = non-data-K.

The symbols are transmitted in the form of differential Manchester-type coding which is characterized by the transmission of two line signal elements per symbol: during the transmission of any binary symbol, the polarity of the line signal changes in the center of the bit time. Furthermore, a binary zero has a transition (a polarity change) at the beginning of the bit time, while a binary one bit has not. The advantages of this are that the resulting signal has no DC component (and therefore can readily be inductively or capacitively coupled), and that the forced mid-bit transition conveys inherent timing information on the channel. The mid-bit transition does not occur at non-data symbols J and K. A violation of the coding rules may therefore easily be detected for error detection purposes. Deliberately introduced code violations are also used to form the special bit patterns that are attached to the frames (see paragraph 3.1.1). Received symbols shall be decoded using an algorithm that is the inverse of the one that is used for coding. The decoded symbols shall be presented to the MAC interface.

In normal operation there is one station on the ring that is called the Active Monitor. All other stations are frequency and phase locked to this Active Monitor. We will discuss the Monitor in paragraph 3.2.2.

More details about the PHY layer can be found in chapter 6 of reference [3]. Chapter 7 of reference [3] describes the station attachment specifications by means of a Medium Interface Connector.

The reader should be aware of the fact that the PHY layer can't be designed with IDaSS, as IDaSS is meant for digital circuits only. This means that we don't actually have to design the PHY layer! But for test purposes, we will have to find some means to provide the four symbols that should be delivered to the MAC sublayer for efficient simulation.

3.2 Token Ring Access Method

3.2.1 Introduction

All right, so far we have seen that the MMTCP will be equipped with a Token Ring LAN controller, which has to be designed as to be fully compatible with the ANSI/IEEE standard family 802. The layers that have to be designed (in hardware) are the Logical Link Control sublayer, the Medium Access Control sublayer, and as much as possible of the Network Management.

So far nothing was told about the actual basic principles of the Token Ring Access Method. For one reason, this was because the preceding chapters only had to introduce the designing of the Token Ring LAN controller by making the reader aware of the design context and the importance of the preliminary work that has already been done on the design of Token Ring LAN controllers. They give a hint on how to handle the huge design job.

Another reason is that the Token Ring Access Method is rather complex if one would fully understand it. Because we probably will not be able to design the whole Token Ring LAN controller, it would have been premature to go into detail in earlier chapters. However, without understanding the contents of reference [3], the IDaSS design of the LAN controller cannot be completed at all. For this reason, the reader is again strongly recommended to study reference [3] in detail. For our convenience, we will only take a quick look at the very basics of the Token Ring Access Method and the Priority Operation;
this will show to be the most important information we'll need to design at least a part of the Token Ring LAN controller. With the help of an example, the Token Ring Access Method will be explained.

3.2.2 General description

A token ring consists of a set of stations serially connected by a transmission medium (see Fig 6). Information is transferred sequentially, bit by bit, from one active station to the next. As mentioned in paragraph 2.2, information is transferred in packets, called 'frames' by standard 802.5.

Figure 6: A Token Ring with 8 stations. Station B is physically disconnected from the ring.

Each station generally regenerates and repeats each bit and serves as the means for attaching one or more devices (terminals, workstations) to the ring for the purpose of communicating with other devices on the network. A station that is physically inserted into the ring, is also called a 'node' (on the ring).

A given station that has access to the medium transfers information onto the ring (therefore this station is called the 'source station'), where the information circulates from one station to the next. The addressed station (the 'destination station') copies the information as it passes. A frame can be addressed to more than one station, in which case all addressed stations will copy the information contained in the frame. The destination station acknowledges the receipt of the frame by setting the address-recognized and frame-copied bits in the frame (see next paragraph). Finally, when the frame has circulated once on the ring, it returns to the source station which effectively removes the information from the ring (the station is said to 'strip' the frame when it circulates back).

A station gains the right to transmit its information onto the medium when it detects a token passing on the medium. The token is a control signal comprised of a unique signalling sequence that circulates on the medium following each completed information transfer. Any station, upon detection of an appropriate token, may capture this token by modifying it to a start-of-frame sequence (SFS) (the token is marked 'busy' by setting the token bit in the token, see next paragraphs).

After capturing the token, the station appends appropriate control and status fields, address fields, information field, frame-check sequence (FCS) and the end-of-frame sequence (EFS). The FCS is a bit pattern that is generated by the source station by means of polynomial calculations, and it is used for error checking by the destination station.

At the completion of its information transfer (so after the frame has circulated back) and after appropriate checking for proper operation, the station initiates a new token, which provides other stations the opportunity to gain access to the ring. Only one station can have control of the token at a
time and only one token is on the ring at a time, thereby preventing data collisions (two or more stations attempting to transmit at the same time).

A **token holding timer** (THT) in every station controls the maximum period of time the station shall occupy the medium before passing the token.

Multiple levels of priority are available for independent and dynamic assignment depending upon the relative class of service required for any given message, for example, synchronous (real-time voice), asynchronous (interactive), or immediate (network recovery). The allocation of priorities shall be by mutual agreement among users of the network.

Error detection and recovery mechanisms are provided to restore network operation in the event that transmission errors or medium transients (for example, those resulting from station insertion or removal) cause the Access Method to deviate from normal operation. Detection and recovery for these cases utilize a network monitoring function that is performed in a specific station (the **monitor station**) with back-up capability in all other stations (standby monitor stations) that are attached to the ring. The active monitor station also contains a latency buffer of variable length, that is used for two distinct functions: it assures a minimum latency of 24 bits to prevent a token for continuously circulating on the ring, and it compensates phase jitter: the cumulative variations in the data signalling rate that can occur instantaneously at segments in the ring are sufficient to cause variations of 3 bits in the latency of the ring. The master oscillator of the ring is supplied by the active monitor station. With the help of the flexible latency buffer, the active monitor can also control the mean data signalling rate around the ring.

3.2.3 Token and Frame formats

Fig 7 shows the Token format and the Frame format. The left-most bit or symbol is transmitted (and received) first. Fig 8 shows the fields of the token and frame formats that are important for our partial design: **Starting Delimiter (SD)**, **Access Control (AC)**, **Ending Delimiter (ED)** and **Frame Status (FS)**.

A frame or token shall be started with the eight symbols of the SD. If otherwise, the frame shall not be considered valid. Note that the SD is for a part comprised of deliberately introduced non-data symbols. So is the ED.

**Figure 7: Token and Frame formats.**

Token Format:

```
SD|AC|ED
```

Frame Format:

```
SD|AC|FC|DA|SA|INFO|FCS|ED|FS
```

- Code violation protected
- CRC protected

The transmitting station shall transmit the ED as shown. Receiving stations shall consider the ED valid if the first six symbols are received correctly. The other two symbols are the **Intermediate Frame bit** (I) and the **Error-Detected** bit (E). The I bit is transmitted as a 1 if this frame is an intermediate (or first) frame of a multiple frame transmission, otherwise it is transmitted as a 0. The E bit shall be set to 1 by any station that detects an error while the token or frame passes the station (for example, FCS error, code violations).
The FS contains the Address-Recognized (A) bits and Frame-Copied (C) bits. The A and C bits shall be transmitted as 0 by the station originating the frame. If another station recognizes the destination address as its own address or its relevant group address, it shall set the A bits to 1. If it manages to copy the frame into its receive buffer, it shall also set the C bits to 1.

The most important field for us will be the AC field, which is comprised of the Priority bits (PPP), the Token bit (T), the Monitor bit (M), and the Reservation bits (RRR).

The priority bits shall indicate the priority of a token and, therefore, which stations are allowed to use the token. The eight levels of priority increase from the lowest (000) to the highest (111) priority. For purposes of comparing priority values, the priority shall be transmitted most significant (left-most) bit first.

The token bit is a 0 in a token and a 1 in a frame.

The monitor bit is used to prevent a token whose priority is greater than 0 (or any frame) from continuously circulating on the ring. This bit shall be transmitted as 0 in all frames and tokens. The active monitor sets this bit to 1. All other stations shall repeat this bit as received. If the active monitor detects a frame or a high priority token with the monitor bit already equal to 1, the monitor decides that the frame or token circulated the ring more than once (or maybe there is a second monitor active, which is considered to be an error). In that case, the frame or token is aborted.

The reservation bits allow stations with high priority PDU's to request (in frames or tokens as they are repeated) that the next token be issued at the requested priority. The precise protocol for setting these bits is described in the next paragraph.

### 3.2.4 Priority operation

The priority bits (PPP) and the reservation bits (RRR) contained in the access control field work together in an attempt to match the service priority of the ring (P) to the highest priority PDU that is ready for transmission on the ring. As the AC field passes a station, the values of PPP and RRR are stored in registers \( Pr \) and \( Rr \), respectively. The current ring service priority (P) is indicated by the priority bits in the AC field, which is circulated on the ring.

The priority mechanism operates in such a way that fairness (equal access to the ring) is maintained for all stations within a priority level. This is accomplished by having the same station that raised the service priority level of the ring (the 'stacking station') also return the ring to the original service priority. We will see presently that the \( Sx \) and \( Sr \) stacks are used to perform this function.

The priority operation is explained as follows: When a station has a priority (a value greater than zero) PDU (or PDU's) ready to transmit, it requests a priority token. This can be done by changing the reservation bits (RRR) as the station repeats the AC field of a frame. This is done as follows: if the
priority level (Pm) of the PDU that is ready for transmission is greater than the value of the passing RRR bits, the station increases the value of the RRR field to the value Pm, overwriting a prior reservation set by some other station which PDU priority evidently is lower than that of the current station. If the value of the RRR bits is equal to or greater than Pm, the reservation bits RRR are repeated unchanged.

If the value of the reservation bits (RRR) is not raised again anymore by another station, a token will arrive at the current station, with its priority bits (PPP) set to the former value of the reservation bits (RRR). Because the priority of the token is now equal to the value of Pm, the station can claim the token. Now the station may transmit PDU’s that are at or above the present ring service priority (P) until it has completed transmission of those PDU’s or until the transmission of another frame could not be completed before timer THT expires (see paragraph 3.2.2). The reader should remark that the priority of all PDU’s that are transmitted should at least be at the present ring service priority! The station will then generate a new token for transmission on the ring. This is the normal way a token is released on the ring.

If the station does not have additional PDU’s to transmit that have a priority (Pm) or does not have a reservation request (as contained in register Rr) neither of which is greater than the present ring service priority (as contained in register Pr), the new token is transmitted with its priority at the present ring service priority and the reservation bits (RRR) at the greater of Rr or Pm and no further action taken.

However, if the station has a PDU ready for transmission or a reservation request (Rr), either of which is greater then the present ring service priority, the token is generated with its priority at the greater of Pm or Rr and its reservation bits (RRR) as 0. Since the station has raised the service priority level of the ring, the station becomes a stacking station and, as such, stores the value of the old ring service priority as Sr and the new ring service priority as Sx. (These values will be used later to lower the service priority of the ring when there are no more PDU’s ready to transmit on the ring whose Pm is equal to or greater than the stacked Sx.)

Note: Since a station may have raised the service priority of the ring more than once before the service priority is returned to a lower priority (for example, from 1 to 3 and then 5 to 6), it may have multiple Sx and Sr values stored and, hence, the term ‘stacked’.

Having become a stacking station, the station claims every token that it receives that has a priority (PPP) equal to its highest stacked transmitted priority (Sx) in order to examine the RRR bits of the AC field for the purpose of raising, maintaining, or lowering the service priority of the ring. The new token is transmitted with its PPP bits equal to the value of the reservation bits (RRR) but no lower than the value of the highest stacked received priority (Sr), which was the original ring service priority level.

If the value of the new ring service priority (PPP equal to Rr) is greater than Sr, the RRR bits are transmitted as 0, the old ring service priority contained in Sx is replaced with a new value Sx equal to Rr, and the station continues its role as a stacking station.

However, if the Rr value is equal to or less than the value of the highest stacked received priority (Sr) the new token is transmitted at a priority value of Sr, both Sx and Sr are removed (popped) from the stack, and if no other values of Sx and Sr are stacked, the station discontinues its role as a stacking station.

Note: A stacking station that has claimed the token may transmit PDU’s as well as examining RRR bits, as described above. Of course only those PDU’s which have a priority equal to or greater than the ring service priority may be transmitted.
The frames that are transmitted to initialize the ring have a PPP field that is equal to 0. The receipt of a PPP field whose value is less than a stacked Sx will cause any Sx or Sr values to be cleared in all stations on the ring.

The complete description of priority operating is contained in the Operational Finite-State Machine (see Appendix B and paragraph 4.2).

3.2.5 Priority operation example

At this point the reader may get a little bit dizzy with all the abbreviations and names that have been thrown at him in the last paragraph. And he will be getting headaches if he realizes that the operation described in the preceding paragraphs merely concern the normal operation mode. Nothing was mentioned about the different actions that should be taken if one of the numerous errors occurs (ring errors, transmission errors, station errors, priority errors, code violations, frame-check-sequence errors, duplicate address errors, multiple monitor errors and so on). Also, the monitor functions and the initialization stages of the ring and stations were omitted. But no panic: the most important knowledge that the reader needs in order to understand the partial design of the Token Ring LAN controller is understanding the priority operation. To be sure that the reader indeed has a grip on the priority operation, we will take a look at an example.

Consider Fig 9, which illustrates a three node ring with the following initial conditions:
1) Station A has a frame (or PDU) queued for transmission with Access Priority Pm = 0.
2) Station B has a frame (or PDU) queued for transmission with Access Priority Pm = 4.
3) Station C has a frame (or PDU) queued for transmission with Access Priority Pm = 6.
4) The initial location of the token on the ring is as shown in Fig 9. The token is priority-free (PPP = 000) and has no priority reservation specified (RRR = 000).

