X.25 TRAFFIC GENERATOR

By

S.J. Aspin, B.Sc.(hons.)

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

SEPTEMBER 1986

Submitted to the University of Cape Town in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

The University of Cape Town has been given the right to reproduce this thesis in whole or in part. Copyright is held by the author.

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

ACKNOWLEDGEMENTS

The author wishes to acknowledge the inspiration of Professor Hugh Bradlow who supervised this thesis. Sincere thanks are also due to Mr. Ventura for his valuable interest, encouragement and assistance.

The author is indebted to the Post Office for providing the topic for this research project.

This thesis was partially funded by the Council for Scientific and Industrial Research (CSIR).

ABSTRACT

Due to the lower cost and error free data transmission capabilities offered by packet switching, numerous countries have installed national packet switched networks. The South African packet network, SAPONET-P, became operational in 1982 and has been growing rapidly, creating a need for network test equipment. This thesis describes the design of a high speed traffic generator which can be used to test and monitor the throughput capabilities of equipment or part of the network as a whole.

To meet the main requirement of the traffic generator, that it should support a number of high speed X.25 lines, a multi-processor architecture was chosen to cope with the high data throughput. An IBM PC was used as the base system, with several specially designed X.25 cards being installed in its expansion slots. The major part of the work done was on the design and development of the X.25 cards, each of which provides two high speed (64 Kbps) X.25 links.

In order to achieve this throughput, the card uses three processors coupled on a local bus to a 256K multi-port memory. Two WD2511 processors implement the link level of the X.25 packet switching protocol (LAPB), with the required software being micro-encoded on the chip. An 8088 processor, the same as is used in the PC, implements the packet level, the traffic generator and overall control of the card. Extensive use was made of programmable array logic (PAL) devices to implement the system logic required.

All programs for the traffic generator are written in the modern and powerful C language which is ideally suited to the application. The software was written in a modular fashion with the various modules being linked together by means of a set of common data structures. Use was made of packet buffers and job queueing to allow the traffic generator to cope with very high peak data rates. As well as programs for the X.25 cards, a monitor program runs on the PC and allows the user to view statistics screens and modify the traffic generator configuration.

While primarily designed for the traffic generator application, the X.25 card may also be configured for a variety of other networking applications. By substituting a local area network (LAN) processor for the X.25 one, the card can be used as a low cost network card or as a network file server. The card can also be configured to provide a low cost means of connecting a PC based workstation to the packet switching network. As all programs are downloaded onto the cards from the PC, it is relatively easy to modify or upgrade the software. Thus while meeting the original project requirements, the traffic generator design has a flexible and expandible nature.

Keywords:

X.25	packet switching	network
multiprocessor	computer	C

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	i
ABSTRACT	ii
TABLE OF CONTENTS	iv
LIST OF ILLUSTRATIONS	viii
GLOSSARY	x
CHAPTER 1: INTRODUCTION	
1.1 Packet switching and the X.25 protocol	1-3
1.2 SAPONET	1-5
1.3 The X.25 traffic generator	1-6
1.4 The future	1-8
CHAPTER 2: SYSTEM DESIGN	
2.1 Traffic generator	2-1
2.2 The X.25 Protocol	2-4
2.3 The WD2511 Link Level Processors	2-15
2.4 Hardware design and the IBM PC	2-20
2.5 Related work at other Universities	2-24
2.6 Design conclusions	2-27
CHAPTER 3: X.25 CARD DESIGN: THE ARBITRATOR	
3.1 The problem	3-1
3.2 Arbitrator design	3-8
3.3 The bus control PAL	3-13

CHAPTER 4: X.25 CARD CIRCUIT DESIGN

4.1 8088 Processor Module 4-2

4.2 WD2511 Processors 4-12

4.3 Memory, ready and write logic 4-19

4.4 PC Bus Interface 4-27

4.5 MUART and Line Drivers 4-33

4.6 Design Conclusions and Testing 4-39

CHAPTER 5: SOFTWARE DESIGN: THE C LANGUAGE

5.1 Introduction to C 5-2

5.2 Compiler Choice 5-9

CHAPTER 6: PROGRAM MODULES

6.1 Data Structures 6-3

6.2 X.25 Packet Level 6-7

6.3 The Executive 6-12

6.4 Traffic Generator 6-13

6.5 Link Level interface 6-15

6.6 Software conclusions 6-16

CHAPTER 7: PC PROGRAM

7.1 DOS Operation 7-1

7.2 Program loading 7-3

7.3 The X25TG program 7-7

CHAPTER 8: SYSTEM TESTING

8.1 Testing on the PC 8-1

8.2 System memory maps 8-5

8.3 Test results 8-7

8.3 Design validation 8-12

CHAPTER 9: CONCLUSIONS

9-1 Enhancing the traffic generator 9-2

9.2 Other applications of the X.25 card 9-4

9.3 Concluding remarks 9-7

LIST OF REFERENCES:

X.25 card related topics

Hardware and IBM PC reference manuals

C References

APPENDIX A: Hardware notes:

A.1 PAL equations and programming A-1

A.2 WD arbitrator design A-6

A.3 X.25 card assembly notes A-7

A.4 Hardware configuration A-8

A.5 X.25 card testing A-15

A.6 Components list and location diagram A-19

APPENDIX B: X25TG Main monitor and control program

APPENDIX C: SYSTEM, ADDRESS constant include files C-1

PACKET, LINK " " " C-5

X25TG.CON " " " C-10

DTYPES data type C-11

DVARS external variable declarations C-17

VDEFS variable definitions C-20

APPENDIX D: SYSTEMS executive D-1

TRAFFGEN traffic generator module D-17

APPENDIX E: PACTRANS packet transmit module E-1

PACREC packet received module E-12

LINKLEV link level interface E-30

APPENDIX F: Software notes:

F.1 Program compiling F-1

F.2 Program loading and TG.BAT F-3

F.3 Program running F-5

F.4 Thesis terms of reference F-6

F.5 WD2511 technical memo F-7

LIST OF ILLUSTRATIONS

PAGE	TITLE
Chapter 1:	
1-2	A microwave repeater station.
1-4	SAPONET structure.
1-7	The X.25 multiple-processor card.
Chapter 2:	
2-4	Fig 2.1 the ISO model and X.25.
2-7	Table 2.1 X.25 frame types
2-12	Table 2.2 X.25 packet types
2-14	Fig 2.2 The role of X.25 in SAPONET
2-17	Fig 2.3 WD2511 block diagram
2-18	Table 2.3 WD2511 register definitions
2-19	Fig 2.4 WD2511 system connection
2-23	X.25 card installed in an IBM PC.
Chapter 3:	
3-2	Fig 3.1 X.25 card local bus structure
3-4	Fig 3.2 Bus synchronization timing
3-11	Circuit diagram 1 : Bus arbitrator and control
3-13	Fig 3.3 The 16L8 PAL
3-16	Fig 3.4 Arbitrator cycle for a PC memory request
Chapter 4:	
4-1	Fig 4.1 X.25 card block diagram
4-2	Fig 4.2 The 8088 architecture
4-6	Fig 4.3 8088 mode timing comparison
4-8	Fig 4.4 The 8284A clock generator
4-9	Fig 4.5 A typical 8088 memory cycle
4-10	Circuit diagram 2 : The 8088 module
4-13	Fig 4.6 WD2511 I/O access cycle
4-17	Fig 4.7 A typical DMA cycle

4-18 Circuit diagram 3 : The WD2511 module

4-19 Fig 4.8 Memory cell structures

4-24 Table 4.1 X.25 card memory map

4-25 Fig 4.9 A typical PC memory access cycle

4-26 Circuit diagram 4 : The memory module

4-27 Table 4.2 PC memory address decoding

4-29 Table 4.3 PC I/O address map

4-30 Table 4.4 PC I/O address decoding

4-30 Table 4.5 PC interrupt table

4-32 Circuit diagram 5 : PC bus interface

4-33 Fig 4.10 MUART block diagram

4-35 Table 4.6 V.35 connector pin allocation

4-37 Circuit diagram 6 : MUART and line drivers/receivers

4-38 Circuit diagram 7 : Miscellaneous

Chapter 5:

5-1 Fig 5.1 Program structure

Chapter 6:

6-2 Fig 6.1 Software architecture

6-5 Fig 6.2 X.25 card system memory map

Chapter 7:

7-1 Fig 7.1 DOS structure

Chapter 8:

8-4 Fig 8.1 A typical statistics display

8-5 Fig 8.2 System memory maps

8-8 Fig 8.3 The main display

Appendix A:

A-22 Component location diagram

GLOSSARY

CCITT	:	International Telegraph and Telephone Consultative Committee.
DCE	:	Data Circuit Terminating Equipment.
DTE	:	Data Terminal Equipment.
ISDN	:	Integrated Services Digital Network.
ISO	:	International Organization for Standardization.
Kbps	:	Kilo bits per second.
LAN	:	Local Area Network.
PAD	:	Packet Assembler/Disassembler
PAL	:	Programmable Array Logic.
RS-232-C	:	Interface between data terminal equipment and data communication equipment employing serial binary data interchange.
SAPONET	:	South African Post Office Network.
SVC	:	Switched virtual circuit
SS No. 7	:	Signalling System Number 7.
V.35	:	Data transmission at 48 kilobits per second using 60-108 KHz group band circuits.
X.25	:	Interface between data terminal equipment (DTE) and data circuit terminating equipment (DCE) for terminals operating in the packet mode on public data networks.
X.29	:	Procedures for the exchange of control information and user data between a packet assembly/disassembly facility (PAD) and a packet mode DTE or another PAD.

INTRODUCTION

This thesis describes the design of a very high speed X.25 traffic generator. It is to be used to test the throughput capabilities of X.25 switching nodes and equipment in the South African X.25 packet switching network (SAPONET). This chapter introduces the concepts and uses of packet switching and the X.25 protocol and describes the applications of the traffic generator.

Digital electronics is a very new and rapidly evolving technology which really started with the invention of the transistor by Shockly and associates at the Bell telephone laboratories in 1948. In 1959 Texas Instruments unveiled the first integrated circuit and in 1971 Intel introduced the first single chip micro-processor, the 4004. Since then micro-computers have quickly become part of everyday life, from robots at street intersections to computerized banking systems.

With the flourishing of computers, it soon became necessary to connect them together. This involves both linking together computers in a building in order to share resources (printers, data bases etc.,) and linking computers on a country wide scale. For example, if a Capetonian draws money from an autobank teller in Johannesburg, the transaction would have to be reported to the computer where the account is kept, e.g., in Cape Town. Clearly a low cost reliable communications media is required to link the computers together.

The obvious communications media to use is the telephone system with the telephone handset being replaced by a modem. However, due to its analog nature, the telephone system is not ideal for computer use - it is noisy, only permits low data transfer rates and for long distance calls it is expensive.

These inadequacies led to the development of digital packet switching networks, which allow numerous users to transfer data over the same physical connection, thereby using it efficiently. Today, with the continuing relative decline in computer processing cost, there is a worldwide trend towards digital switching and transmission of voice as well as data. Research is also being done to adapt packet switching networks for use with voice.

For long distances voice and data are transmitted by means of expensive microwave links. Packet switching allows such links to be used more efficiently.

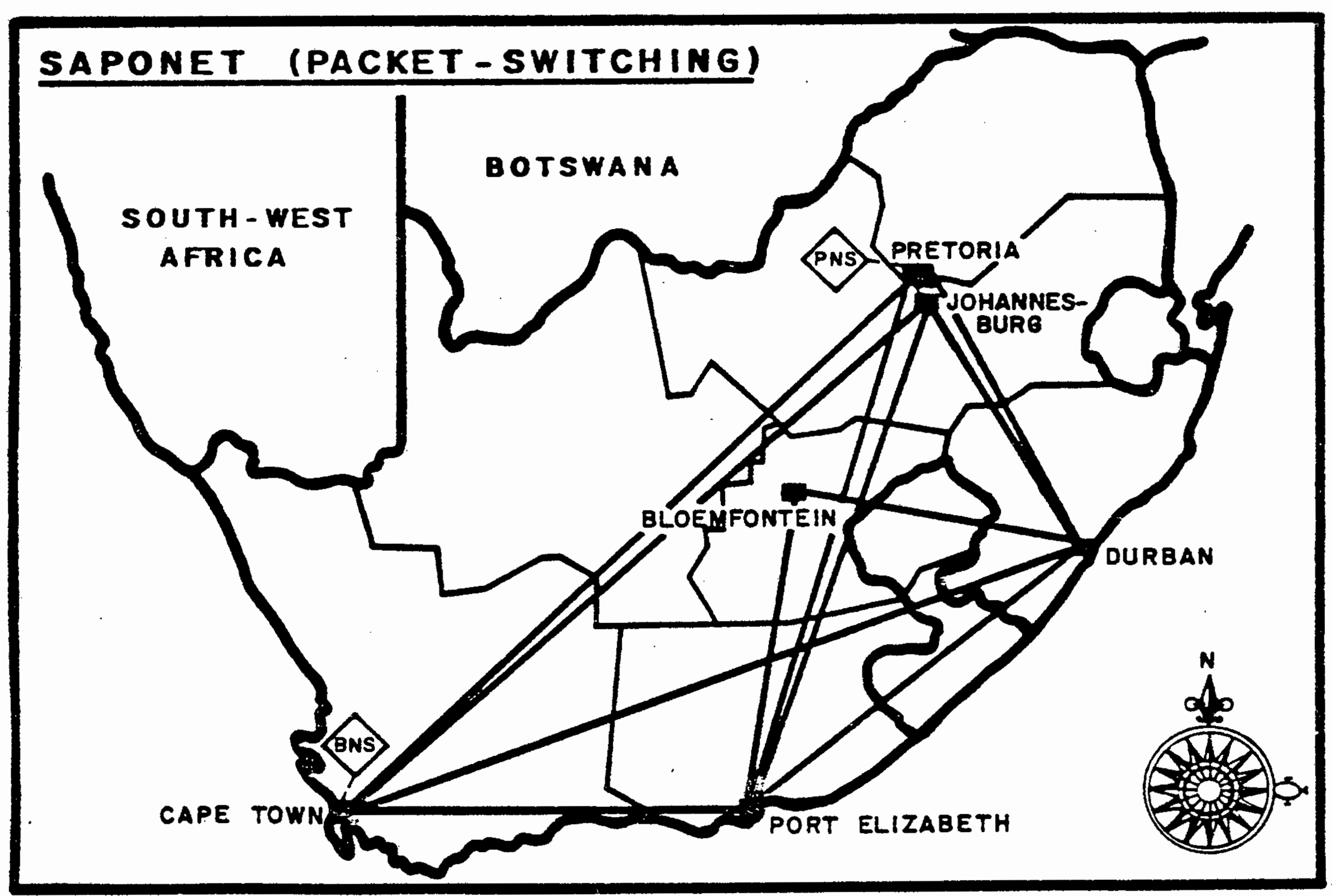
1.1 Packet switching and the X.25 Protocol:

In the early seventies several experimental computer networks were set up to investigate means of providing cheap reliable digital communication. The most notable was the ARPANET network in America which linked together different computers at various research institutes (see Kuo 1981 for an historical introduction). What evolved from this work was the concept of packet switching - splitting the data to be sent into fixed sized addressed packets. Thus rather than hiring a permanent communication link from say Cape Town to Johannesburg, the user sends packets of data to SAPONET Centre Exchange in Cape Town. The exchange or DCE (data circuit equipment) transmits packets from various sources over the same communications link thereby using it efficiently. This utilization of available bandwidth results in a much lower cost for the user (see Bradlow and Verkroost for an analysis of data switching techniques).

Before transferring data on a packet switched network, providing external virtual circuit services, a user must first set up a call by sending a call request packet to the DCE, specifying the called subnetwork address. This system allows the network controller to route calls and thus packets through the network. For example if the Cape Town to Johannesburg primary link is becoming congested, the network controller can re-route via secondary channels (e.g. Cape Town - Durban - Johannesburg). A good routing algorithm thus ensures minimization of the packet delay, with a traffic generator allowing one to determine the throughput characteristics of the network switches.

As well as providing cheaper digital communications, packet switching ensures good data integrity. For each frame of data transmitted, a check character is computed and tagged on with corrupted frames being retransmitted. As there is also provision for flow control and packet sequencing, reliable digital communications are ensured.

Clearly some standard was required to provide these features and in 1976, based on experience gained on the ARPANET and other networks, the CCITT (Committee Consultatif International de Telephonie et Telegraphie) defined the X.25 protocol. X.25 defines precisely how a packet mode DTE is to connect to a packet switched network node (DCE).



1.2 SAPONET-P

X.25 was revised in 1980 and 1984 and has rapidly gained universal acknowledgement. This standardization prompted many countries, including South Africa, to set up their own packet switched networks. Especially with the country's large geographical distances, packet switching results in substantially cheaper digital communication and in 1980 the Post Office contracted a supplier to install the necessary data processing equipment. The packet switching network, SAPONET, became operational in February 1982 and supports the 1980 CCITT X.25 recommendation.

The main control centres of the network are Cape Town and Pretoria, with Pretoria also providing an international gateway to Britain and America. The network started with $20 million of packet switching equipment (Knott Craig 1982) and has been growing rapidly. Transmission cost is distance independent, which makes it particularly attractive for long distance data transfer.

Finally, for those users unable to support X.25, the Post Office supplies asynchronous PAD's at the exchanges. These PADs comply with the X.3, X.28 and X.29 recommendations. However the use of these PADs does unfortunately mean that the user is no longer guaranteed error free transmission; the X.25 card described in this thesis may be enhanced to bring X.25 capabilities direct to an IBM PC based workstation.

1.3 The X.25 Traffic Generator:

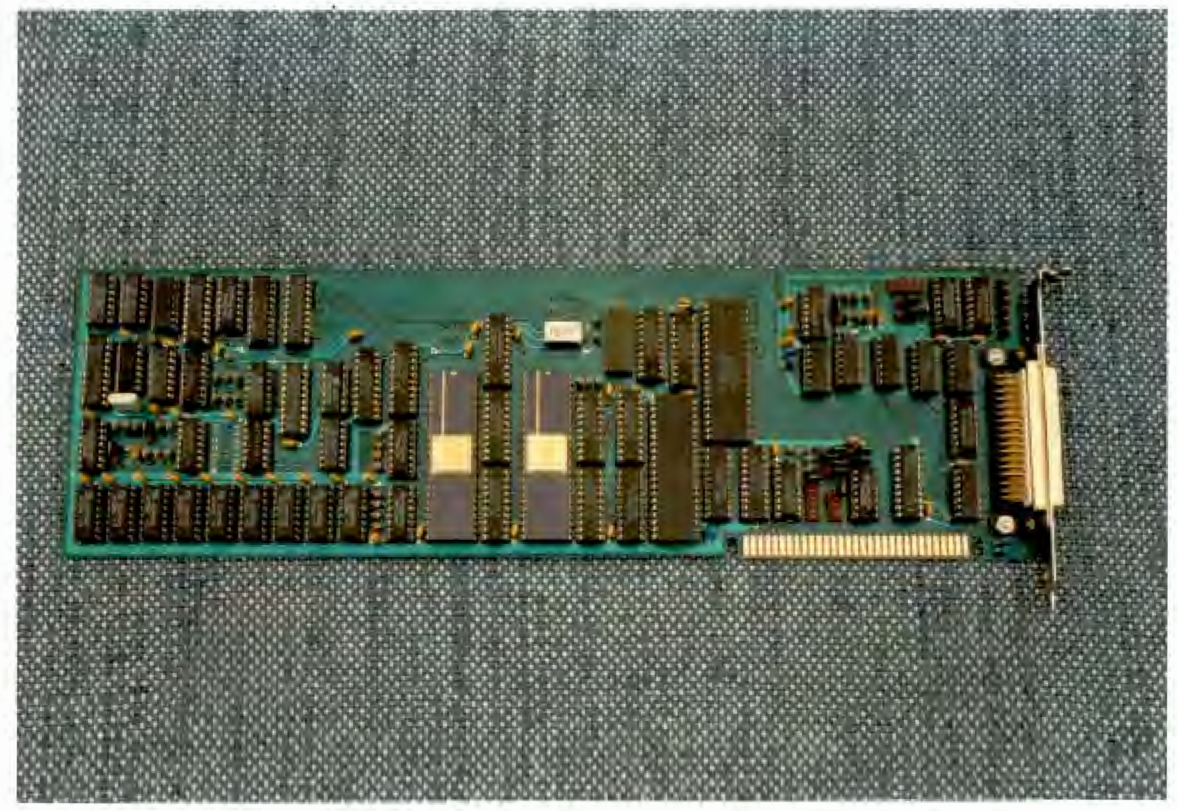
With the establishment and growth of SAPONET, there came a need for network test equipment. This thesis describes the design of a high speed X.25 traffic generator which may be used to exercise or load the packet switching nodes with large volumes of traffic. A specification was provided by the Post Office, (see Appendix F) with the main requirement being that the unit should support four, but preferably up to eight, 64Kbps links.

The intended application of the traffic generator is to load data links with high traffic volumes and to monitor how the network copes. This would enable SAPONET authorities to detect possible bottlenecks under controlled conditions. The traffic generator may also be used in the commissioning of new packet switching processors. It will allow the Post Office to easily test how throughput of the processors vary with traffic loads. Once again, this is a form of preventitive maintenance, enabling the Post Office to detect and correct possible problems and thereby provide a better service to the users.

As well as the speed requirement, the traffic generator had to set up and disconnect calls on several logical channels, generate and receive test data packets and display statistics. Thus is has to implement the basics of the X.25 protocol and a multiple-processor system was designed to provide the required processing power.

As this was a large project, throughout the design every effort was made to make full use of available equipment. Thus, after a review of possible alternatives, such as Multibus cards, the IBM PC was chosen as the base of the system. Most of the project work was then focused on the design and testing of a sophisticated X.25 card, several of which may be installed in a PC. The system design, circuit design, and related software are described in the following chapters.

While pimarily intended as a high speed X.25 sub-system, the card may also be configured for a variety of other applications, including providing high reliability PC to PC communication or implementing a token passing local area network. It may also be used as an asynch. communication card, as an auxilliary processor or simply as a memory board which would be able to double the IBM PC's available memory capacity.



The X.25 card designed in this thesis.
A multiple-processor architecture is used to obtain the high throughput required.

The Future:

Initially there was the analog channel and computer manufacturers have developed means of transmitting data over it (eg., using modems). However, there were problems both with regard to performance and cost. This led to the introduction of sub-networks, such as SAPONET, which provide low cost reliable digital communication. Today, with the continuing relative decline in the cost of computer power, the wheel has turned full circle and manufacturers are now trying to adapt digital networks for use with voice.

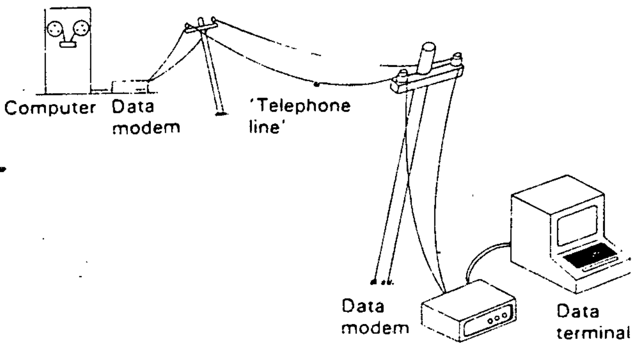
Thus the current research is directed towards digitizing voice efficiently and transferring it over digital networks. For example AT&T and an American packet switching network, Tymnet, are currently conducting tests to transfer voice through packet switched networks (Mier July 1986), while at U.C.T. work has also been done in this regard. Just as the analogue telephone system was not entirely suitable for voice, so the digital networks have their problems - particularly with regard to signal delay and the digital speech coding.

Speech is normally digitized by means of pulse code modulation (PCM) resulting in a 64Kbps data stream. However, this is highly redundant and so work is being done on more efficient coding, or alternatively in interleaving data with the coded voice. Once again, research has been done at U.C.T. in this field (e.g., Irvine 1985), while the CCITT is currently busy trying to formulate the new standards (Schmidt 1986).

The delay problem is due to the error correcting mechanisms employed in digital networks as the mechanism used in X.25 is fairly slow. This has led to the development of new protocols such as Signalling System No. 7, (SS No. 7), specially designed for minimal delays (see chapter 2). It should be noted that X.25 card designed in this thesis can be configured to implement SS No. 7, giving further interesting research possibilities.

In the long term, research is being done to effectively integrate voice and data onto the same digital network. With the increasing usage of optical fibers and satellite links which require digitized voice, one may as well digitize the voice at the user's premises rather than at the exchange. This has resulted in the concept of an integrated services digital network (ISDN): bringing a clear 64Kbps line to the user's premises with SS No. 7 being used for all the signalling requirements (Koehr 1985).

Thus packet switching and X.25 are part of a rapid swing to digital networks, spurred on by the economic advantages offered. The S.A. Post Office has for example recently introduced the circuit switched DIGINET network as well as upgrading SAPONET. It was against this dynamic background that the traffic generator was designed.



Typical representation of a data communications link.

SYSTEM DESIGN

This chapter describes the X.25 interface in more detail and introduces the WD2511 processor used to implement the link levels of the traffic generator. It then discusses the choice of hardware configuration and gives a brief review of related work at other Universities. First, what were the project requirements?

2.1 Traffic generator specifications

The main requirement of the traffic generator is that it should support at least four, but preferably up to eight 64 Kbps links (final design supports up to ten). The network nodes to which the traffic generator is to be connected have a throughput ranging from 300 to 500 Kbps, so eight 64 Kbps links would be required to fully load a 500 Kbps processor.

The network processors can be configured to be either a DCE or a DTE, with the usual configuration being as a DCE. The traffic generator was therefore designed to be a DTE and as such expects the network processor to provide the timing signals for the transmit and receive lines. The physical interface required for the traffic generator is the CCITT recommendation V.35. As well as providing the V.35 interface, the traffic generator also implements the more common RS232C interface thus allowing it to be connected to slower X.25 equipment or a modem (note that the modem acts as a DCE, supplying the required timing).

In addition to being connected to a DCE, the traffic generator links may also be connected to a DTE to provide DTE to DTE communication. This configuration allows two of the traffic generator links to be connected together for loop back testing of the unit. A parameter passed to the traffic generator program at start up informs the program if it is to be configured for loop back testing or for normal operation. When operating in loop back mode, the link timing is easiest obtained from a signal generator, although the card does provide a test clock.

With regard to the software, the traffic generator was required to support at least six logical channels per link with the traffic generator actually being able to support several hundred per link. The traffic generator can in fact support up to 1023 logical channels per link, although in practice one or two hundred is a more practical limit as the increased processing overhead tends to slow down the system. The specified testing procedure for the traffic generator was that it should establish twelve calls every 30 seconds within the same 1 second interval and then disconnect after 25 seconds. This process was then to be repeated for the duration of the exercise. In the actual traffic design, these parameters were all made configurable with the user being able to change them both at system start up and during the test.

The configuration screen allows the user to display and change the configuration for each individual link of the traffic generator. Alternatively, all links may be re-configured at the same time. If configuration is not performed, then default parameters are loaded at start up. Call length and spacing may be varied from 1 second to 50 minutes while from 1 to 100 calls may be set up per second. In addition, the number of active channels on a link may be varied.

There was no specification regarding what test data packets should be sent, the traffic generator currently sends a data field of around 120 bytes ie., just less than the standard maximum of 128 bytes. The software is currently being extended to allow the user to select maximum data field sizes of 256 and 64 bytes in addition to the default 128 bytes, although the length used would depend on the configuration of the network processor. In addition to tranferring packets, the traffic generator was required to display statistics on the link and packet levels. Although a total of only seven statistics were required, the traffic generator has four selectable statistics screens (main, systems, packet level and link level) providing over 30 different parameters, ranging from packet and frame totals to the overal system status. The statistics screens are updated once a second for the duration of the test.

In addition to the above specifications, it is also obviously desirable to have professional looking and reliable piece of equipment. The use of an IBM PC for the system base and the design of a printed circuit board for the X.25 card help ensure the above. Certain features were incorporated in the design to ensure relatively easy software maintenance and upgrading, without the need for any extra software development facilities. Thus the traffic generator is easy to use, configurable and easily expandable.

2.2 THE X.25 PROTOCOL AND THE ISO MODEL:

As communication networks must provide a variety of functions, it becomes convenient to split the network services into a number of layers - each layer performing a well defined protocol. The International Standard of Organization (ISO) proposed the "reference model of Open Systems Interconnection (OSI)"; where X.25 fits into the OSI architecture is illustrated below.

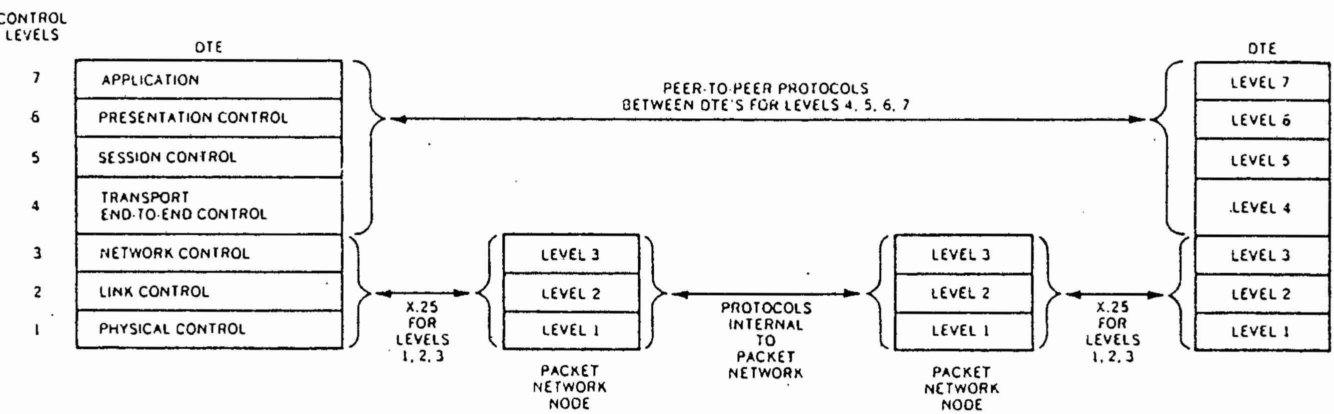


Fig 2.1 X.25 and the ISO reference model

X.25 defines the lower three levels ie. the interface between the user and the network node or DCE. The upper layers define end to end protocols. These would be required (in some form) to set up communication over a network and maintain dialogue between the end users. The functions of layers are thoroughly discussed in Computer Networks (Tannenbaum 1981).

Thus the traffic generator (an end user) must implement the physical, data link and some of the network layer functions as defined by X.25 (CCITT 1980). As this was a large part of the thesis work, a very brief description of important aspects of the layers is given below; the reader is referred to the references (eg. Sloman 1978, Karp 1981, SAPONET 1982) for more information.

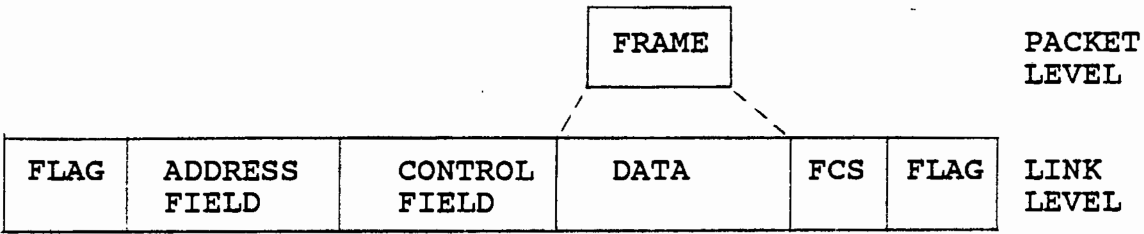
2.2.1 The Physical level

The physical level is concerned with the actual encoding and physical transmission of data bits over the communications channel. The electrical requirement was V.35, which is used for high speed (64 Kbps) data links. The X.25 card also provides the more common RS 232C interface to allow it to be connected to conventional devices (eg. modems).

Note that the actual X.25 specification prescribes the use of X.21, with X.21 bis (similar to RS232C) being used for an interim period. All data transfer is synchronous and full duplex.

2.2.2 The Link level

The link level provides an error free controlled pipeline for transmitting packets between the DTE and DCE. The level is thus responsible for link setup and disconnection as well as error detection and subsequent error recovery. It does this by adding control information to the frame, as illustrated below:



The start and end of the frame is indicated by means of a flag character, coded as 0111 1110. In order to ensure that user data can be transparently transmitted, bit stuffing is performed by inserting a 0 after any sequence of five 1s. On the receiver side, whenever five 1s are received followed by a zero, the zero bit is removed.

When frames are exchanged they are associated in terms of commands and responses as distinguished by the address field. Commands from DCE to DTE and responses from DTE to DCE carry address A (hex 03), while commands from DTE to DCE and responses from DCE to DTE carry address B (hex 01).

Note that X.25 only specifies DCE to DTE communication, so if the traffic generator is used in a loop back test (DTE to DTE), the program will need to set address A equal to address B.

The control field is the next byte in the frame and is used to distinguish between information, supervisory and control frames. In the original 1976 X.25 recommendation a symmetrical configuration, referred to as LAP (for link access protocol), was used. This required the link first to be initialized in one direction and then in the other in order to set it up. Problems were detected with this configuration - particularly in that if the DTE failed and restarted the link in one direction, the DCE may not reinitialize it in the other and a deadlock could occur (Sloman 1978). Hence in 1980 a balanced configuration (LAPB) was adopted in which one command initializes the link in both directions. To avoid incompatibility, it was agreed that all networks should implement the 1980 standard by January 1982 (Drukarch et al 1981), thus in this thesis only LAPB procedures are considered.

			CONTROL FIELD				
FRAME TYPE	COMMAND	RESPONSE	BIT #				
			7 6 5	4	3 2 1	0	
I-FRAME	I-FRAME		N(R)	P	N(S)	0	
S-FRAME	RR	RR	N(R)	P/F	0 0 0	1	RECEIVER READY
	RNR	RNR	N(R)	P/F	0 1 0	1	RECEIVER NOT READY
	REJ	REJ	N(R)	P/F	1 0 0	1	REJECT
U-FRAME	SABM		0 0 1	P	1 1 1	1	SET ASYN-CHRONOUS BALANCED MODE
	DISC		0 1 0	P	0 0 1	1	DISCONNECT
		DM	0 0 0	F	1 1 1	1	DISCONNECT MODE
		UA	0 1 1	F	0 0 1	1	UNNUMBERED ACKNOWL-EDGE
		FRMR	1 0 0	F	0 1 1	1	FRAME REJECT

Only the FRMR and I-frame contain I-fields
P = Poll Bit F = Final Bit

Table 2.1 X.25 LAPB frame types

Therefore to establish a link, in both directions (DTE to DCE and DCE to DTE) the traffic generator's link entity sends a SABM frame (set asynchronous balanced mode) to the DCE and waits for the DCE to return a UA frame (unnumbered acknowledgement). The frame poll bit (P) is used to solicit a response from the combined station. The link level timer (known as T1) is started at the beginning of a transmitted command, provided it has not been previously started. If the timer expires, the frame is retransmitted with P=1 and if after several attempts (counted by a counter N2) there is still no response, then the link will take appropriate recovery action. As one of its statistics, the traffic generator keeps track of how often the T1 timer expired as well as how often the link was initialized.

The actual packets are conveyed by means of the information or (I) frames. The receiver must acknowledge all frames received and this is done by sending back the expected sequence number, N(R) of the next received I frame. As N(R) is three bits long the transmitter can have up to 7 outstanding or unacknowledged frames before is has to stop sending. Acknowledgement can be sent back via a RR (receiver ready) or RNR (receiver not ready) frames, or alternatively it may be "piggyback" onto an I frame. A REJ (reject) frame may be sent back if an out of sequence I frame was received.

Following the control field is the information field containing the actual packet (only in I frames), passed to the link level from the packet level above.

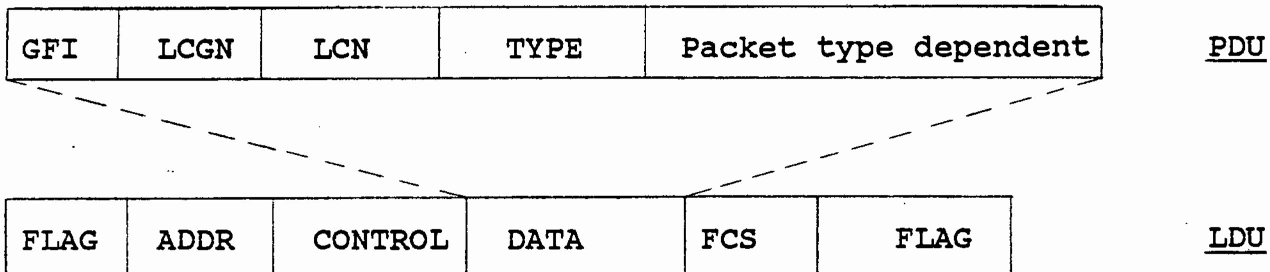
Finally there is the frame check sequence (FCS) which is a sixteen bit cyclic redundancy check character used for error detection. If a frame has an FCS error, then it is ignored by the receiver. The next frame will be out of sequence resulting in a reject frame being returned to the transmitter. The transmitter will go back and retransmit all frames from the bad one onwards. This raises the question of what happens if the REJ frame also gets corrupted? Here the T1 timer will expire and result in retransmission. Thus the link level provides error free communication and can cope with all possible link errors.

It is perhaps interesting to note that the error correction system is slightly different for the link level of Signalling System no. 7. In SS no. 7, a NACK (negative acknowledge) is returned to the transmitter to speed up retransmission of the single bad frame. An alternative procedure for high delay satellite links is to cyclically retransmit unacknowledged frames when there are no new ones to send (Roehr 1985).
As the link level of X.25 has been fully standardised as defined above, there became a real possibility that a manufacturer would cast LAPB on silicon. A company, Western Digital, has in fact done this and the chip, the WD2511, is described later in this chapter.

More recently, Motorola has also introduced a VLSI X.25 chip, the MC68605, designed to be interfaced to the 68000 series of processors (Erikson Sept. 1985). The chip is basically the same as the WD2511 except it has a full 32 bit address bus and a 16 bit data bus.

2.2.3 The Packet level

The basic function of the packet level is to time division multiplex a number of logical channels onto the same link, thus allowing users to share common resources (eg. bandwidth, DCE processors and memory). The packet level also provides facilities for data flow control and packet sequencing. Packets are transported over I frames as shown below:



The first two bytes of a packet contain the 4 bit general format identifier, the 4 bit logical channel group number and the 8 bit logical channel number. Often the channel group number and channel are just regarded as a 12 bit logical channel number and this is the notation followed here.