The Access Control (AC) field is given in the figure, with the PPP, T and RRR values from left to right (note that the shorthand notation 'P T R' is used). A zero value of the token bit (T) indicates a token, also represented by gray shaded data on the ring. A value of 1 for the T bit indicates a frame, also represented by black shaded data on the ring.

Referring to Fig 10, once the token is received by station A, it is used to transmit the enqueued frame. The token was captured because it was not marked busy (T = 0) and the value of PPP was 0, which was not greater than the value of Pm. At the point shown in the figure, the priority of the frame is still 0 (PPP = 000): the priority service level of the ring is thus not increased! Furthermore, the T bit has now been set to 1 (indicating a frame), and the reservation value (RRR) was left at zero.
Figure 10: Example - step 2. Station A captures token and begins transmission of a frame.

Referring to Fig 11, station B has received station A's frame. Because the T bit is equal to one, station B cannot transmit its data. However, station B can change to value of the reservation bits to meet his value of Pm: the RRR value is increased to 4 while() repeating the frame on the ring. Note that if the information in the frame was intended for station B, station B will recognize its own address MA ('My Address') in the DA ('Destination Address') field, and station B will copy the frame into its buffer.

Figure 11: Example - step 3. Station B repeats frame and changes RRR to 4.

Referring to Fig 12, when station C receives the frame, it repeats the frame with RRR equal to 6. This is because station C's enqueued frame has an access priority of 6 which is greater than the value of RRR of the frame which was received. Thus, station C also desires a priority token, but at a higher priority than station B, therefore overwriting the reservation done by station B.

Referring to Fig 13, when the frame originated by station A is returned to station A, station A begins stripping the frame from the ring. Meanwhile station A has to check whether the Source Address field matches its own address MA. If this is not the case, a severe error has occurred. Another kind of error may have occurred if the received priority value was not equal to the value that was transmitted (in this case: 0). Both errors may indicate that another station is also transmitting a frame.

Referring to Fig 14, station A, having now stripped its frame from the ring, releases a new token. Because the priority service level of the ring has to be raised from 0 to 6, the station performs a PUSH operation on it's Sx and Sr stacks. The PUSH operation results in 6 (the 'new priority level', or the 'transmitted priority value') and 0 (the 'old priority level', or the 'received priority value') being pushed onto the Sx and Sr stacks, respectively. The token that is transmitted has the PPP value equal to 6, the
Figure 12: Example - step 4. Station C repeats frame and changes RRR to 6.

Figure 13: Example - step 5. Station A strips frame from the ring.

T bit equal to 0 (indicating a token), and the RRR value equal to 0.

Figure 14: Example - step 6. Station A releases a new token.

Referring to Fig 15, when station B receives the priority 6 token transmitted by station A, it cannot capture the token as its access priority Pm is less than the priority PPP of the token. Thus, station B raises the RRR value, requesting again that a token of priority 4 be issued.
Referring to Fig 16, when station C receives the priority 6 token, it now captures the token and begins transmitting its enqueued frame. The token was captured because the PPP value equaled the access priority Pm of the enqueued frame. Note that the reservation field RRR was left unchanged.

Referring to Fig 17, the frame transmitted by station C circulates the ring normally. Neither station A nor station B modify the RRR value. This is because station A has no frame to transmit and station B need not change the RRR value from its current value.

Referring to Fig 18, station C receives the frame it transmitted and begins stripping the frame from the ring. Meanwhile station C checks whether or not the receiver has copied the frame, and whether or not errors have occurred.

Referring to Fig 19, once station C has stripped its frame, it releases a token at the current ring service priority level (PPP = 6). Because the PPP value is greater than the RRR value of the token, no stack operations are performed at station C prior to token transmission. This is because the ring service priority level need not be raised. On the contrary: it should be lowered. The station that can do this is the stacking station (in this case, station A), which has primary raised the ring service priority level.

Referring to Fig 20, when station A receives the token, it recognizes the PPP value to be equal to the stacked transmitted priority value Sx. Because the PPP value is also greater than the RRR value, the station concludes that some other station requests that a priority token less than the current priority level be circulated. Thus, the station performs a REPLACE function on the stack, substitutes the RRR
Figure 17: Example - step 9. Frame circulates with no modification.

Figure 18: Example - step 10. Station C begins stripping the frame from the ring.

Figure 19: Example - step 11. Station C issues a new token of same priority.

value for the PPP, and transmits the token on the ring. Note that station A lowered the current ring service priority level from 6 to 4, while it initially raised the ring service priority level from 0 to 6 (see Fig 14). To return to the initial state of Fig 9, the ring service priority level will have to be lowered again presently.
Figure 20: Example - step 12. Station A performs a REPLACE and transmits the token.

Referring to Fig 21, station B can now capture the token because the PPP value is now equal to its enqueued frame access priority Pm. Thus the enqueued frame is transmitted on the ring. Station C and A do not modify the RRR value of the frame as they currently do not have a frame enqueued for transmission.

Figure 21: Example - step 13. Station B captures token and transmits frame.

Referring to Fig 22, when the frame transmitted by station B has completed circulation, station B strips the frame from the ring. All stations have transmitted their frames at this point, so the only thing left is to lower the ring service priority level.

Referring to Fig 23, station B now issues a token of priority 4 as originally received. This is because their is no need for raising the current ring service priority level, as no other station requested a higher priority via the RRR field of the frame.

Referring to Fig 24, when station A receives the priority 4 token as sent from station B, it recognizes the PPP value to be equal to the stacked transmitted priority value Sx. Because no other station requested a higher or lower priority level, the station performs a POP of its stacks and returns the token to its original priority-free state. At this point, the ring is in a state similar to that shown in Fig 9, except for the frames enqueued at the stations being all transmitted.

This exercise provided a basic example of the priority operation. Note that each time the priority level of the token changes, each station on the ring is afforded equal opportunity to capture the token for frame transmission.
3.2.6 Considerations on the implementation of the protocol

If we consider the description of the Token Ring Access Method and the priority operation as given in the preceding paragraphs, we discover that a Token Ring LAN controller has to perform a complex, though well defined protocol. This protocol can be implemented as a microprogram contained in a...
ROM memory. To complete the Token Ring LAN controller, we only have to attach a dedicated processor to run the microprogram and add some interface with the host system and an interface with the ring. This is the way Texas Instruments has designed their TMS380 Adapter Chipset for connecting to the IBM token ring LAN (see reference [8]).

In our case, it was decided that the LAN controller should be on the same chip surface as the MMTCP (see paragraph 2.4). That means that if we follow the Texas Instruments design idea, there will be an additional dedicated processor on the MMTCP chip surface, because we can not let the MMTCP perform the protocol software that is contained in the ROM memory.

Furthermore, running a microprogram introduces some unnecessary and unwanted delay in the protocol handling (caused by conditional jumps, instruction fetches, et cetera), unless a fast clock is used. This will be a problem if the data rate is increased to 16 Mbit/s. On the other hand, if one decides to design a dedicated processor that has practically zero wait states or even performs one instruction per clock cycle, the dedicated processor gets too complex.

We probably cannot design the Token Ring LAN controller without any protocol software (that may or may not be contained in ROM), but to take full advantage of services already provided by the MMTCP (interface with the host system and memory for instance), and to keep the interprocessor communication overhead on the MMTCP chip low, we will design the Token Ring LAN controller as a piece of logic that is not based on a microprogram, but on gate-level logic.

If the operations that introduce the delay in every node on the ring can be done in gate-level logic, the delay per node can be made very small. Thus, operations that should be done 'on the fly' (like changing bits before they are repeated on the medium) can be made fast by implementing them in gate-level logic, while other operations (like network management, error handling or frame composition) can be implemented as microprograms (or equivalent; state machines or state controllers), if that will show to be convenient.
4 DESIGN STEPS

4.1 Problem definition

At this point we have seen that a Token Ring LAN controller is to be developed for the MMTCP, and we took a look at the basics of the Token Ring Access protocol and the Priority Operation. We could extend our knowledge with error handling and monitor functions, but at this stage we will first take a look at what could be accomplished in the limited time we have.

Note: the time that is required to design the total Token Ring LAN controller with IDaSS can be estimated to be 2½ years (this estimation is done after a partial design of the LAN controller was completed and it's therefore quite accurate). The time we have for the IDaSS design phase is about three months.

Let's see if we can restrict the design context and define the design problem more precise:

1) The MMTCP is still described in a global fashion: the interface with the Token Ring LAN controller is a little bit fuzzy and can therefore be designed at free will. Nevertheless we must keep in mind what functions are already provided by the MMTCP, and that our hardware structure (like bus widths etc.) will not conflict with the MMTCP internal structure. The easiest thing for us would be to design a part of the LAN controller that is more or less independent of the restrictions imposed upon us by the MMTCP structure. All right, the reader may protest, but this will shift the interface problem with the MMTCP to later stages? Yes, but the reader may also agree that if we start from the MMTCP interface and design towards the lowest (physical) layer, problems may occur on deciding how to group functions at lower levels together, and on introducing new levels or not. These choices are the actual design steps, and they will take quite a lot of time, especially if one takes a non-optimal decision at a high level, in which case the design steps should be traced back from all lower levels to this high level. Even more important may be, that, while designing such a huge system in a top-down fashion and in the mean time preserving the compatibility and the functionality of the standard, designing gets rather complex. On the other hand, if we design bottom-up from the physical layer towards the MMTCP, we can base every current design step on the (clearly and well-defined!) standard, implement it correctly (achieving the compatibility), and then see how higher layers can use the implemented function. This is the same design idea as was mentioned in paragraph 2.1 with routines: once the lowest hardware levels ("functions that do the actual job") have been designed, we have gained a toolbox of elementary functions that can be used to design higher levels. Note that these higher levels can then still be designed in a top-down fashion, but now with most of the lowest levels already provided and grouped. As we already have mentioned in paragraph 3.1.2.4, the choices on grouping elements are thus done for us by the standard. Some readers may object to this being a strong advantage. Not because this grouping could be a non-optimum one (we will leave this out of our discussion), but because the grouping proposed by the standard can theoretically be used directly in a top-down design approach as well. It should be clear however, that a specific lower level of a LAN controller can be tested more easily within IDaSS if the level above it is not designed yet, than it can in the reverse case. This is because a low level can be simulated by simply providing the parameters manually (while they normally would be provided by higher layers), as well as outputting the produced parameters to the terminal (instead of handing them over to the higher levels). While to test a high level, we have to provide for the underlying low-level functions, which is practically impossible without designing the lower levels.

2) Interfaces with higher layers (and thus with the MMTCP) should be compatible with the ANSI 802 standard family. While the higher layers can be implemented by the MMTCP, the Network Management functions should not be shifted too far into the MMTCP. For the least, some basic error handling should be designed within the LAN controller itself. Also the management interfaces with the MMTCP have to be designed. This can easily be done if we design according to the idea proposed above: if we design a low-level elementary function (like loading and checking the error bit in the Frame Status field, see Fig 8), we can decide on whether or not this elementary function has anything to do with Network Management or any other higher level (in the example of the E-bit: yes). If so, we
can provide for an interface to this higher level (in the example of the E-bit, there should be provided means for higher levels to examine and set this bit).

3) It is preferable that an accomplished design can be tested and demonstrated. That is: it should be possible to simulate the design without deadlocking it because it lacks some elementary functions that could not be designed within time. It is also preferable that a partial design provides a complete elementary function. In that case, a designer that attempts to complete the design does not have to finish our partial design first, which is in practice more difficult than designing a part of his own.

4) We know that Texas Instruments achieves a 2.5 bit delay time per node with their TMS380 chipset (see reference [8], page 1-11). If we can only design a part of the Token Ring LAN controller, one good reason to complete our partial design would be reached if we can achieve a smaller delay and therefore predict our total Token Ring LAN controller to be better than the Texas Instruments' complete system. Let's say that a maximum delay of one bit delay time per node should be achieved, even at 16 Mbit/s. The reader that has meanwhile read reference [3] can check for himself that the Token Ring LAN controller standard allows such a small delay: all operations that should be done on the fly can be done within one bit time. Thus, operations on the fly should be made very fast, which is in agreement with our considerations in paragraph 3.2.6 concerning the way the Priority Operation protocol should be implemented.

We know from the previous chapter how data can be transmitted via the LAN medium, and how access on the ring can be obtained. This was the basic information that was needed to understand the Token Ring. Why not base the choice of the part of the LAN controller that should be designed on this basic information? First, the Priority Operations should be done on the fly. So if we try to design the Access Protocol with its Priority Operation, we can try in the meanwhile to achieve our goal of one bit delay time per node mentioned above. Second, the Access Protocol is done at the lowest levels of the lowest layer that we have to design with IDaSS (namely, the MAC sublayer). These lowest levels are indeed the first we want to design in our design perspective. If we also manage to complete the priority operation as an elementary function, we can test it as well. This is because we only have to deal with the interface between the ring and the LAN controller, and not with the interface between the LAN controller and the MMTCP, which does not exist for test purposes. Some parameters (like the requested priority with which a station wants to send its data) can be simulated without designing the higher levels by simply entering the values for these parameters in dummy blocks, as mentioned earlier. These dummy blocks can be exchanged by the interfaces with higher levels at later stages.

So our design job will be designing the Priority Operation of the Token Ring LAN controller Access Protocol, together with some necessary functions and interfaces. The primary goal will be that the Priority Operation should be done within one bit time delay per node.

4.2 Problem tackling

Now we know what we should design, we have to go back to reference [3], which describes the Priority Operation and the Access Method. Reference [3] is a source of information that gives the information in three basically different ways: simple textual information (general and detailed, but not in a formal way), Finite State Machines (FSMs), and the procedural description of the Service Specifications (mentioned earlier in paragraph 3.1.2.4).

We have seen earlier that the procedural description can be used to group elements of the IDaSS design together. Therefore, we will not use this information until we reach the IDaSS design phase.

The FSMs are a perfect starting point for our design route. They don't only give a complete description of the Token Ring LAN controller algorithms (including the Access Priority protocol), but if we design a state controller in IDaSS that is based on such a FSM, a formal verification of the implemented protocol will not be necessary (provided off course that each state is correctly implemented). The 802.5 standard provides three FSMs: the Operational FSM (which is supplemented by the Receive Action
Table and the Bit Flipping Loop State Table), the Standby Monitor FSM and the Active Monitor FSM. Only the first FSM will be needed by us, because that's the one that contains the Priority Operation. The Standby Monitor and Active Monitor FSMs can easily be designed as higher levels upon the Operational level. They are not the basic operations mentioned in the preceding paragraph. Appendix A shows the Operational FSM. The complete description of the states and the transitions in this FSM is given by reference [3], paragraph 4.2.

Unfortunately, this FSM can not be transformed to an IDaSS implementation directly. Not only because of the speed demands that make it necessary to design more than only a state controller, but also because the transitions in the FSM are just protocol requirements. We will need the detailed verbal information in the standard as well to sort of enhance this FSM to a one that deals more with hardware requirements. We could take a lot of examples to clarify this step, but let's take for instance transition number 21 in the FSM depicted in appendix A: it does several tests that, for instance, need the value Pr. Pr is the received priority value that was in the AC field of the frame that has just passed. At the very moment of the test however, the PPP bits in the AC field have just passed the station. The FSM doesn't say anywhere that these bits had to be loaded into register Pr at the moment they passed. The reader may find this an obvious remark, but if the complete FSM is examined this way, the consequences are numerous, especially regarding the on-the-fly operations. Therefore we will transform this FSM to what we will call 'flow charts'. They are just an extra design step that will make the mapping of the FSM onto the IDaSS design easier. The flow charts are given in appendix B. For each state, one part of the total flow chart is given. The names are as much as possible the same as the ones used in the 802.5 standard. With each flow chart, an explanation is given for each action that is depicted in the flow chart. Note that these flow charts are not set up according to some standard or so, but they do reflect the data paths that can be recognized in the LAN protocols. With the 802.5 standard at hand, these flow charts can easily be recognized to implement the Operational FSM.

We will be dealing only with the REPEAT state, as the Access Method stops at the point where a station can transmit its data. The designing with IDaSS will therefore be based on the flow chart of the REPEAT state, which is in fact the very heart of the Priority Operation protocol.
5 IDaSS

5.1 General description

IDaSS is a tool for designing and simulating digital circuits ([9]). Hence the name, which is shorthand for 'Interactive Design and Simulation System'. The IDaSS environment is specifically targeted towards VLSI and ULSI designs of complex data processing hardware, as long as the design is a synchronous machine (a single clock source for all clocked elements in the design). Asynchronous logic with internal feedbacks cannot be simulated with IDaSS.

Note: IDaSS is written in the Smalltalk/V(286) environment. The Token Ring LAN controller was designed using an Intel 386 PC using MSDOS. A UNIX version of IDaSS is also available for Apollo systems, and a stripped version is also available for PC/XT versions running MSDOS.

IDaSS describes a design as a tree-like hierarchy of schematics. One or more lower level schematics can be placed inside an arbitrary schematic as if they were single elements, thus forming the hierarchical tree.

Beside lower level schematics, schematics can contain elements like registers, ALUs, memories, state machine controllers and the like, which we all call '(building) blocks'. These building blocks, as well as lower level schematics, are entered graphically.

An important property of all building blocks is that they are created and edited by the user himself, so they don't come in manufacturer-prepared libraries. A user can instead develop his own libraries with building blocks or schematics, if he wishes to do so.

The rectangles that represent all schematic elements are connected by lines representing the (bidirectional) buses between these elements. The buses are attached to the building blocks with connectors. They come in different shapes to distinguish between input, (disabled) output, bidirectional and control connectors.

State controllers can be placed in a schematic just like all other building blocks can. Their operational characteristics are entered in a textual form, describing a state machine. A controller can test and control the elements of schematics down along the hierarchical tree, and can change its state based upon test results. The language used can describe microprogrammed controllers (including a subroutine stack), Mealy and Moore state machines (or any suitable combination thereof). However, tests done by the controller can only be based upon directly clocked elements in the schematic or in lower level schematics, which restricts the Mealy type state machine somewhat (it becomes a nearly Mealy state machine).

Building blocks (including state controllers) can also be controlled by adding a control connector. This connector can be connected to any bus in the system, the value of which will determine the functions of that block. A textual PLA-like specification 'couples' the values on the bus with the functions to be executed.

Simulation with IDaSS is comfortable because it is immediate: there are no separate compile stages to do. Once an element is placed inside a schematic, it immediately behaves like the hardware equivalent.
5.2 Limitations of IDaSS and their effect on the design

5.2.1 Synchronous machines

In this paragraph we will investigate in what way the limitations of IDaSS influence the design of the Token Ring LAN controller. We will not comment on things as the (graphical) user interface of IDaSS or the way IDaSS is implemented (regarding to size, speed and the programming language it is written in). This is because IDaSS is still in a development stage and therefore these things are of little importance regarding the basics of designing with IDaSS.

The limitations that are really significant are the restriction to synchronous circuits and the (slow) communication between state controllers at the different schematic levels of the design.

We start with the restriction to synchronous circuits: one characteristic point of this restriction is that IDaSS offers us just one single clock source for our design, while we need in fact three clock sources: first, we need one to simulate the 4 Mbit/s data rate, thus running at 4 MHz. Second, we need a clock source to clock the hardware elements of the LAN controller. We will call this the system clock. And third, reference [2] mentions a 32 kHz crystal oscillator to account for the different timers in the Token Ring design (see Appendix A1 of reference [2]).

All timers are checked upon their contents, or they should give some kind of interrupt if they reach a zero value while counting down. Thus by increasing the timer width, a timer can easily be adapted to be used with a higher clock frequency without seriously increasing the design complexity. So we can eliminate one clock source.

Now for the system clock: it should obviously run at 4 MHz or a multiple thereof. If we decide to run the system clock at a higher frequency than 4 MHz, the other clock source can be derived from the system clock by means of a frequency divider. The 4 MHz data rate can also be simulated with some other kind of additional circuitry at the PHY layer in the IDaSS design. However, for simulation purposes we are handed a prettier solution by paragraph 4.1, where we stated our goal of achieving a maximum delay of one bit time per node (this, of course, does not include the latency buffer of the monitor station). We also noted that the system clock should run at a frequency that should be kept as low as possible. Therefore we will transform the delay demand to a maximum delay of one clock cycle per node. That is: a received bit that is clocked into the LAN controller should be clocked out (i.e. transmitted) at the very next clock tick of the system clock. In this case we can operate the Token Ring LAN controller at the low frequency of 4 MHz! Besides the elimination of another clock source, this gives us the additional advantage that the future 16 Mbit/s data rate demand will not introduce too big problems. This is because 16 MHz clocked hardware should be achievable.

Note that, although the LAN controller will be operated at 4 MHz, it can Phase-Locked-Loop-extract a 8 MHz clock signal from the ring, imbedded in the differential Manchester coded data. Fig 25 shows the timing for incoming and outgoing data referred to the extracted 8 MHz clock and two different system clocks: a 16 MHz clock and the proposed 4 MHz clock. Both clocks are derived from a voltage controlled oscillator (VCO) that is synchronized to the received data stream via a phase-locked loop (PLL) which locks to the bit stream signalling rate. Because the system clock is used to time the transmission of the data bits too, the phase jitter between receiving and transmitting bit streams is nearly zero.

The reader knows by now that we don't have to design the physical layer (see paragraph 3.1.2.5). So the asynchronous synchronization between the data rate and the system clock is assumed only, it does not have to be designed. That is as good as well, as this is impossible with IDaSS! Some sort of 'data-ready' line can always be attached later between the PHY layer and our design.

So the simulation with just one single clock source is not a severe problem, neither is the impossibility of designing asynchronous logic.
5.2.2 Communication between levels

IDaSS is being developed to design large synchronous circuits in a top down fashion, as mentioned earlier. In order to let building blocks communicate with each other, buses are used. These can be dragged through the different levels by the means of superconnectors, to let the blocks at different levels communicate with each other as if they were in the same schematic (the actual hardware implementation is in fact one silicon level).

To let state controllers communicate with each other, several options can be used: First, they can have access to a common communication block, such as a memory or registers. Second, state controllers can communicate using a global set of semaphores called signals. These signals can be tested, set and reset by all state controllers in the hierarchy. Third, state controllers can directly give low level commands to other state machines at lower level schematics (like stop and start commands). And finally, fourth: a state controller can manipulate some building block that has its output coupled to a controller bus that can be attached to the other state controller(s).

The third possibility is clearly just a simple form of communication and will often not satisfy. The first two possibilities are time-consuming. For instance, for signals (see also Fig 26a): if state controller A wants to notify state controller B of some internal error that occurred in its schematic, he will try to set some signal at the clock tick after the tick at which the error event initiated. The actual setting is thus done one clock tick later as the actual occurrence of the error. At the third clock tick, state controller B can detect the signal (if he tests it!), and at the fourth clock tick he has made his jump to an error handling routine.

In fact, a signal is nothing more than a specific one bit wide register that is hidden in the schematic (it is not graphically displayed). If a hidden operator or constant generator can be used, communication can be made much faster.

This will then be a hidden form of the fourth possibility that is mentioned above: let us assume the preceding communication situation, but now in the case that controller A uses an operator to output a value on a controller bus that is directly connected to controller B (see Fig 26b). At the detection of the error, state controller A chooses the appropriate function within the operator, which is done
Figure 26: Communication and timing in case of an error event:

a) Two state controllers act upon a signal.
b) Two state controllers communicate by means of an operator.
c) An operator reports the error to a state controller.

...
5.3 Choices within IDaSS

5.3.1 Hardware versus software description

We already decided in paragraph 3.2.6 that the on-the-fly operations should not be designed as a microprogram because of speed demands. We can see now that the IDaSS equivalent of a microprogram, the state controller, can indeed not be used at the lowest level.

For instance, let us see what minimum delay can be achieved if we do decide to design the Priority Operation based on a state controller. First, we have to bear in mind that the state controller has to make decisions about whether to change an incoming bit to what value, or simply repeat it, or whatever. These decisions can only be based upon the contents of directly clocked elements. So the incoming bit has to be clocked into a building block first, which costs us one clock tick. Then an additional clock tick is needed for the state controller to examine the bit value and make some decision, based on an algorithm. In fact, the decision will probably cost us more than one clock tick, because of jumps that go with performing comparisons within the algorithms. But let us not get too pessimistic. Finally, a third clock tick is needed to perform the action that follows from the decision, and output the bit value onto the ring.

For example: the bit (or symbol, if you like) is clocked into a register first. At the second clock tick, the state controller knows its value and decides to change it. Then, at the third clock tick, the register contents is set to the desired value. The register output can be directly coupled to the ring interface, in which case the value change and the transmission can go within one single clock tick.

So at least three clock ticks are needed, which is one too many. Therefore we will need some trick with operators. They are the only building blocks that can perform decisions and actions within one clock cycle, simply by performing their functions. However, the incoming bit does have to be hold for one clock cycle, if only for examining this bit and comparing it with stored values (for instance, comparing a received priority bit with a stored three-bit priority value). So a register seems to be an obvious necessity, also because it can load and store the bit value within one clock cycle, and it can change and output its value within one clock cycle as well. More about this later.

Another point to mention is that (conditional) jumps are not possible within operators. Nevertheless, these jumps are the easiest way, and often even the only way, in which decision based algorithms can be implemented. A state controller will therefore still be needed to guard and control the decisions, while the actions will be performed by controlled operators. These operators need to give feedback to the state controllers that control them. For instance, an operator should notify a controller like this: 'Since you told me to handle the AC field according the Access Control Algorithm, I have now managed to set the Token bit. So now you can go into transmission mode.' So the state controller can in its turn be controlled by one or more operators that they control.

What is described above is just an example of the problems we will encounter while designing using IDaSS. It is typically for designing with IDaSS, that the designer has to make choices whether his design will tend to be a mere 'hardware description' (small operators, registers, memories), or that it will tend to a mere 'software description' (state controllers and/or complex operators with multiple exotic functions).

Note: the terms hardware description and software description are in fact a little bit misleading, as the total IDaSS design will end up on a single chip surface.

A designer that is used to create circuits with the aid of standard hardware libraries will tend to choose the hardware description, as a writer of (simulation) programs will tend to choose the software description. Both descriptions have their advantages and disadvantages. If the designer wants to take full advantage of IDaSS, he has to get familiar with both descriptions, because the optimum choice for his design could be a mixture of the hardware description and the software description. In the first case, the system can be made fast, but the number of building blocks and the number of buses
increases rapidly with increasing system complexity. This goes for the IDaSS design as well as for the final hardware implementation. In the latter case, the grouping of functions will not only be done by introducing schematics within schematics, but also by introducing labelled states and logically grouped function blocks. Such a design will be easy to 'read' and understand.

Beside all these considerations, IDaSS offers a large range of expression operators, which makes it difficult for us to choose how to implement a specific function within a state controller or an operator. This, and the choice between the hardware and software description does not have to be an extra burden to the designer, as long as he uses consistency. If the designer has a set of design rules that will help him to make consistent decisions sort of automatically, he will not only take full advantage of the possibilities of IDaSS, but his design will also stay clear and functional, even if it ends up as a rare mixture of hard- and software descriptions. Therefore, we will try to draw up a set of design rules in the next paragraph.