When setting up a call, the DTE (traffic generator) uses the highest free logical channel allocated to SVCs and sends a call request packet to the DCE. If the call is accepted then from then on that particular logical channel will be assigned to'that user. Similarly a second user will be assigned the second channel number and so on. Thus the channel number provides the mechanism whereby several users can communicate over the same link.

As the channel number is 12 bits long, there is a maximum of 4095 different logical channels (channel 0 being reserved). The actual maximum number used depends on the administration, with SAPONET supporting 1023. Thus the traffic generator must set up calls from the highest configured channel downwards.

At the other side of the network, the remote DCE sends a call indication to the remote DTE using the lowest configured free channel number. This use of high and low channel numbers is intended to avoid possible call collisions.

However, there is an important implication for the traffic generator. For example, suppose a link is being used in a loop back test (ie. the transmit lines are directly connected to the receive lines to give DTE to DTE communication). A call request sent out on channel 1023 would result in an incoming call being received on channel 1023! Thus the traffic generator must have provision, on the receive side, to subtract the channel number from 1024 if in loop back mode.

Finally, for a traffic generator to simulate a large number of users, it is clearly advantageous to be able to set up a large number of logical channels. The minimum specified was six but, by providing ample processing power and a large amount of memory, the traffic generator is actually able to support several hundred.

The next byte in the packet header is the packet type identifier and these are listed in the table 2.2 on the following page.

Two types of virtual circuits are offered: switched virtual circuits (SVC) and permanent virtual circuits (PVC). SVCs have to be set up before information transfer and cleared afterwards, while permanent virtual circuits are brought up in the data transfer state. There are also datagrams, but these have not met with general acceptance and are not supported by SAPONET. Thus the traffic generator supports virtual circuits only.

Packet type		Octet 3 Bits							
From DCE to DTE	From DTE to DCE	8	7	6	5	4	3	2	1
Call set-up and clearing									
Incoming call	Call request	0	0	0	0	1	0	1	1
Call connected	Call accepted	0	0	0	0	1	1	1	1
Clear indication	Clear request	0	0	0	1	0	0	1	1
DCE clear confirmation	DTE clear confirmation	0	0	0	1	0	1	1	1
Data and interrupt									
DCE data	DTE data	X	X	X	X	X	X	X	0
DCE interrupt	DTE interrupt	0	0	1	0	0	0	1	1
DCE interrupt confirmation	DTE interrupt confirmation	0	0	1	0	0	1	1	1
Datagram [a]									
DCE datagram	DTE datagram	X	X	X	X	X	X	X	0
Datagram service signal		X	X	X	X	X	X	X	0
Flow control and reset									
DCE RR (modulo 8)	DTE RR (modulo 8)	X	X	X	0	0	0	0	1
DCE RR (modulo 128)[a]	DTE RR (modulo 128)[a]	0	0	0	0	0	0	0	1
DCE RNR (modulo 8)	DTE RNR (modulo 8)	X	X	X	0	0	1	0	1
DCE RNR (modulo 128)[a]	DTE RNR(modulo 128)[a]	0	0	0	0	0	1	0	1
	DTE REJ (modulo 8)[a]	X	X	X	0	1	0	0	1
	DTE REJ (modulo 128)[a]	0	0	0	0	1	0	0	1
Reset indication	Reset request	0	0	0	1	1	0	1	1
DCE reset confirmation	DTE reset confirmation	0	0	0	1	1	1	1	1
Restart									
Restart indication	Restart request	1	1	1	1	1	0	1	1
DCE restart confirmation	DTE restart confirmation	1	1	1	1	1	1	1	1
Diagnostic									
Diagnostic [a]		1	1	1	1	0	0	0	1

[a] Not necessarily available on every network.

<u>Table 2.2 X.25 Packet type identifiers</u>

Upper layer information is transferred in the data field of a data or an interrupt packet. The coding of the packet type identifier for data packets is depicted below:

Packet type :	P(R)	M	P(S)	0

The P(R) and P(S) sequence number are very similar to the N(R) and N(S) of the link level, but are used for packet sequencing and flow control rather than error control. As shown above, they are three bits wide, with a configurable window size of up to 7 (the default is 2). Two bits of the general format identifier also allow the user to optionally select modulo 128 sequence numbering thereby increasing the window to a maximum of 127 (currently not supported by SAPONET).

The transport handler passes messages to the packet level which then splits the message into packets with a maximum length depending on the current configuration.

The default length is 128 bytes and so the traffic generator's default test data packets were selected to have about 120 bytes.

The packet level is not as clearly defined as the link level and unfortunately no manufacturer has attempted to implement it in silicon. A packet level program was therefore written, with the object code running on the X.25 card's 8088 processor. It should be noted that, being written for use in a traffic generator, the program was designed for high speed operation rather than a full implementation of the X.25 recommendation.

Finally it should be remembered that X.25 only specifies the DTE/DCE interface. Thus in the event of a channel reset, it is up to the upper level (eg. transport handler) to work out what packets, if any, were lost and take appropriate recovery action (Tannenbaum 1981). This consideration does not affect the traffic generator design but should be borne in mind if the system is used in other applications.

We have now looked briefly at the major features and aspects of X.25 with particular reference to the traffic generator design. The diagram below (fig 2.2) serves to summarize the layering of X.25 and its scope within the packet switching network.

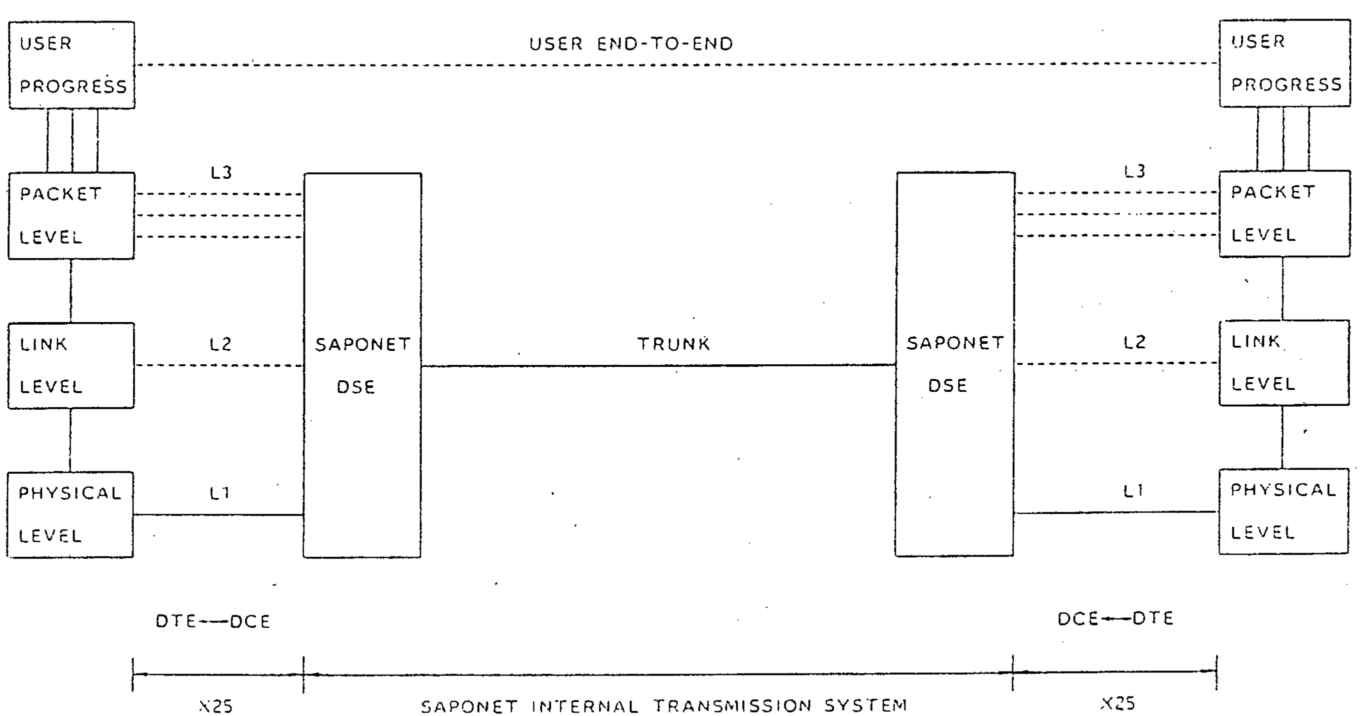


Fig 2.2 <u>The relationship of the three levels of X.25 to the network as a whole (SAPONET X.25 publication 1983)</u>

2.3 THE WD2511 LINK LEVEL PROCESSOR

The objective of the thesis is to design a traffic generator providing 8 high speed X.25 lines. Thus a major part of the work involved was in implementing X.25. Clearly any ready designed systems were not to be ignored.

Most of the protocol controllers on the market, such as the Intel 8273, are able to implement the bit stuffing and flag appending and removing required. They are also able to calculate the frame check sequence and automatically append it to the end of the frame. However, the user still has to code the actual link level protocol ie., send the SABM to set up the link, implement error recovery, provide the T1 timer etc.

In 1982 Western Digital introduced the WD2501 - a single chip processor which implemented the LAP protocol. This was subsequently updated to the WD2511 which implements LAPB which is used in this thesis. The chip, housed in a 48 pin package, implements the entire link level protocol and includes the T1 timer and N2 counter mentioned earlier.

Internally the chip consists of three micro-controllers coupled to an 11K ROM (Ledger 1981). Interfacing to the rest of the system is by means of a DMA (direct memory access) channel, thus allowing the WD2511 to transmit and receive frames without the host intervention. This architecture allows the chip to operate at very high speeds - 100K bps for the standard part and up to 1.1 Mbps for high speed versions.

Currently priced at around $35 each, the processors are not too expensive, although this does work out to R140+ in Rand terms. As well as providing all the link level programs, the extra processing power is most welcome in applications such as a traffic generator. Hence one of the first design decisions taken was to use the WD2511 chip and the traffic generator was accordingly designed around them.

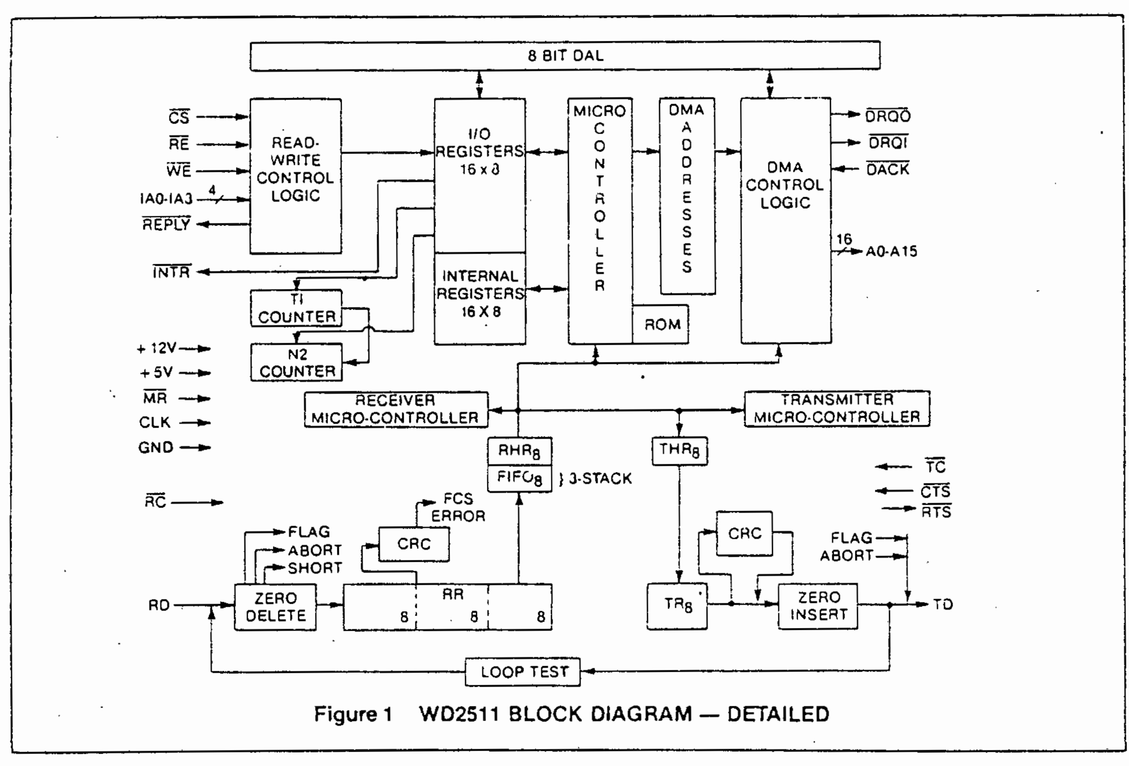


Figure 1 WD2511 BLOCK DIAGRAM — DETAILED

Fig 2.3 <u>WD2511 block diagram</u>

2.3.1 WD2511 hardware interface

The WD2511 acts as a slave processor to a more general purpose CPU (central processing unit) and relies on the host for control. The host processor must also implement the packet level and the traffic generator functions. In the final design it was decided to use an Intel 8088 CPU for each pair of WD2511 processors on a card.

The WD2511 accesses memory by means of two DMA lines, one for read and one for write. However, apart from signalling that it wants memory, the WD2511 has little interaction with its environment. The user must provide circuitry to sense the request lines, get the 8088 CPU off the memory bus, enable the address and data buffers and start the memory cycle. To make it more difficult, the WD2511 processors have a slow memory cycle time when compared to the 8088. While this does not worry the WD2511, which has a small internal FIFO (first in first out) buffer in its receiver, it requires a fair amount of design work to ensure that the 8088 CPU is not unduly held up.

Hence the WD2511 is not an easy chip to design into a circuit and considerable effort was required to design a reliable high speed system. One must remember that the WD2511 is a complete system in its own right rather than just an intelligent peripheral like the 8273.

2.3.2 SOFTWARE INTERFACE

Control and monitoring of the WD2511 is provided by means of its 16 internal registers as shown in table 2.3 below.

REGISTER DEFINITION

REG #	IA3	IA2	IA1	IA0	REGISTER	REGISTER GROUPING
0	0	0	0	0	CR0	OVERALL CONTROL AND MONITOR
1	0	0	0	1	CR1	
2	0	0	1	0	*SR0	
3	0	0	1	1	*SR1	
4	0	1	0	0	*SR2	
5	0	1	0	1	*ER0	
6	0	1	1	0	*CHAIN MONITOR	RECEIVER MONITOR
7	0	1	1	1	*RECEIVED C-FIELD	
8	1	0	0	0	T1	TIMER
9	1	0	0	1	N2/T1	
A	1	0	1	0	TLOOK HI	DMA SET-UP
B	1	0	1	1	TLOOK LO	
C	1	1	0	0	CHAIN/BUFFER SIZE	
D	1	1	0	1	NOT USED	
E	1	1	1	0	XMT COMMAND "E"	"A" FIELD
F	1	1	1	1	XMT RESPONSE "F" (Note 1)	

*CPU READ ONLY. (Write Not Possible)

Table 2.3 WD2511 register definitions

The first two registers are used for actual control of the WD2511 - such as instructing it to set up the link or transmit a frame. There are also four registers which tell the user the memory block number of the next frame to be received, what significant events have occurred etc. Note that the user can program the address field values (thus permitting loop back testing) and link timers. There is also a facility to monitor what frames are being received via register 7.

To access frames the WD2511 uses a system of memory look up tables, one for transmit and one for receive. Each table has 8 segments which contain the memory address, length and control information about a frame. Hence once the packet level has built a packet to be transmitted, a link interface program will have to set up a table segment and tell the WD2511 to send the packet. These routines are described later in chapter 6.

Thus the WD2511 represents a ready proven VLSI design solution for the link level. The chip has been certified for use on the Telenet and Tymnet networks in the U.S.A. as well as several other overseas networks (Western Digital 1986). In addition, Western Digital offer two other pin compatible chips (WD2840 token passing LAN controller and WD2507 for S.S. No 7), thereby increasing the potential applications of the X.25 card.

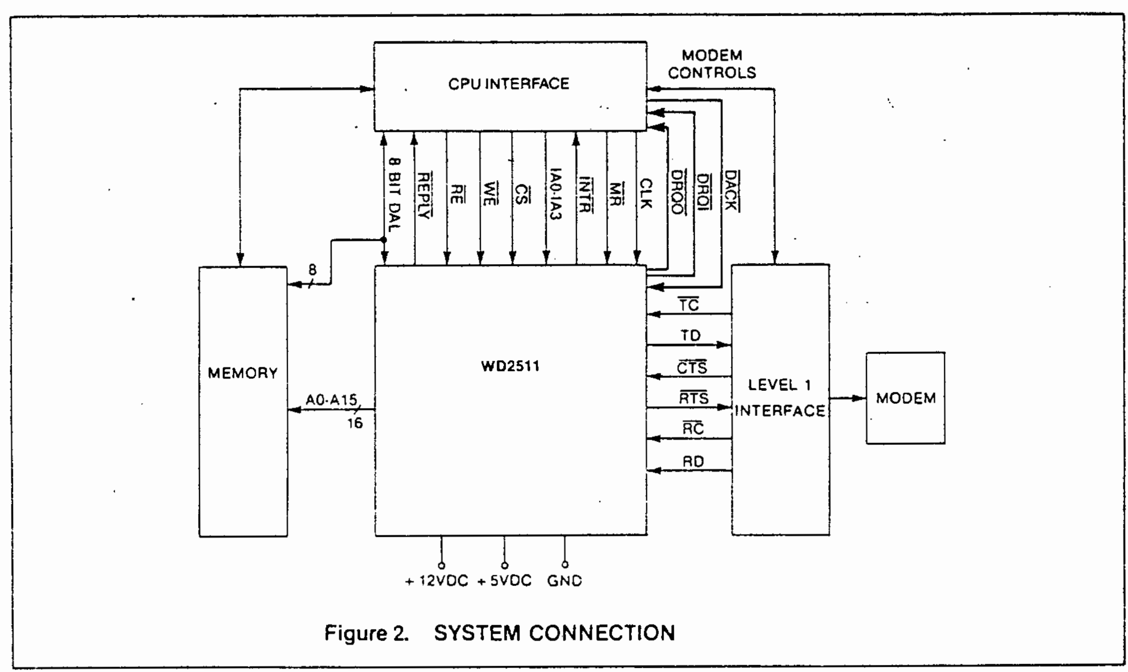


Figure 2. SYSTEM CONNECTION

Fig 2.4 WD2511 system connection

2.4 HARDWARE DESIGN

Having chosen the WD2511 for the link level of X.25, the next decision to make is what hardware system to use. For example, if there are ready designed network cards, then these could be used. ie. every effort was made to avoid re-inventing the wheel. In order to meet the speed requirement, the traffic generator would require eight WD2511 processors, five 8088 processors (one for overall control) and a total of around 1Mbyte of memory - a fairly powerful computer system by any standards. Several alternatives were available and these are described below:

i) SAbus:

There are several cards available for the SAbus including an 8088 board and a 128K memory board. A high speed link level card could be designed and a peripheral card could be used to implement bus control, timers and communication with other modules.

However, a quick calculation shows that one would need around 30 SAbus cards in total and four SAbus kits to house them all in! Even then there would be no disk drives or operating system to build on. Clearly SAbus does not have the power for this application.

ii) Multibus:

The Intel Multibus is a powerful and versatile system with numerous commercial boards available. The "iSBC 88/45 Advanced data communications processor board" (Intel 1983) looked very interesting and the hardware reference manual was accordingly obtained.

This board consists of a high speed (8 MHz) 8088, 16K bytes of dual ported static RAM and 64K bytes of EPROM. In addition there are three communication channels supporting HDLC (i.e. flags and check character as described earlier), various timers, an interrupt controller and a DMA controller chip.

The main problem with the board is that, although only released in 1982, it is now outdated. There is very little memory and, although designed to be flexible, there are very little expansion possibilities and no WD2511 support. Thus this board is unsuitable.

Looking through technical magazines another Multibus vendor was located producing a board called MPA-2000 or "Chairman of the boards" (Metacorp 1985). The board is extremely densely packed and contains virtually everything, including: an 8 MHz 80186 processor, 512K of RAM, 128K EPROM and connectors for expansion modules, including a WD2511 module. However, software development is complex and the card is expensive, making it generally inferior to the next solution.

iii) The IBM PC:

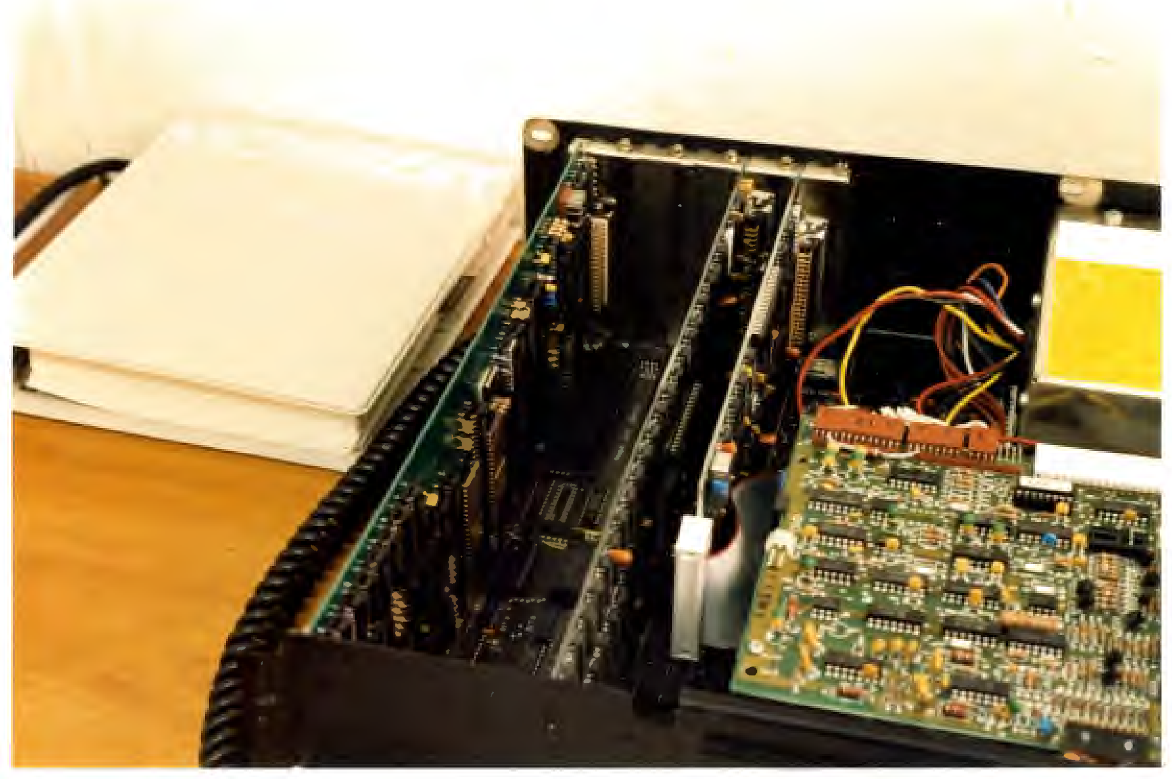
Since its introduction, the IBM PC has become the de-facto standard personal computer. One of the main reasons is its open architecture - the old PC had 5 expansion slots, while later versions have 8. This allows one to easily upgrade or modify one's PC, plugging in display cards, extra memory boards, disk controller cards etc.

There is therefore the possibility of designing an X.25 card for the PC. The main advantage is that one starts with a complete computer system, including disk drives and operating system, on which to build. Furthermore the cost of the traffic generator is reduced to the cost of the X.25 card since the card can be plugged into any PC or compatible if one is available.

Thus on the one hand one has the possibility of designing with the MPA 2000 Multibus board, which would involve having to build up a complete computer system - keyboard, display, disk drives, power supply, card rack and cabinet. At the end of the day one would have an extremely expensive computer system which would only be used as an X.25 traffic generator (note also that Multibus I is now obsolete - having been replaced by Multibus II). On the other hand using a PC would result in the design of an X.25 card while the PC provides the system base which can be used for other applications. Clearly the PC solution is advantageous with regard to hardware.

On the software side, using a Multibus system means that one has to once again build up a system from scratch or alternatively do the development work on a dedicated development system (eg. the Intel MDS). This contracts to the PC where there is a good well documented operating system on which to work and a range of debugging tools. There are also dozens of low priced professional compilers to choose - including the new C language.

Furthermore, using the PC means that programs can be written, compiled, tested and run all on the same PC. No extra facilities such as EPROM programmers are required which makes software updating very easy. Thus the PC, together with the WD2511 processors, formed a solid base on which the traffic generator was developed.



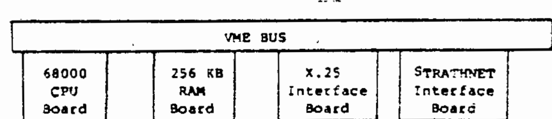
The open architecture of the PC makes it ideal for the development of custome cards. Picture shows X.25 card (far left) installed in a PC together with display and disk drive cards.

2.5 Work at other Universities

As X.25 has become a popular access protocol for wide area network there are several other Universities activily working in this field. Most projects appear to be concerned with the implementation of X.25 gateways to link University local area networks to X.25 and several leads were followed up to try and get new ideas.

2.5.1 LAN to X.25 gateway project at University of Strathclyde:

The aim of this project, sponsored by British Telecom, was to design a gateway between an Ethernet style local area network and British Telecom's packet switched network. The research team consisted of three members and papers describing the project were obtained (Grant et al 1983). The basic architecture of the system is depicted below:



The system basically consists of three processors (one for X.25, one for the LAN and one for overall control) coupled to 256K of memory and the architecture is thus similar to that of the X.25 card. The chosen bus system was the VME bus. Unlike the PC approach, software was developed on the departmental PDP-11 (running UNIX) and then down loaded to the 68000 system EPROMS.

Note that the main difference between this design and the X.25 traffic generator is that the traffic generator effectively implements the whole system on the single X.25 card. This was achieved through the use of VLSI (eg., the new 256K DRAM chips) and the use of an optimised hardware circuit based on PAL devices. Thus from a systems viewpoint the traffic generator allows for several complete multiprocessor X.25 systems to be installed in an IBM PC rather than several cards to be installed on a VME bus. Although the use of a PC is more expensive in the short term, it does offer greater flexibility; for the traffic generator application it is cheaper than a dedicated computer system.

The X.25 card in the Strathclyde project used a Motorola 68120 processor to implement the link level software, which had to be written by the project team. This is perhaps an example of how quickly one can be lapsed by technology as the new MC68605 could easily replace the entire board and save considerable software development time.

While the traffic generator design was primarily concerned with the hardware required to support the high data throughput, the bulk of the work in the Strathclyde project was concerned with the software. Bridging between the two networks was done on the transport level, so both the X.25 and network software up to the network layer was required (the project was not completed). The software design used a system of buffer management and queueing similar to that implemented in this thesis as described in chapter 6.

2.5.2 Work at the University of Catania:

This project, carried out by a research team of 3 (Faro et al 1985), consisted in the design of a general purpose X.25 system, which could be used as a relay system or traffic generator. The objectives are therefore similar to those of this thesis.

For their hardware architecture, a multi-processor system with Z80 CPU's was chosen. Each card contains a Z80 processor, 8K EPROM, 4K RAM, a timer chip, a SIO (serial I/O) chip and interface logic to a common bus. Several of these cards could be installed, with them all being linked together via a 64K shared memory board. Thus once again the choice was to design a dedicated computer system rather than take advantage of the ready built PC.

As for the X.25 card, a bus arbitrator mechanism was implemented to allow access by several processors to the shared memory. The arbitrator was implemented with standard TTL devices rather than the FAST series logic and PAL's, used in this thesis. Note that this system would not be suitable for a high throughput traffic generator design as the single bus architecture would have problems in supporting the large number of processors. The Z80 processors and small amount of memory were also drawbacks.

Once again it is interesting to note that link level processors were not used, so all the associated software would have needed to be developed. Software was written in PLZ, rather than the C language chosen for this thesis, and followed the usual system of buffer management and job queueing.

2.6 Design conclusions:

The two most similar projects at other Universities have been discussed. However, as pointed out, there are substantial design differences. This project is the only one to use an IBM PC for a system base and to use the WD2511 processors. It is also the only one to use the modern C language. Thus, while ideas were obtained from their research, this thesis represents a completely new approach to the problem.

On the commercial side, details of several X.25 protocol analyser systems were obtained from Digilog, Telsaf, Digitech, Dynatech and Tekelec. Interestingly, these systems are all based on the concept of a portable PC with software loaded from disk drives (ie., the same method as employed by the traffic generator). Their main characteristic is that all the systems are primarily designed to have extensive diagnostic capabilities rather than support high bit rate links (usually 64 Kbps max and always for a single line).

Furthermore, few of the products, with the exception of the Tekelec Chameleon range, advertise the availability of simulation packages. The Tekelec product looks promising, but has one major drawback: it supports only a single 64 Kbps line. Furthermore, the unit costs over R50 000, so putting together 8 of them would work out rather expensive!

Thus, while it would be pointless working on another link analyser, the author knows of no commercial traffic generator systems. The unit designed in this thesis makes use of any PC or compatible computer to provide up to 10 high speed X.25 lines (5 free slots required). The design is easily expandable, with the only costs being those of the X.25 cards. Furthermore, the X.25 card can be adapted for use in a variety of other communication applications in the PC.

Finally, having looked at work in other Universities and the commercial market, a literature survey was done. As well as books, this included a Dialog search of dissertations. A search which was conducted by the CSIR for possible similar research projects in the country, revealed a network simulator designed at Wits. However, this thesis designed in 1979 to 81 is concerned with simulating a network (eg the network delays) rather than implementing X.25. The work is now very dated, with the WD2511 VLSI chips replacing a good deal of the software. The IBM PC did not exist at the time and 64K dynamic memories were unheard of, let alone the 256K ones used here. In fact keeping up with technology was found to be one of the major challenges in the traffic generator design.

BUS ARBITRATOR DESIGN

From the discussions in chapter 2, it is clear that an IBM PC is highly suitable for use as the base of the system. The hardware design would therefore consist of a high speed X.25 card containing:

(i) two WD2511 link level processors, one 8088 processor running the packet level and the traffic generator software and a 256K multi-port memory to tie everything together.

(ii) card control mechanisms, timers and the various line interfaces were required to complete the card.

Having defined the requirements and architecture of the X.25 card, this chapter looks at the design of the card's central control system - the bus arbitrator. The arbitrator must reliably and fairly allocate the bus between the four processors requesting it - at a rate of over one million decisions per second.

As this is such an important issue to the traffic generator performance, several references were studied; this chapter discusses how the modern bus systems (Multibus 11 and VME bus) tackle the problem as well as solutions adopted for the newly emerging parallel computers. The circuit design is then discussed with comparisons being made to a Western Digital application note design.

3.1 The Problem:

All the devices on the X.25 card are linked together by means of a common address and data bus as depicted in fig 3.1.

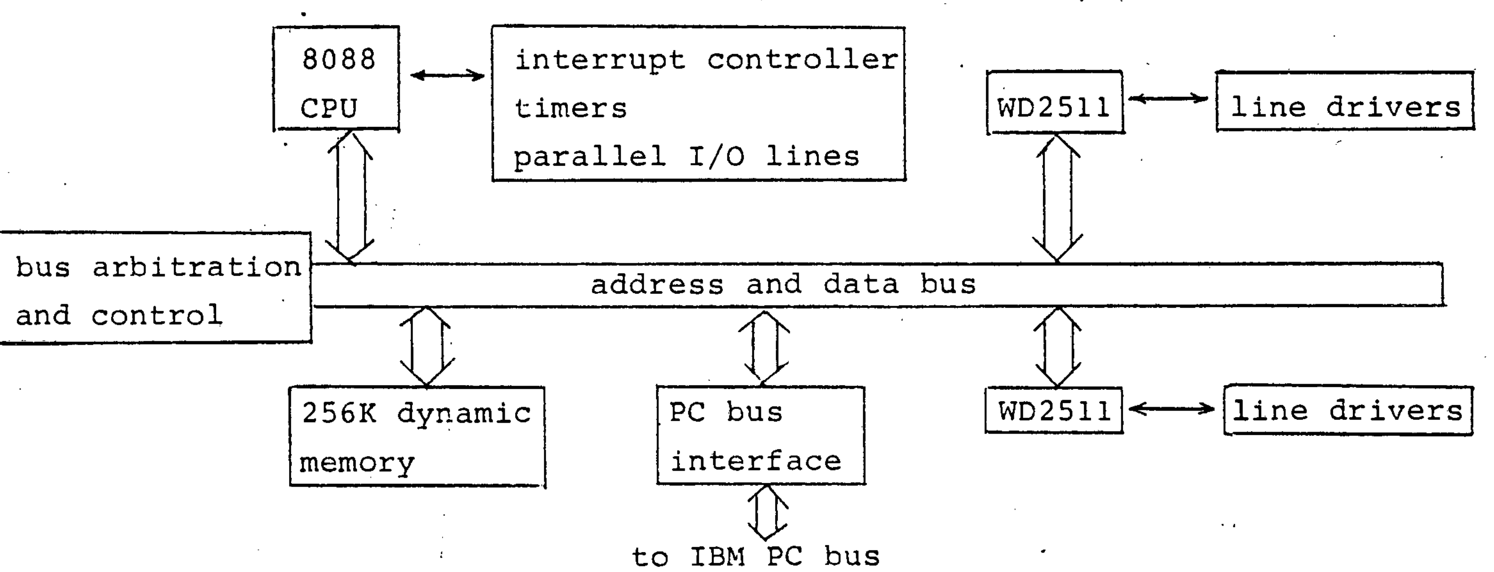


Fig 3.1 X.25 card local bus structure

This local bus is the means whereby all devices access common resources eg., the 8088 processor writing to the control registers of a WD2511 processor, the 8088 accessing memory, the PC accessing memory etc. In total five different devices share the bus (the 8088 processor, the two WD2511 processors and the 8088 PC processor and DMA chip on the IBM PC System's Board); the arbitrator must reliably ensure that each one gets on and off the bus in an orderly manner. As all the processors operate asynchronously, this involves first synchronizing the bus requests to a common clock and then controlling the bus cycle.

To complicate the problem, the card uses dynamic rather than static memories in order to attain the large memory capacity. Failure to perform a refresh cycle or a glitch on one of the memory control lines could result in partial memory erasure. As well as the reliability issue, the arbitrator has to have a parallel priority scheme and be fast. Both the PC and the WD2511 processors are sensitive in this regard so the arbitrator must ensure that one device does not hog the bus.

3.1 The 8088 HOLD line:

Having defined the problem, various solutions were looked at. The most obvious one was to use the hold and hold acknowledge line of the card's 8088 as done in an X.25 card design for the SAbus (Aspin 1983). While simple and ideal for line speeds of around 9 600 to 19 200bps this system is too slow for the traffic generator application. In any case, having removed the 8088 off the bus, one would still need to decide which of the other devices would hold the bus.

Hence this solution was abandoned and the hold line of the card's 8088 is permanently tied to ground. Control of the 8088 is instead implemented by means of its memory ready line. Thus the implemented arbitrator consisted of the following elements:

1. a synchronizer to synchronize all requests to a common 18.432 MHz clock
2. a parallel priority encoder consisting of a 74LS148
3. control via a PAL (programmable array logic) device.

3.1.2 Asynchronous metastibility:

The first design problem encountered was that of asynchronous metastability (the random output from a synchroniser when its setup times are violated). This metastability problem occurs when synchronizing requests; it is common in all multi-processor designs. In an article "Metastability haunts VME bus and Multibus II designers" (Martin 1985) it was described how a VME bus multiprocessor could lock up due to allowing too short a time to arbitrate. There is thus a trade off between performance and reliability; use the fastest available logic and allow the synchronizer the longest possible time to stabilize in.

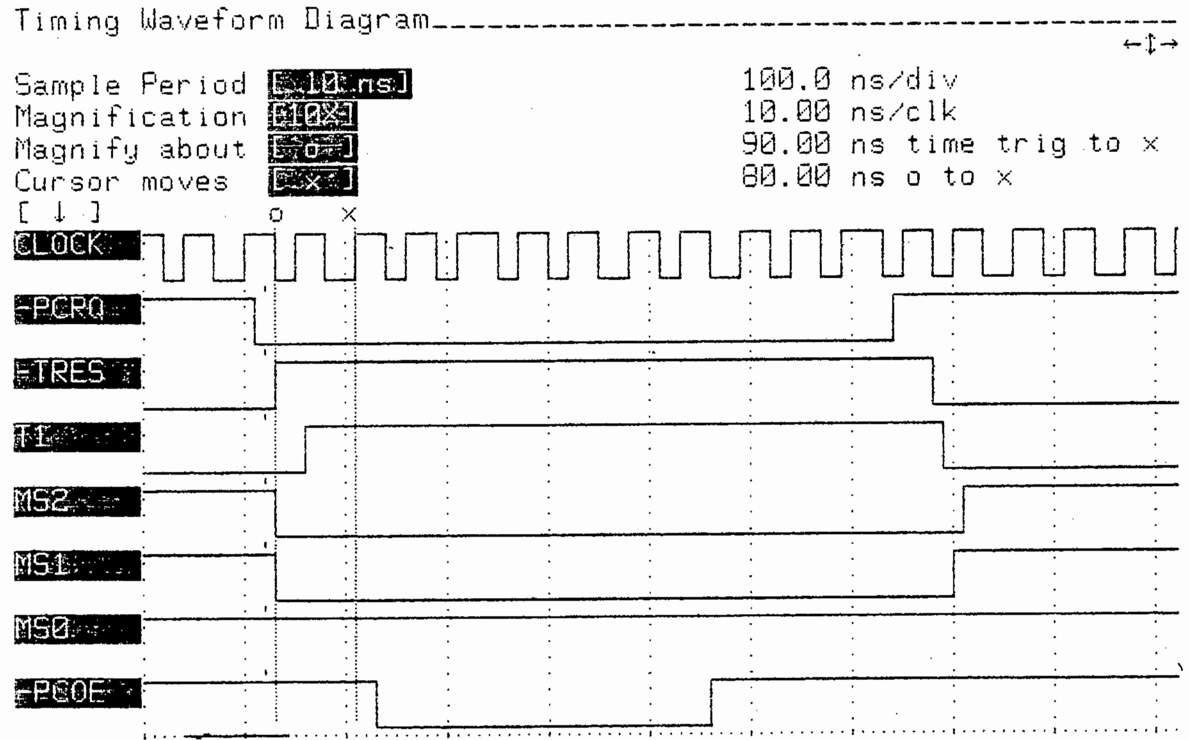


Fig 3.2 X.25 card bus synchronization timing

The article recommended a 50ns delay to be allowed for TTL devices, with the 20ns allowed in the particular VME card causing the problems. It is interesting to note that the method of tackling synchronization is one of the most fundamental differences between the two modern busses. The VME bus allows asynchronous requests which are synchronised by the arbitrator on each card. Multibus II requires all memory requests to be synchronized to a 10MHz system clock ie., the bus operates synchronously. Critics of the multibus point out that the slow 10MHz clock results in a delay of up to 100ns, with an average of 50ns, between a processor requesting memory and the request appearing on the bus. This reduces bus bandwidth.