5.3.2 Design rules

To draw up a list of design rules, it will help if we know in advance what we have to design exactly, and more important, what kind of description of the design is already provided.

In our case, we have seen that the Acces Method with the Priority Operation is handed to us in the form of a Finite State Machine, which we transformed to flowcharts. The shortest design route will therefore be starting with a state controller. If the (main) states of the state controller are designed in such a way that they reflect the states of the FSMs (which is the most obvious way), it will make the IDaSS design easy to understand for the one that is familiar with the 802 standard family, and a formal verification of the implemented access protocol will be superfluous. At points where we have to make concessions regarding to speed demands, we will divert from the state controller concept and develop the necessary 'hardware blocks' (that is, operators et cetera).

On the other hand, one could think of a situation in which the speed demand is to be fulfilled first, making the design of operators a logical first design step. Deviation from this design concept will only be done then in case it is unavoidable in order to conform to the algorithm, or if it is preferred in order to improve the surveyability of the design.

If the prepared information can take us both ways, or if we don't even have some formal prepared description of the design, we will need some reference points or design rules that will help us to design more easily.

Experience with IDaSS has led to the set of design rules that is listed below. These design rules are not strict. Moreover, they don't even reflect the actual design route with IDaSS, as in practice, a lot of backtracking and feedback is used while using these design rules. However, experience has taught these rules to be a good guide along the design route.

First, we should collect all information about the design that is important regarding to speed. If speed shows to play a major role in the system that is to be designed (as in our case), we should group the speed requiring actions on logical and functional terms. For example: detection of the SD and the ED should be done on the fly, and in a similar way. Thus both detection actions can form a functional group which can be denoted as 'detectors'. On the other hand, the AC field contains the PPP and the RRR values, which are both used in quite different ways by the Priority Operation algorithm: different functions operate on them. They can however be logically grouped, as they both concern the Access Mechanism directly.

Second, we should collect all information about the design concerning interfacing. If interfacing plays a major role in the design (as in our case, too), all building blocks that will interface with other schematics, levels, or even with other designs, should be grouped together. For instance, error lines can be packed into one single error bus, which on its turn can be multiplexed with a status bus or interrupt
bus. This will reduce the spaghetti- appearance of a design. All information can then be extracted from this single bus at all places in the design. Of course, if some building blocks don't need all the information on the bus, specific information can be demultiplexed from the central bus before coupling to this building block. We don't want to make connections unnecessary wide.

Note that at this point we still haven't designed any building blocks yet: we only grouped elements. This can be done on paper or in our mind, but it is preferred to group building blocks into a (lower level) schematic. Just by entering the schematic without designing the contents, we make the very most important design steps! It will be clear to the reader that these 'grouping rules' can be used repeatedly at every design step, at any moment, to initiate new 'design branches' on the design route. Remember that at later stages, degrouping of building blocks is always done quicker than trying to group some building blocks, because in the latter case the blocks may not interface well. So in short: in the first design stages, the designer should not hesitate to introduce a lot of (lower level) schematics in the design.

Third, the designer should choose an appropriate architecture for his design. For instance: state controllers or not, one operator with many functions or many operators with one function, et cetera. This choice can be influenced by the design description that is available before designing with IDaSS is started, as in our case. But also of great influence can be the chosen architecture of a partial design that is already designed, or the availability of some elementary blocks (or schematics) in some sort of library. If the designer recognizes some already designed schematic in that library that can be used in his design too, he obviously should consider the adaptation of his design or architecture in order to use this schematic. Note that the choice of an architecture can be done at every schematic, not only the top level schematic. With small systems however, the designer should use the architecture of the top level schematic consistently throughout the other schematics. This because of clearness of the design.

Fourth, when the architecture of the top level schematic is chosen and some important functions are grouped, the actual top down design route is started. One could of course start with pure top down designing without considering the grouping of the elements as discussed above. But this can only be done if the designer has a pretty good idea of his schematic tree at the start of his design. In practice this is not the case, especially not with complex systems. The advantage of the grouping discussed above is that, while designing top down, the designer can now actually work towards these grouped blocks. While designing in a top down fashion, the choice of entering another lower level schematic should be based upon functional or logical relationships between the functions (as with the first design rule), and sometimes space-saving may be a good reason too (that is, space-saving within the IDaSS worksheet that displays the drawing of the schematic).

Fifth, with the essential schematics now being entered, the designer can start entering the building blocks in the schematics. A designer is often inclined to start with the most important and most complex part of the design. But in order not to bite off more than one can chew, it is preferred to start with the small schematics. These are the schematics with little contents or easy to design, obvious contents. These can then quickly be tested (if necessary with the aid of dummy blocks), and after simulation they provide a correctly working toolbox for the more complex schematics. For instance, in our case we should first implement the basic functions that are needed with the Priority Operation (like loading priority values et cetera), so they can be used later with more complex functions (like comparing the loaded priority values with stored values for gaining access to the medium).

Sixth, the choice within every schematic has to be made regarding the actual implementation of a function, within the frame of the chosen architecture. These choices are not straightforward, as an arbitrary function can be implemented in many different ways with IDaSS. These choices are very much influenced by the design experience of the individual IDaSS user. Also, a novice user will rather tend to use the familiar expression operators like AND and NOR in some complex combination, instead of trying to get used to 'intelligent' operators as MSZMASK.

The following considerations are made with the design of the LAN controller:
- If functions should be implemented by operators, and they are functionally related, they should be designed as an operator with multiple functions. This can of course only be done if the functions
should never be activated at the same time, as an operator can only use one of its functions at a time. But if these functions are not to be activated at the same time, the IDaSS compiler that checks the multifunction operator automatically ensures that these functions indeed will not be activated at the same time. In our case for example, we can think of a complex operator that handles all priority actions or the AC field.

- If functions should be implemented that use values that reside in building blocks at or beneath the current schematic level, the choice between state controller or operator has to be made. If speed does not play too big a role in a schematic, a state controller is preferred instead of an operator. This is because the state controller checks the contents of these building blocks without the use of buses. This makes the IDaSS design more easier to read. Two exceptions are made: checking the contents of building blocks at higher level schematics can not be done with a state controller, so in that case an operator has to be used. The state controller can also be moved to that higher level, but that will make the state descriptions within the state controller longer, and is therefore not preferred. The second exception is in case the function that is to be implemented should not only check contents of building blocks, but actually should communicate with state controllers. In the case of state controller to state controller communication, this communication can be done with signals, but they have a disadvantage: a signal is just one bit wide. In order to give feedback to a state controller (as mentioned in paragraph 5.3.1), several signals should be used, which can lead to an error in the choice of a new state. A better solution would be the use of a control connector at the state controller that can be arbitrary wide. Communication with that state controller is thus direct (and fast). The other communication partner can be a state controller too, but an operator does not need an additional building block to provide the signals on the controller bus. Thus the choice we made earlier (a state controller that controls an operator that is coupled back to the state controller by means of a control connector) is not only a good choice regarding to speed, but it also gives us the opportunity of flexible communication with the state controller.

- An additional remark about giving names: in order to clarify the design, the names of buses should be chosen consistently. The following rules should be used as much as possible: First, if there is just one block output connected to a bus, the name of the bus should be identical to the name of the output. Second, buses should have the same name as the superconnectors they are connected to. Third, inputs should carry as much as possible the same name as the buses that are connected to them. When following these rules, all blocks and connectors in all level schematics can quickly be detected to carry the same data.

- Instead of designing a complex function with small operators, a lot of buses and simple expressions, the designer should try to use the 'intelligent' expression operators as much as possible. For instance, one could implement an ALU as a combination of small operators like ANDs and NORs, coupled with the necessary buses, but it is preferred to implement it as one single operator with a slightly more complex textual description. This will eventually show to be the quickest way, because the graphical editing within IDaSS costs quite a lot of time.

- Finally a rather straightforward remark: every designed building block should be simulated as soon as possible, if need be with dummy blocks. This will cost some time, but in the end it will pay back, because if an error occurs when big parts of the designed system are finished, tracing the error will then be done more easily.
6 THE TOKEN RING DESIGN

6.1 Introduction

In this chapter, the IDaSS design of the Token Ring LAN controller will be described. The description is done in such a way that each schematic in the design is described in a separate paragraph. Thus, this description does not reflect the actual design route. However, most of the schematics are designed according the top-down design philosophy, and those which are not are specifically commented as such. Extensive commenting is done only for the first three layers. Important general comments for all lower level schematics are given in a 'parent' layer.

With every schematic, the choice of the implementation is explained with respect to the design rules mentioned in paragraph 5.3.2. And because the complete design is just a (basic) part of the LAN controller, the extendibility of the parts in the schematic is also pointed out (as far as it seems appropriate).

To make the design readable for IDaSS users, the design description resembles the document system output of IDaSS that can be extracted from the design.

6.2 Signals

The Token Ring LAN controller design uses the following signals:

Signal CONNECTED is used to physically connect the LAN controller to the ring. This signal is set high when connected.
Signal INSERTED is used to logically insert the LAN controller into the ring. This signal is set high when inserted.
Signal I simulates the I flag.
Signal MA simulates the MA flag.
Signal SFS simulates the SFS flag.

All signals are operated in the level only mode, and the level state following system reset is reset low (0).

6.3 Level 1 (TOP LEVEL): DATALINK

Figure 27 shows the top level schematic, called 'DATALINK'. The number of building blocks in this schematic is easily kept low because only the most necessary blocks are included: the Token Ring LAN controller, an incoming data link connection, an outgoing data link connection and two small blocks that help the designer with the simulation of the LAN controller. The cover of this report shows the screen view of a simulation session: simulation commands are entered in the state controller, incoming symbols are written into one of the RAMs, the time counter is reset, and simulation starts. The outgoing symbols can be checked visually while they are written into the other RAM.

ADDRESS

'DATALINK\ADDRESS' is a register.
This register is 11 bits wide.
The default function is 'increment'.
The value following system reset is 0.

The value loaded for the 'reset' command is 0.
This counter supplies the increasing address that is used to write into the RAM memory DATA_OUT and to read from the RAM memory DATA_IN. The counter is reset by the state controller INITIATOR and then increases indefinitely (although both the memories only contain 2048 words).

DATA_IN

'DATALINK\DATA_IN' is a RAM.
This RAM contains 2048 words of 2 bits each.
This RAM holds its contents through a system reset.
Output 'data' reads at the address input by 'read'.

This memory can be filled for simulation purposes with the symbols that the user wants to apply to the Token Ring LAN controller after system reset. The memory is read out by the increasing address produced by 'ADDRESS', starting at address 000h. In this way a data link is simulated on which a maximum number of 2048 data symbols (bits) and non-data symbols (J and K) appear and then subsequently are fed to the LAN controller elements. It should be obvious that this memory is for test purposes only, because in practical situations, this memory will be substituted by the appropriate receiving part of the physical layer of the Token Ring design (with an additional element to provide for a 'symbol ready'-kind of signal).

![Top level schematic DATALINK.](image)

DATA_OUT

'DATALINK\DATA_OUT' is a RAM.
This RAM contains 2048 words of 2 bits each.
This RAM is loaded with unknown values after system reset.
Data input 'data' writes at the address input by 'write'.

This memory will be filled with the symbols that the Token Ring controller elements produce after system reset. The symbols will be written at the increasing address produced by 'ADDRESS', starting at address 001h (accounting for the one bit delay that is introduced by the LAN controller design). It should be obvious that this memory is for test purposes only, because in practical situations, this
memory will be substituted by the appropriate transmitting part of the physical layer of the Token Ring LAN controller.

INITIATOR

'DATALINK\INITIATOR' is a state machine controller that is enabled following system reset. Within this state controller, several primary functions and values can be edited, which are used throughout the design to accomplish an overall system state which can be thought of as to be set by the extern host system. For instance: Is the LAN controller physically connected to the ring? Is the LAN controller logically inserted into the ring? And so on. Some of these primary functions are already provided by means of signals. If the design evolves in the future, this state controller may become superfluous. This is because all commands will then be given by the attached host system.

The text for the states is as follows:

START:  "Start simulation."
"!CONNECTED;  "Physically connect to the ring."
"!INSERTED;  "Logically insert into the ring."
ADDRESS reset;  "Initiate time."
TOKNRING\MAC2LLC\REQUESTED_SERVICE_CLASS setto:4;  "Enqueue a PDU with priority 4"
TOKNRING\MAC2LLC\MA_DATArequest setto:1;  "...and make request to transmit."
><

IDLE:  "Don't do anything."
<<

TOKNRING

'DATALINK\TOKNRING' is a schematic. This is the actual design of the Token Ring LAN controller. Data symbols and non-data symbols enter the controller, are processed, and then they are retransmitted back onto the ring. No attachment to any host system is provided yet. This is in agreement with the assumption made in paragraph 4.1, where we concluded that the partial design should for now be more or less independent of the restrictions imposed by the MMTCP structure. It will be clear that in the future, the interface with the MMTCP will be easier to design if one knows more specific details about the communication demands and the hardware structure of the MMTCP. Nevertheless, one could design a general system interface with adaptive widest of the buses. Compatibility with existing I/O should be examined. For instance, 68000 and 80X86 processor systems should be attached without any serious problems. Because of this compatibility, The Texas Instruments TMS380 Chipset User's Guide ([8]) could help with the interface design.

6.4 Level 2: TOKNRING

Figure 28 shows a level 2 schematic, called 'TOKNRING'. In this schematic, the reader can recognize the grouping done by the procedural description mentioned earlier in paragraph 4.1. Three schematics are already provided: 'PHYSICAL', 'PHY2MAC' and 'MAC2LLC', the last two schematics implementing the service specifications provided by the physical layer to the MAC sublayer and by the MAC sublayer to the LLC sublayer, respectively. Schematic 'PHYSICAL' is designed according to the considerations made in paragraph 4.1 in combination with the fourth design rule: 'PHYSICAL' should contain all necessary physical layer related functions that have to be simulated but that at the other hand will not be part of the actual synchronous Token Ring LAN controller design. The extendibility can be accomplished by adding an additional schematic for
Network Management functions ('ALL2NMT') and Monitor functions ('MONITOR'). Between the 'PHYSICAL' and 'PHY2MAC' schematic, the monitor controlled latency buffer can be placed.