3.1.3 Performance considerations:

From the above it will be clear that the multi-port memory takes a good deal longer to access than ordinary memory. This can be a major problem, especially if there are several processors accessing the memory. The memory has a limited number of access cycles per second which leads to a system bottleneck.

This memory bottleneck is a particular problem in the next generation of computers - parallel processors. The solution adopted is for each processor to have its own private memory as well as memory which may be accessed by the other processors.

Even this is fairly restrictive and several manufacturers have opted for other architectures such as the "Hypercube". An example is the Inmos transputer which only requires local memory and has four high speed communication links to access its nearest four neighbours.

For the X.25 card design, the problem was partially solved by pipelining memory requests and optimising the circuits to get the highest possible throughput. It should be noted that the X.25 card is classified as a loosely coupled multiple processor system - its processors work on different parts of a job rather than all working simultaneously on a single task.

Having looked at the arbitrator problem, it is worthwhile to mention a simple solution to dual port memories - dual port static RAM chips especially designed for the job. A example is the SY2131 which is a 1K byte chip with a 100ns access time (Drumm 1984). This chip achieves such high speed by arbitrating on the byte level rather than for the entire memory system. Thus two processors can access memory at full speed and arbitration is only required if they both access the same byte simultaneously.

An initial design used one of these chips for the X.25 card/PC connection while using the hold line to allow the slower link level processors to access the card's private memory. However, this solution made software downloading much more complex and would have required the use of an EPROM for system start up. Also, memory refreshing would become a major problem (the current X.25 card design uses the PC's refresh mechanism). Finally, the PC would not be able to use the card as a memory board and program debugging would be much harder. Hence this design alternative was rejected.

3.1.4 Arbitrator problems summary:

The choice of bus control is an important issue as it virtually determines the system architecture. The seemingly simply solution of the hold line and dual port RAMs creates as many problems as it solves. Thus it was decided to have a completely shared bus with programs for the 8088 being downloaded from the PC rather than stored in EPROMs.

As the 8088 is working from shared memory performance considerations become important and one needs the shortest possible bus cycle times to avoid system congestion. At the same time a reliable arbitration method is required to avoid the metastability problem and to ensure that all processors get a fair share of the memory. It was with these objectives in mind that the arbitrator was designed.

3.2 Arbitrator Design:

While designing the arbitrator, an application note in one of the Western Digital data books (Network handbook April 1984) was consulted. The design of a single WD2840 circuit coupled to an 8K byte of dual port static memory was described. The circuit used D type flip flops to latch to two request lines, a few gates to priority encode and 273 octal flip flops to implement the state generator (see Appendix A). While similar in principal, an examination of the application note revealed a few problems, particularly with regard to metastability. The arbitrator circuit for the X.25 card is therefore described with comparisons being drawn to the WD design.

The arbitrator for the card arbitrates between 7 possible memory requests and these are presented to an F373 octal latch (IC6) (see circuit diagram 1). The first thing to note is the use of the FAST (advanced Schottky) series logic. This series is typically 30% faster than the Schottky TTL series yet uses 1/4 of the power. Use of this logic series significantly reduces the chance of metastability (eg. the F373 has a minimum data setup time of 2ns as opposed to 20ns for the LS175 used in the WD application note).

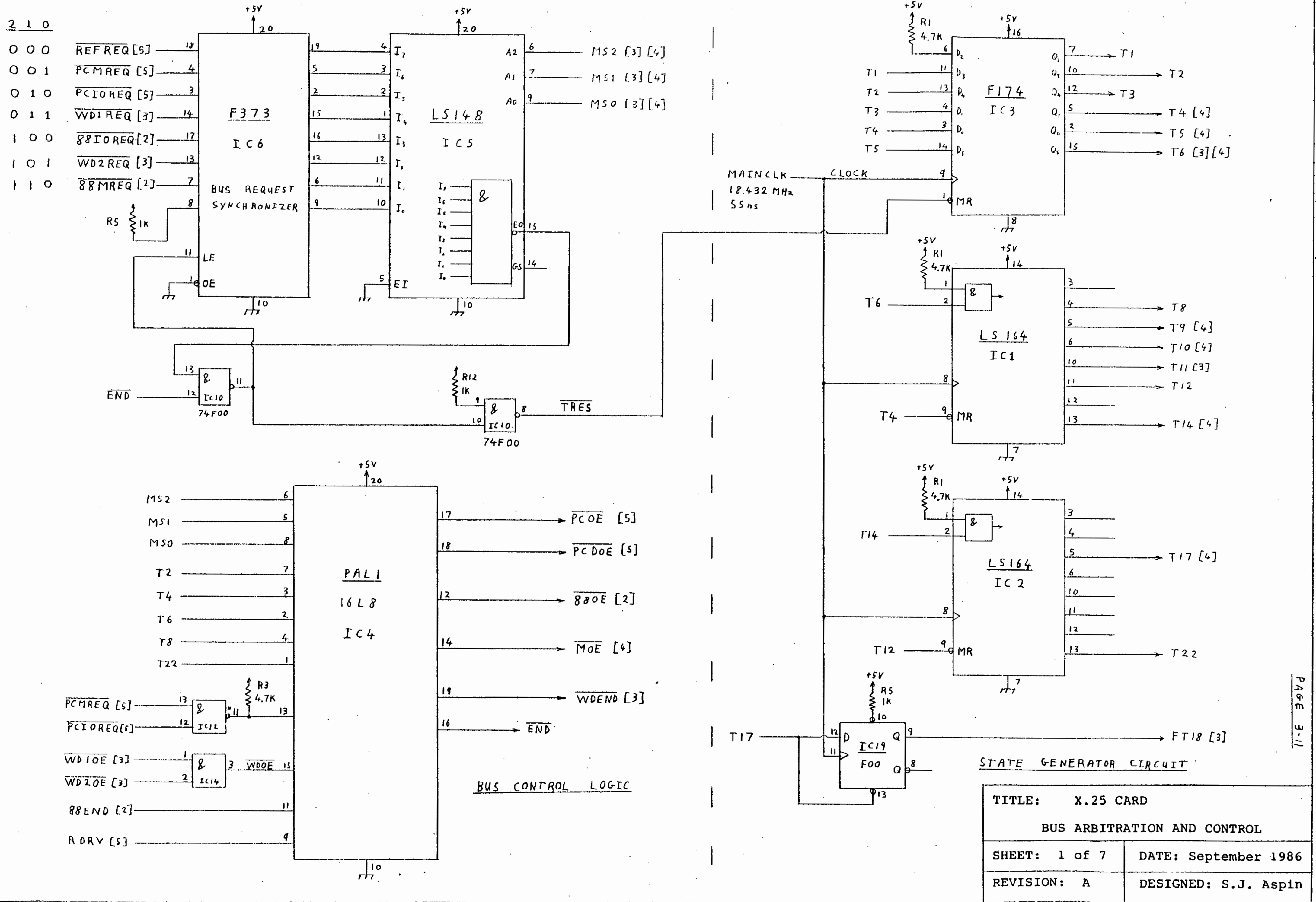
The first request through IC6 of circuit diagram 1 latches it and activates the state generator circuit. This means that, if a request just misses the clock, the F373 simply has a further 55ns to stabilize in (ie., the "dead time" is put to good use rather than just wasted as in the WD application note and in Multibus II systems).

The F00 gate delay and the F174 enable time also add to the available stabilizing time. These considerations allow a typical stabilization time of around 70ns which should be adequate for the F373. This contrasts to the WD design which has three TTL gates between latching the requests and starting the state generator. Adding the typical gate delays to the 273 setup time gives only 15ns for the 175 to stabilize in, which, considering the number of bus cycles involved, could well lead to circuit failure. As far as stability is concerned, once a request line goes active, the F373 will latch irrespective of the others. The requesting logic ensures that a request is only deactivated once it has been acknowledged. There is thus no positive feedback.

The F373 latches the requests, but these still have to be synchronized to the clock (18,432 MHz) - a function performed by the F174. This is done in a two stage process, with the second flip/flop sampling the first one's output. Thus T1 (see circuit diagram 1) is not used at all and the bus cycle starts on T2. To be quite safe, the memory and I/O access lines, as well as the data buffer enable lines, are only actually activated on T4 (in the WD design, all control lines are activated immediately).

Thus one clock period, the same as for the WD design, is required for the basic arbitration. However, the use of the unclocked 373 effectively extends the available arbitration time. Furthermore, by connecting the output of the bus request synchronizer to the reset line of the F174 rather than just to an input, gate delays are prevented from accummulating. These considerations allow more time for the F373 to stabilize and result in a more reliable circuit.

For the priority mechanism, a LS148 parallel priority encoder is used with PC memory requests having the highest priority and the card's 8088 the lowest. This system was chosen because the PC should not have more than 10 wait states (IBM Technical Ref.) and so it was important to give it the shortest possible memory access time. The refresh request cycle has the highest priority to allow a pipelining system to be implemented. Finally, note that as well as multi-port memory, the I/O lines of one of the WD2511 devices are dual ported, with both the card's 8088 and the PC being able to control the processor. This was found to be a particularly useful feature during software testing (see chapter 8).


2 1 0
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
REFREQ [5]
PCMREQ [5]
PCIOREQ [5]
WD1REQ [3]
88IOREQ [2]
WD2REQ [3]
88MREQ [2]
F373
IC6
BUS REQUEST SYNCHRONIZER
LS148
IC5
MS2 [3] [4]
MS1 [3] [4]
MS0 [3] [4]
END
74F00
TRES
MAINCLK
18.432 MHz
55ns
CLOCK
F174
IC3
T1
T2
T3
T4 [4]
T5 [4]
T6 [3] [4]
LS 164
IC1
T6
T8
T9 [4]
T10 [4]
T11 [3]
T12
T14 [4]
LS164
IC 2
T14
T17 [4]
T12
T22
T17
IC19
F00
FT18 [3]
STATE GENERATOR CIRCUIT
MS2
MS1
MS0
T2
T4
T6
T8
T22
PAL1
16L8
IC4
PCOE [5]
PCDOE [5]
88OE [2]
MOE [4]
WDEND [3]
END
PCMREQ [5]
PCIOREQ [5]
WD1OE [3]
WD2OE [3]
WDOE
88END [2]
RDRV [5]
BUS CONTROL LOGIC
TITLE: X.25 CARD
BUS ARBITRATION AND CONTROL
SHEET: 1 of 7
DATE: September 1986
REVISION: A
DESIGNED: S.J. Aspin
PAGE 3-11

3.2.1 The state generation logic:

The clock logic consists of a number of flip flops and generates the timing states used by the rest of the circuit. 18.432 MHz was chosen as a high but easily manageable clock frequency. The clock is also divided down to supply the baud rate for the card's asynchronous communication link (provided in addition to the X.25 lines).

Note that the flip flops are not all reset on the /TRES line, as they are in the WD design (see Appendix A). The reason is that some logic lines are activated on one clock state and deactivated on another (eg. for a WD2511 memory access the memory RAS line is activated on T9 and deactivated of T17). Thus should, at the end of a memory cycle, T17 be reset before T9 a glitch would result on the RAS line which could partially wipe out the memory. The WD design solves this by inserting extra gates, relying on gate delays being larger than possible latch skew.

The chaining of the timer reset lines guarantees that T9 is reset before T17, the RAS line in fact being reset on T4 just to make quite sure. In fact, looking through the PAL equations in Appendix A, the reader will note that all clock related circuits are reset on either T2, T4 or T6. This ensures that they are all reset promptly at the end of a cycle before the LS164 outputs are reset. Using FAST series logic for the 174 (IC3) ensures minimal skewness between the resetting of its outputs, while the 164 shift registers allow for a more compact circuit.

3.3 BUS CONTROL PAL

The final part of the arbitrator circuit is the bus control PAL. A PAL, or Programmable Logic Array, is basically a device which can be programmed to provide user defined functions. Five PALs are used in the X.25 card design, one for each section of the circuit. The 16L8 PAL device, whose internal structure is shown in fig 3.4 below, was used.

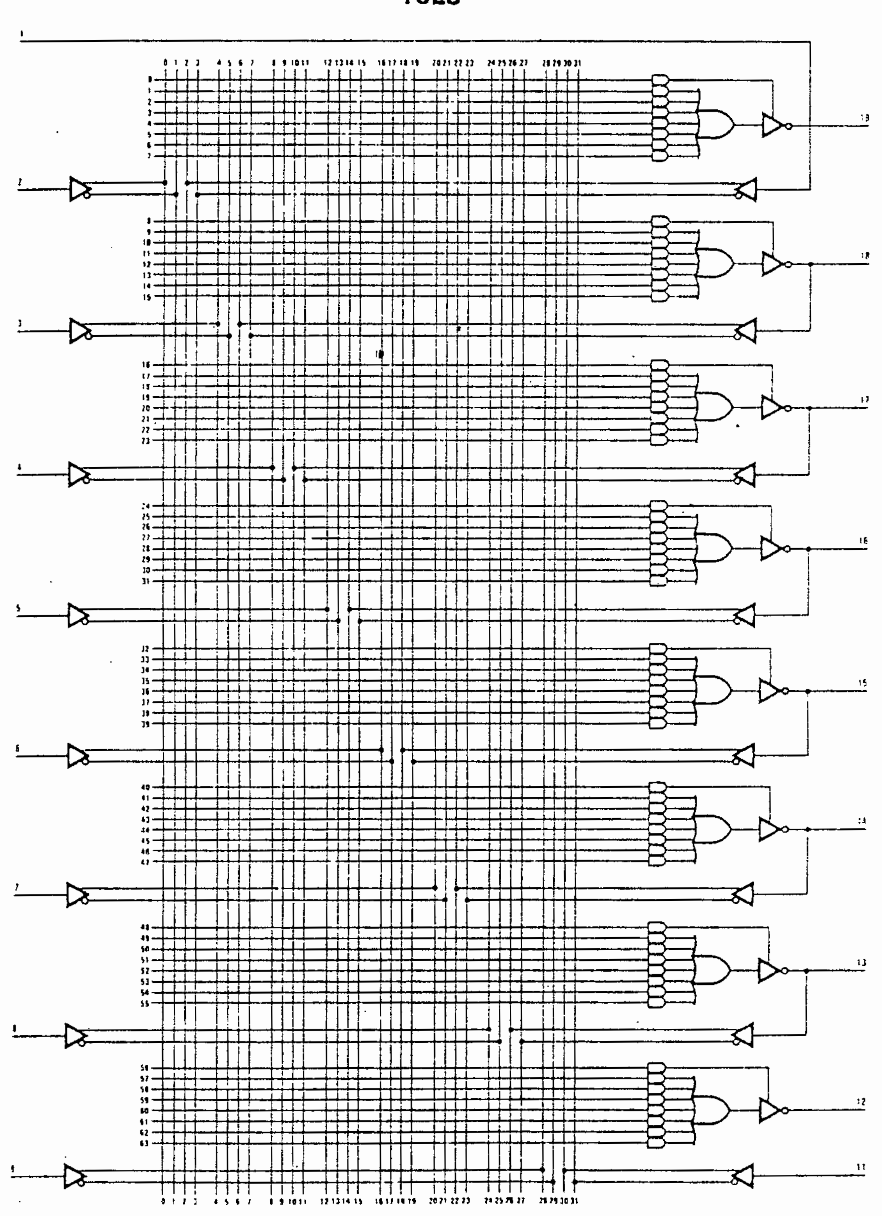


Fig 3.3 The 16L8 PAL

The PAL logic family consists of 29 chips and is thoroughly described in the PAL handbook (Monolithic Memories 1983). The 16L8 device chosen has a total of 16 possible inputs and 8 active low outputs (hence the part name). Note that 6 of the input lines are feedbacks from output pins which makes the PAL very flexible. In fact for the bus control PAL, two of the outputs are re-configured as inputs while feedback inputs from four of the remaining outputs are used in the PAL equations.

The big advantage of PALs over standard TTL logic is that one can build up complex logical equations involving a large number of terms. An example of this is the three multiplexed bus select lines MS0 - MS2 which are input to the PAL devices conveying to the latter which bus cycle is in progress. Thus the bus control PAL is able to activate /MOE (memory output enable) only for those cycles involving memory and similarly for the other output lines.

Thus PALs do not simply replace a few gates, but rather can replace a complete digital sub-system. This results in a substantial reduction in chip count and hence a reduced PCB area. Certainly without the use of PALs, the X.25 card would have required two circuit boards to fit all the logic.

PAL1 (IC4) therefore implements the complete bus control sub-system, activating the various bus buffers and terminating each bus cycle.

Particular problems were encountered mainly with the cycle termination procedure. When /END goes active, it could be possible for the arbitrator outputs to change before the timing chain is reset, which in turn could result in a glitch on the buffer enable lines. This problem was solved through careful construction of the /END signal, which only goes active after the buffers are disabled. The /END term appears in all the other bus equations ensuring that they remain disabled.

Note that the /END signal is not timed - it cannot be as it results in the resetting of the timing chain. /END starts the cycle termination and is cleared once the timing chain is completely cleared - ie. once everything is reset. This means that the cycle ending is very quick while at the same time it guarantees that everything is reset, irrespective of gate propagation delays.

Finally the use of the reset line from the PC (RDRV) ensures that everything is properly reset on power up.

Thus the use of a PAL results in a single chip solution to the fairly complex bus control problem, ensuring that all processors get on and off the bus in an orderly manner. Together with the actual arbitrator and the state generating logic this circuit forms the hub of the X.25 card design, linking all the circuit modules together. These modules are described in the next chapter.

CONCLUSIONS:

The bus control system therefore consists of three sections: the synchroniser and arbitrator, the timing chain and the bus control logic. The synchronizer was designed using F series logic with particular attention having been paid to the metastability problem. The bus control logic, designed using a PAL device, controls the bus buffers. Care was taken here to ensure a quick and fullproof cycle termination method. A logic analyser trace (from an HP 1630G system) of a typical bus arbitration cycle is shown in fig 3.4 below.

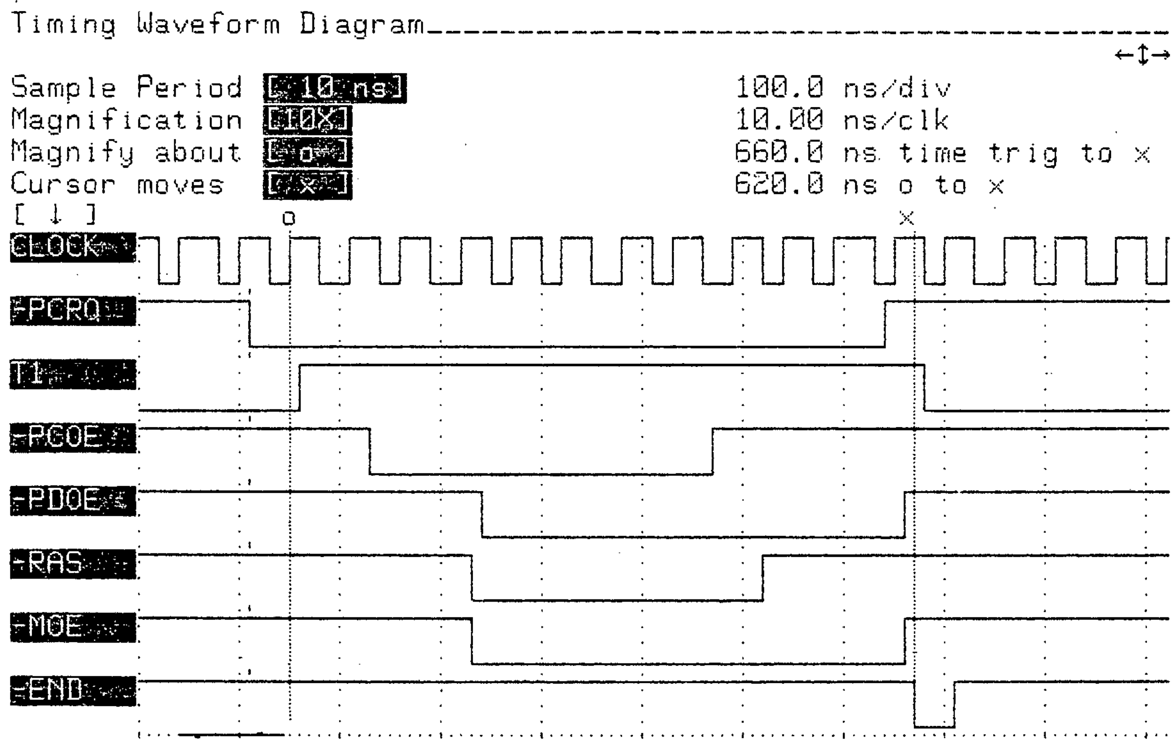


Fig 3.4 The arbitration cycle for a PC memory request

X.25 CARD CIRCUIT DESIGN

This chapter describes all the remaining circuit elements in the X.25 card as shown in the block diagram below.

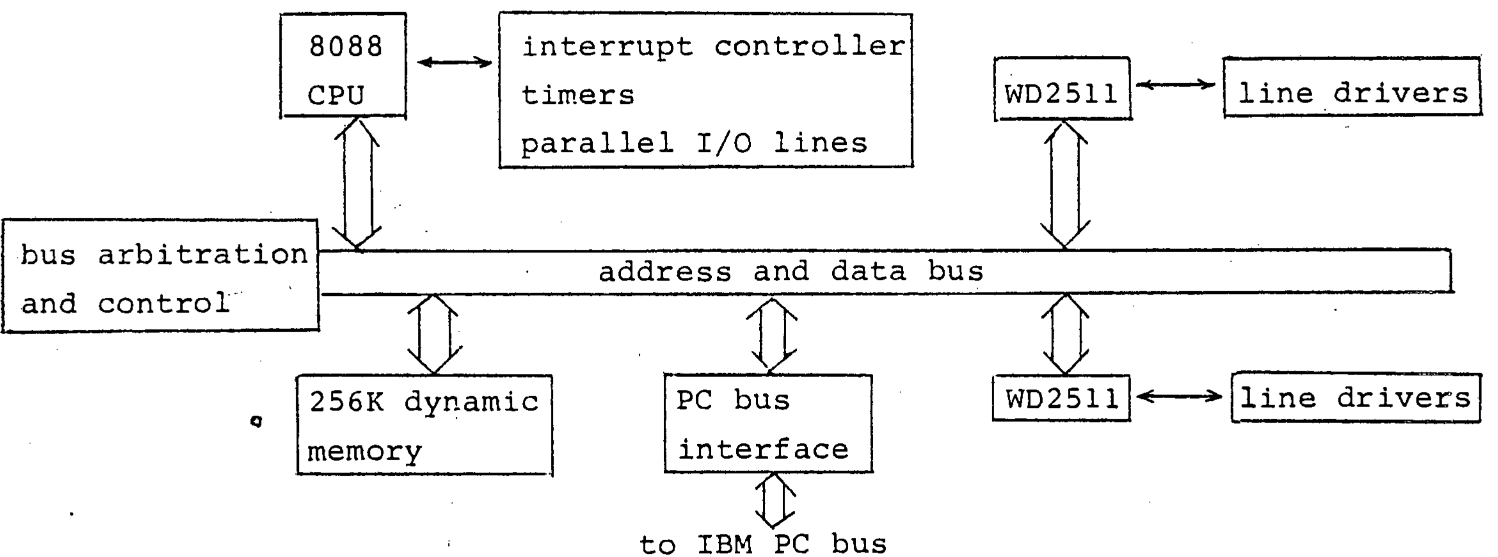


Fig 4.1 X.25 card : block diagram

Only the important aspects of the circuit operation will be emphasized. The reader is referred to the circuit diagrams and databooks for more details. Timing diagrams in the text are the results from circuit tests described in appendix A. These tests would normally be carried out after the construction of an X.25 card.

The control functions in the above diagram are implemented mostly with PALS. The equations for the PAL devices, as well as programming notes, are listed in Appendix A.

4.1 THE 8088 PROCESSOR

The main reason for choosing the Intel 8088 processor is that it is the same processor as used in the PC and hence is directly software compatible. This means that programs can be developed on the PC and then loaded to the X.25 card (refer to chapter 7 for a description of program loading).

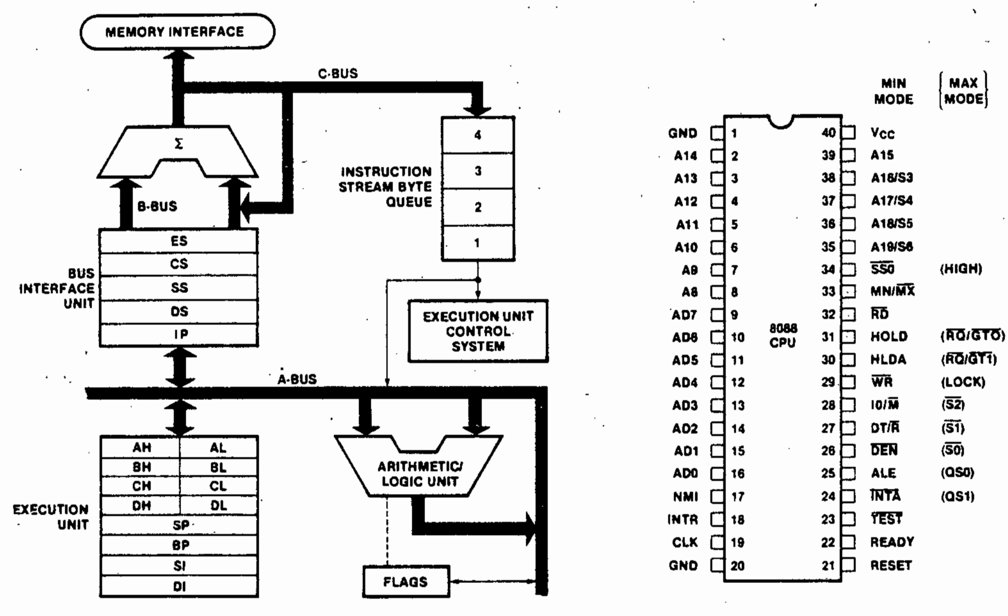


Fig 4.2 The 8088 Architecture

From the block diagram of the 8088 (fig 4.2), note how the bus interface unit is separate from the actual execution unit. There is also a bus queue of 4 bytes to interleave the instruction fetch and executing cycle. Thus the queue acts as a very small cache memory. If the next instruction to be executed is sequential then the 8088 will execute it from the queue. To execute a program control instruction the 8088 execution unit must wait while the instruction is fetched from memory.

This parallel instruction fetch and execution is one of the reasons for the 8088's superior performance compared to older processors such as the Z80 or the 8085. It is extremely important for the X.25 card design as it helps reduce the memory bottleneck problem discussed in the last chapter. Often the 8088 will be able to execute from its instruction queue while the instruction fetching unit is waiting for a WD2511 or refresh cycle to finish. This is particularly important as the WD2511 processors have a long memory access cycle compared to the 8088.

Other important aspects of the 8088 depicted in the diagram are the segment registers in the bus interface unit and the two pinouts modes available. The segment registers allow the 8088 to access 1M byte of memory and have very important implication both with regard to data structure definitions (chapter 6) and program loading (chapter 7).

4.1.1 Circuit design : min/max mode

Looking first at the 8088 pinouts, the reader will note that 9 of the control pins have dual meanings depending on the mode selected. The final circuit uses a hybrid combination of both to achieve high performance and a low chip count.

In minimum mode (selected by pulling the MN/MX pin high), designed for small systems, the 8088 outputs the usual read, write, IO/Memory signals. This system is easy to implement and at first sight seems the obvious choice for the X.25 card design. However, there is one problem - it is slow.

The 8088 has a 4 clock (each clock being 210ns) memory access cycle. During T1 (CPU clock), the address is placed on the bus, at the start of T2 the control lines are activated, at the end of T3 the data is latched in (for a read) and during T4 the control lines are disabled. For a full description the reader is referred to application note 67 in the iAPX 86/88 User's Manual. The problem with the minimum mode is that the timings are very lax eg., the RD active delay is between 10 and 165ns. As one has to work with the worst case timings, this effectively means that most of T2 is gone leaving very little memory access time.

To speed up the system, the 8088 is often run in maximum mode with an 8288 bus controller decoding the 8088 status signals and outputting the control signals. The 8288 has a RD output delay of 10 to 35ns from the start of T1. The maximum mode is used

The only problem with the 8288 is that the chip is expensive (about the same as the 8088 itself) and uses up very scarce PCB space or "real estate". For this, all it provides is faster control lines which still need to be decoded and altered to suit the timings of the X.25 card environment.

Hence in the final design a PAL was used to implement the following functions: 8288 bus controller, I/O line decoding, address latching and cycle start and end control lines.

Called the alternate mode and suggested by Intel for use in high speed dynamic memory interfacing (AP97 in the memory components book, Intel 1983), the status lines of the 8088 are directly decoded. Thus the control lines from the 8088 are all passed directly to PAL4. As well as a reduced package count, this hybrid mode (an 8088 minimum mode set up with an alternate timing) provides superior performance to both the min and max modes while still allowing easy connection of the 8256 multi function peripheral chip (see circuit diagram 6).
A description of the circuit operation is now given.

4.1.2 Memory request signal generation

At the start of every memory cycle, the 8088 pulses the ALE (address latch enable) line to latch in the address. This, combined with the IO/M line can be used to indicate a memory request. Hence the /MREQEN (memory request enable) output from PAL4.

This line is then clocked on the rising clock edge in T1 to the /88MREQ signal. Note that this results in a signal 65ns <u>before</u> the start of T2 as opposed to up to 35ns after T2 for the 8288 bus controller. This circuit is significantly faster than both the standard modes.

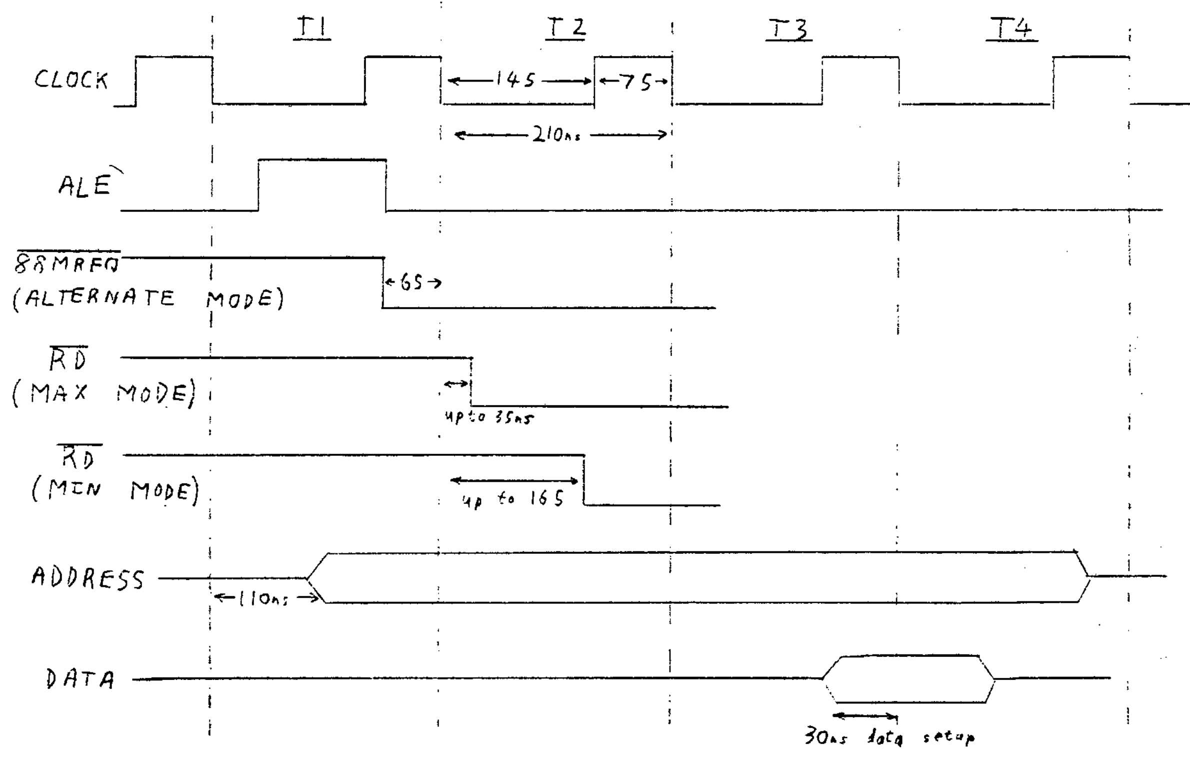


Fig 4.3 <u>Speed comparison for various 8088 modes.</u>

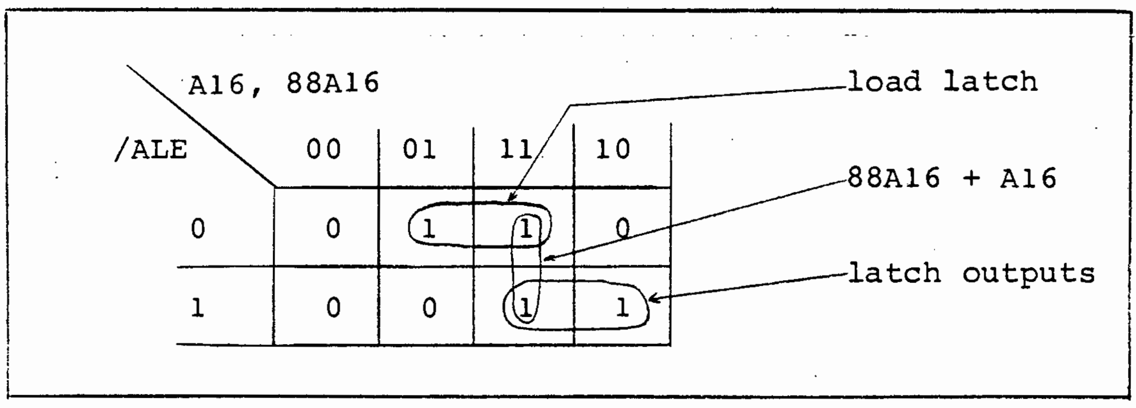
While faster, this method does have its problems and critics are quick to point out the loose timings of the ALE signal. Hence the use of the F74 (with a setup time of 3ns compared to 20ns for the LS counterpart) to allow the ALE line ample time to stabilize in. The minimum inactive delay of ALE from the rising clock edge is not given so to make double sure the /MREQEN signal is latched in the PAL to give an equation of MREQEN = /IOM * ALE + MREQEN * /IOM * /88DOE (in all PAL equations, + is a logical OR while * is a logical AND). Thus all timings are guaranteed to be met for worst case propagation delays.

4.1.3 Other functions of PAL4:

The address and data buffers are directly controlled by the bus control PAL discussed in chapter 3. However, due to lax timings of the 8088 control lines, some timing requirements for the data buffer are not reliably met, hence the /88DOE and /88READ lines from the PAL4.

The PAL also performs the I/O line decoding and latches the A16 and A17 address lines which are multiplexed with status information. Without the use of a PAL this would have required a latch such as a 373, taking up a lot of PCB space.

One interesting point to note here is the use of feedback lines on the 16L8 PAL to latch signals. The usual terms are A16= 88A16 * ALE + A16 * /ALE. These two terms cover the cases for ALE high or ALE low (A16 latched) but not the case for ALE changing. This state is covered by a third term 88A16 * A16 as shown in the Karnaugh map below. The third term therefore ensures glitch free latching and is used in several of the PAL equations. By using the 16L8 PAL devices no extra circuits are required to implement the latching.



4.1 4 The Clock Generator:

The 8284A implements the ready, clock and reset logic for the 8088, and its internal architecture is shown in fig 4.4 below.

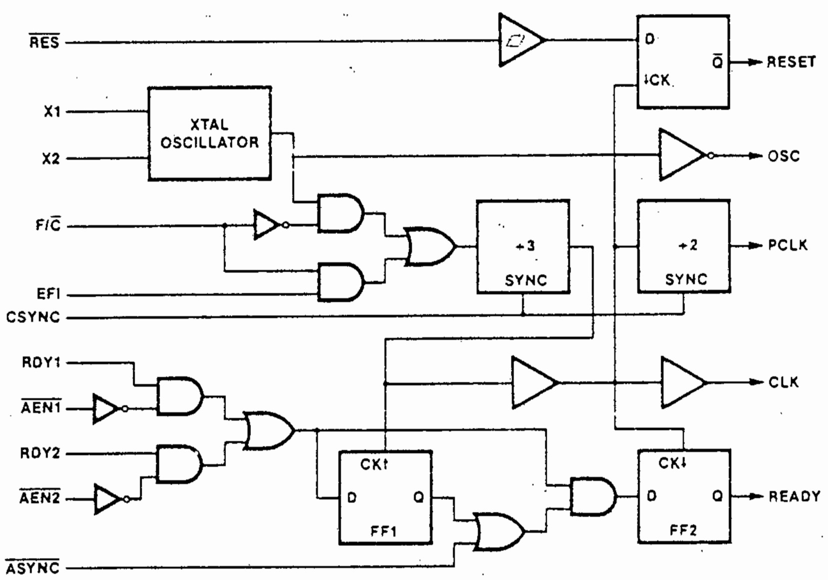


Fig 4.4 The 8284A clock generator

In the 8284A clock circuit logic note that the oscillator section can be separated from the 8088 clock circuit. The oscillator section provides the 18,432 MHz system clock used by the state generator circuit, producing a sharp clean clock signal. The clock for the 8088 was simply obtained from the PC bus via a schmitt trigger buffer.

As the 8088 is not synchronized to the system clock, it becomes necessary to synchronize the ready line to the 8088. This is done by means of two flip flops inside the 8284A, with the second flip flop sampling the first one's output 70ns after it latches. The circuit is designed for synchronization, so there are no metastability problems. A similar circuit is used to synchronize the ready signal on the PC.