**Figure 28: Level 2 schematic TOKNRING.**

**MAC2 LLC**

This block simulates the services required of the MAC sublayer by the LLC to allow the Logical Link Control sublayer entity to exchange LLC data units with peer LLC sublayer entities. Note that the procedural descriptions of the service specifications are used here to group building blocks together (see reference [3], chapter 5): 'MA_DATA.request' is implemented in the signals 'FRctrl', 'DESTaddr', 'MACsdu', 'REQclass' and 'request'. 'MA_DATA.indication' is implemented in the signals 'FRctrl', 'DESTaddr', 'MACsdu', 'Rstatus' and 'indicate'. 'MA_DATA.confirmation' is implemented in the signals 'Tstatus', 'PRVclass' and 'confirm'.

**PHY2MAC**

This block simulates the services provided by the physical layer that allow the local Medium Access Control sublayer entity to exchange MAC data units with peer MAC sublayer entities. Note that the procedural descriptions of the service specifications are also used here to group building blocks together (see reference [3], chapter 5): 'PH_DATA.request' is implemented in the signals 'bin_out' and 'sym_out'. 'PH_DATA.indication' is implemented in the signals 'bin_in' and 'sym_in'. 'PH_DATA.confirmation' is embedded in the design, as 'DATALINK\DATA_OUT' is designed to accept all symbols generated at lower levels. We will concentrate on this schematic, as it should contain the Priority Operation algorithm.
This schematic incorporates the basic features provided by the PHY layer to the MAC sublayer. In this case, incoming symbols are always present (UNKNOWN values do not enter this design). The symbols are decoded into a binary value (bin_in) and a symbol value (sym_in), which are passed to the MAC sublayer. With respect to the FCS calculation, the non-data symbols J and K are treated as binary 1 and 0 respectively. The coding/decoding scheme therefore is as depicted in table 1:

<table>
<thead>
<tr>
<th>Symbol:</th>
<th>bin_in:</th>
<th>sym_in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>data symbol 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>data symbol 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>non-data symbol J</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>non-data symbol K</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

6.5 Level 3: MAC2LLC

Figure 29 shows a level 3 schematic, called 'MAC2LLC'. In this schematic, several dummy blocks are entered to provide the parameters that are needed to give transmit commands to the MAC sublayer, such as the destination address and a transmit request line. It will be clear to the reader that this schematic is far from completely designed. This schematic should provide the (for now still vague) interface with the MMTCP.

Figure 29: Level 3 schematic MAC2LLC.

**MA_DATArequest**

'DATALINK\TOKNRING\MAC2LLC\MA_DATArequest' is a register. This register is 1 bit wide.
The default function is 'hold'. The value following system reset is 0. The value loaded for the 'reset' command is 0.

'MA_DATArequest' should be set to 1 if there is a PDU queued for transmission, i.e.: if there is a request for a transmission.
FRAME_CONTROL

'DATALINK\TOKNRING\MAC2LLC\FRAME_CONTROL' is a register.
This register is 8 bits wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

The 'FRAME_CONTROL' parameter specifies the value of the frame's FC octet.

DESTINATION_ADDRESS

'DATALINK\TOKNRING\MAC2LLC\DESTINATION_ADDRESS' is a register.
This register is 15 bits wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

The 'DESTINATION_ADDRESS' parameter may specify either an individual or a group MAC entity address. It shall contain sufficient information to create the DA field that is appended to the frame by the local MAC sublayer entity as well as any lower-level address information.

REQUESTED_SERVICE_CLASS

'DATALINK\TOKNRING\MAC2LLC\REQUESTED_SERVICE_CLASS' is a register.
This register is 3 bits wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

The 'REQUESTED_SERVICE_CLASS' parameter specifies the priority (Pm) desired for the data unit transfer.

M_SDU

'DATALINK\TOKNRING\MAC2LLC\M_SDU' is a register.
This register is 16 bits wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

The 'M_SDU' parameter specifies the MAC service data unit to be transmitted by the MAC sublayer entity. There is sufficient information associated with 'M_SDU' for the MAC sublayer entity to determine the length of the data unit.

6.6 Level 3: PHY2MAC

Figure 30 shows a level 3 schematic, called 'PHY2MAC'.
In this schematic, the Priority Operation algorithm should be implemented. This schematic therefore took the most effort to design and is designed towards the deepest level. Building blocks 'DETECTORS' and 'PRIORITIES' can be seen to be designed as an in-between-step towards the sub-schematics 'SD_DET' and 'ED_DET', and 'P' and 'R', respectively. These four sub-schematics are
designed according to the first design rule. This is because the very first design step that was taken, was designing the on-the-fly detectors and shift registers that are contained by 'DETECTORS', 'PRIORITIES', 'P' and 'R' (remember the-toolbox though mentioned earlier!). The fourth and sixth design rule were used frequently at lower levels, as the reader can check out for himself throughout the descriptions later in this chapter. Before designing the most important and complicated blocks in this schematic, 'AC_FILTER' and 'REPEAT', the fifth design rule was used to design the smaller blocks 'TIMERS' and 'STACKS' first. Later on, as the design of 'REPEAT' was finished as far as the normal error-free operation was concerned, 'ERROR' was designed first before designing error functions within 'REPEAT', according to the second design rule.

AC_FILTER

'DATALINK\TOKNRING\PHY2MAC\AC_FILTER' is an operator.

This operator filters the AC field in order to gain access to the medium. This filter actually performs the Access Method protocol by trying to set the token bit in the AC field as well as changing the value of the reservation bits. Because of the feedback to the state controller 'REPEAT', two dummy functions are needed to prevent the absence of a signal on the feedback controller bus.

The text of the functions is:

'repeat':

setRr := PmMAX not \ " If Pm > Rr then Rr > Pm is false. "

( " First bit (MSB) " ...)
(BitCnt = 6) \land 
\text{bin\_in} \land (Pm \text{ at:2} \ not)

) \quad \text{"} Rr=1XX, Pm=0XX = Rr > Pm. \text{"} 
\lor 

( " Second bit ...")
(BitCnt = 7) \land 
\lor 
(RrMAX \quad " Rr=1XX, Pm=0XX = Rr > Pm is already determined at the MSB. "
\lor 
(bin\_in \land (Pm \text{ at:1} \ not)) \quad " Rr=A1X, Pm=A0X = Rr > Pm." 
)
)
).

c : = RrMAX \lor " If Rr > Pm then Pm > Rr is false. " 
(
( " First bit (MSB) ...") 
(BitCnt = 6) \land 
(bin\_in \not) \land (Pm \text{ at:2})

) \quad " Pm=1XX, Rr=0XX = Pm > Rr. "
\lor 

( " Second bit ...")
(BitCnt = 7)
\lor 
(PmMAX \quad " Pm=1XX, Rr=0XX = Pm > Rr is already determined at the MSB. "
\lor 
((bin\_in \not) \land (Pm \text{ at:1})) \quad " Pm=A1X, Rr=A0X = Pm > Rr." 
)
)
).
).

\text{state : = " } A \text{ change in state is made if a proper token can be captured. This change is reported to the state controller who then decides what action is to be taken. } \text{ "}
(
(BitCnt=4) \land 
(bin\_in \not) \land 
(SkMT \not)
\land 
(Pr=Sx)) \land 
(notQMT \land (Pr\leq Pm))\not 
), \quad " \ldots \text{ in this case the stacks should be modified ... } " 
(
(BitCnt=4) \land 
(bin\_in \not) \land 
notQMT \land 
(Pr\leq Pm)
).
\quad " \ldots \text{ and in this case data can be transmitted. } \text{ "}

\text{modify :=(}
(BitCnt=4) \lor " Only the token bit and the three ... "
(BitCnt=6) \lor " \ldots \text{ reservation bits can be modified. } "
(BitCnt=7) \lor 
(BitCnt=8)
),
\quad 1 \text{ zeroes, } " \text{ Non-data symbols are ignored in this filter. } " 
(
( " \text{ Modification of token bit : } "
(BitCnt=4) \land " \ldots \text{ fourth bit in AC ... } "
(bin\_in \lor " \ldots \text{ bit is already set, or ... } "}
(bin_in not) \(\land\) "... bit need only be set if it is zero ...
" ... and if one of these conditions are forfilled: ...
(notQMT \(\land\) (Pr \(\leq\) Pm)) \(\lor\) "... token captured for transmission ...
(StkMT not) \(\land\) (Pr=Sx)) "... token captured for modifying stacks ...
) "... so far for the token bit.
)
)
)
\v "Modification of reservation bits:"
(notQMT \(\land\)
( "... first reservation bit ...
(BitCnt=6) \(\land\) "... sixth bit in AC ...
(bin_in \(\lor\) (Pm at:2)) "... first reservation bit :=MAX(Pm,Rr) ...
) "... so far for the first reservation bit ...
\v
( "... second and third reservation bit ...
(((BitCnt=7) \(\lor\) (BitCnt=8)) \(\land\) "... seventh or eighth bit in AC ...
( (RrMAX \(\land\) bin_in) \(\lor\)
( PmMAX \(\land\) (Pm at:(8-BitCnt))) \(\lor\)
( (RrMAX not) \(\land\)
( PmMAX not) \(\land\)
(Pm at:(8-BitCnt)) \(\lor\) bin_in
) ) ) )
) "... so far for the last two reservation bits."
)
)
ClrSx:=(BitCnt=4) \(\land\)
(bin_in not) \(\land\)
(StkMT not) \(\land\)
(Pr=Sx)\(\land\)
(notQMT \(\land\) (Pr \(\leq\) Pm)) not.

'send0':
modify :=%100.
setRr :=%0.
setPm :=%0.
state :=%11.
ClrSx :=%0.

'wait':
modify :=%000.
setRr :=%0.
setPm :=%0.
state :=%11.
ClrSx :=%0.
DATALINK\TOKNRING\PHY2MAC\BIN_DELAY is a register.
This register is 1 bit wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register serves as a buffer to give the MAC sublayer a one clock tick delay time to determine what to do with the incoming bit. Notice that the maximum delay imposed by this register equals one bit duration time. The buffer is controlled by a three bit wide bus that consists of the following multiplexed control bits: one bit determines whether the incoming bit should be altered or whether it simply can be repeated, one bit determines the value to which the incoming bit should be altered, and one bit is not used here (see 'SYM_DELAY'). Notice that the decision that the incoming bit will be altered does not mean that the incoming bit will be inverted! It can thus end with the same value it had when it was entering the Token Ring controller. It merely indicates that a situation has occurred in which incoming data can be modified (for instance, if the reservation bits enter while the station is requesting a transmission, these reservation bits can be modified if some specific demands are fulfilled).

Control specification:
%0XX load.
%1X0 setto:0.
%1X1 setto:1.

_DETECTORS_
Detection of the Starting Delimiter (SD) and the Ending Delimiter (ED) should be done on the fly. As soon as the last bit of the SD arrives at the LAN controller, all other circuits and state controllers should be notified (especially the ones that should act upon the priority bits that directly follow the SD). In order to get control over the different steps that should be taken at the receipt of the SD, a bit counter is started as soon as the SD is detected. This bit counter is reset again if the ED is detected. An additional signal EDval is needed to check whether or not the received ED is valid. This is the case if all but the I bit and the E bit of the ED are received.

_ERROR_
This schematic is a collection of several error events that can occur throughout the (partial) design. It can use the error information to steer the state controller 'REPEAT' when deviation from normal (error-free) operation is requested.

PmMAX
'DATALINK\TOKNRING\PHY2MAC\PmMAX' is a register.
This register is 1 bit wide.
The default function is 'load'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register is used by the operator 'AC_FILTER' at determining which of the values Pm or Rr is the highest one. If Pm>Rr,'PmMAX' is set to 1.

_PRIORITIES_
This schematics contains two shiftregisters that load the values of the priority bits and the reservation bits contained by the AC field, respectively.

**RECEIVE**

This schematic is prepared to be controlled by the state controller 'REPEAT' as soon as the AC field has passed. The contents will thus be all the elements that are needed to copy incoming data from the ring and calculating the FCS. Quite some effort has to be put into the block that calculates the Frame Check Sequence. This 'FCS generator' is likely to be placed inside this schematic if 'SEND' has its own specific FCS calculation (because 'SEND' need not calculate it on the fly). If 'SEND' and 'RECEIVE' will use the same FCS calculation, the FCS generator should best be placed at the same level as 'RECEIVE'.

**REPEAT**

'DATALINK\TOKNRING\PHY2MAC\REPEAT' is a state machine controller that is enabled following system reset. This state machine controller is the actual implementation of the Access Method protocol as it is given by the Operational FSM in the standard 802.5. From here, all other elements are controlled in order to perform the Priority Operation. The state controller is controlled by the 'AC_FILTER' and the error block 'ERROR' as well, to provide a form of feedback.

Control specification:
%000 goto: REPEAT.
%001 goto: TRANSMIT.
%010 goto: MODIFY.
%1XX goto: ED_error.