4.1.5 An 8088 Memory Cycle:

The actual 8088 processor timings are given in the data sheets so, rather than repeat them here, a picture of an actual memory cycle is given. It is important to remember that no cycle is the same as the bus is shared by several devices, all operating asynchronously. The picture shows two 8088 memory cycles with memory refresh cycle in between.

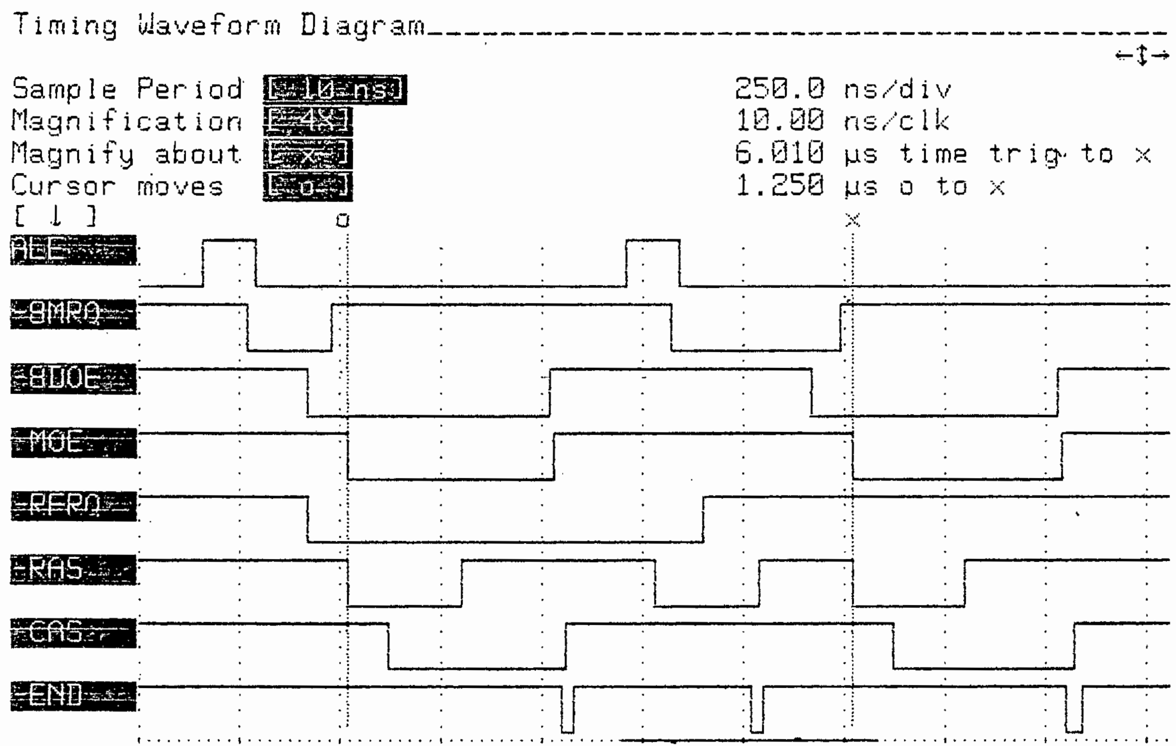


Fig 4.5 A Typical 8088 Memory Cycle

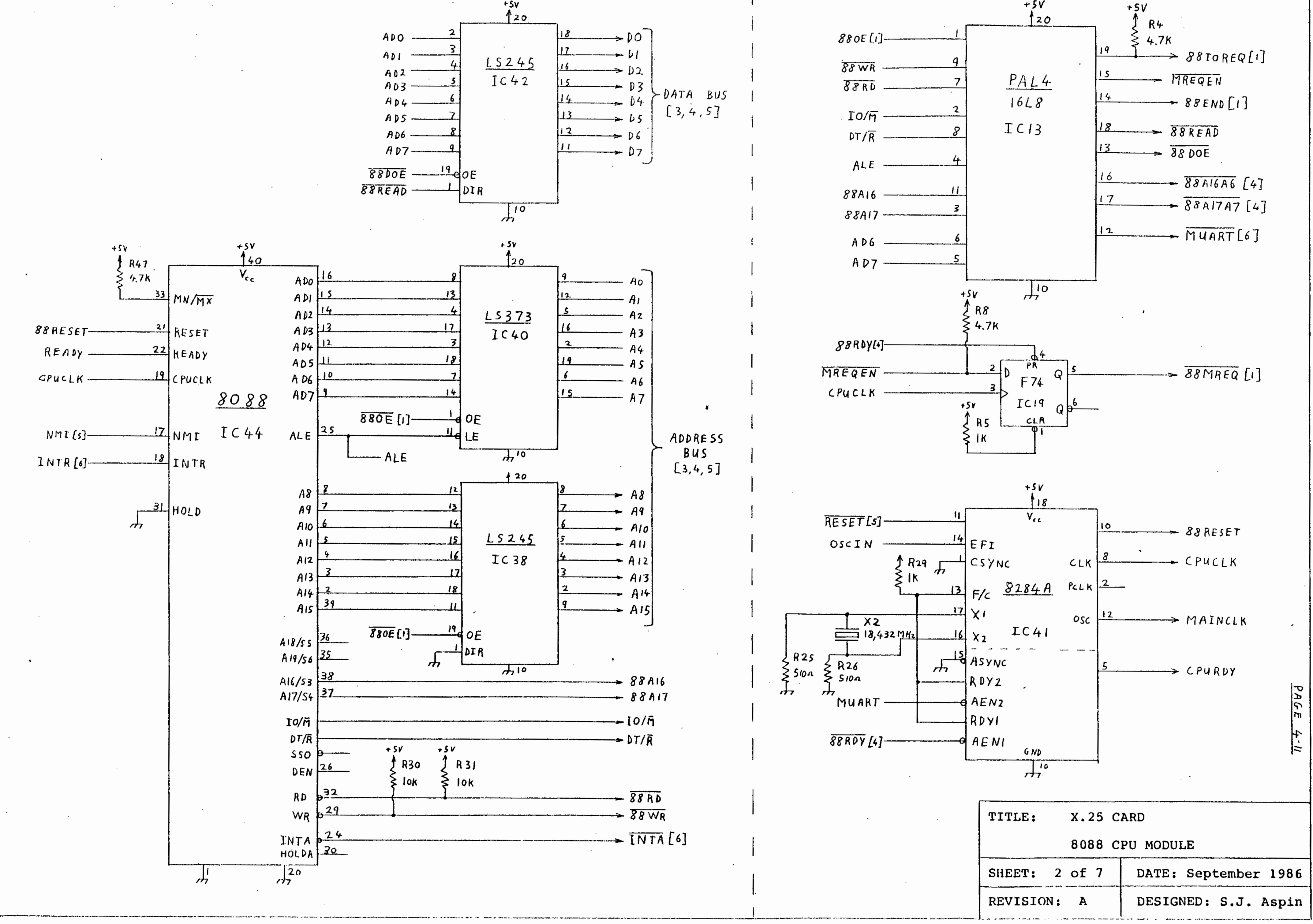
Looking at the above traces, the cycle starts with an ALE pulse. This is clocked on the rising edge of T1 to activate /88MREQ as described earlier in this section. After arbitration has completed, the memory cycle starts and the memory /RAS and /CAS lines are activated.

After accessing memory, the 8088 control PAL signals to the bus control PAL which in turn activates /END (see chapter 3). This terminates the cycle and the arbitrator is ready for the next request. The /88DOE signal, generated by the 8088 control PAL, is used to enable the 8088 data buffer.

While the above cycle was taking place, a refresh request was pending. Thus the next cycle to be executed is a memory refresh, indicated by the second /RAS pulse and the second /END pulse. Note how the refresh is slipped in during the remaining T4 of the 8088 cycle and the T1 of the next 8088 cycle, resulting in the addition of only 1 wait state. Finally, the second 8088 cycle takes place.

In conclusion, each bus cycle is a complex intermingling of various circuit blocks. Thus the 8088 bus cycle described involved the bus arbitrator and dynamic memory circuits as well as the 8088 logic itself. The memory circuit is described later in this chapter.

Finally, note that the entire 8088 subsection (8088, buffers and PAL4) is optional and may be ommitted if the X.25 card has one X.25 line or if the card is configured as a LAN (Local Area Network) card. In this case the 8088 processor on the PC's system board will control the link level or LAN chip. Pullup resistors R4 and R8 ensure that the /88IOREQ and /88MREQ lines are deactivated in this configuration.


LS245
IC42
AD0
AD1
AD2
AD3
AD4
AD5
AD6
AD7
88DOE
88READ
OE
DIR
D0
D1
D2
D3
D4
D5
D6
D7
DATA BUS
[3,4,5]
+5V
R47
4.7K
MN/MX
88RESET
RESET
READY
CPUCLK
8088
IC44
NMI[5]
NMI
INTR[6]
INTR
HOLD
Vcc
ALE
A8
A9
A10
A11
A12
A13
A14
A15
A18/S5
A19/S6
A16/S3
A17/S4
IO/M
DT/R
SSO
DEN
RD
WR
INTA
HOLDA
LS373
IC40
88OE[1]
OE
LE
A0
A1
A2
A3
A4
A5
A6
A7
ADDRESS
BUS
[3,4,5]
LS245
IC38
88A16
88A17
R30
10K
R31
10K
88RD
88WR
INTA[6]
PAL4
16L8
IC13
88WR
88RD
R4
4.7K
88TOREQ[1]
MREQEN
88END[1]
88DOE
88A16A6 [4]
88A17A7 [4]
MUART[6]
R8
4.7K
88RDY[4]
CPUCLK
PR
D
Q
F74
IC19
CLR
R5
1K
88MREQ [1]
RESET[5]
OSCIN
EFI
CSYNC
R29
1K
F/C
8284A
IC41
X1
X2
X2
18,432 MHz
R25
510Ω
R26
510Ω
ASYNC
RDY2
MUART
AEN2
RDY1
AEN1
GND
CLK
PCLK
OSC
88RESET
MAINCLK
CPURDY
PAGE 4-11
TITLE: X.25 CARD
8088 CPU MODULE
SHEET: 2 of 7
DATE: September 1986
REVISION: A
DESIGNED: S.J. Aspin

4.2 THE WD2511 LINK LEVEL PROCESSORS (LAPB)

The biggest problem encountered when designing with the WD2511 processors is their relative slow bus speed. A good deal of time was therefore spent trying to reduce the bus cycle duration to an absolute minimum. The required control logic is contained in PAL3, which is again the 16L8 type described in the last chapter.

The first WD2511 processor may be replaced by a WD2840 or WD2807 and the interface lines are accordingly brought out onto a row of wire wrap pins. The second WD processor is optional and together with its buffers need not be installed. Pull up resistors are used to ensure that the request lines are held inactive if the second WD processor chip is not mounted.

A feature of the circuit, which proved very useful in later software development, is that the first WD2511 processor may be controlled either by the PC or by the card's 8088 processor. Thus the operation of PAL3 may be viewed from two aspects - its control of the WD2511 being accessed by the 8088 or PC and the WD2511 DMA cycles. The WD2511 I/O cycles are described first.

4.2.1 <u>WD2511 I/O Cycles</u>:

A WD2511 I/O cycle is indicated by the activity of /CS and /RE (a CPU read) or /CS and /WE (a CPU write). Thus PAL3 need only be concerned with the chip select (CS) and buffer outputs enable lines, the write control line being supplied by PAL2. A typical CPU write cycle is shown below:

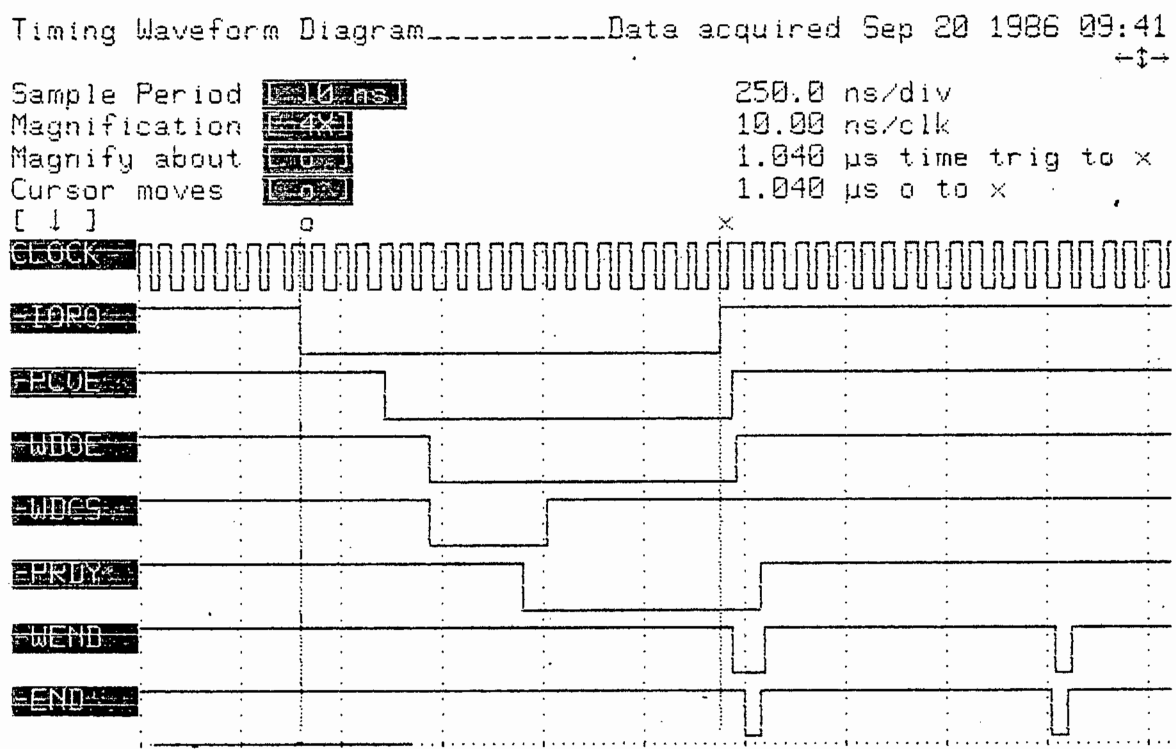


Fig 4.6 <u>The PC writing to the WD2511 Processor</u>

After arbitrating for a bus cycle, the /PCOE signal generated by PAL1 is activated on T4, putting the address and data onto the bus. The chip select line is lowered on T6 and raised on T11, the data being then latched. This period allows for a 200ns data setup time which is required by the WD2511. Note how the buffer enable line /WD1OE is kept active to ensure the data hold time required by the WD2511 processor.

As well as the above timing considerations, one must also ensure a 500 ns period between a CPU write and the next DMA cycle and 300ns between CPU writes. The former is achieved by only activating the PC ready line on T10 (hence the long /PCIOREQ in fig 4.6), which ensures a minimum period of 230 ns from the inactive edge of /WDCS to the cycle ending (fig 4.6). There is then a delay of at least a clock period (55 ns) for the bus arbitration. As /DACK is activated on T6, or 275ns from the start of a cycle, this gives a total of 560 ns minimum between a write and a /DACK. Clearly the 300 ns between CPU accesses is also guaranteed.

Calculations such as the one above were done for all major timing parameters. These calculations can be fairly long, especially as worst cases propagation delays of all the devices must be taken into account. They are however fairly easily re-constructed by working through the circuit diagrams and PAL equations and so are not discussed here.

The CPU read cycle is much the same as the write cycle described above, except that one must ensure an adequate data setup time for the CPU concerned. This was achieved by activating the particular ready line fairly late in the bus cycle.

Finally, note that PAL3 controls the buffer direction (via /WDWRITE) as in a CPU write the buffer presents an opposite direction to that of a memory write cycle. Note also the /WDEND line from the bus control PAL which is used to completely disable the WD2511 bus buffers before the bus cycle is terminated.

4.2.2 WD2511 DMA cycle:

The main timing requirements stated in the Western Digital databook (April 1984) are 500ns between a DACK (DMA acknowledge) and the next CPU access as well as various data setup and hold times. However, on the chip's arrival an attached technical memo (Western Digital 1985) required two additional timing parameters (see Appendix F). The first was that the clock had to be exactly 2 MHz (rather than 0.5 to 2.1MHz) and the second was a 500ns period between DACKS. The circuit was modified to accommodate these.

These 500ns delays are a slight problem as they make the WD2511 processor memory cycles rather long. The first means of solving this was to activate both the /DACK and /CS lines on the T6 state. By allowing time from when these lines are deactivated to the end of the cycle, effectively only a single 500ns delay is required. (ie., the circuit ensures 500ns from one WD2511 operation to the next).

The second means of reducing the WD2511 bus cycle time is to have slightly different read and write cycles. For the DMA read, memory is accessed as soon as possible with the data being valid of T15. The data is only latched during T18 (when /DACK goes high) to ensure that the WD2511 data setup time is met.
For the DMA write, the problem is that one must allow a period of some 375ns from /DACK going active for the WD2511 processor to put the data onto the bus. Hence the memory /CAS signal is delayed until T14, the dynamic memory latching in the data on the active edge of /CAS.

Finally, to check on the timing discussed above, one can calculate the minimum delay from one DMA cycle to the next:

-	25ns	PAL propagation delay T18 to /DACK high
-	10ns	max F74 delay clock to T18
4 x	54ns	T18 to T22
1 x	54ns	T22 to next T1
5 x	54ns	T1 to T6 of next cycle
	5ns	min propagation delay clock to /DACK

	510ns min from one DMA cycle to the next.	

Thus the 500ns delay between DMA cycles is safely met for worst case chip propagation delays. Note that a spare F74 flip flop was used to ensure that T18 appears promptly on the clock edge. Using the worst case delay of 32ns for the LS164 shift register would have meant a longer delay to /DACK going inactive and hence the circuit being unable to guarantee the required 500ns delay.

Note that no time is wasted and the minimum time for the one cycle to end and the next bus cycle to begin is also taken into account in the calculation. Thus the arbitration delay is effectively put to use.

Having discussed the timing problems involved with WD2511 processors, one can now look at a typical DMA cycle.

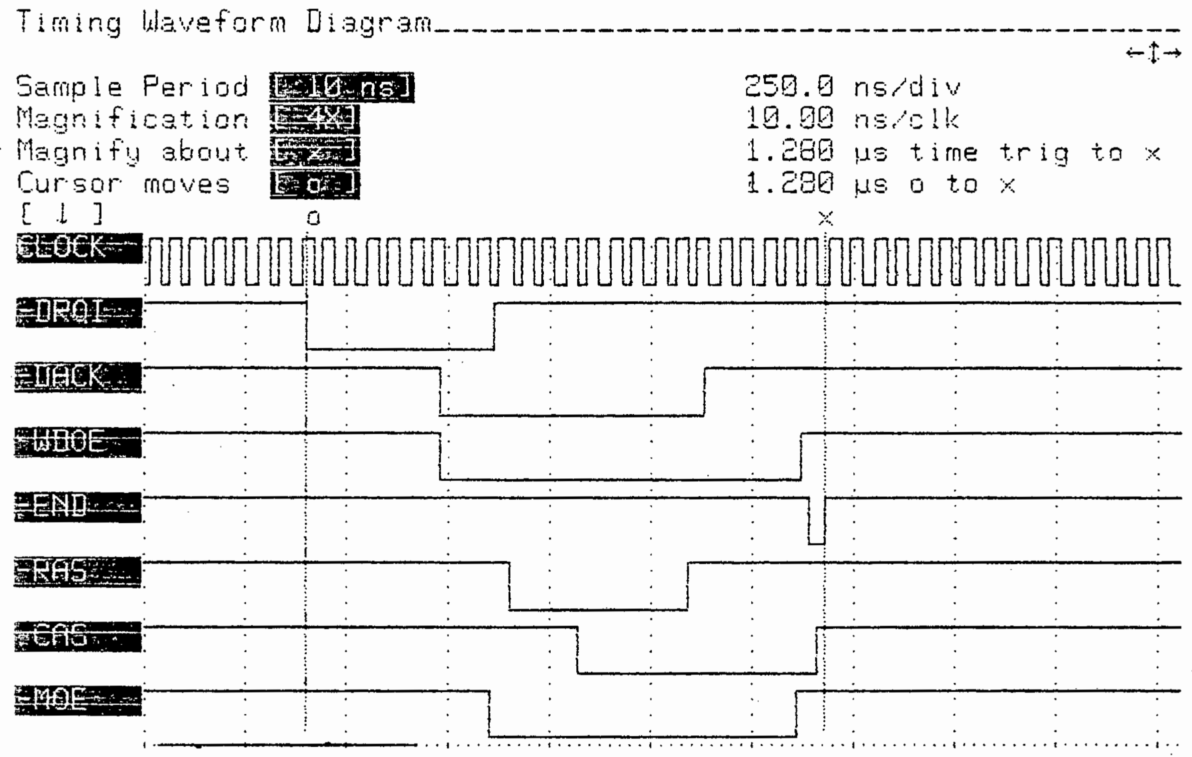
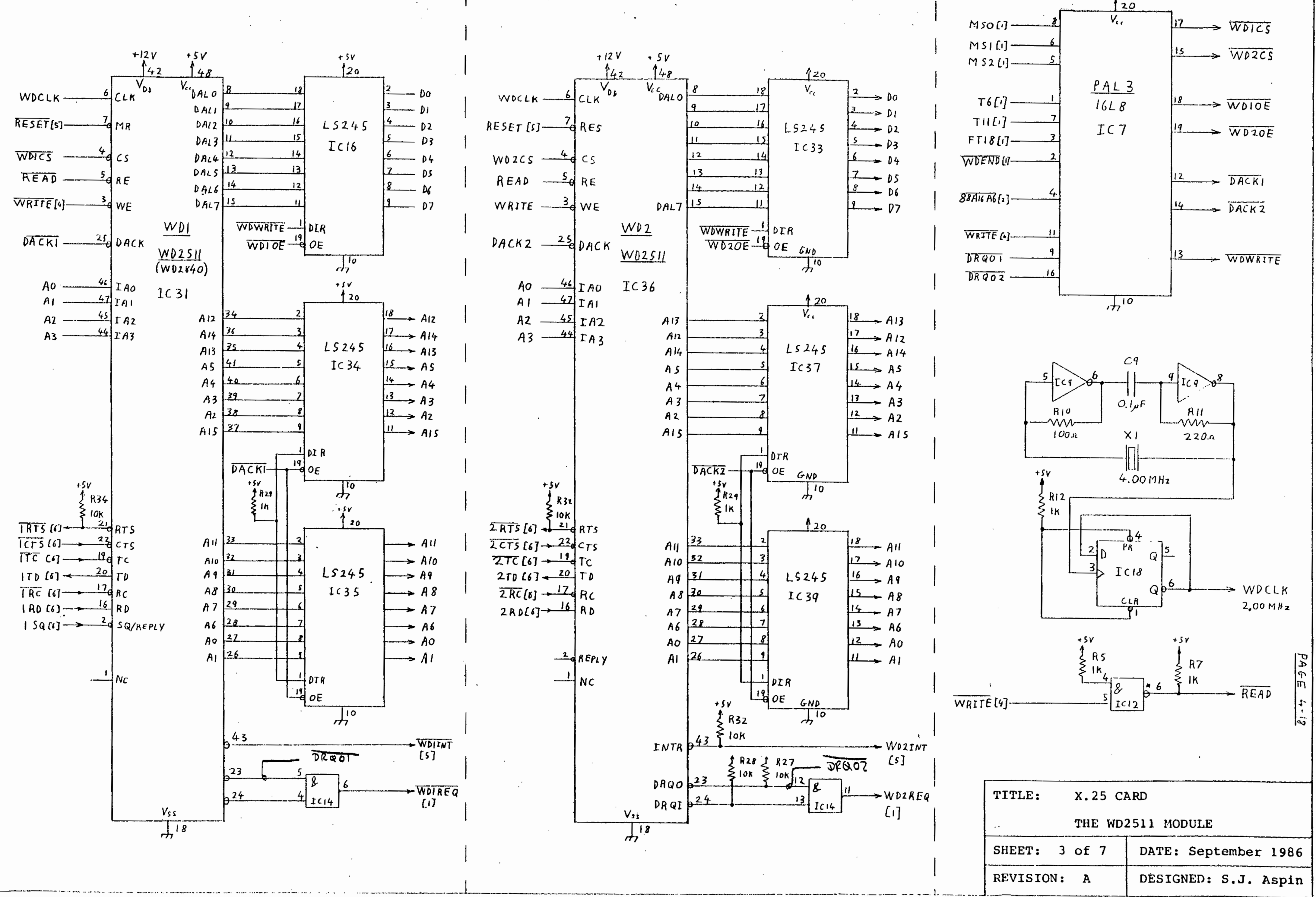


Fig 4.7 <u>A Typical DMA Cycle</u>.

This is a memory read cycle and is signalled by the WD2511 processor activating the /DRQI line (for a memory write /DRQO is activated). Having been granted the bus the /DACK line is activated on T6. The memory lines are activated as described earlier.


WD1
WD2511
(WD2840)
IC31
WD2
WD2511
IC36
LS245
IC16
LS245
IC34
LS245
IC35
LS245
IC33
LS245
IC37
LS245
IC39
PAL3
16L8
IC7
IC18
IC12
IC14
WDCLK
RESET [5]
WD1CS
WD2CS
READ
WRITE [4]
DACK1
DACK2
WDWRITE
WD1OE
WD2OE
WD1INT [5]
WD2INT [5]
WD1REQ [1]
WD2REQ [1]
DRQ01
DRQ02
MS0 [1]
MS1 [1]
MS2 [1]
T6 [1]
T11 [1]
FT18 [1]
WDEND [1]
C9
0.1μF
R10
100Ω
R11
220Ω
X1
4.00MHz
WDCLK
2.00MHz
PAGE 4-18
TITLE: X.25 CARD
THE WD2511 MODULE
SHEET: 3 of 7
DATE: September 1986
REVISION: A
DESIGNED: S.J. Aspin

4.3 MEMORY, READY AND WRITE LOGIC

So far we have looked at the various processors accessing the bus and memory. This section now looks at the memory itself. In order to achieve the large amount of memory required, the new 256K dynamic memories were used in the circuit. The book Semiconductor Memories (Prince 1983) describes the various implementations with the diagrams below showing the main differences.

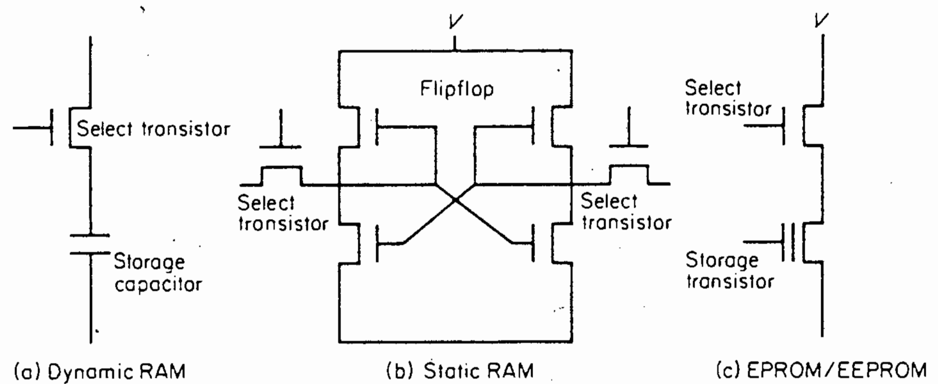


Fig 4.8 Memory cell structures

The conventional static RAM is by far the easiest to design with, but due to the large number of transistors in a cell, has a low package density. In fact some 32 of the new 8K by 8 chips would be required to give a total of 256K, taking up at least half of the available circuit board space.

Dynamic memories work by storing charge on a very small capacitor - typically 50 fF. Due to the small cell size, very high densities and a low cost per bit are achieved. However, there are several problems:

Firstly the charge leaks off from the capacitor and hence the memory must be continuously refreshed. It is up to the user to perform this refreshing, thus requiring extra circuitry.

In early dynamic memories it was found that the decay of minute radioactive sources in the chip package material were resulting in a random memory erasure. A stray alpha particle travelling through a memory cell could cause ionization and discharge the storage capacitor. To counter this problem, manufacturers coat the chip die with a special substance which resists alpha particles (Prince 1983).

However, alpha particles are still a problem and so most computer designs, including the PC, employ a system of parity checking to detect any possible errors. Some system designs even go as far as implementing an error correction system using a special Hamming error correction code. The X.25 card implements a parity checking system, which also serves to verify that the DRAMS have not failed.

A further problem with dynamic memories is that, unlike static memories, the address lines are multiplexed. The first 9 address lines are sampled on the falling edge of /RAS and the second 9 address lines are sampled on the falling edge of /CAS. Once again it is up to the user to ensure that correct address lines are presented to the DRAMS at the correct time.

In order to make dynamic memories easier to use, several manufacturers have designed special memory controller chips, such as the Intel 8203 or 8207 and the Motorola MC 3480. For the X.25 card design, PAL 2 implements the necessary memory control while refresh timing is obtained from the PC system board.

4.3.1 Refresh Pipelining:

One of the DMA channels on the PC system board is used for the memory refresh function. It is connected to a timer output and performs dummy DMA cycles at regular intervals, providing a relatively straighforward method to refresh the DRAMS. The refresh is available on the PC bus and hence a refresh cycle is very similar to that of a PC memory request.

In the PC circuit, the DMA controller chip gets control of the bus by means of a hold request line, similar to the bus control alternative described for the X.25 card. Thus the whole bus is inactive while a refresh cycle is in progress, decreasing the bus bandwidth by approximately 7% (IBM technical reference).

Clearly if the PC has to wait for a X.25 card to finish a memory cycle its performance will be degraded. Furthermore, if there are 4 cards installed, they will all have to wait for the slowest card to be refreshed. Hence one quickly ends up with the whole system being slowed down with very long refresh cycles.

To solve this, the refresh cycles are pipelined. When the PC does a refresh, the PC bus control PAL intructs the lower address buffer to latch in the refresh address. Thus the PC does its refresh cycle without waiting, irrespective of the activity on the X.25 card bus. The X.25 card then performs a refresh as soon as the current cycle is finished (a refresh request having the highest bus priority).

Thus, by utilizing the flexibility of the PAL chips, a very quick and efficient memory refresh is performed. From one of the previous pictures it was noted that the 8088 only had one wait state for the refresh cycle to take place.

4.3.2 Memory interfacing:

PAL2 (see X.25 schematic, sheet 4 of 7) provides the memory /RAS and /CAS control lines required for the memory, with the row address being latched on /RAS and the column address on /CAS. A problem here is that one must take into account the various delays to ensure that there is a stable column address when /CAS goes active. Hence the use of the F series gates on the address select, /MSEL and /CAS lines.

As an example, one can calculate the available column address setup time as follows:

```
 54ns    clock period from T5 to T6
- 6ns    F00 gate and F174 output skew
-32ns    max LS151 or LS158 gate delay
- 4ns    for capacitive loading
 ----
 12ns absolute min (DRAM's typically require 0)
```

It is interesting to note how worst case delays soon mount up, with a full clock period quickly diminishing to 12ns. Note the added delay due to capacitive loading : 45pF for the DRAM chips (5pF per input) and 10pF for the tracks (using 1pF per cm). With the LS chips being rated for 15pF, there is an extra 40pF loading which represents a 4ns delay. The 33 ohm resistors on the memory lines are used to dampen the bus so as to avoid overshoots.

For the WD2511 processors, an extra clock period was allowed between /MSEL and /CAS to allow for the increased skewness of the LS164 state generator, the PAL and the inverters. As the WD devices only use about 5% of the bus bandwidth (fully loaded at 64Kbps) the extra delay is neglible.

Finally, a clocked memory system is not the only solution; the PC for example uses a delay line. These are however very expensive and would only save a few ns in memory access time. Another solution, used in an IBM PC compatible design (Byte magazine Nov, Dec 1982) is to simply use a string of gates to generate the required delays. The problem here is that one has to rely on all chips having standard delays which seriously impares circuit reliability.

The design method here was therefore to take into account all possible delays and design accordingly. Performance is then achieved by reducing the number of delays (eg. /CAS comes directly on T6 with only the single F00 gate delay). Extensive memory tests carried out on the X.25 card's memory have never resulted in failures.

4.3 3 Memory Map:

To select the top 2 address lines, a LS151 multiplexer is used giving the memory map shown below:

First 64K block	:	8088 (channels status array, interrupt table)
Second	:	WD1 buffers and WD2 (selectable)
Third	:	WD2
Fourth	:	8088 programs, downloaded from PC.

Table 4.1 X.25 card Memory Map.

Note that only the top 64K block need be dual ported with the PC, with the PC loading the programs into this area of memory. Jumper J1 is used to select the position of the second WD2511 processor. Normally the link to ground will be used selecting the third memory block. Finally note that the card can also be used with 64K memory devices (configured by simply installing them) in which case the memory map above reduces to a single common 64K block.

4.3.4 Ready Logic:

As well as controlling the memory, PAL2 also generates the master WRITE line (used by all bus data buffers except those of WD processors mentioned earlier) and the ready logic.

With the ready logic, it is important to note that the PC sees all memory as being normally ready, so the circuit must deactivate the ready line as soon as possible. This justifies the use of /PCREQ line to directly control the I/O CHRDY line. Referring to the 8284A diagram given earlier, it is to be noted that an inactive edge of the ready line is only passed through a single synchronizing flip flop and so the circuit is able to generate the required setup time. The active edge of the ready line is given on T4 to ensure adequate data setup times for the 8088 processors.

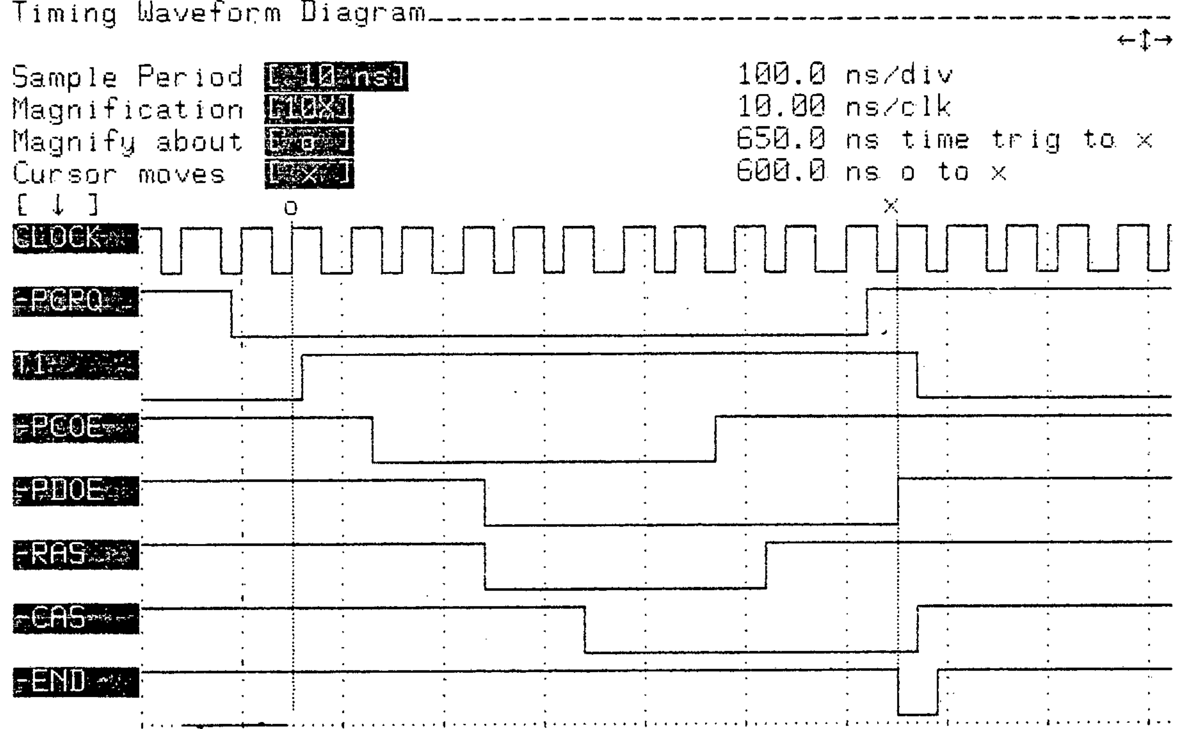
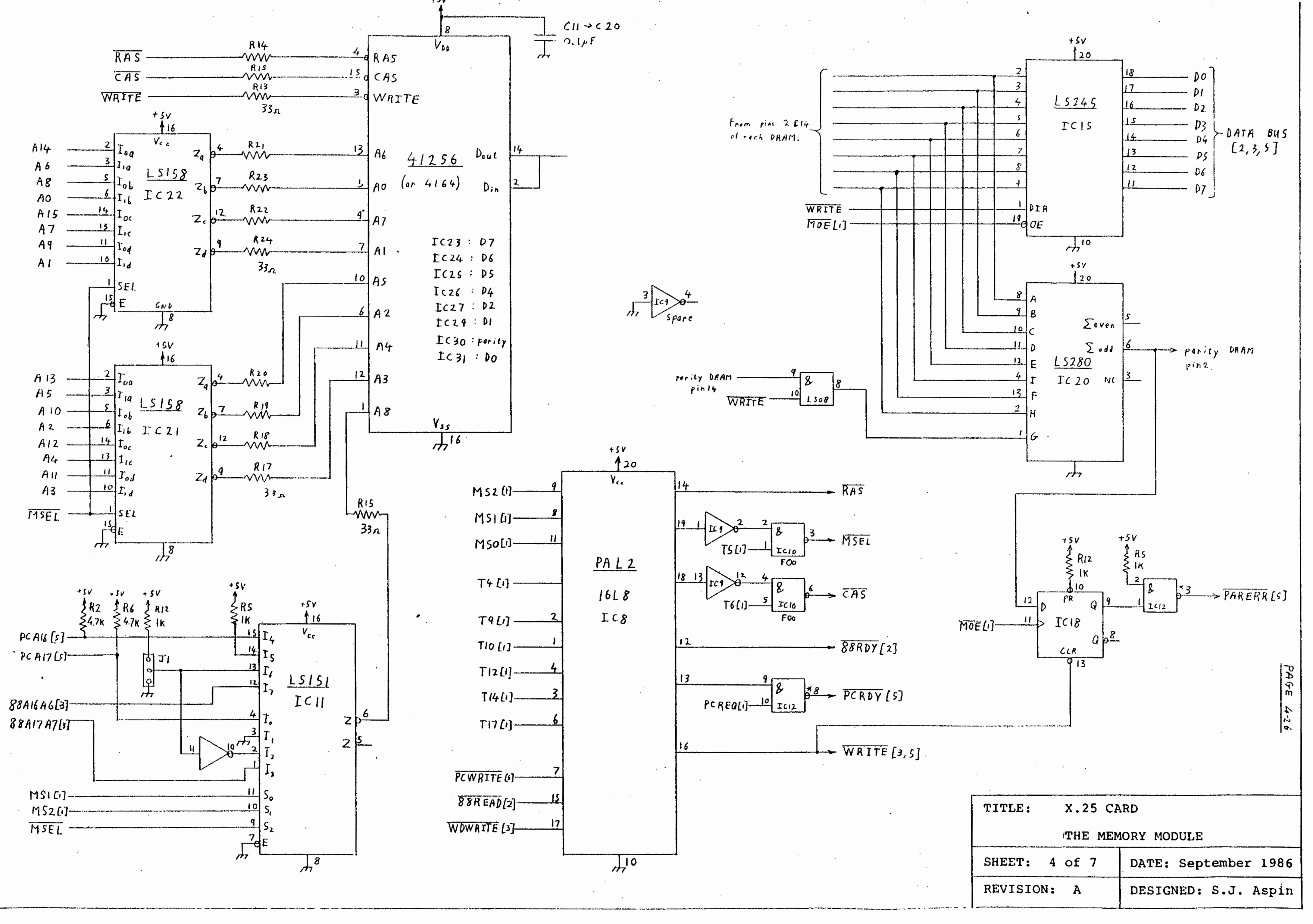


Fig 4.9 A typical PC memory access cycle

Note the timing of the /RAS and /CAS lines relative to the T1 state line. One wait state was inserted in the memory cycle.



TITLE: X.25 CARD

THE MEMORY MODULE

SHEET: 4 of 7	DATE: September 1986
REVISION: A	DESIGNED: S.J. Aspin

4.4 PC BUS INTERFACE LOGIC

The PC bus interface block performs two functions: firstly it selects the memory orientation as seen by the PC and secondly it provides I/O registers for the actual control of the card. Once again, a PAL is used to implement all the required logic functions (see listing of PAL5 equations in Appendix A).

4.4.1 Address decoding:

The memory configuration is selected by jumpers J7 to J10 and jumpers J12 and J13. J12 and J13 determine how much memory is shared with the PC as shown below:

J12	J13	amount of memory
closed	closed	64K
open	closed	128K
open	open	256K

The actual address of the memory in the PC's memory map is given by J7 to J10 as shown below (fig 4.10) for a card with 128K dual port memory. (Refer to Appendix A).

J7(A11)	J8(A18)	J9(A17)	J10(A16)	address	range
ON	ON	OFF	OFF	001X	128 - 256K
ON	OFF	ON	OFF	010X	256 - 384K
ON	OFF	OFF	OFF	011X	384 - 512K
OFF	ON	ON	OFF	100X	512 - 640K

Table 4.2 PC memory address decoding

Usually the card's DRAM will be located in the memory area from 128K to the DOS limit of 640K and so the jumper configurations are only listed for this address range. Jumpers on the PC will also need to be set to tell the PC that more memory has been installed. Documentation on these settings is given in the Installation manual of the PC.

If multiple cards are installed, as for the final traffic generator, each card will need to be located at a different address range. Typically 128K would be dual ported in a system of three X.25 cards. Together with 256K on the PC, this brings the DOS memory to its maximum of 640K bytes. With the maximum of five cards installed, the amount of dual ported memory would need to be reduced to 64K.

Thus the use of the card is very flexible and the actual configuration will depend largely on what is already installed in the PC and how many X.25 cards are used.

A final note is that, when 256K of RAM is dual ported, J9 and J10 may be used to invert the order in which the PC sees the memory. Usually J9 and J10 will be off so that the order is the opposite as seen by the card's 8088 processor. ie. the card's program area appears as the lowest dual port memory block. The reasons for these settings are described more fully in chapter 7.

4.4.2 I/O Control:

The PC controls the X.25 card by writing to or reading from certain I/O locations. J6 allows the user to select from six possible I/O address ranges, all of which are unused by the PC. The address ranges selected by J6 are given in the circuit diagram, while the I/O address map of the PC is given below:

Hex Range	Usage
000-00F	DMA Chip 8237A-5
020-021	Interrupt 8259A
040-043	Timer 8253-5
060-063	PPI 8255A-5
080-083	DMA Page Registers
0Ax*	NMI Mask Register
0Cx	Reserved
0Ex	Reserved
100-1FF	Not Useable
200-20F	Game Control
210-217	Expansion Unit
220-24F	Reserved
278-27F	Reserved
2F0-2F7	Reserved
2F8-2FF	Asynchronous Communications (Secondary)
300-31F	Prototype Card
320-32F	Fixed Disk
378-37F	Parallel Printer
380-38F	SDLC Communications
3A0-3AF	Reserved
3B0-3BF	IBM Monochrome Display/Printer
3C0-3CF	Reserved
3D0-3DF	Color/Graphics
3E0-3E7	Reserved
3F0-3F7	Diskette
3F8-3FF	Asynchronous Communications (Primary)

Table 4.3 PC I/O address map.

Thus the user will need to check the I/O address map if additional optional cards are installed in the PC. The card address may be selected from one of six areas, ranging from 220 to 2DF here. This is a reserved area but does not clash with any currently available IBM cards.

The actual use of the I/O address map is given below in table 4.4. The address values assume that J6 is in the top position (ie. 220 to 23F here).

Address	A4	A3	A2	A1	A0	IOR	IOW	action
544	0	0	x	x	x	1	0	Reset card
544	0	0	x	x	x	0	1	Enable card
552	0	1	x	x	x	1	0	8088 NMI
552	0	1	x	x	x	0	1	8088 Interrupt
560	1	0	0	0	0	x	x]	
							]	WD2511 chip
							]	registers
575	1	1	1	1	1	x	x]	

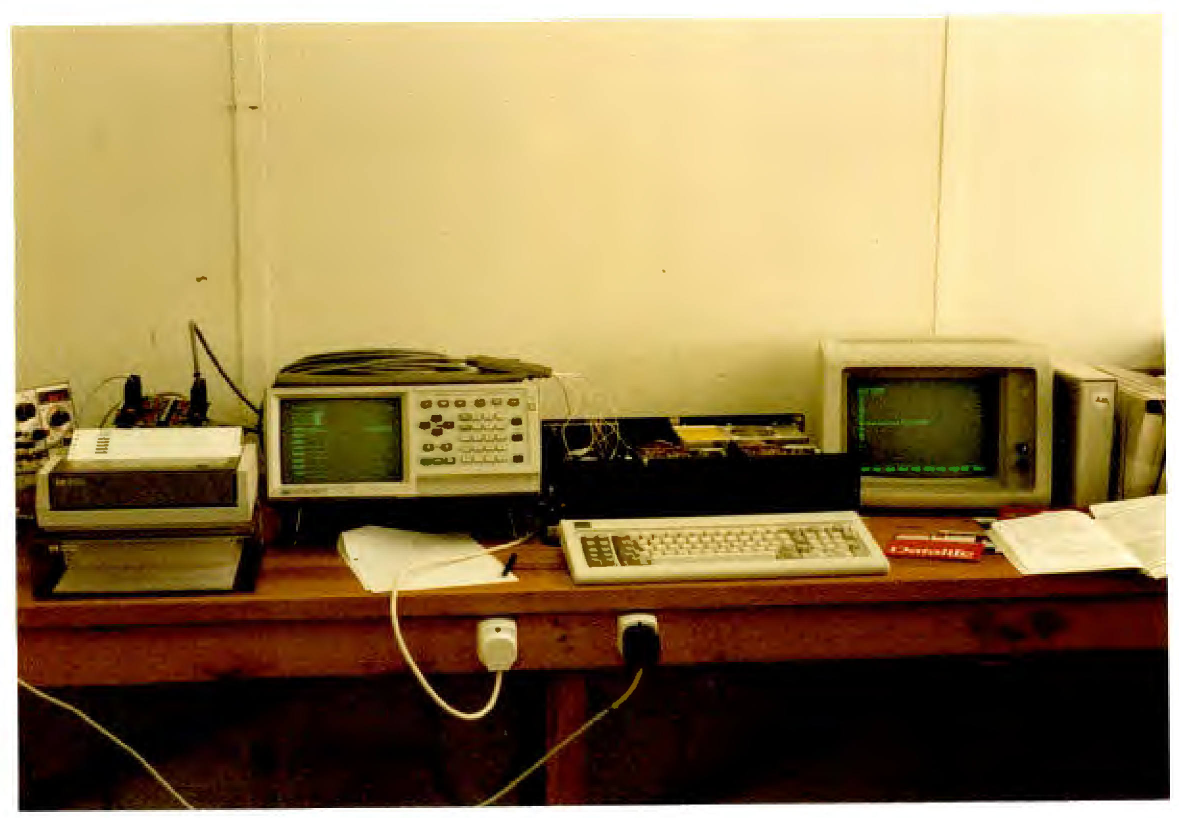
Table 4.4 PC I/O address decoding.

On power up, the circuit will be in the reset state, so the card's processors are disabled. This allows the card to be permanently installed in a PC, acting as a standard memory board.

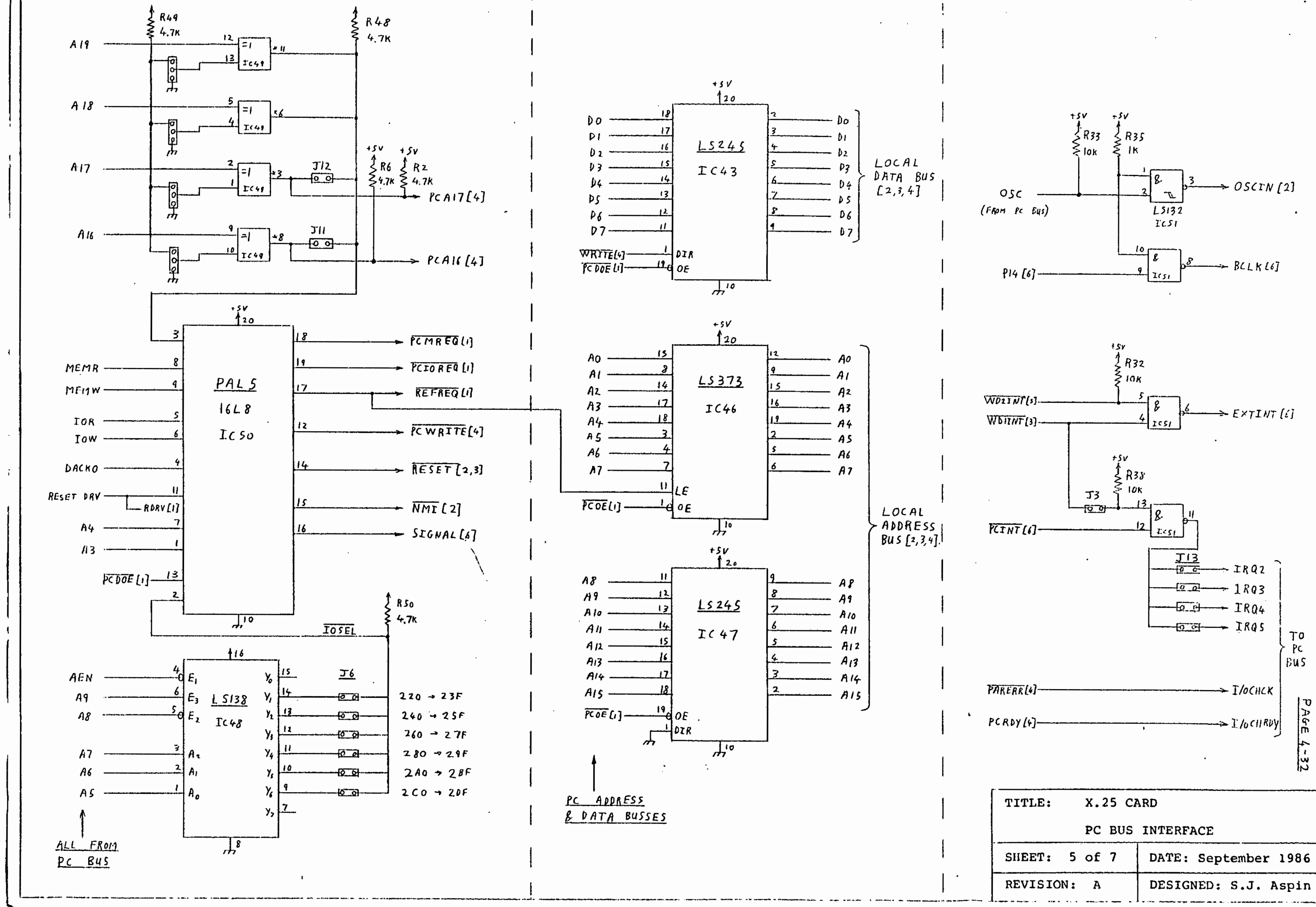
Number	Usage
NMI	Parity
0	Timer
1	Keyboard
2	Reserved
3	Asynchronous Communications (Secondary) SDLC Communications
4	Asynchronous Communications (Primary) SDLC Communications
5	Fixed Disk
6	Diskette
7	Parallel Printer

Table 4.5 PC interrupt table.

The final hardware configuration is that the circuit allows an interrupt request to be given to the PC. This may come either from the 8088 or the WD2511 processor (selected by J3) and is given on interrupt lines IRQ2 to IRQ5 (selected by J11). As there are few interrupt lines, one must be particularly careful about clashes: no 2 is reserved, no3 and no4 are for comms. cards and no5 is for the fixed disk (the other four lines on the PC are not available at all). The current system does not use interrups as the exact time the PC updates a display is irrelevant.



Picture of the test setup used to obtain the traces presented in this chapter.



R49 4.7K
R48 4.7K
A19
A18
A17
A16
IC49
J12
J11
R6 4.7K
R2 4.7K
PCA17 [4]
PCA16 [4]
PAL 5
16L8
IC50
MEMR
MEMW
IOR
IOW
DACKO
RESET DRV
RDRV [1]
A4
A3
PCDOE [1]
PCMREQ [1]
PCIOREQ [1]
REFREQ [1]
PCWRITE [4]
RESET [2,3]
NMI [2]
SIGNAL [6]
R50 4.7K
IOSEL
AEN
A9
A8
A7
A6
A5
LS138
IC48
J6
220 → 23F
240 → 25F
260 → 27F
280 → 29F
2A0 → 2BF
2C0 → 2DF
ALL FROM
PC BUS
LS245
IC43
D0
D1
D2
D3
D4
D5
D6
D7
WRITE [4]
PCDOE [1]
DIR
OE
LOCAL
DATA BUS
[2,3,4]
LS373
IC46
A0
A1
A2
LE
PCOE [1]
LOCAL
ADDRESS
BUS [2,3,4]
LS245
IC47
A10
A11
A12
A13
A14
A15
PC ADDRESS
& DATA BUSSES
R33 10K
R35 1K
OSC
(FROM PC BUS)
LS132
IC51
OSCIN [2]
P14 [6]
BCLK [6]
R32 10K
WD2INT [3]
WD1INT [3]
EXTINT [6]
R38 10K
J3
PCINT [6]
J13
IRQ2
IRQ3
IRQ4
IRQ5
TO
PC
BUS
PARERR [4]
I/OCHCK
PCRDY [4]
I/OCHRDY
PAGE 4-32
TITLE: X.25 CARD
PC BUS INTERFACE
SHEET: 5 of 7
DATE: September 1986
REVISION: A
DESIGNED: S.J. Aspin

Finally, the chip has a programmable asynchronous communications interface. This turns the card into a true multifunction communications system, able to connect up to standard asynch. equipment as well as synchronous X.25 systems.

4.5.1 The V.35 physical interface:

According to Post Office specifications, the card interface adheres to the CCITT V.35 recommendation. The V.35 specification is entitled "Data transmission at 48 Kbps using 60-108KHz group band circuits" and only briefly describes the interface in an appendix. The actual pin allocation is defined in the ISO specification 2593 and is given in table 4.6.

V.35 is an old standard, initially defined in 1968, and uses a mixture of balanced and unbalanced circuits. The signal lines (request to send, calling indicator etc.) use the standard V.24 or RS232C specification while the clock and data lines use a balanced configuration similar to RS422. The circuit was therefore designed accordingly and supports the required signals.

An interesting point about the interface is the clock signals. The clock is always supplied by the modem (or DCE) to which the DTE is connected. For the receive clock this is fairly straightforward but for the transmitter the DCE supplies the clock on SCT and looks at the data on the SCTE clock from the DTE (see circuit diagram 6). This arrangement ensures that the transmit clock is in phase with the transmit data. On the card J4 and J5 allow the clock to be obtained from the MUART for initial testing purposes.

Pin	Function	CCITT circuit No.	Direction
A	Protective ground or earth	101	common
B	Signal ground or common return	102	common
C	Request to send	105	from DTE
D	Ready for sending	106	to DTE
E	Data set ready	107	to DTE
F	Data channel received line signal detector	109	to DTE
H	Connect data set to line	108/1	from DTE
	Data terminal ready	108/2	from DTE
J	Calling indicator	125	to DTE
K	F_1	–	–
L	F_2	–	–
M	F_1	–	–
N	F_2	–	–
R	Received data A-wire	104	to DTE
T	Received data B-wire	104	to DTE
V	Receiver signal element timing A-wire	115	to DTE
X	Receiver signal element timing B-wire	115	to DTE
Y	Transmitter signal element timing A-wire	114	to DTE
AA	Transmitter signal element timing B-wire	114	to DTE
P	Transmitted data A-wire	103	from DTE
S	Transmitted data B-wire	103	from DTE
U	Transmitter signal element timing A-wire	113	from DTE
Z	F_3	–	–
W	Transmitter signal element timing B-wire	113	from DTE
BB	F_3	–	–
CC	F_4	–	–
DD	F_5	–	–
EE	F_4	–	–
FF	F_5	–	–
HH	N_1	–	–
JJ	N_2	–	–
KK	N_1	–	–
LL	N_2	–	–
MM	F	–	–
NN	F	–	–

Table 4.6 Pin Allocation for V.35 Interface.

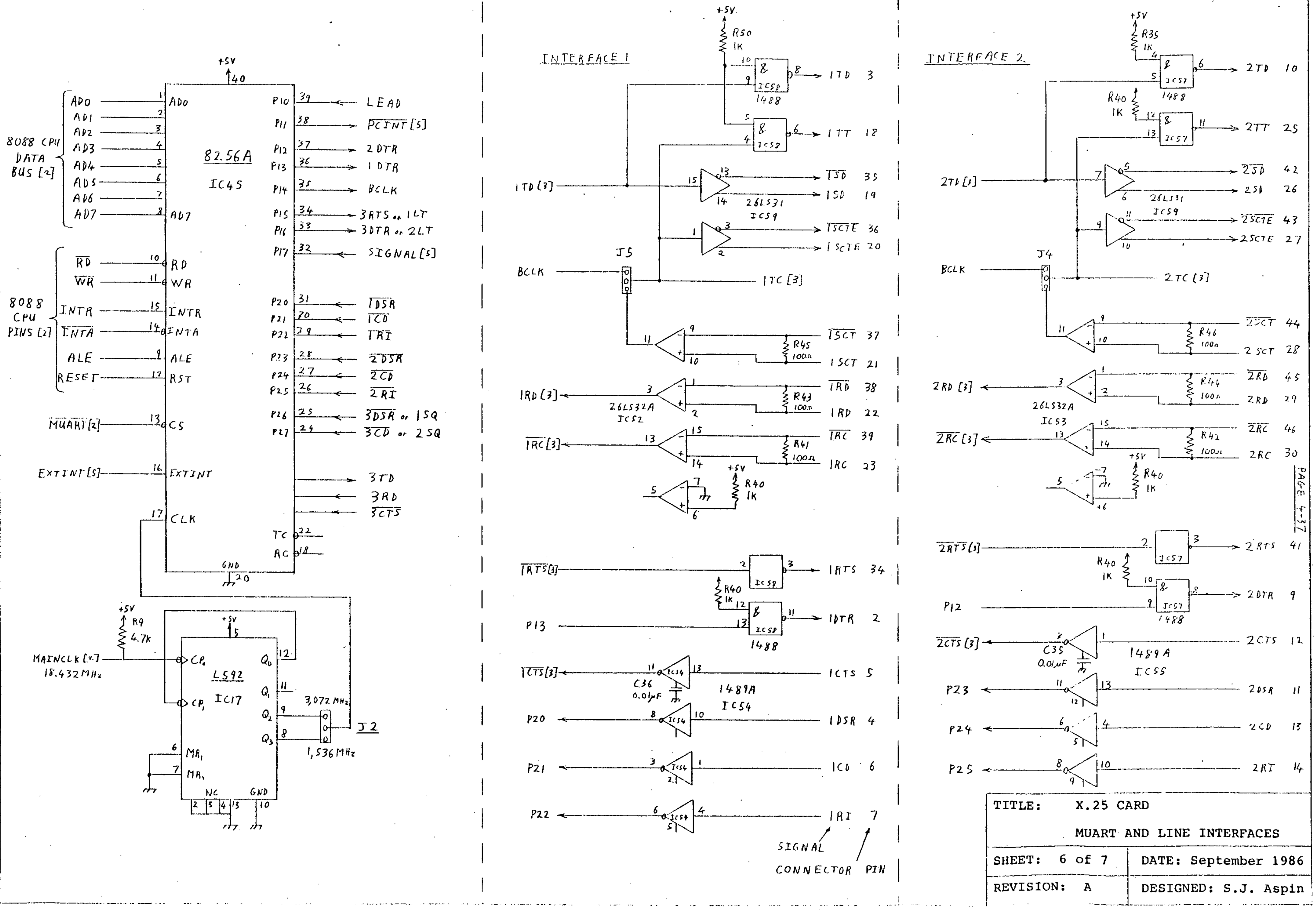
4.5.2 The RS 232 interface:

Although V.35 was all that was required, the use of this interface limits the card to being used with high speed modems and communication equipment. Thus it was decided to also implement the more common RS232C standard.

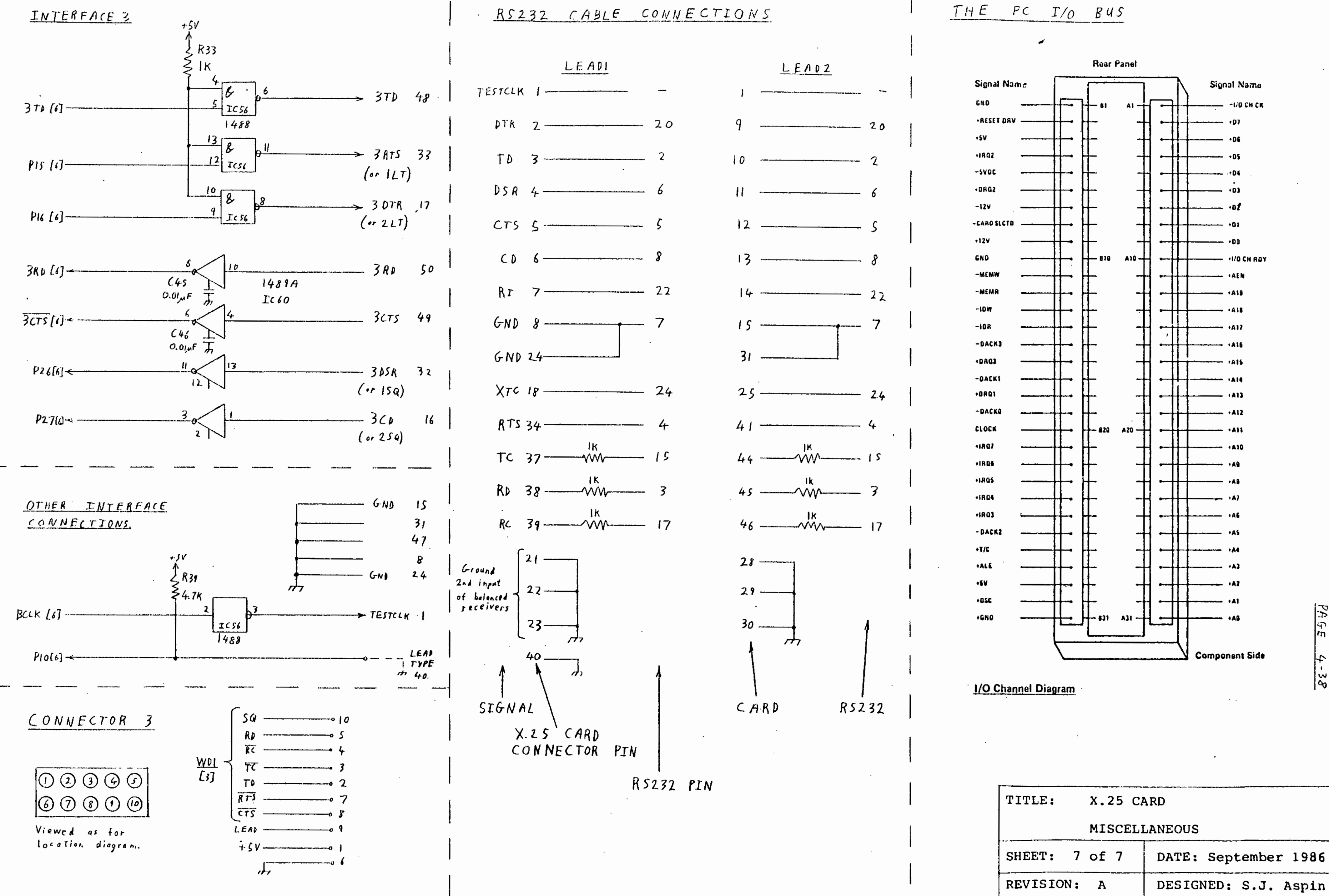
Both interfaces are brought out on the cards 50 pin connector with one of the pins selecting which interface is being used. Thus the user simply plugs in a different cable to select the RS232 interface with the software being able to detect which one is present. For the transmit circuits (transmit clock and data) standard RS232 drivers are connected in parallel with the balanced ones. Clearly this cannot be done for the receive circuits, so the balanced receivers are used for both RS232 and V.35 interfaces. The line receivers used are suitable for RS232, but unfortunately the 100 ohm termination specified for V.35 is far too low.

To solve this problem, three resistors are wired into the RS232 plug acting as a voltage divider. This arrangement reduces the load current to acceptable levels while at the same time magnifying the input hysterises of the 26LS32A receivers.

The card supports all the RS232 signals required for connection to a synchronous modem (Witten 1983) allowing the X.25 card to be used for other applications, such as using the PC as a PAD. The asynchronous communication link is also buffered and brought out on the connector, thereby increasing the versatility of the card.



8088 CPU DATA BUS [2]
8088 CPU PINS [2]
82.56A
IC45
LEAD
PCINT [5]
2DTR
1DTR
BCLK
3RTS or 1LT
3DTR or 2LT
SIGNAL [5]
1DSR
1CD
1RI
2DSR
2CD
2RI
3DSR or 1SQ
3CD or 2SQ
3TD
3RD
3CTS
MUART [2]
EXTINT [5]
MAINCLK [2]
18.432MHz
LS92
IC17
3,072 MHz
1,536 MHz
J2
INTERFACE 1
INTERFACE 2
1488
26LS31
IC59
26LS32A
IC52
IC53
1489A
IC54
IC55
SIGNAL
CONNECTOR PIN
PAGE 4-37
TITLE: X.25 CARD
MUART AND LINE INTERFACES
SHEET: 6 of 7
DATE: September 1986
REVISION: A
DESIGNED: S.J. Aspin



INTERFACE 3
+5V
R33
1K
IC56
1488
3TD [6]
P15 [6]
P16 [6]
3TD 48
3RTS 33
(or 1LT)
3DTR 17
(or 2LT)
3RD [6]
3CTS [6]
P26[6]
P27[6]
C45
0.01μF
C46
0.01μF
1489A
IC60
3RD 50
3CTS 49
3DSR 32
(or 15Q)
3CD 16
(or 25Q)
OTHER INTERFACE CONNECTIONS.
GND 15
31
47
8
GND 24
R39
4.7K
BCLK [6]
P10[6]
TESTCLK 1
LEAD TYPE 40.
CONNECTOR 3
Viewed as for location diagram.
WD1 [3]
SQ 10
RD 5
RC 4
TC 3
TD 2
RTS 7
CTS 8
LEAD 9
+5V 1
6
RS232 CABLE CONNECTIONS
LEAD1
LEAD2
TESTCLK 1
DTR 2
TD 3
DSR 4
CTS 5
CD 6
RI 7
GND 8
GND 24
XTC 18
RTS 34
TC 37
RD 38
RC 39
1K
Ground 2nd input of balanced receivers
SIGNAL
X.25 CARD CONNECTOR PIN
RS232 PIN
CARD
RS232
THE PC I/O BUS
Rear Panel
Signal Name
Component Side
I/O Channel Diagram
PAGE 4-38
TITLE: X.25 CARD
MISCELLANEOUS
SHEET: 7 of 7
DATE: September 1986
REVISION: A
DESIGNED: S.J. Aspin

4.6 CIRCUIT DESIGN CONCLUSIONS AND TESTING

The main design aspects of the X.25 card have now been covered. It should be noted however that the card design went through several versions before reaching the state described. Initially the card started off with 64K of memory and a single link level processor but it was soon realised that, from a software point of view, 64K was rather restrictive. Also by putting two WD2511 processors on the card, the overall cost of the traffic generator would nearly be halved.

Circuit design was done in TTL with the Western Digital application note providing the basic arbitrator circuit. It was only after reading through the literature that the problem of metastability was discovered and the WD design was consequently rejected. Most of the control logic required was redesigned around PALs. As well as reducing the chip count, PALs are far more versatile and allow for precise control of the card's operation.

Having finalized the design, work started on the circuit board layout. Although it took considerable time to design due to the circuit complexity, the circuit board does ensure easy manufacture of the X.25 card. This was particularly important as several cards would be required for the traffic generator and it allows the card to be built for other applications. Also, a circuit board is more reliable as it is less noisy and there is less chance of short circuits. This reliability is particularly important for a design using dynamic memories and having a multiple processor architecture.

Having designed and constructed the card, the system was thoroughly verified using a logic analyser. A summary of the tests conducted is given in Appendix A, while the traces in this chapter depict typical bus cycles. As the card was designed taking into account worst case delays, no timing problems were encountered. Errors were however discovered in two of the PAL equations, but these were easily fixed by reprogramming them. If a new card is constructed, it is strongly recommended that the tests be repeated as it is all too easy to make an error in entering the PAL equations.

The card was designed primarily to have a high throughput for its 8088 and WD2511 processors. This was achieved with the use of the "alternate configuration" for the 8088 processor and by optimising the WD2511 processors to have the shortest possible bus cycle times. The circuit was also designed with reliability in mind and adequate timing margins are allowed for critical circuit parameters such as the data bus control.

The X.25 card therefore provides a versatile high performance multiple processor system with required software being directly downloaded form the host PC. This software is described in the chapters to follow.

THE C LANGUAGE

So far the overall design of the traffic generator and the required hardware have been considered. In chapter 4 the design of the X.25 card was described. This chapter deals with its software requirements.

The basic functions executing in the X.25 card are: the X.25 protocol, generating test packets and monitoring line activity. The layered structure of the software is depicted below. A separate program (x25TG) running on the 8088-PC, controls and monitors the cards and drives the display.

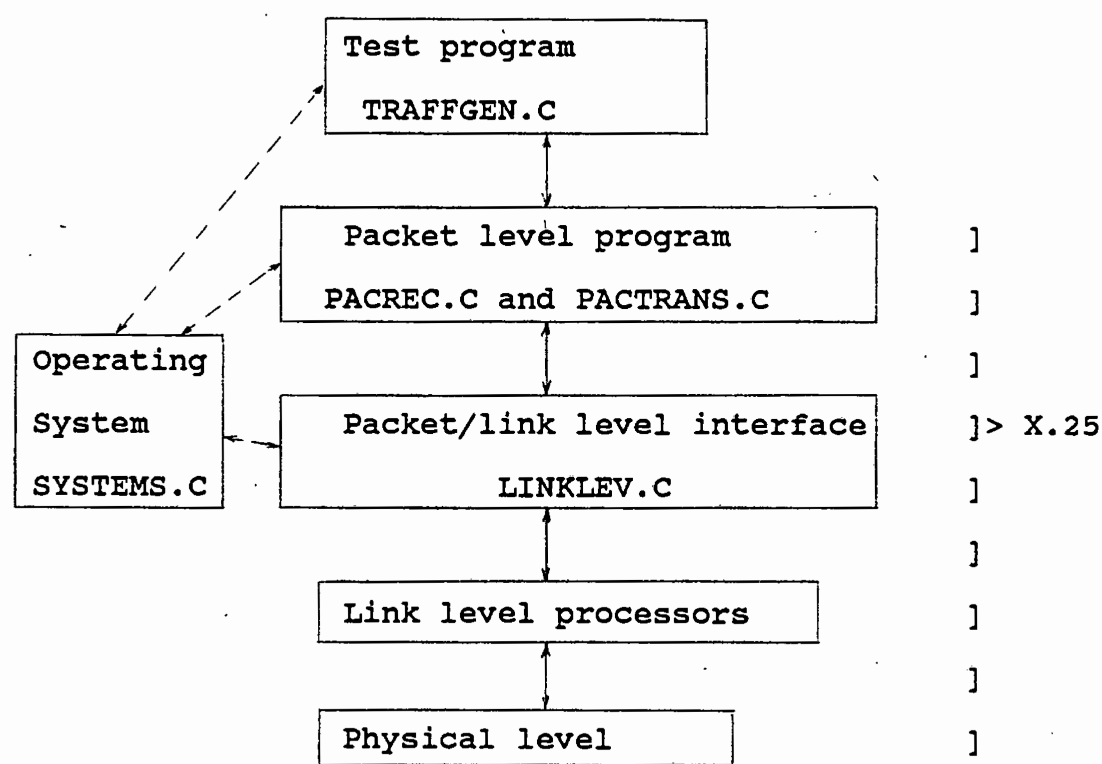


Fig 5.1 Program Structure

In this layered architecture each entity performs a specific set of tasks. For example, the test program will pass a primitive/event to the packet level requesting a call set up on a particular logical channel. The packet level requests a buffer from the operating system to build the call request packet. The buffer is then added to the link level queue and finally presented to the link level. The link level adds the header and the frame check sequence before transmitting the frame over the V.35 interface.

Software requirements:

For the traffic generator application the main software requirement is high speed data transmission/capture rather than a complete implementation of X.25. Thus only the essential packet level services required by the traffic generator were coded. There is for example no need for a transport level interface as there is no transport level. It is however desirable for the code to be modular and written in a good structured language.

5.1 THE C PROGRAMMING LANGUAGE

Clearly the language choice is an important decision to make when writing a large piece of software. It would for example be inefficient to write complex scientific programs in COBOL, or business applications in FORTRAN. For this application the most suitable languages initially appeared to be Pascal or Modula 2.

The coding was in fact started in Pascal, but it was noted that almost all the modern commercial X.25 system boasted the use of the new "C" language e.g. "Portable-written in the language C" (TITN 1984).

Numerous articles have been written on the use of C, with an issue in Byte magazine (April 1983) and, in PC magazine (March 1984) devoted to modern languages. Perhaps one sentence sums it all up "Lean, fast, and powerful, the language C is emerging as the Ferrari of modern programming language for the PC". It was soon decided to change to C and this thesis is problably the first in the department to be written entirely in this modern language. Before continuing it is worthwhile to look at the development of C and what it offers and why it was chosen for the traffic generator application.

5.1.1 History of C:

C was developed in the 1970's almost entirely at AT&T's Bell Laboratories. It started in 1969 when Ken Thompson started developing the UNIX operating system, based on a slower and older operating system called MULTICS. The first versions of UNIX were written in assembler for a PDP mini-computer and so the operating system was not portable. To solve the portability problem, Thompson developed the B language. This was then modified and improved by Dennis Ritchie to become the C programming language. By 1980 almost all of UNIX was written in C, making it highly portable.

In 1981 the first "commercial" UNIX system was released by AT&T, with the System V version being released in 1983. (For a good introduction see Thomas 1985). Since then UNIX has been ported to numerous computers and has become a popular operating system for new mini-computers such as the Digital VAX range. In addition, Microsoft has written a version of UNIX, called XENIX, for use on micro-computers such as the IBM PC. Thus UNIX and C developed hand in hand and the growing popularity of UNIX encouraged the use of C. In 1978 the definitive text "The C programming language" was published. The book, written by Brian Kernighan and Dennis Ritchie of Bell Laboratories, is exceptionally well written as it clearly defines all aspects of the C language. Having a good standard also encouraged the use of C.

Unlike Pascal, which was intended to be used as a model to teach structural programming, C was developed by a professional program developer and is therefore oriented towards the needs of the programmer. C is a "low level language" and allows the programmer to get close to the target processor while still providing the features (such as complex data structures) associated with high level languages.

Because it is close to the processor, C compilers are very efficient. This feature is very important for commercial programmers where execution speed helps sell the product. Many programms for the PC, such as Wordstar and dDASE III, are written in C (Thomas 1985). Execution speed is particularly important for the traffic generator where, due to the high data rates, there is a lot of processing to be done.

5.1.2 Programming in C:

Clearly the features described above make C ideal for writing the X.25 traffic generator programs in. However, first one has to familiarize with the fairly terse C source code.

A few general differences between C and other languages (eg. Pascal) are worth noting. Firstly, C permits and encourages the construction of terse source code. A simple example is the use of "{" rather than the Pascal BEGIN, or the compact "for" loop statement (e.g. "for (linkno = 0; linkno <2; linkno ++)")

This generally results in short, compact programs.

Secondly, C is not as strongly typed as Pascal. While this can be dangerous it does greatly increase the flexibility. It is the programmer's responsibility to effect the conversion (e.g., one must be very careful of the sign bit). On this subject, C can generally cast one data type into another as for example in this statement:

```
rlook[0] = (struct buff_table far *) (WD1TAB_ADDRESS + 64);
```

Here the address of the receive look up table of a WD2511 processor is cast to a far pointer to a data structure.

The above example leads to another distinguishing feature of C, its extensive use of pointers. These are manipulated via the * and & operators as follows: & returns the address of a variable while * treats its operand as the address of the ultimate target.

```
eg. the statments        px = &x
                         y  = *px
are equivalent to        y  = x.
```

(See Chapter 5 of C programming language, Kernighan 1978).

5.1.3 Program Structure

C has a block structure similar to Pascal. One important difference is that C only has functions with parameters being passed by value. In order to update the actual variable, the programmer must pass its address to the function via a pointer.