The text for the states is as follows:

**REPEAT:**

```
[ DETECTORS\BITCOUNTER=8
  | 1 AC_FILTER wait;
  !!SFS;
  [ RECEIVE ERROR\TK_DET
    | 1 ->RECEIVE ]
  | 0 AC_FILTER repeat ];
< <
```

**RECEIVE:**

```
AC_FILTER disable::modify;
RECEIVE\TEMP enable;
AC_FILTER wait;
[ RECEIVE \READY | ->REPEAT ];
< <
```

**TRANSMIT:**

```
[ DETECTORS\BITCOUNTER<8
  | 1 AC_FILTER send0; "M=R=0"
  | 0 AC_FILTER send0;
  "Start transmitter ..."
  TIMERS\THT reset;
/MA;
```
TX_DATA:
AC_FILTER send0;
AC_FILTER disable; modify;
SEND\TEMP enable;
"Start transmitter ..."
[SEND\READY | ->REPEAT ];
<<<

MODIFY:
[ DETECTORS\BITCOUNTER < 8
| 1 AC_FILTER send0;
| 0 AC_FILTER send0;
"Start transmitter ...
STACKS\TRANSMIT pop;
TIMERS\TRR reset;
/SFS;
->MOD_STCK
]
<<<

MOD_STCK:
AC_FILTER send0;
<<<

ED_error:
AC_FILTER wait;
"TX abort"
<<<

RrMAX

'DATALINK\TOKNRING\PHY2MAC\RrMAX' is a register.
This register is 1 bit wide.
The default function is 'load'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register is used by the operator 'AC_FILTER' at determining which of the values Pm or Rr is the highest one. If Rr > Pm, 'RrMAX' is set to 1.

SEND

This schematic is prepared to be controlled by the state controller 'REPEAT' as soon as the AC field has passed and access to the ring is gained to transmit data. The contents will thus be all the elements that are needed to transmit the enqueued PDU onto the ring and calculating the FCS for the data that is to be transmitted. Sending data can be seen to be a quite straightforward sequence of sequentially executed steps without (conditional) jumps. For instance: 1) transmit ready-to-go FC field, 2) transmit ready-to-go addresses, et cetera. A simple state controller that takes over the control from 'REPEAT' should do the job.
STACKS

This schematic contains the two stacks Sx and Sr that are used in the Priority Operation algorithm. This schematic will probably be worked out in more detail in the future, because not all the stack operations are designed yet.

STATE_MUX

'DATALINK\TOKNRING\PHY2MAC\STATE_MUX' is an operator.
This operator is simply used to multiplex the different feedback sources to the state controller 'REPEAT' onto one controller bus.

SYM_DELAY

'DATALINK\TOKNRING\PHY2MAC\SYM_DELAY' is a register.
This register is 1 bit wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register serves as a buffer to give the MAC sublayer a one clock tick delay time to determine what to do with the incoming 'symbol' bit. Notice that the maximum delay imposed by this register equals one bit duration time. The buffer is controlled by a three bit wide bus that consists of the following multiplexed control bits: one bit determines whether the incoming bit should be altered or whether it simply can be repeated, one bit determines the value to which the incoming bit should be altered (that is, a '1' for symbol J or a '0' for symbol K), and one bit is not used here (see 'BIN_DELAY'). Notice that the decision that the incoming bit will be altered does not mean that the incoming bit will be inverted! It can thus end with the same value it had when it was entering the Token Ring controller. It merely indicates that a situation has occurred in which incoming data can be modified (for instance, incoming bits can be modified to non-data symbols if a sudden abort sequence has to be transmitted).

Control specification:
%0XX load.
%10X setto:0.
%11X setto:1.

TIMERS

This schematic contains the different timers that go with the LAN controller design, including additional registers that can be loaded with default reset values.
6.7 Level 3: PHYSICAL

Figure 31 shows a level 3 schematic, called 'PHYSICAL'. The functions that should be implemented within this schematic are bypassing data in the case the station is physically not connected to the ring, and transforming the simulated asynchronous differential Manchester coded line signals into two binary bit streams. The reason for choosing two one bit wide bit streams instead of one two bit wide bus, is for the simple reason that the non-data symbols are not needed throughout the whole Token Ring LAN controller, as are the data symbols. Because the non-data symbols 'J' and 'K' are treated as binary '1' and '0' respectively at the calculation of the FCS, the decoding scheme is designed such as to make another decoding at the 'FCS schematic' unnecessary.

BYPASS

'BATALINK\TOKNRING\PHYSICAL\BYPASS' is a TS buffer with width 2. This buffer is used to simulate the situation in which the station is not physically connected to the ring. In that case, the incoming bits should be passed to the ring without any delay. The default state is enabled. It can be disabled by means of the signal 'CONNECTED'.

![Figure 31: Level 3 schematic PHYSICAL.]

DECODER

'DATALINK\TOKNRING\PHYSICAL\DECODER' is an operator. The symbols from the data link are decoded into the bit streams 'bin_in' and 'sym_in' as given in table II. The function 'bypass' is used to output dummy values to the MAC sublayer. Note: the data code is the code that is used to store the symbols into the RAM memories at the top level schematic.

The text of the functions is:

'bypass'
bin_in := 0.
sym_in := 0.

'decode':
bin_in := in at: 0.
sym_in := in at: 1.

CODER

'DATALINK\TOKNRING\PHYSICAL\CODER' is an operator.
Table II: DECODER/CODER (de-)coding scheme.

<table>
<thead>
<tr>
<th>sym_in/</th>
<th>bin_in/</th>
<th>data code:</th>
<th>symbol:</th>
</tr>
</thead>
<tbody>
<tr>
<td>sym_out:</td>
<td>bin_out:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>J</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>K</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>

The bitstreams are coded into the symbols 0, 1, J and K as shown in table II. Note: the data code is the code that is used to store the symbols into the RAM memories at the top level schematic.

The text of the default function is:

'code':
out := sym_out, bin_out.

CONNECT

'DATALINK\TOKNRING\PHYSICAL\CONNECT' is a state machine controller. This state machine controller continuously checks the signal 'CONNECTED' to see whether the Token Ring LAN controller is physically connected to the ring or not. If so, the 'BYPASS' buffer is switched off, and the decoder is activated.

The text for the state is as follows:

TEST_ON_CONNECTED: "Test if the LAN controller is physically connected to the ring or not."
[ ?CONNECTED " Test signal CONNECTED ..."
  | 1 BYPASS disable; " If connected, switch on ..."
  | CODER enable; " ...enable the coder ..."
  | DECODER decode ]; " ...and start decoding."
<<

6.8 Level 4: DETECTORS

BITCOUNTER

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\BITCOUNTER' is a register. This register is 15 bits wide.
The default function is 'reset'.
The value following system reset is 0.
The value loaded for the 'reset' command is 1.

This important counter counts the bits after the SD has been detected, so it starts with the first priority bit which is regarded bit number 1. The contents of the counter is used throughout the whole LAN controller to determine what frame field or specific bit is processed at the moment.
Control specification:
%000 hold.
%1X0 reset.
%0X1 setto:0.
%010 inc.

BUSY

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\BUSY' is a register.
This register is 1 bit wide.
The default function is 'reset'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register is used as a flag by 'DET_MUX' to record the situation where SD is detected, yet ED is not.

Control specification:
%1X0 setto:1.
%0X1 setto:0.
%010 hold.

DET_MUX

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\DET_MUX' is an operator.
This multiplexer serves as a controller for 'BITCOUNTER'. Based on the occurrence of the SD and ED, the counter has to be reset, started or increased. The flag 'BUSY' is used to record the situation where SD is detected, while ED isn't detected yet.

The text of the function is:
'multiplex':
out :=SDdet,busy,EDdet.

ED_DET

This schematic incorporates the on-the-fly Ending Delimiter detector. The output 'EDval' equals 1 at the moment the sixth bit of the ED is detected, otherwise it is 0. To prevent timing problems, it is necessary that this value is known at the very moment of the arrival of the sixth bit of the ED. If the eighth bit of the ED is detected (the E-bit), EDdet is set high for one clock cycle.
SD_DET

This schematic incorporates the on-the-fly Starting Delimiter detector. The output 'SDdet' equals 1 at the moment the SD is detected, otherwise it is 0. To prevent timing problems, it is necessary that this value is known at the very moment of the arrival of the last bit of the SD.

6.9 Level 4: ERROR

DECODE

'DATALINK\TOKNRING\PHY2MAC\ERROR\DECODE' is an operator.
This decoder is used to set the 'ERRbit' register, which is used to record the occurrence of a symbol error (an unwanted code violation).

The text of the function is:

'decode':
ctrl :=(BitCnt =0),sym.

ERR_CHK

'DATALINK\TOKNRING\PHY2MAC\ERROR\ERR_CHK' is an operator.
This operator is used to check all sorts of errors that can occur while repeating the data. For now, checking is only done for symbol errors and the absence of the ED.

The text of the function is:

'error_check':
ED :=(EDval not) \ " No valid ED detected ..." 
((BitCnt=14) \ (TKbit not)) \ " ...while token is processed, or ..." 
((Bitcnt=10000) \ (TKbit not)) \ " ...while frame is processed. "

Figure 33: Level 4 schematic ERROR.
sym := sym \_in \ \_ "Symbol error in ..." \\
( \\
\_BitCnt > 0) \_\_\_ (BitCnt < 4) \_\_\_ priority bits or token bit. " \\
) \_\/_
( \\
(TKbit not) \_\_\_ "In case of a token : ..." \\
( \\
\_BitCnt > 4) \_\_\_ (BitCnt < 8) \_\_\_ monitor bit or reservation bit ..." \\
\_BitCnt = 15) \_\_/ (BitCnt = 16) \_\_\_ ... intermediate frame bit or error bit ... " \\
) \\
) \\
).

ERRbit

'DATALINK\TOKNRING\PHY2MAC\ERROR\ERRbit' is a register.
This register is 1 bit wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register is used to record the occurrence of a symbol error (an unwanted code violation). This is necessary because the E-bit in the ED field has to be set when all other fields have passed. Also, controllers can check the contents of this register for other network management functions.

Control specification:
%1X reset. "BitCnt=0(reset) = reset ERRbit. "
%01 setto: 1. "If an symbol error occurs, set ERRbit. "

TK_DET

'DATALINK\TOKNRING\PHY2MAC\ERROR\TK_DET' is a register.
This register is 1 bit wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register is used to store the token bit in the AC field. If this bit is '1', a token is detected. If this bit is '0', a frame is detected.

Control specification:
0 reset. "BitCnt=0(reset) = reset TK_DET. "
4 load. "BitCnt=4(token bit) load token bit. "
6.10 Level 4: PRIORITIES

Figure 34: Level 4 schematic PRIORITIES.

\[\text{DATA}\text{LIN}\text{K}\text{\backslash T}\text{OK}\text{N}\text{R}\text{I}\text{N}\text{G}\text{\backslash P}\text{HY2}\text{MAC}\text{\backslash PRIORITIES}\text{\backslash P} \]

\[\text{DATA}\text{LIN}\text{K}\text{\backslash T}\text{OK}\text{N}\text{R}\text{I}\text{N}\text{G}\text{\backslash P}\text{HY2}\text{MAC}\text{\backslash PRIORITIES}\text{\backslash R} \]

6.11 Level 4: RECEIVE

CONTROLLER

\[\text{DATA}\text{LIN}\text{K}\text{\backslash T}\text{OK}\text{N}\text{R}\text{I}\text{N}\text{G}\text{\backslash P}\text{HY2}\text{MAC}\text{\backslash RECEIVE}\text{\backslash CONTROLLER} \]

\[\text{DATA}\text{LIN}\text{K}\text{\backslash T}\text{OK}\text{N}\text{R}\text{I}\text{N}\text{G}\text{\backslash P}\text{HY2}\text{MAC}\text{\backslash RECEIVE}\text{\backslash ERROR} \]

The text for the state is as follows:

START:

* Receive the data *

ERROR

This schematic contains the 3 bit wide shift-register that loads the priority value in the AC field. The loading is controlled by the 'BitCnt' value: the 3 priority bits are the bit numbers 1, 2 and 3.

This schematic contains the 3 bit wide shift-register that loads the reservation value in the AC field. The loading is controlled by the 'BitCnt' value: the 3 reservation bits are the bit numbers 6, 7 and 8.

This state machine controller is prepared for future use. It will take over the controlling of the incoming bit stream as soon as the AC field has passed and the Priority Operation algorithm has been executed. Control may be handed back to the 'REPEAT' state controller at the 'PHY2MAC' schematic. From within this state controller, things as checking addresses, copying the incoming data and calculation of the FCS should be controlled.

The text for the state is as follows:

START:

* Receive the data *

ERROR

This schematic is nearly identical to the 'PHY2MAC\ERROR' schematic. It is used for test purposes only and will in future be replaced by a more specific error block. This future version will concentrate on errors other than those that occur at receiving the AC field. For this purpose, a multiplexer is provided to multiplex all kinds of error signals onto one bus. Because this schematic has no further functional value in our partial design, we will not look at it in detail.
READY

'DATALINK\TOKNRING\PHY2MAC\RECEIVE\READY' is a register.
This register is 1 bit wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 1.

This register is used for test purposes only. Its functions is to notify the 'PHY2MAC\REPEAT' state controller that all the receive actions within this schematic are done and control can be taken over. In future, this register will be replaced by a control bus or a more specific status register.

Figure 35: Level 4 schematic RECEIVE.

TEMP

'DATALINK\TOKNRING\PHY2MAC\RECEIVE\TEMP' is a register.
This register is 3 bits wide.
The default function is 'hold'.
The value following system reset is 7.
The value loaded for the 'reset' command is 0.

This register is used for test purposes only. It merely puts a dummy control value on the bus that controls the one bit delay registers 'PHY2MAC\SYM_DELAY' and 'PHY2MAC\BIN_DELAY'. In future, this register will be controlled in such a way that the incoming bit stream will be repeated when operating in normal repeat mode.