As the traffic generator program has a large number of data structures, such as buffers, channel status, statistics etc., all type declarations were put into a file called DTYPES while the variable declarations are in a file DVARS (see Appendix A). These files are included in all program modules, with the actual structures being defined in the operating system module. The big advantage of this progam structure is that if an alteration is made, all program modules are updated. The program constants are similarly contained in several separate constant files (see Appendix G).

One should exercise care when accessing data structures, as illustrated in the following code:

```
        extern struct buff_control  rcontrol [2][NO_REC_BUFFS];
        for (i =0; i <=NO_REC_BUFFS; i++)
                rcontrol[0][i]. busy = FALSE;
```

This code, similar to that used in the program (see rbuff_unit function in SYSTEMS module), declares all buffers for a link as being free. The programming error in the code is that, due to the use of <=, it will initialize one data structure too many. C does not check array bounds and will overwrite whatever happened to be in the way. Similarly, if a function in a different module has passed an 'int' but expects a 'long', it will simply take a 'long' from the stack.

The traffic generator program therefore consists of several modules and a common set of data structures. All the variables are passed to the various functions when they are called.

5.1.4 Programming technique:

As C is very flexible, there are usually several methods of tackling a problem. An example (from Purdun 1986) is the code to set y to i if x equals 5, otherwise to set y to 0. The usual constructs are

```
Y = 0;
if (x = = 5)
        y = 1;
```

A second method uses the ternary operator

```
y = (x ==5) ? 1 = 0;
```

Finally, a third method is to make use of the hierarchy of the C operators to give:

```
y = x = = 5;
```

Thus, proficiency in the language is a function of experience.

In this thesis the author tended to avoid the third method of coding. For example, there are often unnecessarily equates to TRUE, which are made simply to improve code readibility (e.g. if ((master).block == TRUE)). The ternary operator is however extensively used in the traffic generator programs. An example follows:

```
work_in = (++work_in == MAX_QUE) ? 0 : work_in;
```

This code increments the work queue pointer work_in, testing if it is equal to the maximum queue size. If true, then the pointer is set to 0, otherwise it retains its new value. Thus the queue is circular in nature with a size given by the constant MAX_QUE.

An interesting point to note here is the use of the ++ to pre-increment work_in rather than post-increment it as in

```
work_in = (work_in ++ == MAX_QUE) ? 0: work_in;
```

This performs the comparison before incrementing the pointer, giving the possibility of work_in being equal to MAX_QUE. This will result in accessing a non existing structure element with the corresponding unpredictable program behaviour.

Thus C is a very powerful and flexible language and is ideally suited to applications such as the traffic generator. So far this chapter has given a general introduction to the C language and program structures.

5.2 COMPILER CHOICE:

Having looked at the development of C, its features and some typical program statements, the choice of compiler needs to be evaluated. There are literally dozens of C compilers on the market, ranging from small introductory versions to professional development systems such as the Microsoft C compiler. There were several factors why this package is most suited to the application and these are now discussed:

5.2.1 Memory models:

This first important criterion in the compiler choice is the question of memory models. This problem arises from the memory segmentation used by the 8086 family of processors as was depicted in the 8088 block diagram (see chapter 4). The registers in the 8088 are 16 bits wide and thus only address 64K. In order to access more than 64K, the particular segment register must be changed. The memory segmentation is a major problem particularly as the traffic generator data structures are spaced out over the card's 256K memory block.

Early compilers limited the user to 64K of program and 64K of data. Today most professional compilers have a system of memory models and, for example, the large memory model of Microsoft C allows code and data to be accessed over the full mega byte address range. Microsoft C also allows the use of the FAR keyword, enabling one to use the small memory model while still being able to occasionally access far data structures. This useful feature is used in the traffic generator application.

5.2.2 Language continuity:

As Microsoft also developed the DOS operating system their C compiler is fully compatible with it the linker and other utilities. This is important as the code needs to be downloaded onto the card in a stand-alone configuration. None of the available compilers supported stand alone code.

Also Microsoft C can be used directly with other Microsoft products, including Pascal, Fortran and assembler.

5.2.3 Library functions:

The C language does not support standard I/O facilities. Instead they are in a library. For example, all keyboard and display operations are performed by calling the relevant functions which in turn call the necessary BIOS routines.

5.2.4 Conclusions:

C is a very new language developed together with the UNIX operating system. It is a system's programming language and is ideally suited to writing programs for applications such as the traffic generator. Used on the PC it provides a very powerful programming environment with the ability to interface easily to the DOS operating system. Programs are both developed and run on the PC, making it easy to modify and upgrade the code.

SOFTWARE DESIGN

Having looked at the C language and the overall concept of the software, this chapter looks at the actual traffic generator program design. The objective of the program is to generate and receive test data packets and monitor the line activity.

There are two ways of obtaining this objective. The first and, in the short term the simplest solution, is to lump all the programs into a monolithic unit. The program would loop around monitoring the status of the link level and creating test packets accordingly. The problem with this sequential approach is the difficulty of modifying or extending the software. Also, it becomes tricky to keep track of exactly what is going on in the system.

The second solution, adopted for this project, is to have a layered software structure with each module implementing a specific set of functions. The modules then interact by means of a set of shared data structures. A simple kernel program is then written to dispatch work from a work queue, thereby activating a specific module or process. Thus to extend the program, one just needs to add further modules to the existing ones.

The program was therefore divided into the following compilation modules:

SYSTEMS	operating system
TRAFFGEN	actual traffic generator
PACTRANS	packet transmit
PACREC	packet received
LINKLEV	link level interface

These modules, listed in appendicies D and E, are compiled to become the CARD run file which executes on the X.25 cards. Another program, X25TG, runs on the PC, controlling the cards and interacting with the user. Finally there are the data structure and variable declaration files (appendix C) as well as several constant files (appendix F). The layered software structure is illustrated in fig 6.1 below.

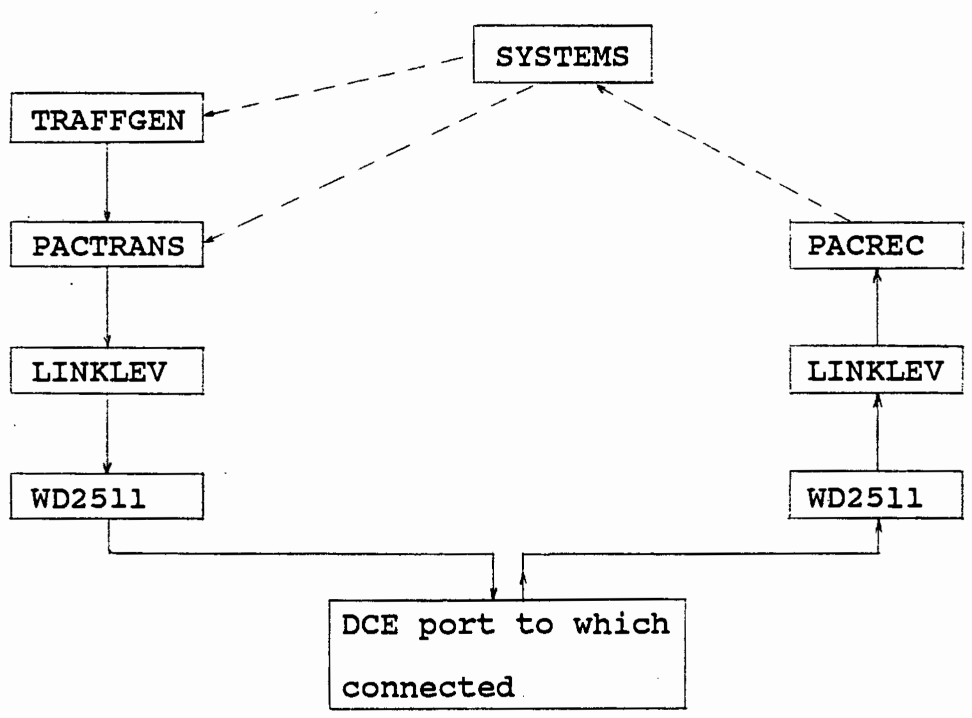


Fig 6.1 Software architecture

6.1 Data structure declarations

This is perhaps the most important compilation module. The flow of data (signals) conveying information between processes/modules are instantiations of these data structures or types.
The following data structures are declared (see Appendix C):

- the work queue
- the buffers and buffer control structures
- the link level look up tables and queue
- the channel status array
- a statistics structure
- a control structure

6.1.1 Locating data structures

Several of the data structures, particularly those related to the link level processors, need to be located at specific memory addresses. For example, the memory look up tables must be located in the 64K block accessible by the WD2511 processors. As there is no locator in the Microsoft C package this represents quite a problem. An initial idea was to link in some form of assembly routine which would perform the necessary location. This method is however rather messy and would basically involve getting the C compiler to store the data structure in different groups, with assembler routines assigning the addresses to the groups.

A further problem is that C initializes all external data structures, resulting in a huge program. Perhaps even more serious is that, when several X.25 cards are installed, not all the shared memory is necessarily dual ported. This means that even if one could locate the data structures it would be impossible to load them.

The problem was eventually solved by simply utilizing the power of C. As mentioned in the last chapter, one characteristic of C is its extensive use of memory pointers to point to strings and other data items. Thus one can easily use pointers to point to the required data structures and this led to data structure definitions such as:

```
extern struct  buff_table (far * rlook[2])[8];
```

This statement declares an array of 2 far pointers to an array of 8 structures. The actual structure, buff_table, is the memory look up table structure defined for the WD2511 processors. The base structure address is easily assigned to the pointers and this was in fact done in the last chapter.

In the data declarations it is generally important to note the use of the brackets. For example, the declaration

```
extern struct  chan_state (far * chans)[2][MAX_CHANS];
```

refers to a pointer to an array of arrays rather than an array of pointers as above. The use of an array of two is a result of two link level processors per card.

Thus pointers provide a neat and easy way of locating data structures, while the 'far' keyword allows the structures to be located anywhere in the 8088's memory map. To conclude, a typical memory map for the X.25 card is given below:

Address	Contents
3FFFF – 3FFF0	Assembler start up routines
3FFF0 – 30000	Statistics and control data structures C programs and data
30000 – 20000	Link processor 2 buffers and Memory look up tables
20000 – 10000	Link processors 1 buffers and Memory look up tables
10000 – 00000	Channels status array 8088 interrupt table

Fig 6.2 X.25 card system memory map

In practice, the system was configured to have a slightly different memory map to avoid having to reload DOS (see chapter 7). This involved altering a hardware jumper so that the second link level processor shares the same 64K block as the first one (refer to appendix A for jumper settings). The channels status array was then located in the third 64K block leaving the first block free. Note the size of the channels status array - it is 64K long with each element containing 32 bytes of data about the channel state. Thus the traffic generator can monitor 2048 channels or 1023 per link (channel 0 is reserved). It is therefore primarily due to the large amount of available memory that the traffic generator can support so many channels (6 was the minimum required for the traffic generator).

The type and variable declaration files (DTYPE and DVAR, listed in Appendix C) are used by all the traffic generator programs and are included in the files using the "# include" pre-processor directive. Having a single file of variables outside of the function blocks allows for easy program updating. The external variables are defines in the VDEFS file, which must be included in one module, usually SYSTEMS.

6.1.2 Data structure access time

An important consideration with the data structures is the access time. Although invisible to the programmer, the compiler must produce assembler code to calculate the actual address of each data element. Exactly what code is produced is very important for applications such as the traffic generator where speed is essential and data structures are used frequently.

Consider accessing say the third element of a structure which is seven bytes long. The address is computed by using the integer multiply instruction 'imul' to multiply the structure length by the element number. The only problem is that the imul instruction is one of the longest in the 8088's instruction set, taking around 140 clock cycles to execute (iAPX 88 Book, Intel 1981). Considering the large number of data structure accesses required, this will certainly use a lot of the 8088's time. Furthermore, it makes no difference whether the data structure is near or far, 7 bytes long or 240, the 'imul' is still required.

However, the Microsoft C compiler checks if the field length is a multiple of 2 and if so does a shift (taking 2 clock cycles) instead of an 'imul'. Hence wherever possible data structures have been constructed, to have a field length which is a multiple of 2 (e.g., the channel status array has a length of 32 bytes). Note that the compiler pack option must be used to pack the data structures otherwise all elements will be word aligned which could result in holes and a larger structure size. This is particularly important for the WD2511 memory look up tables.

6.2 X.25 packet level:

The X.25 packet handler is a complex but well defined protocol. The link level is fully implemented by the WD2511 processors, saving considerable development time. The packet level software requirement for the traffic generator is not too large. As the software is just sending and receiving packets, there is no need for an interface to the transport layer. In fact, after receiving a packet and updated the system status and statistics, the packet received module simply discards the received data field. Also, the software requirement was reduced by following the SAPONET X.25 specifications rather than the full CCITT X.25 recommendation for (e.g., only mod 8 packet sequencing is supported). Thus the software by no means implements a complete X.25 packet handler.

6.2.1 Packet received

The main function of the PACREC module is pac_received (see appendix D). The module is dispatched by the program 'systems' and gets the packet information from the work queue.

This data, entered by the link level on reception of the frame, provides the packet received module with the link and buffer number of the received frame. As the packet buffers are not a multiple of 2 in length (e.g., the data field may be 256 bytes, but an extra 3 bytes are required for the packet header), a pointer to the buffer is first computed.

Using this pointer, the packet header is read in and checked. If correct (ie. mod 8 sequencing, not a datagram and channel number within the configured range), a function is called to service the packet and control is passed back to 'systems'. Thus the rest of the PACREC module just consists of the service functions, listed in the same order as they appear in the X.25 recommendation. Note that the required information is passed to the functions when they are called.

As an example, the data_packet function is called to service a data packet. Firstly it checks that the channel is in the correct state (ie., data transfer state, flow control ready). If in the wrong state, the state error counter is incremented and appropriate action is taken (e.g., discard packet if in clear request state).

Having validated the channel state, the function calls the pr_seq_check and ps_req_check functions to check the receive and send sequence numbers. If correct, these functions update the channels status array. If there is a sequence error, then a function (chan-reset) is called to reset the logical channel. This resetting involves changing the channel state, resetting the channel sequence numbers and adding a reset request to the work queue (PACTRANS will then build and transmit the packet).

Finally, the packet level window is checked via a call to win_check. If required, this function will send a RR packet to the DCE in acknowledgment. Having completely validated the packet, the statistics counter is incremented and control returned to the modules main function. Note that in a loop back test (i.e., DTE to DTE) data is transferred in one direction only (e.g., from channel 1023 to channel 1) and hence an RR packet is returned after every two data packets received.

Thus a fair amount of processing is done on received packets. If a packet is received in the wrong state or an acknowledgment is required then an entry is made on the work queue for the packet transmit module. For resets and clears this is done by calling a function which adds the job to the work queue. As well as counting the types of packets received, statistics are also kept on the number of bad packets received and the cause of call clears.

In normal mode of operation the traffic generator is connected to a DCE, but for testing purposes it may be configured to "talk to itself". The following code, at the start of pac_received () allows for this:

```
        if ((* master [linkno]).block == TRUE)
           lcn = NO_CHANS - lcn + 1;
```

If in loop back mode, then an incoming packet with channel number 1023 will be processed as channel 1. Similarly, a packet with channel 1 will be processed as channel 1023.

Thus the PACTRANS module can transmit data packets on channel 1023 with PACREC effectively processing them as channel 1. If an RR packet needs to be returned, then the entry will be added to the work queue and transmitted by PACTRANS. PACREC will then effectively process the RR packet on channel 1023 and will update the sequence numbers accordingly. Channel 1023 is therefore effectively communicating with channel 1 of the traffic generator. This is possible as PACREC is completely independant of PACTRANS with both getting channel information from the chans status array. Finally, note that, although DTE to DTE communication is not defined in X.25, the packet type coding does allow it (e.g., an incoming call and a call request packet have the same header).

6.2.2 Packet transmit:

Once again, the module's main function pac-transmit is the only entry point. The function retrieves details from the work queue and unless a message was supplied by the traffgen module, it requests a free transmit buffer from the operating system. As there are typically 60 transmit buffers, this allows the transmit program to work ahead of the link level, thereby ensuring a very high throughput.

Having obtained a buffer, a function is called to build the packet. This is done by first calculating the buffer address and then filling in the required packet header and other information. The packet is then added to the link queue which adds the frame information and transmits it. If required, a packet level timer is started.

Note the use of a very large transmit buffer. Typically each link level has nearly a full 32K of buffer space which is divided into a number of fixed sized buffers. The buffer length is set to 259 bytes allowing the maximum SAPONET data field length of 256 bytes. The use of large buffers helps keep the software modules independent of each other and allows for high peak data rates. To increase the throughput and simplify control, separate transmit and receive buffers are used, although these could easily be amalgamated to form a single buffer pool.

6.3 Operating system module:

The module 'systems' contains the executive that calls the various other modules. In addition, it manages the buffers, system timer and packet level timers.

Following the example of the gateway project at the University of Strathclyde (Grant et al 1983), a simple executive is used. The program simply loops around calling various processes to be run, such as the link level module or the packet received module. Each process runs to completion before returning. Thus if there is a job in the work queue for the packet transmit module, the required packet is fully built before returning control to the operating system. This ensures that all processes get a fair share of CPU time and that processing is done sequentially. Furthermore it ensures minimal CPU overhead. Note that there are no real time requirements as the link level processors take care of that aspect.

'Systems' uses a simple first-in-first-out non-preemptive scheduler. Thus for example, if a link level processor is able to acknowledge a block of packets, these are simply added to the work queue for later processing. Obviously some mechanisms to manage the work queue and buffers are required, the functions being provided by 'systems'.

'Systems' allocates and frees buffers keeping a count of the remaining free buffers. If the X.25 card is getting behind in processing received packets, then the particular link is

declared congested. A flag in the WD2511 register is then set, resulting in an RNR frame being sent to the DCE. Similarly when the transmit buffers start getting congested, the variable 'tcongestd' is set true and the traffic generator module will temporarily stop passing test messages to the packet level.

As well as controlling the buffers, 'systems' also implements the system and packet level timer functions. The system timer calls the traffic_gen function at regular intervals so that calls can be set up and cleared at fixed times. Test packets are also sent at multiple intervals of the system clock.

Finally, 'systems' contains all the traffic generator initialization functions. These functions initialize the frame buffers and channel control block as well as locating the far data structures.

6.4 The traffic generator module:

The traffic-gen function is called every 10 milli seconds and calls three functions to implement the traffic generation. (Appendix D). The first function looks through all the active channels and decrements the send timer in the chans status array. If it is time to send a test data packet, then an appropriate function is called. A standard test data packet is used and is first copied into the buffer segment to be used. This is one of the reasons for having separate transmit and receive buffers - the test packet is copied into the buffers at initialization, so that only the first 20 bytes need be reloaded.

The other two modules decrement timers to see when a call on a channel should be cleared or a new one set up. Thus the user is able to configure the number of logical channels to use, the call duration and the call interval as well as the number of data packets sent.

The chans status array is used to store all the data about a particular logic channel e.g., channel state, packet sequence numbers, packet level timers and send timers. Each record is 32 bytes long, with the whole structure being 64K. This allows for an extremely large number of logical channels - up to the SAPONET limit of 1023 per link (channel 0 being reserved). Usually PAD's and X.25 equipment support in the region of 20 channels with the traffic generator specification requiring at least 6. Thus the traffic generator can easily handle a large number of users. Note that the extra processing required will to an extent limit the number of logical channels to one or two hundred, depending on the configuration (eg. the call duration) and what data rate is required.

6.5 Link level interface:

This module is used to interface the WD2511 processors to the other modules. It looks after the initialization and monitoring of the link levels.

When the WD2511 processors need attention, such as after a frame has been correctly received, they signal the host by means of an interrupt and store the cause in the second status register. Although the hardware provides an interrupt facility, it was not used and the devices were simply polled. Polling maintains the sequential processing of the system and simplifies debugging. As the interrupt is only used to indicate an event, the service time is not important (e.g., up to 7 frames can be received before the WD2511 sends an RNR frame).

Status information in the WD2511 memory look up tables inform the WD2511 whether the particular buffer is free or not. The 8088 checks to see if a frame has been received and if so the buffer is added to the work queue for later processing by the PACREC module. This is why, when viewing the systems display of the traffic generator, at least 8 receive buffers will be activated per link. These buffers have been allocated and are waiting in the WD2511 receive look up table for packets to be received. On the transmit side, a call is made to buff_free to free the buffer and details of the next buffer are filled into the look up tables. The WD2511 processors have their own transmit queues (send_queue).

6.6 Software conclusions:

This chapter has looked at the overall software design, with the actual listings being given in the appendixes. The software has been written in modular fashions consisting of the systems, traffic generator, packets level and link interface modules.
The data structures, listed in appendix C, are the key to the program operation. To summarize, the work queue enqueues jobs for the packet level modules; the buffer control structure is used by the executive which allocates and frees the transmit and receive buffers; the memory look up tables are used by the WD2511 processors and follow their standard definition; the channel control block is used by all modules and stores the state of each logical channel; and finally the statistics and control structures are used to communicate with the monitor program running on the PC.

For program operation, the 8088 processor performs a loop, calling the main functions of each of the modules as required. These functions in turn call other functions to do the processing. Thus the basic operation of a particular module can be relatively easily understood by reading through the module's main function.

The software meets the specification for the traffic generator and has been designed for high speed operation and the support of a large number of logical channels. Finally, the modular design approach and use of the C programming language should allow for easy software maintenance and upgrading.

PC PROGRAM

This chapter describes the IBM PC environment and the X25TG program which runs on the PC. The program monitors and controls the card, displays system statistics and allows the user to configure the traffic generator.

7.1 DOS Operation

Before looking at the X25TG program it is worthwhile to look very briefly at some of the aspects and features of the MS DOS operating system. Several articles have been written on the operating system, with one of the best being "An inside look at MS-DOS" (Paterson T. 1983) written by one of the original designers.

The first thing to note is that, like X.25, DOS has a layered structure :

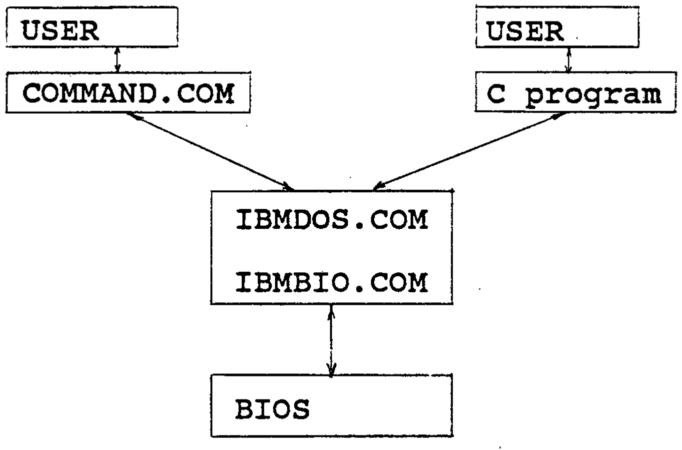


Fig 7.1 DOS structure

The BIOS is a collection of I/O routines permanently stored in the PC's ROM (read only memory). It therefore performs the actual disk access, keyboard fetch etc. The IBMDOS and IBMBIO serve to interface the 'command.com' to the BIOS. These two files are loaded from the DOS disk on system startup.

The actual program which the user interacts with is the command.com. This program is essentially hardware independent and is loaded from the DOS disk at system startup. As well as providing the common commands, such as DIR, it contains the basic error trapping mechanism and allows other programs to be loaded and run. A characteristic feature of COMMAND is that consists of two parts - the resident and the transient sections. The resident portion contains the essentials while the transient portion, located at the top of memory, may be overlaid with an application program or data. This is why, for example, DOS must be reloaded after doing a diskcopy on a PC which has only 256K of memory installed. Note that the transient portion is located in the X.25 cards memory, so it may be necessary to reload it after running the traffic generator. The system was in fact configured so as to avoid this.

7.2 Program loading:

To run the X.25 program, one has to do the usual system booting to get COMMAND installed. Running the X25TG program is then simply a matter of typing the program name but what about the programs for the X.25 card? The Microsoft C package does not contain a locator and COMMAND loads programs only in the next free memory segment.

To make matters worse, the traffic generator programs may need to be loaded into several X.25 cards each located at different addresses. An initial solution was therefore to write code in the X25TG module to do the required loading. This could be done by making use of the EXEC DOS function call (see DOS technical reference manual, version 3.0 or 3.1). To do this one basically sets up the various parameters in the registers and then issues an interrupt type 21H to invoke the DOS function call.

A simpler alternative is to load the X.25 card programs under control of the DEBUG utility. DEBUG is a standard DOS debugger and is documented in the DOS manual, together with EDLIN and LINK. Furthermore, DEBUG can also be used to load in the required assembler start up routines in the X.25 cards.

7.2.1 BAT files and redirection

The loading process is automated through the use of a short batch file called TG.BAT (see appendix F.2).

DOS allows three different program file extensions, COM and EXE for programs which are loaded and run and BAT for text files containing commands to be executed by the operating system. Thus the DOS commands required for program loading and starting are stored in the batch file and the user simply types 'TG' to start the traffic generator.

For a system with a single X.25 card installed, 'TG.BAT' consists of the following two statements:

```
debug   X.25prog.EXE  <  startup.txt  > null

X25TG   1    4    0
```

Note the use of the DOS redirection function, which is described in the DOS reference manual (IBM 1985). The reason for using this is that the batch file can only load the debug program - it cannot pass any commands to it. Thus these debug commands are stored in the startup file to instruct debug to perform the required loading by getting the program into memory and then copying it onto the installed X.25 cards.

Finally, when debug has finished, the X25TG program is loaded and started. Hardware configuration parameters are passed to X25TG to enable it to calculate the I/O and memory addresses of the cards and to start them running. From there onwards the program monitors and controls the cards and displays the statistics.

7.2.2 Program startup

The standard DOS utilities provide an easy to use, easy to modify method of loading programs onto the X.25 cards. However, there is still one further problem: the program itself is structured to run in a DOS environment rather than stand alone.

When control is passed by DOS to a C program, it is actually passed to a startup routine which then calls the main function. This startup routine basically sets up the environment and provides error trapping mechanisms. When called a program is allocated all available memory and thus, for example, one of the first things the startup routine does is to release all unused memory. Clearly for the traffic generator such facilities are not required, particularly as the program requires no I/O facilities.

Hence the program is simply started by calling the main function directly. The only alteration required is to select the compile option inhibiting the stack check routine (see Appendix J). A short assembler routine is loaded into the top of the card's memory to initialize all the 8088 registers before calling main, which is located at address 30002 in the card's memory map. Thus when the card is enabled, the 8088 jumps to address 3FFF0, executes the startup routine and then jumps to the traffic generator program.

This loading scheme is only possible because a small memory model is used so all the code, data and stack are present in a single 64K segment.

There are therefore no intersegment calls to worry about. The data structures which are located in different memory segments (WD tables, buffers and channels status array) are all referenced via far pointers which are initialized by the cards operating system. The card's startup routine loaded by the PC therefore only needs to initialize the four segment registers and jump to the start of code.

As well as the startup routines, a short interrupt service routine is also loaded. Currently, as interrupts are not used, this routine does a register dump and then halts. This routine is used primarily to assist in program debugging, but can easily be replaced by other routines at a later date.

Rather than having a separate assembly program module, the assembler statements for the startup and interrupt modules are just listed in the startup.txt file and are directly coded into memory by the 'debug' utility. This allows for very easy modifications with the only disadvantage being a second or so extra load time. It also means that all that is required for complete software maintenance is an IBM PC and the Microsoft C compiler package.

In conclusion, the user starts the traffic generator by typing TG. The batch file loads all the traffic generator programs and the assembler startup routines. The program for the PC is then loaded and control is passed to it. All hardware configuration parameters are stored in the TG.BAT file with notes in Appendix G describing how to select different configurations.

7.3 The X25TG program

Following the program loading method, the program which runs on the PC, 'X25TG', is now described. There is only one program module and this is listed in Appendix B. Note that this program is completely independent of the others - it runs in the MSDOS environment and is intended to interact with the user rather than implement X.25.

When invoked, the hardware configuration parameters are passed to X25TG in the form of a series of digits. The first digit is set to 1 if the traffic generator is being used in a loop back mode ie., DTE to DTE communication. Then follows a list of address block numbers for the installed cards and a list of I/O bank numbers. The program automatically calculates how many X.25 cards are installed (eg. X25TG 1 4 0 means that there is one X.25 card in loopback test mode with control structures in the PC's fifth 64K memory block and the cards I/O select jumpers set to the first location - refer to Appendix G).

Thus the first function called by the program is the one needed to compute the addresses of the shared memory. More precisely it calculates the address of the statistics and control structure values in the X.25 cards, assigning the addresses to an array of pointers. These two structures enable the intercommunication between X25TG (under 'MSDOS') and the programs running in the X.25 card (under 'systems').

The actual I/O addresses of the cards are also calculated and the X.25 cards are then started. Finally, if an invalid hardware configuration was passed (e.g., told that card is in the PC ROM area), then the program aborts with an error message.

7.3.1 Statistics display

After X25TG initialization the program executes a loop displaying information on the console (see Appendix B). There are a total of four different displays, each showing various aspects of the traffic generator operation. The displays offered are listed below with the user being able to select a different display at any given time.

Note that the statistics on each display are continuously updated throughout the operation of the traffic generator.

1. Main display — for overall system totals
2. Systems display — to monitor the actual operation of the cards
3. Packet level display — for packet level statistics
4. Link Level display — for link level statistics

In addition to the above, the user may also select a configuration display which may be used to alter traffic generator parameters (eg. number of logical channels to use) at run time. Hence the user also has full control of the operation of the traffic generator.

7.3.2 Statistics updating

Two problems were encountered in updating the statistics. Firstly, one has to ensure that the X.25 card is not updating the statistics at exactly the same time as they are being read and secondly one needs to update the current display as opposed to scrolling up a new one.
To solve the first problem a software semaphore was used. The program sets a flag to signal the particular X.25 card. When the card acknowledges, the program reads in the required statistics and then releases the card. This system is also used when changing the card control parameters.

The second problem was a little more tricky as the Microsoft C package has no functions to manipulate the screen. Thus while one can easily write to the display, there is no provision for clearing it or positioning the cursor. One possible solution was to make direct calls to the display part of the BIOS, but this gets rather complicated and reduces program portability (IBM has copywrite of the BIOS routines and therefore entry points may be different on compatible computers).

The DOS technical reference manual was therefore consulted with chapter 2 describing the use of the "extended screen and keyboard" control. Basically this involves installing a device driver called ANSI.SYS. This driver, which comes standard with DOS, provides a whole set of cursor control sequences and provides a very neat and easy method of controlling the cursor.

Thus the actual text of the screen remains static with only the figures being updated. If the user selects a new display, then the screen is first cleared before being written to. Note that ANSI.SYS must be installed by including the command "device =ANSI.SYS" in the DOS configuration file CONFIG.SYS (see Appendix G).

The statistics screens are updated once every two seconds, with a C library call being made to get in the time. As the time of the last display updating needs to be remembered, a static variable is declared in the function. This is a useful feature of C. One can declare a static variable which is completely private to the particular function. Furthermore the variable may also be initialized. Another application of this is the main display function which must remember the starting time in order to compute the average data rate. The following code does this with the initialization only being done on the first call to the display function:

```
static  long   start_time =0;
if (start_time ==0)
      start_time = time(NULL);
```

The keyboard buffer is regularly checked to see if the user has requested a new display. Note that the buffer is being polled and not the keyboard so it does not make the slightest difference if, for example, the user hits a key just as the display is being updated. DOS will also process key sequences such as control break. Thus, as with the display functions everything is built on top of DOS and there are no assembler routines. This makes the traffic generator program portable and so it may be run on any IBM PC compatible computer.

7.3.3 Conclusions

The X25TG program was written in a modular fashion with the listing being divided into the following sections:

main function
setup functions
display functions

The program provides the user with four different statistics screens displaying the activity of the traffic generator as well as a configuration screen.

SYSTEM TESTING

The first unit of software to be tested was the X25TG program which runs on the PC. This involved getting the screen layout correct, ensuring that the screen selection method worked and checking that the statistics were updated properly. Print statements were then inserted to check that the program was computing the correct card's memory addresses and could control the cards properly. Next the configuration screen was debugged and tested.

The X25TG program is menu driven with the user being able to select what screen he wishes to view. To quit, the X or ESC may be hit and the user is returned to DOS. The main point about this testing is that it is relatively straightforward. The problem arises when trying to debug the X.25 programs running on the X.25 card.

8.1 Testing on the PC

Even though the X.25 card is installed in the PC, it is still a completely separate entity and thus a development system with an 8088 emulator would be required to debug the program. This is where one hardware feature of the X.25 card was particularly useful - one of the WD2511 processors I/O control registers are dual ported with the PC. Thus, for initial testing, all programs can be run directly on the PC with only the following restrictions:

1. One link instead of two
2. Clock from PC timer rather than MUART
3. Lower data rate as only a single 8088 processor is active.

This implies that all the PC debugging tools were available, with the high level language symbolic debugger 'CODEVIEW' being the most effective. This menu driven program, designed for use with the Microsoft C package, allows one to perform all the debugging directly on the C statements. For example one can load in a program module and set a breakpoint at a particular C statement. When the program is run and has stopped at the breakpoint, one can single step through the C statements monitoring program execution. Alternatively, one can step through the assembler code corresponding to the C statement viewing the processor registers.

Setting a breakpoint in the PACREC module one could for example monitor exactly what packets were being received and on what logical channels. The first few packets would always be the call requests on channels 1, 2, 3 etc. Then these would be intermingled with call accept packets on channels 1023, 1022 etc. A little later the large data packets would start coming through. Similarly on the transmit side, one could trace through exactly what packets were being built.

An example of one problem encountered was that, by observing the traffic generator packet level display, a large number of reset packets were being received. Using Codeview to monitor the PACREC module showed that indeed this was the case, the cause being traced to packets with invalid sequence numbers. Further tracing revealed the problem to be in the RR function of the PACTRANS module - an '&' had been used in place of a | when constructing the receive sequence numbers.

Thus the cards hardware architecture of multi port memory and the dual ported I/O registers of the WD2511 processor allow the PC to be used as the complete software development system. The use of the latest debugging package for the PC greatly assisted in initial software testing.

8.1.2 Testing configuration

When running programs on the PC's 8088 processor instead of the cards, a few changes are required specifically :

1. Data structure address constants are different (ADDRESS.CON, listed in appendix F)
2. The PC program must call the systems program and all initialization must be done in the PC program (X25TG.C)
3. The main function of the systems program needs altering as does the timer function (clock from PC rather than the MUART) (SYSTEMS.C)

The above are the only three modules requiring changes.

The standard method of configuring is to have different functions which are called depending on the configuration. However, this leads to problems as only one main function is allowed per C program.

The solution to this was to use the # if preprocessor directive

eg.
```
# if TESTING == TRUE
        X.25_card()
# else
        main()
# endif
```

This code causes the function to be given a different name at compile time, while the actual function contents remain the same. All that is therefore required to run the program on the PC is to set the TESTING constants to TRUE (in SYSTEM.CON), recompile the X25TG and SYSTEMS modules and relink the program.

```
B>
                         +-------------------------+
                         | X.25 TRAFFIC GENERATOR  |
                         +-------------------------+

 SYSTEM DISPLAY:
 ==============                 Link 0          Link 1
     BUFFER ACTIVITY:
 System congestion flag   :      0               0
 Active receiver buffers  :      9               0
 Active transmit buffers  :      10              0

     REC. ERROR COUNTERS:
 Number of state errors   :      0               0
 Number of seq. errors    :      0               0
 Number of rec. errors    :      0               0

     PROGRAM STATUS:
 Test data packet length  :      118 bytes
 Time traff. gen. active  :      1 mins 18 secs.
 Prog. flag (buff_find)   :      17339           0

 Display options:  S: system,  P: packet level,  L: link level
                   C: configuration,  X: exit,   other: main  ...>s
```

Fig 8.1 A typical traffic generator display

8.2 System memory maps

The memory maps in fig 8.2 show the configuration for one X.25 card installed in a PC with 256K on the mother board (the PC dip switches would be set to 512K memory total). This is the configuration for normal program running ie. programs being run on the X.25 cards processor.

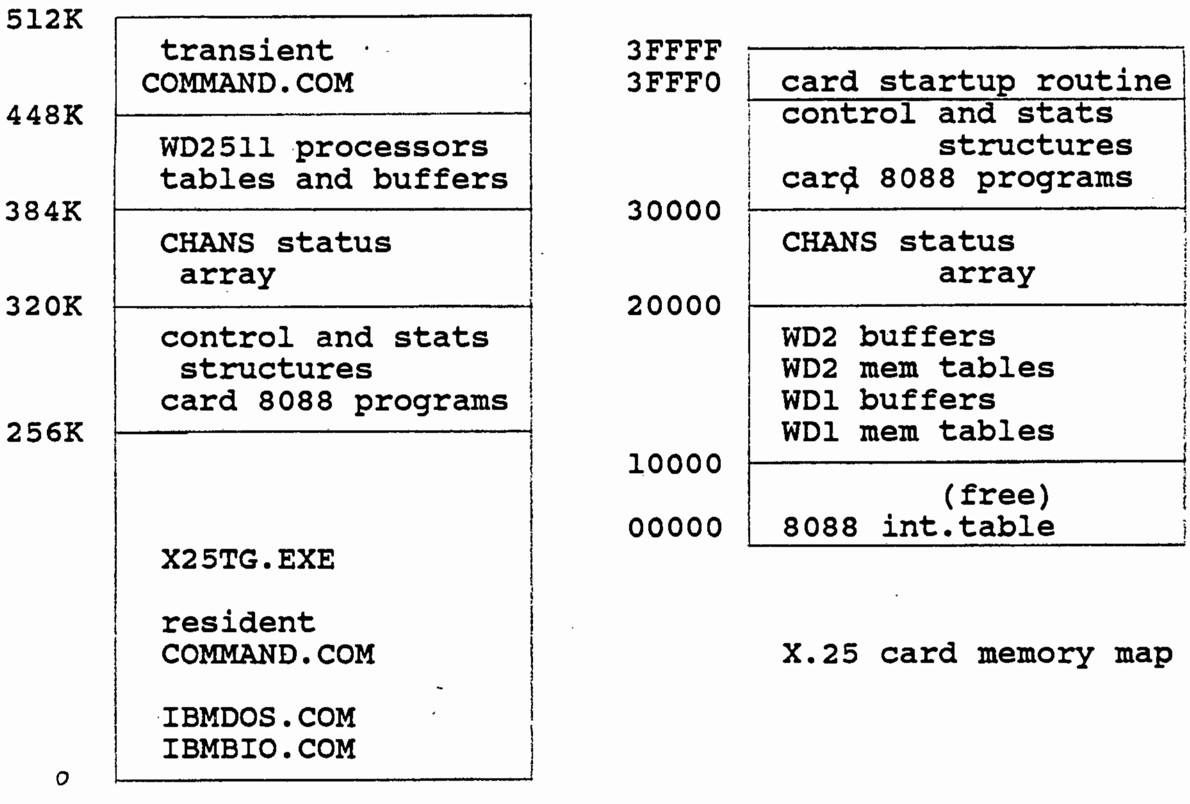


Fig 8.2 System memory maps

Note that the memory map is not altered if the programs are run on the PC's processor instead of the card's 8088 CPU - only the address constants are changed. The actual addresses used are given in the ADDRESS.CON file (see Appendix F).

The hardware configuration chosen was to have the X.25 card memory blocks (64K) appear the opposite way round in the PC's memory map (J11 and J12 on), eg. to the cards 8088 processor the code appears in the top 64K block (30000 to 3FFFF) while in the PC's memory map it is the lowest dual ported block (refer to appendix F). Other configurations are possible but this one allows the card's programs and startup routines to be loaded lower down in the PC's memory map.

In the initial tests, it was found that 32K of buffer space per link was more than ample - allowing around 60 transmit and 60 receive buffers per link (259 bytes per buffer). Usually there were only around 12 active receive buffers (8 in the WD2511 receive look up table) and 4 transmit buffers when operating at 64Kbps. The only time the buffer usage did increase a lot was when the frequency of the transmit clock was reduced by adjusting the signal generator. This meant that the link could not transmit fast enough and so the transmit buffers started queueing up until the congestion limit of 40 was reached at which the traffic generator stopped sending test data packets. (Note that the number of active buffers is displayed on the systems display of the traffic generator. Also note that a fairly long call length is required as buffers are flushed when a call is cleared).

Thus jumper J3 was closed so that the second WD2511 processor shares a 64K block with the first processor. This meant that, in the PC's memory map, the top 64K is free and so the transient part of COMMAND need not be reloaded after the traffic generator program is run.

8.3 Test results

The system was set up for loop back testing on one link with the RS232 cable being connected to a "breakout box" to monitor the line activity. The loop back connections were set in the breakout box with a signal generator providing the timing. The system was tested in this configuration, making sure that the statistics displays were showing roughly what was expected. The debugger was also used to examine the packets received in order to double check the statistics.

The following system configuration was used:

Number of logical channels : 10
Number of packets per call : 80
Call length : 10 seconds
Interval between calls on same channel 2 seconds

Together with a test packet length of about 120 bytes, this should give a data rate of approximately 64Kbps. Observing the main traffic generator display, one would see the number of calls jump up by 10, the number of data packets go up, followed by 10 more clears. At this stage the number of active channels would go down (depending on when the display was updated) as all channels become deactivated for a period of about a second. After this the next batch of calls is sent and the cycle is repeated.

It was also observed that RR packets were being transferred to acknowledge the received data packets. As the packet level window size was set to 2, there is approximately one RR packet for every two data packets. Note that there are actually slightly fewer RR packets as not all data packets will have resulted in an RR packet when the call is cleared (eg. if the logical channel has received one data packet since the last RR was sent). This operation is correct as the RR packets are used for flow control rather than error recovery.

To check the error control mechanism in the link layer a simple line corrupter circuit was built and inserted in the line. The result of this test was to verify that the link level FCS error correction and retransmission did not affect the packet level statistics. On the link display the number of frames received with a bad FCS was seen to increase as the frequency of the pulse generator (used to gate out the RD line) was increased.

```
B>                         X.25 TRAFFIC GENERATOR

 MAIN DISPLAY:
 ============
     SYSTEM TOTALS:
 Number of active links   :   1
 Number active channels   :   10
 Number of data packets   :   1971

     PACKET TOTALS:
 Total calls set up       :   40
 Total DTE call clears    :   30
 Number of RR packets     :   965

     LINK TOTALS:
 Total corrupted frames   :   0
 Total link timeouts      :   0
 Total data rate (bps.)   :   79375

 Display options:  S: system,  P: packet level,  L: link level
                   C: configuration,  X: exit,   other: main  ...>
```

Fig 8.3 The traffic generator main display

The traffic generator was also connected to a commercial PAD (configured as a DCE) to verify the operation of the packet handler. Thus the traffic generator would send a call request to the DCE, receiving a call connected packet to enter the data transfer phase. The test packets were then transmitted to a port of the PAD. Note that the PAD initializes the packet level by sending a restart packet to the traffic generator, this being reported on the packet level statistics display.

Also important is the system configuration. For example the PAD used supported 16 asynch ports, with the port number corresponding to the X.25 logical channel number. Thus it is necessary to configure the traffic generator to set up a call on say channel 8 rather than 1023 (done by changing the NO_CHANS constant in the SYSTEM.CON file). The called DTE address of the traffic generator (in PACKET.CON) must also be set to the configured address of the PAD. Note that the port sub-address should also be included in the called address field and the PAD configured accordingly.

8.3.1 Speed tests

As the primary requirement of the traffic generator is to support a number of high speed (64Kbps) links, data throughput tests are important. As the program runs slower when running in the debugging environment, the program was relinked and executed as a standard program under DOS. The result was a maximum throughput of only 50Kbps on a single link, which is far from adequate.

The reason for this slow speed was due to the fact that the PC's 8088 processor was used to run the display program as well as the traffic generator programs. It was found that the display updating in particular took up a large amount of the CPU time, and one could actually clearly see a flicker in the transmit and receive LED's on the breakout box when the display was being updated. This was due to the link level processor sending a continuous stream of flags.

The main reason for the long display updating is that it is all being done on a high level. For example, the X.25 program calls a function to output a string of characters to the display. This function (eg. cprintf) processess the data and in turn makes interrupt calls to IBMDOS which finally calls the BIOS routines which do the actual display updating. In addition, the ANSI.SYS device driver intercepts the calls to IBMDOS to provide the cursor positioning and screen clearing functions (again actually executed by the BIOS routines - refer to BIOS routine listing in the hardware reference manual). While the above is completely transparent to the actual program, it does use up a lot of CPU time.

As well as display updating, the PC's 8088 processor also has to do such housekeeping chores as updating the system clock and servicing the keyboard, which again take up time. Thus the X.25 card's 8088 processor is vital to the success of the traffic generator project. Accordingly the TESTING flag was set FALSE and the program was recompiled to run on the X.25 card.

8.3.2 Final speed tests

In addition, two other alterations were done to improve execution speed. Firstly, some of the 'int' declarations in functions were replaced with 'register int' declarations. This results in the variable being stored in either the SI or DI registers of the 8088 rather than in memory. Thus instead of taking two memory cycles to access the variable, the CPU can use it immediately. The register variables are particularly useful for variables which are used often in a function, such as linkno. The program assembler listings were also checked to ensure that most of the 'imul' instruction had been replaced with shifts. Finally, the code was compiled using the optimisation feature of the compiler.

This time the traffic generator was configured to support 20 logical channels rather than the 10 used previously. This results in more packets than the card is capable of sending and thus one may monitor the buffer activity. The signal generator frequency, used for the line clock, was set to 100 KHz.

From the main display, the total data rate was around 180Kbps or 75Kbps per link. Reducing the signal generator frequency to 80KHz caused no change, while reducing it further resulted in the number of active transmit buffers increasing and the data rate decreasing. Tests were also done using larger numbers of logical channels with similar results. Thus the traffic generator meets the speed requirements with a safe margin.

8.4 Traffic generator design validation

Having designed the traffic generator, one can look back and see whether the correct design decisions were made. In particular, was it worthwhile to put an 8088 processor on the X.25 cards or would it have been easier just to use the PC to control the link level processors? Also, could the card have been designed with just say 16K of static memory in place of the 256K of multi-port memory? These questions are important as a great deal of the project time was spent on the hardware design and optimising the circuit for high speed operation and flexibility. Was this work actually necessary?

From the test results, it is clear that, had the PC's CPU done all the controlling, it would never have been able to support more than two links at 64Kbps. Even to achieve such data rate extensive software modifications to control the keyboard and display would have been required, reducing program portability. Thus the 8088 processor on the card allows the traffic generator to meet the required throughput and allows all software to be written on an "upper level", making full use of available library functions.

Similarly the large amount of multi-port memory allowed the traffic generator to support the large number of logical channels and have large buffer area. Other hardware features also proved very useful, particularly the ability to download all software and to run the programs on the PC for testing purposes.

CONCLUSIONS

The objective of this thesis was to design a high speed traffic generator capable of supporting from four to eight 64 Kbps lines. The design approach was to use an IBM PC as the base of the system with several high performance X.25 cards being installed in the PC's expansion slots. Five cards may be installed in a PC (3 in older models) or compatible computer for a total of ten X.25 lines.

Use was made of the WD2511 link level processors to implement the link level of X.25, while the remaining software was written in the modern C language. There are four statistics displays displaying a super set of the required statistics as well as a configuration screen which may be used to change test parameters at run time. The software supports several hundred logical channels per X.25 card, although only six were required in the original specification.

The traffic generator designed meets and exceeds the original project requirements. In addition, the use of a good high level language and a modular software structure allows for easy software upgrading. All programs are directly loaded from the PC's disks onto the X.25 cards so the software development can be done on the PC without requiring additional facilities. Also, the ability to bypass the cards 8088 processor and directly control one of the card's link level processors from the PC allows one to use modern debugging tools such as the codeview program.

As well as being easily enhanced, the programs are also portable. All I/O is performed through the DOS operating system while the use of far pointers allows for the access of data structures located at specific memory addresses. On the hardware side, the use of a printed circuit board ensures card reliability and easy manufacture. The multiple processor architecture and multi-port memory allows a variety of X.25 communications applications to be supported.

9.1 Enhancing the traffic generator

At the time of write up, the following enhancements were being implemented:

1. Extending the traffic generator configuration screen to allow the packet level window size, called DTE address field and the test data packet length to be selectable at run time.
2. Increasing the throughput by streamlining the C source code for the minimum execution time.
3. Writing the memory initialization function, which would mean that only 64K of the card's memory need be shared with the PC.
4. Further testing of the packet level software, particularly with regard to how it copes with packet level procedural errors.

Other improvements could also be made, but the biggest enhancement would probably be to combine the X.25 traffic generator with a commercial link analyser package.

Rather than supply a complete computer as part of the test equipment, some manufacturers supply a small communications card and a software package which runs on the PC. This reduces the cost of link analyser as the user is buying the software package without having to purchase a dedicated computer system as well. Examples of such products are the Datahawk line monitor from Renex and the Feline package, locally distributed by Duxbury Transmission Equipment.

A feature of the X.25 traffic generator design is that the X.25 cards run completely independent of the PC - they have their own processors, memory and system clocks. Thus once the traffic generator is started, another program can be run on the PC. Furthermore, if the other program is also "well behaved" (ie. loads properly under DOS) it is possible to have both the traffic generator display program and the link analyser program, loaded and run under a windowing environment, such as the Microsoft windows package. The user could then switch between running the traffic generator display and the link analyser programs.
The only alteration required for the traffic generator to run in the above environment would be to set the hardware configuration so that the X.25 cards reside in unused memory areas out of the DOS 640K range in order to allow the commercial package adequate memory. A typical setup could then be to have say three X.25 cards installed (providing six 64 Kbps links) while the commercial link analyser package monitors one of the links. Thus one would effectively be combining the raw data throughput and processing power of the traffic generators X.25 cards with the software expertise of a commercial package.

9.2 Other applications of the X.25 card:

Although primarily designed for the traffic generator application, the X.25 card is flexible and can be configured for a variety of networking functions. A few possibilities are suggested below:

9.2.1 Bringing X.25 to the PC

Perhaps one of the biggest problems with SAPONET is the relative difficulty in connecting to the network. The user must buy an expensive PAD to implement the X.25 and associated protocols. Alternatively, PAD facilities at the Post Office exchanges may be hired, but this means that the end to end communications path is once again susceptible to errors. Thus there is a need for a low cost PAD and the X.25 card can be used as a PAD for the PC. A user could then connect up to SAPONET via an X.25 to X.25 multiplexer (to reduce the port charges).

The card would be configured with a single link level processor and RS232 interface. Thus all the hardware for the project is complete and the software structure and basic packet level is provided. Additional software would need to be written to implement X.29 and upgrade the packet level. This software could be developed and tested on the PC using the C language and making use of the Microsoft C library functions.

As well as connecting a single PC to SAPONET, other cards could be installed in the PC to make it into a more general purpose PAD. Although the software requirement would be very much larger, this configuration could effectively use the processing power of the X.25 card to turn the PC into a high performance communication subsystem.

On the other side of the coin, the 8088 processor could be removed from the card and the circuit slightly altered to achieve the lowest possible cost. With a realistic software requirement, this would provide a low cost solution to linking remote PC workstations to central hosts.

9.2.2 Local area networking

Work is currently being done on designing a local area network for the department, based on the token passing protocol. This protocol provides good performance at a modest cost and, as for X.25, Western Digital produce a special VLSI processor chip to implement the link level of the network (Stieglitz 1982). This chip is used in the proposed departmental network.

The token passing processor has virtually identical pinouts to the WD2511 and may be installed on the X.25 card. For low cost, the 8088 processor could be omitted leaving the card with just the token passing processor and 64K of memory installed. The lines drivers would be omitted and a special cable interface would need to be plugged onto connector 3 on the X.25 card. A network node address input would also be required.

Thus the X.25 card is easily modified to become a LAN card suitable for connection to the proposed network. Upper level software would be run on the PC's processor, while the 64K of dual ported buffer memory would ensure adequate system performance. On the software side, the latest version of DOS (DOS 3.1) supports networking (Data Comms. Nov 1985) while the token passing processor implements the lower software level. However all the middle layers of software would still be required, perhaps based on current commercial programs.

The card could also be used as a network file server, with the 8088 and 256K of memory being installed to improve performance. As data can be DMA'ed directly to the cards buffer memory, the host PC would still be able to do other tasks. With 30 mega byte hard disks now common place, a PC makes an excellent low cost file server. Finally, the Microsoft C compiler provides a complete library of functions for file locking, releasing etc. and so eases the software requirement.

9.2.3 As an X.25 gateway

A much more ambitious project from the software point of view would be to link the LAN to SAPONET. Here one could either have a single X.25 card with both an X.25 and token passing processor installed, or alternatively two X.25 cards could be used. While the hardware is complete, the software requirement for such a project would be large.

9.3 Concluding remarks

The topic of this research project was the design of a high speed X.25 traffic generator. The project requirements are detailed at the start of chapter 2. The main requirement was a very high data throughput and work was accordingly focused on the design of a high performance X.25 card.

Each card supports two high speed X.25 lines, using two WD2511 processors for the physical levels of X.25, while an 8088 subsystem implements packet level functions and overall control. This multiple processor design provides adequate processing power and accordingly the speed requirements of the traffic generator were met. Up to five X.25 cards may be installed in an IBM PC or compatible computer for a total of 10 X.25 lines.

All programs were written in the C language with a modular design allowing for software upgrading. With four statistics screens and a configuration screen, the monitoring requirements of the traffic generator were met. In addition the traffic generator can support a substantially larger number of X.25 logical channels than were required.

Thus the X.25 traffic generator design described in this thesis meets all the original project goals. In addition the design is flexible in nature, providing scope for future development.

LIST OF REFERENCES

1. Aspin S. Triple-X PAD 1983
   Undergraduate thesis, University of Cape Town
2. Braden R. and Cole R. Some problems in the Inter-Connection of Computer Networks. University College of London (Proceeding of 6th Int. Conf. on Comp. Comms. 1982)
3. Bradlow H.S. and Verkoost R.F. A comparison of communication techniques. (The X.25 packet network protocol seminar. Data communications - Sponsored by Barlow Data, February 1983)
4. CAMTEC JNT-PAD and JNT-MINIPAD product publications
   Camtec Electronics, Melton Street, Leister LE1 3NA, Britain
5. CCITT Recommendations X.25 1980
   (Union Internationale des Telecommunications, Place des Nations, CH-1211 Geneva, 20, Switzerland)
6. Datapro Corp. CCITT X.25 Packet Switching Interface
   Datapro Research Corp., Delron, NJ08075, U.S.A. April 1983
7. Digilog Product Information on Digilog Protocol Analysers
   (Digilog Inc., 1370 Welsh Road, Mongomeryville, PA 18936 U.S.A.)

Note: Protocol analyzer product information also obtain from:
Digitech Industries (Encore 200 portable analyser)
Dynatech Data Systems (Simon 5 protocol analyser)
Telsaf Data (Comstate I and II protocol analysers)
Tekelec - see reference

8. Drukarch C.Z. et al X.25: The Universal Packet Network

9. Emerson S.L. Implementing X.25
Mini -Micro Systems, September 1981

10. Erickson I. "Protocol controller chip manages X.25 Interface" Computer Design September 1985 P 78-81

11. Faro A. et al A Multimicrocomputer - based structure for Computer Networking April 1985 IEEE Micro

12. Grant A. et al Implementation of a Local Area Network X.25 Gateway Local Networks: Strategy and Systems (Online Publications, 1983) P 149-162

13. Grant A. et al A Gateway for linking Local Area Networks and X.25 Networks ACM Communications Architecture and protocol Vol 13 No 2 March 1983

14. Green J.H. "Microcomputer programs for data network design" Data communications, April 1986 P 116-145

15. Green P.E. Computer Network Architectures and protocols
Plenum press 1982

16. Higgonson P. and Cole R. Issues in Interconnecting Local and Wide Area Networks
Department of Computer Science, University College, London (Presented at Business Telecon conference, London 1983)

17. Irvine J.M. Delta modulation techniques for digital speech encoding. MSC. thesis University of Cape Town 1985

18. Karp P.M. and D.F. The international standard X.25 interface protocol for packet networks, and related network protocols. (Presented at a CSIR conference in Pretoria, February 1981).

19. Knott-Craig A.D.C. SAPONET, an overview April 1982
A SAPONET publication.

20. Kuo F.F. Protocols and techniques for data communication networks Prentice Hall 1981

21. Ledger G.L. "LSI ready to make a mark on packet switching networks" Electronics, Dec 20 1979 P172-177

22. Lowndes R. The JNT-PAD terminal concentrator and interworking standards. Interfaces in computing 1983 P 275-280

23. Micom PAD product publication, Micom System, 1983
(Details of Gandalf X.25 equipment also obtained).

24. Mier E. "How AT&T plans to conquer voice-data intergration" (devolopments in voice packet switching)
Data Communications, July 1986 P 51-56

25. OSI Data transfer - A Tutorial on the Open Systems Interconnection reference Model June 1982
(Omnicom, 400 Hooloway Ct, Vienna VA 22180 USA)

26. Patel A. Higher Level Support in X.25
(Seminar on X.25 for Computer Society of S.A.)
Dept. of Computer Science U.C.T. October 1982.

27. Roehr W.C. "Inside SS No. 7: A detailed look at ISDN's signalling system plan"
Data Communications, October 1985 P 120-128

28. Rosner R.D. Packet Switching; Tomorrows communication today Wadsworth 1982 (ISBN 0534979653)

29. SAPONET X.25 Packet Switching 2nd edition Oct. 1982
SAPONET Technical Publication No. 1
(Obtainable from the Director,
Digital Services,
Private Bag X74, Pretoria 0001).

30. SAPONET The Triple-X PAD 1st edition Feb. 1983
SAPONET Technical publication No. 4.

31. Schmidt W. "Can the world agree on the next digital-speech coding algorithm" Data Communications July 1986

32. Sherif M. et al X.25 Conformance Testing - a tutorial
IEEE Communications Jan 1986 Vol 24 no. 1 P16-27

33. Sloman M.S. X.25 Explained Standards and protocols - Computer communications (Vol 1 No. 6 Dec. 1978 IPC Business Press)

34. Stieglity M. "Local Networks: Token passing cashes in with controller chip" Electronic Design, Oct. 14 1982
P 189-196

35. Tannenbaum A.S. Computer Networks 1981
Prentice Hall Software series (ISBN 0-13-164699-0)

36. Tekelec Product information on the Chameleon range of protocol analyzer test equipment Nov. 1984
(Tekelec Incorporated, 2932 Wilshire Blvd., Santa Monica, CA 90403, U.S.A.)

37. TITN Twice communications software product pamphlets
500 Airport Boulevard, Suite 138, Burlingame, CA 94010, U.S.A. (Also obtained was System Strategies software product pamphlets)

38. Verkroost R. & Hattingh C. A case study of a protocol system implementation April 1984 (Protocol specification/ implementation seminar, University of Pretoria 1984).

39. Western Digital WD2511A network certification ad. Data Communications, June 1986 P 67

40. Wood E. Data Communications in South Africa (Paper presented at SACAC meeting, June 1984). printed in Pulse Nov. 1984 P 6-13.

Hardware reference manuals and related topics

(By author or company)

1. Belius R. "Practical Dynamic - Memory System Design"
   BYTE Publications, Dec. 1982 P 372- 385
2. CCITT Recommendation V.35
   Also, ISO standard 2 593 for the pin allocation.
3. Ciarcia S. "Build the Circuit Cellar MPX-16 Computer System"
   BYTE Publications, Nov. 1982 P 78-114
4. Drumm M.J. et al "Dual-port static RAM's can remedy contention problems" Computer Design August 1984 P 145-151
5. IBM IBM PC Hardware Technical Reference Manual
6. Intel iAPX 88 Book 1981
   (Order No. 210200, Local agent: E.B.E.)
7. Intel iAPX 86, 88 User's Manual (includes AP67) July 81
   (Order No. 210201 - 001)
8. Intel Memory Components Handbook 1983
   (AP 97 and other application notes).
9. Intel Microsystem Components Handbook (for MUART data) 1984
10. Intel 88/45 Advanced data communications processor board hardware reference manual 1983 (Order No. 143824-002)
11. Kavunu S. "Designing a modular computer the IBM PC"
    BYTE Publications June 1983 P 194-204
12. Martin K. "Metastability haunts VME bus and Multibus II system designers" Computer Design August 1985 P 29-32
13. Monolithic Memories PAL Handbook 3rd edition 1983
14. Motorola Schottky TTL data book 1984/85
    (Book includes data on FAST logic series).
15. National Semiconductor Corp. Interface Databook 1983

16. Prince B. and Due-Gundersen G. Semiconductor Memories John Wiley & Sons, 1983 (ISBN 0-471-90146-6)

17. Riedel N.K. A signal Processing Implementation for an IBM-PC based Workstation. IEEE Micro, October 1985 P 52-66

18. Western Digital WD2511/WD2840 Technical Package April 1984 (Note: specs. in the older 1983 Network Products handbook are out of date)

19. Western Digital Technical memo no. 192 on the WD2511A February 1985 (WD2511, used in thesis, replaced the WD2511A).

20. White G.P. "Bus structure eases multiprocessor integration" Computer Design June 15 1984 P 129-135

21. Witten I.H. "Welcome to the standards jungle (an in-depth look at connection standards) BYTE Publications, Feb, 1983 P 146-179

C Language and related topics

(By author)

1. Brown D.L. From Pascal to C Wadsworth Publishing Co.1985,

2. Field T. "A peek into the PC" Byte, March 1983

3. Hogan T. The C Programmers handbook
   Brady Communications Company, 1984 (ISBN 0-89303-3065-0)

4. Hunter B.H. Understanding C
   Sybex computer books, 1984 (ISBN 0-89588-123-3)

5. Hurwicz M. MS-DOS 3.1 spells easier PC networking
   Data Communications November 1985 P 223-237

6. IBM DOS 3.1 reference manual 1985

7. IBM DOS 3.0 Technical Reference Manual 1984

8. Jackson D.L. Cowan J. The Proposed IEEE 855 Microprocessor
   Operating Systems Interface Standard.
   August 1984 IEEE Micro.

9. Kernighan B.W. Ritchie D.M. The C programming language.
   Prentice Hall Software Series 1978 (ISBN 0-13-110163-3)

10. Microsoft C Compiler Version 3.0

11. Microsoft Macro Assembler Version 4.0

12. Paterson T. "An Inside Look at MS-DOS"
    BYTE Publications, June 1983 P 230-252

13. Pundum J. et al C Programmer's Library
    Que, 1984 (ISBN 0-88022-048-1)

14. Roskos J.E. "Writing Device Drivers for MS-DOS 2.0"
    BYTE Publications, February 1984 P 370-380

15. Thomas R. & Yates J. A user guide to the UNIX system
    Osborne Mr Graw Hill, 1985 (ISBN 0-88134-109-6)

HARDWARE APPENDIX

A.1 PAL equations

The equations for the five PAL devices are listed in numeric order. All equations are in the standard PALASM format (see PAL handbook).

PAL1

(IC4, circuit diagram 1)

```
PAL16L8                                          PAL DESIGN SPECIFICATION
PAL1                                             S. J. ASPIN   10/05/1986
BUS CONTROL PAL
X.25 COMMS. CARD  (U.C.T.)
T22    T6    T4    T8    MS1   MS2   T2    MS0    RDRV    GND
88END /88OE PCREQ /MOE /WDOE /END /PCOE /PCDOE /WDEND  VCC

IF(VCC) 88OE  = T2*MS2*MS1*/MS0*/88END*/END*/RDRV         ;88MREQ buff enable
              + T2*T4*MS2*/MS1*/MS0*/88END*/END*/RDRV     ;88IOREQ

IF(VCC) PCOE  = T2*/MS2*/MS1*/MS0*/T8*/END*/RDRV          ;REFREQ buff enable
              + T2*/MS2*/MS1*MS0*PCREQ*/T8*/END*/RDRV     ;PCMREQ
              + T2*T4*/MS2*MS1*/MS0*PCREQ*/END*/RDRV      ;PCIOREQ

IF(VCC) PCDOE = T2*T6*/MS2*/MS1*/MS0*/T8*/END*/RDRV       ;Reset REFREQ line
              + T2*T4*/MS2*/MS1*MS0*PCREQ*/END*/RDRV      ;PCMREQ data buff
              + T2*T4*/MS2*MS1*/MS0*PCREQ*/END*/RDRV      ;PCIOREQ data buff

IF(VCC) MOE   = T2*T4*/MS2*/MS1*MS0*PCREQ*/END*/RDRV      ;PCMREQ mem buff
              + T2*T4*MS2*MS1*/MS0*/88END*/END*/RDRV      ;88MREQ
              + T2*T8*/MS2*MS1*MS0*/T22*/END*/RDRV        ;WD1REQ
              + T2*T8*MS2*/MS1*MS0*/T22*/END*/RDRV        ;WD2REQ

IF(VCC) WDEND = T8*88END*MS2*/MS1*/MS0                    ;88IOREQ cycle end
              + T8*/PCREQ*/MS2*MS1*/MS0                   ;PCIOREQ
              + T22*/MS2*MS1*MS0                          ;WD1REQ
              + T22*MS2*/MS1*MS0                          ;WD2REQ
              + RDRV                                      ;Reset cycle end
              + END                                       ;Deactivate last

IF(VCC) END   = /PCOE*/PCDOE*/MOE*/WDOE*T8                ;Normal cycle end
              + /PCOE*/PCDOE*/MOE*/WDOE*RDRV              ;Reset end from PC
              + /T2*T8*/RDRV                              ;Reset failure end
              + /T2*T22*/RDRV                             ;Reset failure end
              + T8*END                                    ;Latch end signal
              + T22*END                                   ;Ensure restarted

DESCRIPTION
```

PAL2

(IC8, circuit diagram 4)

```
PAL16L8                                              PAL DESIGN SPECIFICATION
PAL2                                                 S. ASPIN  10/05/1986
Memory, ready and write logic PAL
X.25 COMMS. CARD  (U.C.T.)
T10  T9     T14     T12   T4      T17    /PCWRITE MS1   MS2     GND
MS0  /88RDY /PPCRDY /RAS  /88READ /WRITE /WDWRITE /PCAS /PMSEL  VCC

IF(VCC) RAS    = T4*/T9*/MS2*/MS1*/MS0                   ;REFREQ
               + T4*/T9*/MS2*/MS1*MS0                    ;PCMREQ
               + T4*/T9*MS2*MS1*/MS0                     ;88MREQ
               + T4*T9*/T17*/MS2*MS1*MS0                 ;WD1REQ
               + T4*T9*/T17*MS2*/MS1*MS0                 ;WD2REQ

IF(VCC) PMSEL  = T4*/MS2*/MS1*MS0                        ;PCMREQ
               + T4*MS2*MS1*/MS0                         ;88MREQ
               + T10*/MS2*MS1*MS0                        ;WD1REQ
               + T10*MS2*/MS1*MS0                        ;WD2REQ

IF(VCC) PCAS   = T4*/MS2*/MS1*MS0                        ;PCMREQ
               + T4*MS2*MS1*/MS0                         ;88MREQ
               + T12*/MS2*MS1*MS0*/WRITE                 ;WD1REQ read
               + T12*MS2*/MS1*MS0*/WRITE                 ;WD2REQ read
               + T14*/MS2*MS1*MS0*WRITE                  ;WD1REQ write
               + T14*MS2*/MS1*MS0*WRITE                  ;WD2REQ write

IF(VCC) 88RDY  = T4*MS2*MS1*/MS0                         ;88MREQ
               + T10*MS2*/MS1*/MS0                       ;88IOREQ

IF(VCC) PPCRDY = T4*/MS2*/MS1*MS0                        ;PCMREQ
               + T10*/MS2*MS1*/MS0                       ;PCIOREQ

IF(VCC) WRITE  = PCWRITE*/MS2*/MS1                       ;PCMREQ and REFREQ
               + PCWRITE*/MS2*MS1*/MS0                   ;PCIOREQ
               + /88READ*MS2*MS1*/MS0                    ;88MREQ
               + /88READ*MS2*/MS1*/MS0                   ;88IOREQ
               + WDWRITE*/MS2*MS1*MS0                    ;WD1REQ
               + WDWRITE*MS2*/MS1*MS0                    ;WD2REQ
               + T4*WRITE                                ;Latch the WRITE line

DESCRIPTION
```

PAL3

(IC7, circuit diagram 3)

```
PAL16L8                                              PAL DESIGN SPECIFICATION
PAL3                                                 S. ASPIN 10/05/1986
WD2511 CONTROL LOGIC PAL
X.25 COMMS. CARD  (U.C.T.)
T6      /WDEND FT18     A16A6  MS2    MS1    T11    MS0    /DRQO1  GND
/WRITE /DACK1 /WDWRITE /DACK2 /WD2CS /DRQO2 /WD1CS /WD1OE /WD2OE  VCC

IF(VCC) WD1CS    = /A16A6*T6*/T11*MS2*/MS1*/MS0*WRITE*/WDEND       ;8088 write
                 + /A16A6*T6*MS2*/MS1*/MS0*/WRITE*/WDEND           ;8088 read
                 + T6*/T11*/MS2*MS1*/MS0*WRITE*/WDEND              ;PC write
                 + T6*/MS2*MS1*/MS0*/WRITE*/WDEND                  ;PC read

IF(VCC) WD2CS    = A16A6*T6*/T11*MS2*/MS1*/MS0*WRITE*/WDEND        ;8088 write
                 + A16A6*T6*MS2*/MS1*/MS0*/WRITE*/WDEND            ;8088 read

IF(VCC) WD1OE    = /A16A6*T6*MS2*/MS1*/MS0*/WDEND          ;8088 read or write
                 + T6*/MS2*MS1*/MS0*/WDEND                 ;PC read or write
                 + T6*/MS2*MS1*MS0*/WDEND                  ;WD mem read or write

IF(VCC) WD2OE    = A16A6*T6*MS2*/MS1*/MS0*/WDEND           ;8088 read or write
                 + T6*MS2*/MS1*MS0*/WDEND                  ;WD mem read or write

IF(VCC) DACK1    = T6*/FT18*/MS2*MS1*MS0*/WDEND            ;WD mem read or write

IF(VCC) DACK2    = T6*/FT18*MS2*/MS1*MS0*/WDEND            ;WD mem read or write

IF(VCC) WDWRITE  = /MS2*MS1*MS0*DRQO1               ;WD1 memory write
                 + MS2*/MS1*MS0*DRQO2               ;WD2 memory write
                 + WDWRITE*T6                       ;Latch the WDWRITE signal
                 + WDWRITE*T11                      ;ensure last to deactivate
                 + MS2*/MS1*/MS0*/WRITE             ;8088 read from WD1 or WD2
                 + /MS2*MS1*/MS0*/WRITE             ;PC reading from WD1

DESCRIPTION
```

PAL4

(IC13, circuit diagram 2)

```
PAL16L8                                             PAL DESIGN SPECIFICATION
PAL4                                                S. ASPIN  10/05/1986
8088 CONTROL PAL
X.25 COMMS. CARD  (U.C.T.)
/88OE IOM    88A17  ALE   88A7   88A6   /88RD DTR    /88WR    GND
88A16 /MUART /88DOE 88END /MREQEN A16A6 A17A7 /88READ /88IOREQ  VCC

IF(VCC) 88IOREQ = 88WR*IOM*A17A7              ;8088 write to WD1 or WD2
                + 88RD*IOM*A17A7              ;8088 read from WD1 or WD2

IF(VCC) MREQEN  = /IOM*ALE                    ;Signal a memory request to F74
                + MREQEN*/IOM*/88DOE          ;Latch the request on 88DOE

IF(VCC) 88READ  = /DTR*ALE                    ;Sample the DTR line of the 8088
                + 88READ*/ALE                 ;and latch it
                + /DTR*88READ                 ;Ensure a glitch free latching

IF(VCC) 88DOE   = 88WR*88OE                   ;8088 data buffer controlled by
                + 88RD*88OE                   ;the 8088 WR and RD lines

IF(VCC) MUART   = IOM*/88A7*/88A6             ;8088 accessing MUART registers

IF(VCC) /A17A7  = /IOM*/88A17*ALE             ;Sample the A17 line
                + /A17A7*/ALE                 ;Latch it when ALE goes low
                + /IOM*/88A17*/A17A7          ;Ensure a glitch free A17 latch
                + IOM*/88A7*ALE               ;Sample the A7 line
                + IOM*/88A7*/A17A7            ;Ensure a glitch free A7 latch

IF(VCC) /A16A6  = /IOM*/88A16*ALE             ;Sample the A16 line
                + /A16A6*/ALE                 ;Latch it when ALE goes low
                + /IOM*/88A16*/A16A6          ;Ensure a glitch free A16 latch
                + IOM*/88A6*ALE               ;Sample the A6 line
                + IOM*/88A6*/A16A6            ;Ensure a glitch free A6 latch

IF(VCC) /88END  = 88WR                        ;8088 cycle ends when the WR
                + 88RD                        ;and RD lines are deactivated
                + MREQEN                      ;Allow for WR,RD active delay

DESCRIPTION
```

```
PAL16L8                                                 PAL DESIGN SPECIFICATION
PAL5                                                    S. ASPIN  10/05/1986
PC BUS INTERFACE PAL
X.25 COMMS. CARD  (U.C.T.)
A3   /IOSEL   MEMSEL /DACK0 /IOR /IOW   A4      /MEMR   /MEMW    GND
RDRV /PCWRITE /PCDOE /RESET NMI  SIGNAL /REFREQ /PCMREQ /PCIOREQ  VCC

IF(VCC) PCMREQ  = MEMSEL*MEMR*/DACK0                    ;PC memory read
                + MEMSEL*MEMW*/DACK0                    ;PC memory write

IF(VCC) PCIOREQ = IOSEL*A4*IOR*/MEMW                    ;PC I/O read
                + IOSEL*A4*IOW*/MEMR                    ;PC I/O write

IF(VCC) REFREQ  = DACK0*MEMR                            ;DACK0 signals a refresh
                + DACK0*MEMW
                + REFREQ*/PCDOE*/RDRV                   ;Latch the REFREQ signal

IF(VCC) PCWRITE = MEMW                          ;Write PC data to memory or to WD1
                + IOW*/MEMR                     ;IOW is active during DMA reads
                + REFREQ                        ;Write during a refresh

IF(VCC) RESET   = RDRV                          ;Reset line from PC bus
                + IOSEL*/A4*/A3*IOR             ;Reading I/O location gives a reset
                + RESET*/IOSEL
                + RESET*A4                      ;RESET*/(IOSEL*/A4*/A3*IOW)
                + RESET*A3                      ;Latch RESET signal until the PC
                + RESET*/IOW                    ;writes to the I/O location

IF(VCC) /NMI    = /IOSEL                        ;NMI= IOSEL*/A4*A3*IOR
                + A4                            ;use ORs for active high output
                + /A3                           ;NMI restarts 8088 programs
                + /IOR                          ;PC reads I/O location for a NMI

IF(VCC) /SIGNAL = /IOSEL                        ;SIGNAL= IOSEL*/A4*A3*(IOW+SIGNAL)
                + A4                            ;use ORs for active high output
                + /A3                           ;General purpose interrupt to 8088
                + /IOW*/SIGNAL                  ;Latch for adequate MUART hold time
```

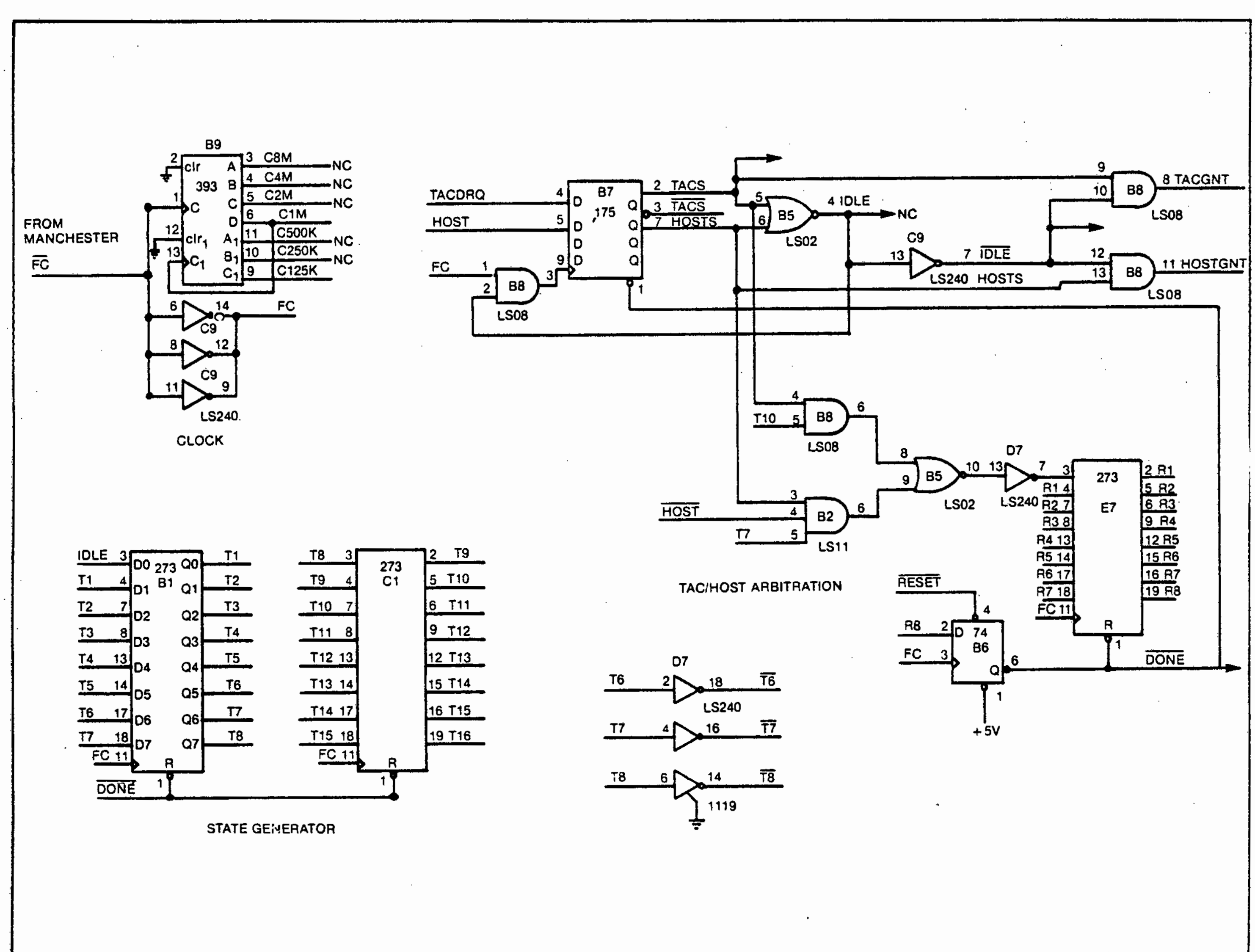
DESCRIPTION

A.1.1 PAL programming

The PAL equations must first be compiled by the PALASM program (for this thesis this was done using the P20 command on the departmental PDP computer). The resulting hex file is then downloaded to the PAL programmer (see BSc. thesis by Sherlock, 1984 for details on the PAL programmer). Note that when programming, PAL devices from MMI need a lower verify voltage than do PALs from other manufacturers.