6.12 Level 4: SEND

CONTROLLER

'DATALINK\TOKNRING\PHY2MAC\SEND\CONTROLLER' is a state machine controller.
This state machine controller is prepared for future use. It will take over the controlling of the outgoing bit stream as soon as the AC field has passed and the Priority Operation algorithm has been executed in such a way that the station may transmit its enqueued data. Control may be handed back to the 'REPEAT' state controller at the 'PHY2MAC' schematic. Within this state controller, things as collecting data from memory, creating frames and calculating the FCS should be controlled.

The text for the state is as follows:
SEND:
*Send the data*
<<

![Schematic of DATA_UNIT and CONTROL block](image)

Figure 36: Level 4 schematic SEND.

**DATA_UNIT**

'DATALINK\TOKNRING\PHY2MAC\SEND\DATA_UNIT' is a RAM. This RAM contains 2048 words of 8 bits each. This RAM is loaded with value 0 following system reset. Output 'out' reads at the address input by 'read_addr'.

This memory is for test purposes only. It can be used to collect addresses and PDU data in order to create the frame fields that should be transmitted sequentially. The use of this memory is limited because of its insufficient length and because the memory has to be filled in manually. It is recommended to design more lower, basic building blocks first, before going on with designing and using operations that rely on this memory. In fact, the memory that will be used eventually will probably be a memory block that is part of the MMTCP memory.

**READY**

'DATALINK\TOKNRING\PHY2MAC\SEND\READY' is a register. This register is 1 bit wide. The default function is 'hold'. The value following system reset is 0. The value loaded for the 'reset' command is 1.

This register is used for test purposes only. Its functions is to notify the 'PHY2MAC\REPEAT' state controller that all the transmit actions within this schematic are done and control can be taken over. In future, this register will be replaced by a control bus or a more specific status register.
TEMP

"DATALINK\TOKNRING\PHY2MAC\SEND\TEMP" is a register.
This register is 3 bits wide.
The default function is 'hold'.
The value following system reset is 7.
The value loaded for the 'reset' command is 0.

This register is used for test purposes only. It merely puts a dummy control value on the bus that controls the one bit delay registers 'PHY2MAC\SYM_DELAY' and 'PHY2MAC\BIN_DELAY'. In future, this register will be controlled in such a way that the outgoing bit stream will be correctly transmitted when operating in normal transmit mode.

6.13 Level 4: STACKS

CTRL

"DATALINK\TOKNRING\PHY2MAC\STACKS\CTRL" is a state machine controller.
This state machine controller's job is to guard the stacks in the schematic. Because only state controllers can check whether a stack is empty or not, this state controller is used to provide the line signal 'StkMT' ('stack empty'). The state controller is also used to reset the stack 'Sx' ('TRANSMIT' in the schematic). The command to clear the stack is given by the operator 'PHY2MAC\AC_FILTER'.

Control specification:
%1 goto: CLEAR_Sx

The text for the states is as follows:

TEST: "Test if TRANSMIT is empty."
[TRANSMIT?
  0 SxEMPTY setto: 1
  1..7 SxEMPTY setto: 0 ];
<<

CLEAR_Sx:
SxEMPTY setto: 1;
TRANSMIT reset;
>>
RECEIVE

'DATALINK\TOKNRING\PHY2MAC\STACKS\RECEIVE' is a LIFO.
This LIFO contains 7 words of 3 bits each.

This LIFO is the implementation of the stack Sr. Because the Ring Service Priority level can be raised no more than 7 times in a row, this LIFO is big enough to handle even the most extreme situations.

SxEMPTY

'DATALINK\TOKNRING\PHY2MAC\STACKS\SxEMPTY' is a register. This register is 1 bit wide.
The default function is 'hold'.
The value following system reset is 1.
The value loaded for the 'reset' command is 1.

This register holds a zero value if the stack Sx is not empty. It is controlled by the state controller 'CTRL'.

TRANSMIT

'DATALINK\TOKNRING\PHY2MAC\STACKS\TRANSMIT' is a LIFO.
This LIFO contains 7 words of 3 bits each.

This LIFO is the implementation of the stack Sx. Because the Ring Service Priority level can be raised no more than 7 times in a row, this LIFO is big enough to handle even the most extreme situations.

6.14 Level 4: TIMERS

TAM

'DATALINK\TOKNRING\PHY2MAC\TIMERS\TAM' is a register.
This register is 24 bits wide.
The default function is 'decrement'.
The value following system reset is 12582912.
The value loaded for the 'reset' command is 12582912.

This register implements the Active Monitor Timer. Each station shall have a timer 'TAM' which is used by the active monitor to stimulate the enqueueling of an 'AMP PDU' ('Active Monitor Present') for transmission. The default time-out value of timer 'TAM' shall be 3 seconds, according to the 802 standard.

The register 'TAM' is accompanied by a similar register 'TAMdefault' that holds the default value that the user (or the host system) wants to load into 'TAM' at the reset (or re-loading) of timer 'TAM'. The default value contained by 'TAMdefault' at initialization of the station is the value that is calculated by H. Vos (see reference [2], appendix A1), which is 750 milliseconds.

THT

'DATALINK\TOKNRING\PHY2MAC\TIMERS\THT' is a register.
This register is 16 bits wide.
The default function is 'decrement'.
The value following system reset is 41943.
The value loaded for the 'reset' command is 41943.
This register implements the Holding Token Timer. Each station shall have a timer 'THT' to control
the maximum period of time the station may transmit frames after capturing a token. A station may
initiate transmission of a frame if such transmission can be completed before timer 'THT' expires. The
operation of 'THT' is described in the Operational Finite-State Machine (see Appendix A). The default
time-out value of timer 'THT' shall be 10 milliseconds, according to the 802 standard.

The register 'THT' is accompanied by a similar register 'THTdefault' that holds the default value that
the user (or the host system) wants to load into 'THT' at the reset (or re-loading) of timer 'THT'. The
default value contained by 'THTdefault' at initialization of the station is the value that is calculated by
H. Vos (see reference [2], appendix A1), which is 150 microseconds.

Figure 38: Level 4 schematic TIMERS.

TNT

'DATALINK\TOKNRING\PHY2MAC\TIMERS\TNT' is a register.
This register is 23 bits wide.
The default function is 'decrement'.
The value following system reset is 4194304.
The value loaded for the 'reset' command is 4194304.

This register implements the No Token Timer. Each station shall have a timer 'TNT' to recover from
various token-related error situations. 'TNT' shall have a time-out value equal to 'TRR' plus n times
'THT' (where n is the maximum number of stations on the ring). The operation of 'TNT' is described
in the Monitor
Finite-State Machines (see reference [3]). The default time-out value of timer 'TNT' shall be 1 second,
according to the 802 standard.

The register 'TNT' is accompanied by a similar register 'TNTdefault' that holds the default value that
the user (or the host system) wants to load into 'TNT' at the reset (or re-loading) of timer 'TNT'. The
default value contained by 'TNTdefault' at initialization of the station is the value that is calculated by
H. Vos (see reference [2], appendix A1), which is 150 microseconds.
TQP

'TDATALINK\TOKNRING\PHY2MAC\TIMERS\TQP' is a register.
This register is 16 bits wide.
The default function is 'decrement'.
The value following system reset is 41943.
The value loaded for the 'reset' command is 41943.

This register implements the Queue PDU Timer. Each station shall have a timer 'TQP' for the purpose of timing the enqueuing of an 'SMP PDU' ('Standby Monitor Present') after reception of an AMP or SMP frame in which the A and C bits were equal to 0. The default time-out value of 'TQP' shall be 10 milliseconds, according to the 802 standard.

The register 'TQP' is accompanied by a similar register 'TQPdefault' that holds the default value that the user (or the host system) wants to load into 'TQP' at the reset (or re-loading) of timer 'TQP'. The default value contained by 'TQPdefault' at initialization of the station is the value that is calculated by H. Vos (see reference [2], appendix A1), which is 150 microseconds.

TRR

'TDATALINK\TOKNRING\PHY2MAC\TIMERS\TRR' is a register.
This register is 14 bits wide.
The default function is 'decrement'.
The value following system reset is 10486.
The value loaded for the 'reset' command is 10486.

This register implements the Return to Repeat Timer. Each station shall have a timer 'TRR' to ensure that the station shall return to REPEAT state. 'TRR' shall have a value greater than the maximum ring latency. The maximum ring latency consists of the signal propagation delay around a maximum-length ring plus the sum of all station latencies. The operation of 'TRR' is described in the operational Finite-State Machine. The default time-out value of 'TRR' shall be 2.5 milliseconds, according to the 802 standard.

The register 'TRR' is accompanied by a similar register 'TRRdefault' that holds the default value that the user (or the host system) wants to load into 'TRR' at the reset (or re-loading) of timer 'TRR'. The default value contained by 'TRRdefault' at initialization of the station is the value that is calculated by H. Vos (see reference [2], appendix A1), which is 150 microseconds.

TSM

'TDATALINK\TOKNRING\PHY2MAC\TIMERS\TSM' is a register.
This register is 25 bits wide.
The default function is 'decrement'.
The value following system reset is 29360128.
The value loaded for the 'reset' command is 29360128.

This register implements the Standby Monitor Timer. Each station shall have a timer 'TSM' which is used by the stand-by monitor(s) to assure that there is an active monitor on the ring and to detect a continuous stream of tokens. The default time-out value of timer 'TSM' shall be 7 seconds.

The register 'TSM' is accompanied by a similar register 'TSMdefault' that holds the default value that the user (or the host system) wants to load into 'TSM' at the reset (or re-loading) of timer 'TSM'. The default value contained by 'TSMdefault' at initialization of the station is the value that is calculated by H. Vos (see reference [2], appendix A1), which is 2 seconds.
TVX

'DATALINK\TOKNRING\PHY2MAC\TIMERS\TVX' is a register.
This register is 16 bits wide.
The default function is 'decrement'.
The value following system reset is 52429.
The value loaded for the 'reset' command is 52429.

This register implements the Valid Transmission Timer. Each station shall have a timer 'TVX' which is used by the active monitor to detect the absence of valid transmissions. The operation of 'TVX' is described in the monitor Finite-State Machine. The time-out value of 'TVX' shall be the sum of the time-out value of 'THT' plus the time-out value of 'TRR'.

The register 'TVX' is accompanied by a similar register 'TVXdefault' that holds the default value that the user (or the host system) wants to load into 'TVX' at the reset (or re-loading) of timer 'TVX'. The default value contained by 'TVXdefault' at initialization of the station is the value that is calculated by H. Vos (see reference [2], appendix A1), which is 750 microseconds.

6.15 Level 5: P

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{schematic.png}
\caption{Level 5 schematic P.}
\end{figure}

REG3

'DATALINK\TOKNRING\PHY2MAC\PRIORITIES\P\REG3' is a register.
This register is 3 bits wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register will contain the priority value of the received AC field for the purpose of making decisions within the Priority Operation algorithm. The first three bits following the SD will be loaded.

Control specification:
1,2,3 load.

SHFT3

'DATALINK\TOKNRING\PHY2MAC\PRIORITIES\P\SHFT3' is an operator. This operator is used to shift the contents of register 'REG3'.

Text for the function:
6.16 Level 5: R

Figure 40: Level 5 schematic R.

REG3

'\texttt{DATALINK\backslash TOKNRING\backslash PHY2MAC\backslash PRIORITIES\backslash R\backslash REG3}' is a register.
This register is 3 bits wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

This register will contain the reservation value of the received AC field for the purpose of making decisions within the Priority Operation algorithm. The sixth, seventh and eighth bit following the SD will be loaded.

Control specification:
6,7,8 load.

SHFT3

'\texttt{DATALINK\backslash TOKNRING\backslash PHY2MAC\backslash PRIORITIES\backslash R\backslash SHFT3}' is an operator. This operator is used to shift the contents of register 'REG3'.

Text for the function:

'shift3':
out :=(in from:0 to:1),bit.

6.17 Level 5: ED_DET

DATA

This schematic contains an 8 bit wide shift-register. The contents, being the last eighth received data symbol-indicating bits ('bin_in'), are shifted to the left, while the 'bin' bit is inserted at the right side.
NDATA

This schematic contains an 8 bit wide shift-register. The contents, being the last eighth received non-data symbol-indicating bits ('sym_in'), are shifted to the left, while the 'bin' bit is inserted at the right side.

```
Figure 41: Level 5 schematic ED_DET.
```

ED_TEST

'DATALNK\TOKNRING\PHY2MAC\DETECTORS\ED_DET\ED_TEST' is an operator. It is used to compare the contents of the shift registers with the ED specific values. That is, the occurrence of the sequence 'JK1JK1' is tested. If the test succeeds, 'EDval' will go high, by definition for one clock cycle. If all eighth bits of the ED are received, 'EDdet' will go high for one clock cycle.

The text of the default function is:

'test on ED':
EDval :=((in1 from: 0 to:5) = %110110) ∧
        ((in2 from: 0 to:5) = %101101).
EDdet :=((in1 from: 2 to:7) = %110110) ∧
        ((in2 from: 2 to:7) = %101101).

6.18 Level 5: SD_DET

DATA

This schematic contains an 8 bit wide shift-register. The contents, being the last eighth received data symbol-indicating bits ('bin_in'), are shifted to the left, while the 'bin' bit is inserted at the right side.

NDATA

This schematic contains an 8 bit wide shift-register. The contents, being the last eighth received non-data symbol-indicating bits ('sym_in'), are shifted to the left, while the 'bin' bit is inserted at the right side.

INSERTED

'DATALNK\TOKNRING\PHY2MAC\DETECTORS\ED_DET\INSERTED' is a state machine controller that is enabled following system reset. This state machine controller is used to test the signal 'INSERTED' to check whether the station is logically inserted into the ring or not. If not, a gate is closed just before the SD detector. In that way, no random data can enter the station to start some unwanted process.
The text for the reset state is as follows:

TEST:
\[
\begin{cases}
0 & \text{SD KEY setto: 0} \\
1 & \text{SD KEY setto: 1}
\end{cases}
\]

The text of the default function is:

'and':
out := in1 \& in2

SD_GATE

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\SD_DET\SD_GATE' is an operator. Through SD_KEY this simple logic AND gate can be set to decide whether the result of the SD_TEST can be seen by higher levels or not.

The text of the default function is:

'and':
out := in1 \& in2

SD_KEY

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\SD_DET\SD_KEY' is a register. This register is 1 bit wide.
The default function is 'hold'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

The contents of this register can be set to 'open the gate' SD_GATE.
SD_TEST

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\SD_DET\SD_TEST' is an operator. It is use to compare the contents of the shift registers with the SD specific values. That is, the occurrence of the sequence 'JK0JK000' is tested. If the test succeeds, 'out' will go high, by definition for one clock cycle.

The text for the default function is:

'\SD_test':
out :=(in1 = %11011000) \land
       (in2 = %10010000).

6.19 Level 6: DATA/NDATA

REG8

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\ED_DET/SD_DET\DATA/NDATA\REG8' is a register.
This register is 8 bits wide.
The default function is 'load'.
The value following system reset is 0.
The value loaded for the 'reset' command is 0.

Figure 43: Level 6 schematic DATA/NDATA.

SHFT8

'DATALINK\TOKNRING\PHY2MAC\DETECTORS\ED_DET/SD_DET\DATA/NDATA\SHFT8' is an operator.
This operator is used to shift the contents of register REG8.

The text for the default function is:

'shift8':
out :=(in from:0 to:6), bit.
7 CONCLUSIONS AND REMARKS

Looking back at the achieved work, we can conclude that we achieved several goals (some of them not stated specifically in this report):

- We have gained a lot of knowledge about the design tool IDaSS. One addition to the tool is recommended, namely a sort of hidden operator-state-controller communication that resembles communication by means of signals.

- To achieve consistency in designing with IDaSS (making designing clearer and faster), a set of design rules was drawn up that can be used by any designer. This set is based on experience and is therefore especially a good guide to novice users of IDaSS.

- We have designed a part of the Token Ring LAN controller that fulfills the demands that were imposed: interfaces with external hardware (the MMTCP, for instance) are prepared or possible, the design is easy to read by those who have read the standard, compatibility with the standard is achieved, the design can be tested and demonstrated and it is a completed elementary function, and last but not least: the design is fast. Our goal was to achieve a one bit time delay per station, where we in fact reached a maximum delay of that duration. This makes the design interesting, as Texas Instruments achieves a minimum delay of $2^{\frac{1}{2}}$ bit times.

Furthermore, the following remarks can be made:

- The design can be extended as to fully implement the Operational FSM. The data flow charts for this FSM are already prepared.

- Later on, Monitor functions can be added. Reference [2] should then be used instead of reference [3].

- IDaSS has been proven to be a powerful tool, although some remarks can be made: First, a tool by itself makes the tool not very accessible: the manual of IDaSS should be accompanied by a discussion on how to use IDaSS in some specific situations. This is because especially novice users are probably staggered by the many possible solutions that IDaSS reaches for a given problem. Second, the graphical user-interface could be made more user-friendly, although this is not a strong item to concentrate on. Both items had their influence on the design rules drawn up in this report: using the design rules will lead to designing with as less drawing as possible, as drawing (and changing drawings) takes a lot of time, if one wants to do it neatly.
REFERENCES

[1] Verschueren, A.C.


THE UNIX SYSTEM. Addison Wesley Publishing Co., Reading, Ma., USA, 1982.


[8] Texas Instruments
TMS380 ADAPTER CHIPSET USER'S GUIDE, 1986.

[9] Verschueren, A.C.
IDaSS for ULSI V0.07 (A bare bones manual for the IDaSS package), Group Digital Systems, Department of Electrical Engineering, Eindhoven University of Technology, Netherlands, 1990.
APPENDIX A: THE OPERATIONAL FINITE STATE MACHINE. (as given by reference [3])

STATE 0 : REPEAT

- PDU_QUEUED & TK(PS\text{Pm})
- SFS(P=Pr,M=R=0), RESET (THT, MA_FLAG)
- TOKEN_ERROR V FR_FLAG V FR_FLAGS V FR_FLAGS CL_TK V STATION_ERROR
- TX_ABORT

BIT FLIPPING LOOP

- PDU_END & (QUEUE_EMPTY V TEST_THT)
- EFS(I=E=A=C=0), RESET (TRR, I_FLAG)

STATE 1 : TX DATA_FR

- RX DATA_FR
- TX_ABORT

STATE 2 : TX FILL & AWAIT MA

- RX FILL & AWAIT MA
- TX_ABORT

STATE 3 : TX FILL & STRIP FRAMES

- MA_FLAG_SET & Pr2Rr/Pm
- RESUME

STATE 4 : TX ZEROS & MOD STACKS

- RX>Sr
- TX_ABORT, STACK Sx=P

STATE 5 : TX FILL & STRIP SFS

- SFS_FLAG_SET V TRR_EXPIRED
- RX>Sr
- TX_ABORT, STACK Sx=P

Fig 4-3

Operational Finite-State Machine Diagram

<table>
<thead>
<tr>
<th>REF</th>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>02A</td>
<td>PDU_QUEUED &amp; (FR(R&lt;Pm) V TK(P&gt;Pm&gt;R,P≠Sx))</td>
<td>SET R=Pm</td>
</tr>
<tr>
<td>02B</td>
<td>FR_FLAG</td>
<td>SET E=1</td>
</tr>
<tr>
<td>02C</td>
<td>DA=HA (ADDRESS RECOGNIZED)</td>
<td>SET A=1</td>
</tr>
<tr>
<td>02D</td>
<td>FR_COPIED</td>
<td>SET C=1</td>
</tr>
</tbody>
</table>

Fig 4-4

Bit Flipping Loop State Table
APPENDIX B : DATA FLOW CHARTS FOR THE OPERATIONAL FSM.

Appendix B1: Alphabetical list of flow chart actions.

AC.R:=0 : Transmit a zero value for the reservation bits.
AC.R:=Pm : Transmit the PDU priority value as the new reservation value.
AC.R:=Rr : Transmit the receive reservation value as the new reservation value.
AC.P:=Pm : Transmit the PDU priority value as the new priority value.
AC.P:=Pr : Transmit the received priority value as the new priority value.
AC.P:=Rr : Transmit the received reservation value as the new priority value.
AC.T:=0 : Transmit a binary zero for the Token bit.
AC.T:=1 : Transmit a binary one for the Token bit.
AC.M:=0 : Transmit a binary zero for the Monitor bit.
AnyError : Test if any of the possible errors has occurred while the data passed.
BitsOK:=1 : Test if there were no code violations.
BYTE_E : Report the occurrence of a code violation.
CHECK BitsOK : Start a process that checks the validity of incoming symbols (code violations).
CHECK EDdet : Start a process that keeps track of an (abrupt) occurrence of an Ending Delimiter.
CHECK FCS : Start a process that calculates the Frame Check Sequence.
CHECK OnByte : Start a process that checks if data is placed correctly at byte boundaries.
CHECK TK_OK : Start a process that checks if the (rest of the) token is valid.
COMP ycs : Compare the calculated Frame Check Sequence with the received one.
COPY_FR : Start a process that copies the frame into an internal buffer.
DAr:=DA : Load the Destination Address into a register.
DAr:=MA : Test if the Destination Address equals My Address.
DQ PDU : Dequeue the enqueued PDU.
EDdet:=1 : Test if an Ending Delimiter is detected.
ED_CHK : Test the validity of the Ending Delimiter if appeared.
ED.E:EDx.E : Send the ready-to-go E bit in the ED.
ED.E:=1 : Test if the Error bit in the ED is set.
EDx.E:=0 : Set the Error bit that is to be transmitted to a binary zero.
EDx.E:=1 : Set the Error bit that is to be transmitted to a binary one.
ED.I:=0 : Test if the Intermediate Frame bit in the ED is set.
EDx.I:=0 : Set the Intermediate Frame bit that is to be transmitted to a binary zero.
EDx.I:=1 : Set the Intermediate Frame bit that is to be transmitted to a binary one.
FCr:=FC : Load the Frame Control field into a register.
FCs_E : Report the occurrence of a FCS error.
FrCopd:=1 : Test if the frame is correctly copied.
FS.A:=FSx.A : Send the ready-to-go A bits in the FS field.
FS.C:=FSx.C : Send the ready-to-go C bits in the FS field.
FSx.A:=0 : Reset the Address Recognized bits in the transmitted Frame Status field.
FSx.A:=1 : Set the Address Recognized bits in the transmitted Frame Status field.
FSx.C:=0 : Reset the Frame Copied bits in the transmitted Frame field.
FSx.C:=1 : Set the Frame Copied bits in the transmitted Frame field.
INIT_PDU : Initiate the transmission of the enqueued PDU.
I~ :=SET : Test if the Intermediate Frame flag is set.
LENGTH_E : Report the occurrence of a length error.
MA~ :=SET : Test if the My Address flag is set.
MONITR_E : If a Monitor Error occurs, report this.
MON_CHK : Check the monitor functions.
Mr:=AC.M : Load the Monitor bit into a register.
NrByte:=1 : Test if there were no length errors.
Appendix B2: Alphabetical list of flow chart actions (continued).

- **OnByte=1**: Test if there were no byte-boundary errors.
- **PLEVEL_E**: Report a priority level error.
- **Pm>Pr**: Test if the PDU priority value is greater than the received priority value.
- **Pm>Rr**: Test if the PDU priority value is greater than the received reservation value.
- **POP(Sr)**: Perform a POP operation on the Sr stack.
- **POP(Sx)**: Perform a POP operation on the Sx stack.
- **Pr:=AC.P**: Load the priority bits into a register for use by the Priority Operation Protocol.
- **Pr>Pm**: Test if the received priority value is greater than the PDU priority value.
- **Pr>Sx**: Test if the received priority value is greater than the stacked Sx value.
- **Pr=Sx**: Test if the received priority value equals the stacked Sx value.
- **PUSH(P)**: Perform a PUSH operation onto the Sx stack.
- **PUSH(Pr)**: Perform a PUSH operation onto the Sr stack.
- **QMT=1**: Test if the queue is empty (if there are no enqueued PDUs to transmit).
- **RESET I~**: Reset the Intermediate Frame flag.
- **RESET MA~**: Reset the My Address flag.
- **RESET SFS~**: Reset the Start of Frame Sequence flag.
- **RESET THT**: Reset the Token Holding Timer.
- **RESET TRR**: Reset the Return to Repeat Timer.
- **Rr:=AC.R**: Load the reservation bits into a register for use by the Priority Operation.
- **Rr>Pm**: Test if the received reservation value is greater than the PDU priority.
- **Rr>Pr**: Test if the received reservation value is greater than the received priority.
- **Rr>Sr**: Test if the received reservation value is greater than the Sx value.
- **SAr=MA**: Test if the Source address equals My Address.
- **SAr:=SA**: Load the Source Address into a register.
- **SET I~**: Set the Intermediate Frame bit flag.
- **SET MA~**: Set the My Address flag.
- **SET SFS~**: Set the Start of Frame Sequence flag.
- **SDdet=1**: Test if Starting Delimiter is detected.
- **STACK(P)**: Push the received priority value on the Sx stack.
- **START BCNdet**: Start a process that checks for the arrival of a BEACON frame.
- **START CalcFCS**: Start a process that calculates the FCS of the data that is to be transmitted.
- **START CLTdet**: Start a process that checks for the arrival of a Claim Token Frame.
- **START NrBits**: Start a counter that counts the incoming bits (needed for several purposes).
- **START NrByte**: Start a counter that counts the number of bytes (checking for length errors).
- **START PRGdet**: Start a process that checks for the arrival of a PURGE frame.
- **START StatnE**: Start a process that checks for station errors.
- **STATUSRA**: Make a status report.
- **STATUS24**: Report the specific status (see reference [3]).
- **StkMT=1**: Test if the highest stacked transmitted priority stack (Sx) is empty.
- **StkOK=1**: Test if all stack operations have finished (correctly).
- **Sx>Pr**: Test if the Sx value is greater than the received priority value.
- **Sx=Pr**: Test if the Sx value equals the received priority value.
- **THTexp=1**: Test if the Token Holding Timer has expired.
- **TRRexp=1**: Test if the Return to Repeat Timer has expired.
- **Tr:=AC.T**: Load the Token bit into a register.
- **Tr=0**: Test if the received AC field is part of a token or a frame.
- **TX ABORT**: Transmit an Abort Sequence.
- **TX DA**: Transmit the Destination Address field.
- **TX ED**: Transmit the Ending Delimiter.
- **TX FC**: Transmit the Frame Control field.
- **TX FCS**: Transmit the calculated Frame Check Sequence.
- **TX FS**: Transmit the Frame Status field.
- **TX FILL**: Transmit a FILL symbol (a zero).
Appendix B2: Alphabetical list of flow chart actions (continued).

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX INFO</td>
<td>Transmit the Information field.</td>
</tr>
<tr>
<td>TX SA</td>
<td>Transmit the Source Address field.</td>
</tr>
<tr>
<td>TX ZERO</td>
<td>Transmit a binary zero.</td>
</tr>
</tbody>
</table>
Appendix B4: Flow chart for the 'REPEAT' state.
Appendix B5: Flow charts for the 'MODIFY STACK' and 'STRIP FRAME' states.
Appendix B6: Flow chart for the 'TRANSMITDATA' state.
Appendix B7: Flow chart for the 'AWAIT MY ADDRESS' state.

[Flowchart image]

This situation should not be possible.