A.2 WD arbitrator design

This is the arbitration and state generating logic from an application note by Western Digital for the WD2840 (WD2511/WD2840 handbook, 1984). This note described a circuit to connect a WD2840 device to a computer system using 8K of dual port static memory. The circuit was analysed and found to have reliability problems. The design in this thesis solves these problems and is presented in chapter 3 with comparisons being made to the WD design.



A.3 X.25 card assembly notes:

The PCB is double sided, through-hole plated and solder masked. The edge connectors are gold plated. Dimensions are given on the layout diagram and the board should be cut to the smallest possible tolerances. Before actual construction, the board should be usefully checked for possible shorts. Construction follows in the usual manner, using the location diagram to locate the components. It is strongly recommended that sockets be used for all ICs.

Due to the space problems when laying out the PCB, the following jumpers need to be installed:

1. +5V at centre of PCB (ie. power track from pin 20 of IC 33) to pin 18 of IC 41.
2. Extra ground line near edge connector : pin 10 of IC 47 to hole next to pin 8 of IC 48.
3. /88READ line to 8088 data buffer : pin 18 of IC 13 to pin 1 of IC 42.
4. MUART clock line : centre of J2 to pin 17 of IC 45.
5. Pullup for IC 11 input : R5 to pin 14 of IC 11.

Due to a misunderstanding that pin 12 of the 16L8 PAL devices could be used as a feedback input, the following layout errors occurred:

i) On PAL 1 (IC 4) pin 12 needs to be swopped with pin 15

ii) On PAL 3 (IC 7) pin 12 needs to be swopped with pin 16

iii) On PAL 5 (IC50) pin 12 needs to be swopped with pin 16

Before soldering in the 50 way connector, it is recommended that the metal cover be drilled and cut first. This will ensure that the cards is easily slotted into the PC. Finally, note that the crystal X2 is mounted using picture mounting tape (ie. double sided adhesive strip).

All ICs, with exception, are installed vertically with pin 1 at the top left hand corner (see location diagram).

A.4 Hardware configuration

Having constructed an X.25 card, it will be necessary to configure it prior to installation in a PC. The configuration is performed by setting jumper switches J1 to J13. This section looks at the jumpers in ascending order, giving the function of each, together with the default setting. The component location diagram should be consulted to locate the jumpers on the X.25 card. A reference is also made to the circuit diagram in which the jumpers appear (diagrams are in chapters 3 and 4).

Note that all jumper setting diagrams are drawn viewing the card from the component side with the edge connector on the right side ie., the same orientation as used for the component location diagram and card photo.

A.4.1 Jumper J1: (circuit diagram 4)

This jumper selects into which 64K memory block the second WD2511 processor (IC 36) is mapped. The jumper has two settings defined as follows:

on : The second WD2511 processor is mapped into memory block 1, sharing it with the first WD2511.

off: The second WD2511 processor is mapped into memory block 2.

The default setting is on, giving the card memory map as depicted in fig 8.2.

A.4.2 Jumper J2: (circuit diagram 6)

Selects the clock frequency supplied to the MUART device. This clock is divided down by the MUART and used for its timers and the baud rate of the asynch. communications line.

on : MUARTCLK is 3,072 MHz (the default setting).

off: MUARTCLK is 1,536 MHz

A.4.3 Jumper J3: (circuit diagram 5)

Selects whether the first WD2511 processor (IC 32) can give an interrupt to the PC or not.

closed : WD1 can give an interrupt to the PC. (see J11 setting for which interrupt line is used).

open : WD1 cannot interrupt the PC.

Default setting is open. This jumper may be closed if the card does not have its own 8088 processor installed (eg. if being used as a low cost LAN card).

A.4.4 Jumper J4: (circuit diagram 6)

Selects the clock source for the transmit signal element timing of the physical interface for WD2.

on : clock supplied by the MUART on the X.25 card (only used for initial testing of the MUART and WD2511 processor).

off: transmit clock is supplied externally from the DCE or modem to which the card is connected.

Default setting is off. An external clock (from a signal generator) should be supplied when doing loop back testing.

A.4.5 Jumper J5:

Same function and settings as for J4, but for interface 1.

A.4.6 Jumper J6: (circuit diagram 5)

X.25 card I/O address bank selection. This jumper selects which I/O address range in the PC's I/O map the particular X.25 card is to occupy. There are 6 different settings as defined below:

setting	I/O range (hex)	I/O block number
0	220 - 23F	0
1	240 - 25F	1
2	260 - 27F	2
3	280 - 27F	3
4	2A0 - 2fF	4
5	2C0 - 2DF	5

Notes:

a) Each card installed must have a unique I/O location.

b) If additional non IBM cards are installed in the PC their I/O location should be checked to avoid possible overlaps (see table 4.3 for the IBM PC's I/O map).

c) The I/O block number used should be entered into the TG.BAT system startup file (see Appendix G).

A.4.7 Jumpers J7 and J8 (circuit diagram 5)

These select into which 256K memory block the X.25 card is to be mapped, according to the table below:

	J7	J8	address range
setting:	on	on	0 - 3FFFF hex
on	on	off	40000 - 7FFFF
off	off	on	80000 - BFFFF
	off	off	C0000 - FFFFF

The default setting is J7 on and J8 off (for a PC with 256K installed on the motherboard and the X.25 card supplying a further 256K).

A.4.8 Jumpers J9 and J10: (circuit diagram 5)

The setting of these jumpers has a dual meaning, depending on whether J11 and J12 are open or closed. For J11 and J12 closed (only 64K of the cards memory dual ported with the PC), J9 and J10 select the location of the dual ported 64K block according to the table below.

	J9	J10	address range
setting:	on	on	00000 - 0FFFF
on	on	off	10000 - 1FFFF
off	off	on	20000 - 2FFFF
	off	off	30000 - 3FFFF

Note that the address range refers to the base address selected by J7 and J8.

If 256K of the cards memory is accessible by the PC (J11 and J12 open), then J9 and J10 may be used to vary the order in which the cards individual 64K blocks are mapped into the PC's memory space. The options are listed in the table below (as a reference, the card's 8088 processor always sees the memory blocks in the order 0,1,2,3,).

	J9	J10	memory block order
setting:	on	on	3,2,1,0
on	on	off	2,3,0,1
off	off	on	1,0,3,2
	off	off	0,1,2,3

Usually J9 and J10 will be on, simply reversing the memory block order.

A.4.9 Jumpers J11 and J12: (circuit diagram 5)

These select how much memory is dual ported with the PC according to the table below.

	J12	J11	amount dual ported mem.
o o open	closed	closed	64K
o o closed	closed	open	128K
	open	open	192K
	open	open	256K

For a single card installed in a PC, J12 and J11 will usually be open.

A.4.10 Jumper J13: (circuit diagram 5)

This selects which of the PC's available interrupt lines it to be used by the card. The lines are selected as depicted below.

o o IRQ2

o o IRQ3

o o IRQ4

o o IRQ5

As interrupts are not used in the traffic generator configuration, this jumper is simply left open.

A.4.11 Address selection:

The physical address of the X.25 cards is set by jumpers J6 to J12. This section summarises the address selection procedure.

a) Set the cards I/O address via J6.

b) Determine the amount of dual ported memory required and set J11 and J12 accordingly.

c) Select the card address via J7 to J10.

d) Edit the TG.BAT file so that the traffic generator programs are loaded to the correct locations (see Appendix F).

e) Set the PC's dip switch so it can use the shared memory (see IBM installation manual).

This completes the address selection.

A.4.12 Circuit options

If only a single WD processor is required, then ICs 16,31,34 and 35 need not be installed on the card. If the 8088 is not installed, then ICs 13,38,40,42 and 45 should not be installed. Pullup resistors ensure that the memory request lines are deactivated.

If 64K of memory is all that is required, then 4164 DRAM chips may be installed in place of the 41256 devices.

Finally, if the card is to be used for other applications (eg. a token passing LAN card), then the appropriate WD processor (WD2840 for a LAN, WD2507 for SS no. 7) should be installed in place of IC32. The physical level interface lines are available on connector 3 (see component location diagram and circuit diagram no 7).

A.5 X.25 card Testing procedure:

The first test is to do a continuity check on the PALs and other important chips. Next a high speed logic analyser (an HP 1630 G was used), set to 10ns sample intervals, will be required to carry out the tests below. Note that tests should preferably be conducted in the order shown:

A.5.1 PAL5 (PC bus interface logic:

Program PAL5 and insert it and the relevant logic (except the data buffer IC47) onto the card.

Connect the logic analyser to monitor the /IOW, /IOSEL, /RESET and SIGNAL lines. Load BASIC on the PC and use the statements OUT 544,0 and A = INP(544) to check that the reset line can be latched and unlatched. An OUT 552,0 statement can be used to check the SIGNAL line.

Next connect the logic analyser to look at /REFREQ, /PCMREQ, /PCIOREQ and /PCWRITE. As statements such as OUT 560,0 or A=INP (560) will allow the I/O request logic to be checked. For a memory test, give the commands DEF SEG = 16384 and POKE 10,0 to check the memory request and write lines. The /REFREQ line should pulse active with a period of around 15 micro seconds.

If any tests do fail, check for correct I/O and memory address jumper selections. If other tests are done, note that /IOW goes low together with /DACKO and /MEMR. This is because the DMA chip on the PC also activates the I/O write line when doing a memory read (DACKO used for refresh).

A.5.2 PAL1 (Bus controller):

With PAL1, PAL2 and the bus arbitrator and state logic inserted, connect the logic analyser to monitor /PCIOREQ, /PCOE, /PCDOE, /END. Once again a cycle can be initiated from BASIC via say A=INP(560). Check for connect cycle operation in particular the /END line pulsing low after /PCOE and /PCDOE have been deactivated. The cycle should be around 1040 ns long or five CPU cycles of 210ns.

If these tests are passed, then the remaining logic may be inserted with the exception of all the data buffers.

A.5.3 PAL2 (DRAM Controller):

Monitor the /PCMREQ, /RAS, /CAS and /MOE lines and write to memory via DEF SEG=16384 and POKE 10,0. Check for connect /RAS, /CAS and /MOE timing (refer to fig 4.9). Note that /RAS goes high prior to /CAS to conserve power and ensure an adequate precharge time.

The second test is to monitor the /REFREQ, /PCDOE, /WRITE and /MOE lines, checking that /WRITE changes state after /PCDOE and /MOE have gone inactive (note that /WRITE is latched on T4 to ensure this - see PAL2 listings). During a refresh cycle, check that /WRITE is active and that /PCDOE pulses active the end of the cycle to unlatch PAL5.

If everything is working, insert the data buffers (IC15 and IC43) and adjust the DIP switches on the PC for an extra 64K of memory. When the system is rebooted, the dynamic memory will be initialized and the transient part of DOS loaded onto the card. The DOS CHKDSK command may be used to verify that the memory has been dual ported.

A.5.4 PAL3 (WD2511 Controller):

Monitor the /PCIOREQ, /PCOE, /WD1OE and /WD1CS lines and write to the WD2511 processor. Check the timings against the PAL listings eg. /WD1OE comes 110ns from /PCOE on T6 and /WD1CS is low for around 275ns from T6 to T11.

Next monitor the /WRITE and /WDWRITE lines for connect polarity (During I/O accesses /WDWRITE will have the opposite polarity of /WRITE). Note that a DMA cycle cannot be checked until the connectors are set up - the WD2511 sets up the link before going to the memory look up tables.

A.5.5 PAL4 (8088 Control):

The basic test for the 8088 is to monitor the ALE, /88MREQ, /88DOE and /MOE lines (see fig 4.5). A typical cycle will be around 1050ns, indicating one wait state. Other tests should also be done, particularly to monitor the /88DOE and /88READ lines. From the trace note that only 1 wait state is required for a refresh while an adequate RAS precharge time of around 220ns is allowed (see fig 4.5).

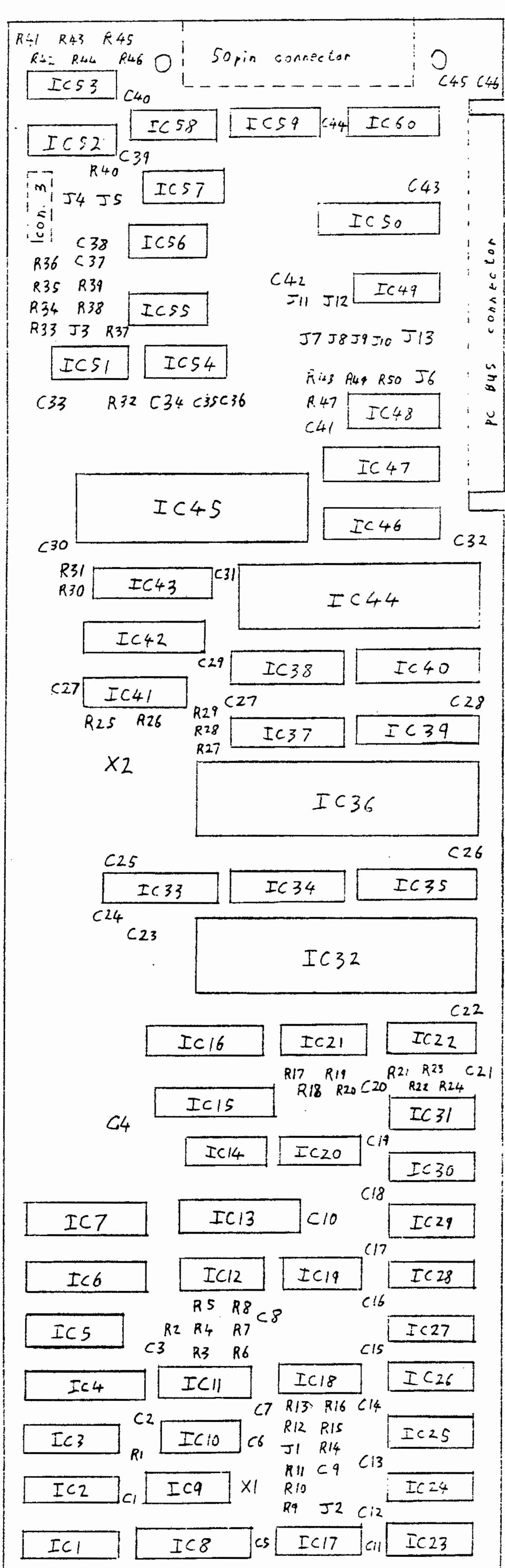
A.5.6 Final Tests:

Having checked the basic circuit operation, the remaining chips may be inserted onto the X.25 card. The first test is to connect the link interface lines for loop back testing and to provide a clock. A BASIC program may then be written to program and set up the link (see WD2511 manual). DMA cycle timings may be monitored and compared to fig 4.7.

For the 8088, instructions may be packed into memory before starting the processor via a OUT 544,0. The final test is to POKE in a jump instruction so that the 8088 loops continuously. The PC can be made to execute the same loop allowing one to monitor the bus arbitrator performance.

This concludes the testing of the card. The traffic generator programs can now be loaded (refer to Appendix G).

Component location diagram.



COMPONENT LIST

Integrated circuits:

NO.	TYPE	IC NUMBER	NOTES
2	WD2511A	IC32; IC36	S.C.D.slow speed 100 KHz version E.B.E.
1	8088	IC44	
1	8284A	IC41	
1	8256 MUART	IC45	
5	16L8A PALs	IC4; IC7; IC8; IC13; IC50	Fast (A) version Promilect
9	41256-15 DRAM's	IC23 to IC31	150ns version
	(or 4164-15)	IC23 to IC31	S.C.D.
1	F174	IC3	Protronix
1	F373	IC6	
1	F00	IC10	
1	F74	IC19	
2	LS164	IC1; IC2;	
1	LS148	IC5;	
1	LS04	IC9	
1	LS151	IC11	
1	LS03	IC12	
1	LS08	IC14	
1	LS92	IC17	
1	LS74	IC18	
1	LS280	IC20	
2	LS158	IC21; IC22	
2	LS373	IC40; IC46;	
1	LS138	IC48	
1	LS136	IC49	
1	LS132	IC51	
11	LS245	IC15; IC16; IC33; IC34; IC35; IC37; IC38; IC39; IC42; IC43; IC47;	
3	1488	IC56; IC57; IC58	
3	1489A	IC54; IC55; IC60;	
1	26LS31	IC59	Protronix
2	26LS32A	IC52; IC53	"

I.C. Sockets:

NO.	TYPE
4	24 pin (or 2 x 48 pin)
2	40 pin
19	20 pin
1	18 pin
17	16 pin
19	14 pin

Miscellaneous:

Capacitors

2	10nf		50V	ceramix caps	
41	100nF	50V		ceramic caps.	Electrolink
5	10,uF	16V		tantalum caps	
1	47uf		16V	tantalum caps	

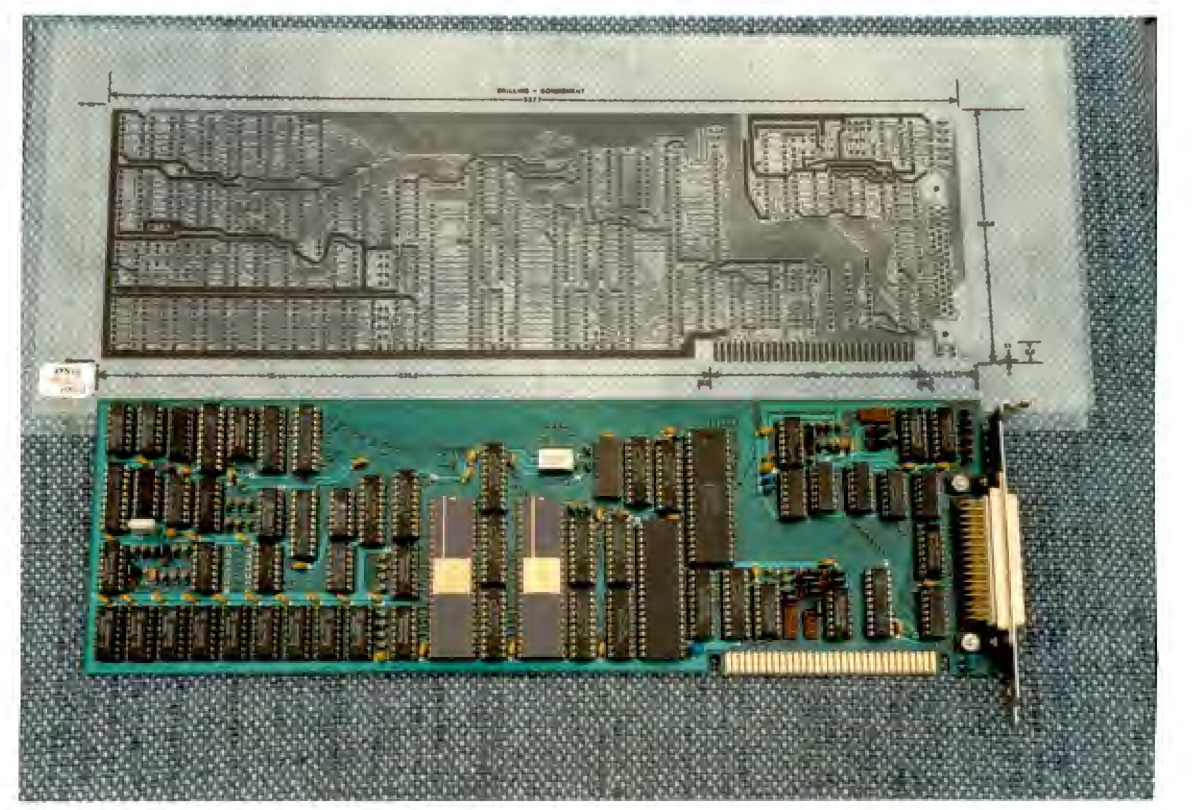
Resistors

12	33	1/4W	5% resistors
2	510	"	"
10	1K	"	"
12	4.7K	"	"
10	10K	"	"
1	100	"	"
1	220	"	"
2	27K	"	"
8	100	1/2W	5%

Berg Jumpers: 2 x 6 pin
2 x 5 pin
2 x 4 pin
12 x 3 pin (some cut to 2 pin)

Connectors:

1	50 pin 90" female connector	Strand tracking
2	50 pin male plugs & covers	
3	male 28 pin plugs & covers	
2	male V.35 connectors (for V.35 leads)	
Cable:	10m of 25 conductors, shielded (V.35 leads)	
	10m of 16 conductors, " (RS232C leads)	
	5m of 12 conductors, " (asynch RS232 lead)	



Picture of the X.25 card and P.C.B. layout.

APPENDIX B:

The X25TG program which runs on the PC, controlling and monitoring the X.25 cards.

File : X25TG.C

(This program runs on the IBM PC)

Contents:

B-1 : Data structure and variable declarations

B-2 : Main function

B-4 : Setup functions.

Functions to locate the shared data structures and start the X.25 cards.

B-7 : Display functions.

Display and update the statistics screens.

B-16 : System configuration.

Allows user to change test parameters.

```
/* X25TG.C */

/***************************/
/* IBM PC CONTROL PROGRAM */
/***************************/

/* This program runs on the PC's 8088 processor rather than the cards.
   It is called by the TG.BAT batch routine after the traffic generator
   programs have been loaded onto the installed X.25 cards.
   The program first checks for valid card addresses parameters.
   It then initializes the  card communication structures and, once the user
   has confirmed that he is ready, it starts the card's processors.
   The programs main job is to interact with the user, monitoring the
   keyboard and displaying the required statistics. */

                          /* Include C library routines to handle the console
                             and keyboard I/O. */
#include "STDIO.H"
#include "CONIO.H"
                          /* Include the required constant files. */
#include "SYSTEM.CON"
#include "X25TG.CON"
#include "ADDRESS.CON"


/*********************/
/* DATA DEFINITIONS */
/*********************/
/* As this program is running on the IBM PC rather than on an X.25 card, the
   data structures are different.  However, the type file is included to
   define the mcontrol and statistics data types. */

#include "DTYPES.C"

                                  /* Card control and stats structures. */
struct mcontrol  far *control[10];
struct statistics  far *pcstats[10];
                                          /* Card I/O address pointers. */
int  ioloc[5];

                                  /* Define the configuration structure. */
struct tg_control {
        int  no_chans;            /* Number of logical channels to use. */
        int  no_pacs;             /* Number of data packets to try and send per
                                     call set up. */
        int  call_len;            /* Call length in seconds. */
        int  cinterval;           /* Interval between a clear and the next call
                                     request on a channel (in seconds). */
        int  stimer;              /* Number of 10msec intervals between sending
                                     test packets on a channel. */
}  tg_config;

                                  /* Global variables used by the program. */
int  no_cards;            /* Number of X.25 cards installed (1 to 5). */
int  lback;               /* True if running in loop back mode. */
int  card;                /* Number of the card being displayed (0 to 4). */
int  line;                /* Number of line last configured (0 to 9). */
long tstart;              /* Test starting time. */

/**************************************************************************/
```

```
/* This is the main function of the control program.  Main is passed the
   hardware configuration when invoked, including a parameter for selecting
   loop back testing (DTE to DTE) rather than DTE to DCE communication (see
   TG.BAT listing).
   After calling functions to do the system initialization, the program loops
   around displaying the required statistics screens. */

main (argc, argv, envp)
  int  argc;
  char *argv[];
  char *envp[];
{
  char correct, old, ans, new;
  int  i;
                      /* The first function called sets up the address
                         pointers to the control and stats data structures
                         on the X.25 cards. */
  correct = (char) setup (argc, argv);

                      /* Return to DOS if invalid hardware configuration.*/
  if (correct == FALSE) {
    cputs("\n\rInvalid hardware configuration parameters passed. \n\r");
    cputs("Program aborted. \n\r");
    return;
  }

#if TESTING == FALSE
                              /* Dissable the X.25 cards' processors. */
                /* For testing, cards 8088 halted by loading routine TG.BAT*/
  for (i = 0; i < (no_cards * 2); i++)
    inp (ioloc[i] + RESET);
#endif

                      /* Load the default configuration onto the cards. */
  cfig_init();

  do {
                      /* Display the introductory screen, asking the user
                         if he wishes to configure the traffic generator.*/
    ans = (char) intro_display();
    if ((ans == 'c') || (ans == 'C'))
      configure();
  }
  while  ((ans == 'c') || (ans == 'C'));

  cputs("\n\r\n");
  cputs("System initialization, please wait...");

                              /* Enable the X.25 cards' processors. */
  for (i = 0; i < (no_cards * 2); i++)
    outp ((ioloc[i] + RESET), 0);

#if TESTING == TRUE
  initialize();
#else
                      /* Wait until all cards have finished initializing.*/
  for (i = 0; i < (no_cards * 2); i++)
    while ((*pcstats[i]).system.init != TRUE) {
    }
#endif
```

```
ans = CR;                        /* Initialize reply variables. */
old = 's';

do {
                                 /* Call up the system display required by the
                                    user.  Each function returns the value of
                                    the next display function requested. */

                                 /* Check if a new display. */
  new = (old == ans) ? FALSE : TRUE;
  old = ans;

  switch  (ans)  {

      case 's':
      case 'S':
              ans = (char) system_display (new, ans);
              break;
      case 'p':
      case 'P':
              ans = (char) packet_display (new, ans);
              break;
      case 'l':
      case 'L':
              ans = (char) link_display (new, ans);
              break;

      case 'c':
      case 'C':
              ans = (char) configure (new, ans);
              break;

      default:
              ans = (char) main_display (new, ans);
              break;
  }
}
                                 /* Return to DOS if user hits Esc. or 'X'. */
while  (ans != 'x' && ans != 'X' && ans != ESC);

                         /* Home the cursor (allows screen to be printed). */
cputs("\x1B[1;1H");

                         /* Shut down packet level and disconnect the link. */
#if TESTING == FALSE
  terminate();
#else
  for (i = 0; i < (no_cards * 2); i++) {
    ans = inp (ioloc[i] + NMI);
    outp (WD1 , 1);
  }
#endif

}

/***************************************************************************/
```

```
/*******************/
/* SETUP FUNCTIONS */
/*******************/

/* This function checks and loads the address pointers to the control and
   statistics data structures on the cards.
   Parameters passed to main program and hence to setup:
   the first selects loop back test (1) or connect to DCE (0),
   then follows card address block numbers (e.g. 4 if card control structures
   in fifth 64K block), followed by the card's  I/O block number.  (Note that
   the notation used is for block numbers to start from 0 rather than 1). */

setup (argc, argv)
  int  argc;
  char *argv[];
{
  register int i;
  int data[11];
  unsigned long int address[5];

                                /* Compute how many cards are installed. */
  no_cards = (argc / 2) - 1;
                                /* Exit if an invalid number of parameters.*/
  if ( (argc % 2 != 0) || (no_cards < 1) || (no_cards > 5) )
    return (FALSE);

                        /* Read in digits passed converting to decimal. */
  for (i = 0; i < (argc - 1); i++) {
                        /* First argument is program name, so use (i + 1). */
    data[i] = *argv[(i + 1)];
                                /* Check for a digit. */
    if (data[i] < '0' || data[i] > '9')
      return (FALSE);
                                /* Convert to decimal. */
    data[i] = data[i] - '0';
  }


                /* First digit selects loop back test or connected to a DCE*/
  if (data[0] == 1)
    lback = TRUE;


                                /* Compute the control and stats addresses
                                   for the installed cards. */
  for (i = 0; i < 5; i++) {

    if (i < no_cards) {
                                /* Check for PC reserved memory areas.  Card
                                   must be between 128K and 640K. */
      if (data[(i + 1)] < 2)
        return (FALSE);
                                /* Fill in address block number. */
      address[i] = data[(i + 1)];
    }
                                /* Unused cards given address of first one.*/
    else
      address[i] = data[1];
```

```
                                /* Now compute the address in segment:offset
                                   notation.  e.g. block 4 gives 4000:0000.*/
    address[i] = (address[i] << 28);

                        /* Assign the addresses to the memory pointers. */
                                /* Offset for stats structure 1 is F800. */
    pcstats[(i * 2)] = (struct statistics far *) (address[i] + S1_OFFSET);
                                /* Offset for stats structure 2 is F900. */
    pcstats[(i * 2) + 1] = (struct statistics far *) (address[i] +S2_OFFSET);

                        /* Offset for first lines control structure FA00. */
    control[(i * 2)] = (struct mcontrol far *) (address[i] + M1_OFFSET);
                                /* Second control structure at FA80. */
    control[(i * 2) + 1] = (struct mcontrol far *) (address[i] + M2_OFFSET);

  }

                                /* Compute the actual 16 bit I/O address. */
  for (i = 0; i < 5; i++) {

    if (i < no_cards) {
                                /* Check for a valid I/O block number.
                                   ( 6 allowed by hardware.) */
      if (data[(i + no_cards + 1)] > 5)
        return (FALSE);
                                /* Fill in the I/O block number. */
      ioloc[i] = data[(i + no_cards+1)];
    }
                                /* Unused cards have same address as first.*/
    else
      ioloc[i] = data[(no_cards + 1)];

                                /* Compute the physical address. */
    ioloc[i] = PCIOBASE + (ioloc[i] * PCIOLEN);
  }

                                /* Return to main program, indicating address
                                   set up completed. */
  return(TRUE);
}



/* This function loads the default traffic generator configuration onto the
   installed X.25 cards.  The configuration may be changed later by calling
   the configure function. */

cfig_init()
{
  register int i, j;
  int (far *eraser)[128];
                                /* Fill the default configuration parameters
                                   into the tg_config structure (X25TG.CON)*/
  tg_config.no_chans = C_CHANS;
  tg_config.no_pacs  = NO_PACS;
  tg_config.call_len  = CALL_LEN;
  tg_config.cinterval = CINTERVAL;
```

```
                                /* Clear the stats structure of all installed
                                   lines.  This is done by erasing the 128
                                   byte block reserved for the structure. */
  for (i = 0; i < (no_cards * 2); i++) {
    eraser = (int far *) &(*pcstats[i]);
    for (j = 0; j < 128; j++)
      (*eraser)[j] = 0;
  }
                                /* Fill the configuration details into the
                                   control structure of each card. */
  for (i = 0; i < (no_cards * 2); i++) {
    (*control[i]).no_chans = tg_config.no_chans;
    (*control[i]).no_pacs  = tg_config.no_pacs;
    (*control[i]).call_len  = 10 * tg_config.call_len;
    (*control[i]).cinterval = 10 * tg_config.cinterval;
                                /* Calculate time interval (in 10msecs units)
                                   between test data packets on a channel.*/
    (*control[i]).stimer = (100 * tg_config.call_len) / tg_config.no_pacs;
    if ((*control[i]).stimer == 0)
      (*control[i]).stimer = 1;
                                /* Fill in other control data. */
    (*control[i]).lback = lback;
    (*control[i]).pad = FALSE;
    (*control[i]).exit = FALSE;
    (*control[i]).pcread = FALSE;
  }
                                /* Initially display for line 0, card 0. */
  card = 0;
  line = 0;
}


/* A function to shut down the cards in an orderly manner. */
terminate()
{
  int i;
  long stime;
                                /* Signal to all cards to shut down. */
  for (i = 0; i < (no_cards * 2); i++)
    ((*control[i]).exit = TRUE);
                                /* Delay for 4 secs while the cards clear all
                                   calls and disconnect their links. */
  stime = time(NULL);
  while (time(NULL) <= (stime + 4)) {
  }
}


/* This function is called by the display functions to signal a particular
   card that statistics are to be read in.  It is also called by the
   configure() function prior to writing to a card's control structure. */
lhold(lineno)
  int lineno;
{
#if TESTING == FALSE
  (*control[lineno]).pcread = TRUE;     /* Signal the particular link. */
                                        /* Wait until card acknowledges. */
  while ((*pcstats[lineno]).system.stopped == FALSE) {
  }
#endif
}
```

```
/* Used with lhold, this function signals to a card that it may continue. */
lfree(lineno)
                /* This function frees a line after PC has finished. */
{
#if TESTING == FALSE
                                        /* Free the line. */
  (*control[lineno]).pcread = FALSE;
                                        /* Wait for card to respond. */
  while ((*pcstats[lineno]).system.stopped == TRUE) {
  }
#endif
}

/****************************************************************************/




/**********************/
/* DISPLAY FUNCTIONS */
/**********************/


/* This function displays the X25 traffic generator introductory screen.
   The user is prompted to alter the default test configuration. */

intro_display()
{
  int reply;

  title ('i');                  /* Display the title block. */

  cputs("Welcome to the X.25 traffic generator on the IBM PC.\n\r");
  cputs("\n\n");

  if (no_cards == 1)
    cputs("One X.25 card installed. \n\r\n");
  else
    cprintf("%d X.25 cards installed. \n\r\n", no_cards);

  if (lback == TRUE)
    cputs("Configured for loop back testing (DTE to DTE).\n\r");
  else
    cputs("Configured for normal testing.\n\r");

                                /* Display current software configuration. */
  cputs("\n");
 cprintf("Number of logical channels : %d per link.\n\r",tg_config.no_chans);
  cprintf("Number data pacs per call  : %d \n\r", tg_config.no_pacs);
  cprintf("Average call length        : %d secs.\n\r", tg_config.call_len);
  cprintf("Interval between calls     : %d secs.\n\r", tg_config.cinterval);

                        /* See if user wants to alter configuration.*/
  cputs("\n\n");
  cputs("Hit C to alter default test configuration, ");
  cputs("any other key to start > ");

                                /* Return the reply to the main program. */
  return ((int) getche());
}
```

```
/* This function displays the main display, giving overall system totals. */

main_display (new, ans)
  char new, ans;
{
  int  nlink=0, nchans=0;                  /* System totals. */
  long ndpac=0;
  long pcall=0;                            /* Packet totals. */
  int  pres=0, pdteclr=0, prr=0;
  int  fcs_errs=0, timers=0;               /* Link level totals. */
  long bytes=0, drate=0;

  register unsigned i;

                                           /* Get in all the data required. */
  for (i = 0; i < (no_cards * 2); i++) {
                                           /* Signal the particular link. */
    lhold(i);
                                           /* Number of active links. */
    if ((*pcstats[i]).link.lstate == TRUE)
      nlink++;
                                           /* Total no. logical channels. */
    nchans = nchans + (*pcstats[i]).system.active;
                                           /* Total no. data packets. */
    ndpac = ndpac + (*pcstats[i]).packet.data;
                                           /* Packet level totals. */
    pcall = pcall + (*pcstats[i]).packet.calls;
    pdteclr = pdteclr + (*pcstats[i]).packet.dte_clears;
    prr = prr + (*pcstats[i]).packet.rr;
                                           /* Link level totals. */
    fcs_errs = fcs_errs + (*pcstats[i]).link.fcs_err;
    timers = timers + (*pcstats[i]).link.t1_timeout;
    bytes = bytes + (*pcstats[i]).link.byte_tot;
                                           /* Release the link. */
    lfree(i);
  }

                                   /* Calculate the total average data rate. */
  i = (unsigned) (time(NULL) - tstart);
  if (i != 0)
    drate = (bytes * 8) / i;


                          /* If a new display, then write out the headings.*/
  if (new == TRUE)
    title (CR);
                                   /* Position the cursor to start of text. */
  cputs("\x1B[9;1H");

                                                  /* Display all the data. */
  cputs("     SYSTEM TOTALS:\n\r");
  cputs(" Number of active links    :   \x1B[K");
  cprintf(" %d \n\r", nlink);
  cputs(" Number active channels    :   \x1B[K");
  cprintf(" %d \n\r", nchans);
  cputs(" Number of data packets    :   \x1B[K");
  cprintf(" %ld \n\r\n", ndpac);
```

```
  cputs("     PACKET TOTALS:\n\r");
  cputs(" Total calls set up      :  \x1B[K");
  cprintf(" %ld \n\r", pcall);
  cputs(" Total DTE call clears   :  \x1B[K");
  cprintf(" %d \n\r", pdteclr);
  cputs(" Number of RR packets    :  \x1B[K");
  cprintf(" %d \n\r\n", prr);

  cputs("     LINK TOTALS:\n\r");
  cputs(" Total corrupted frames  :  \x1B[K");
  cprintf(" %d \n\r", fcs_errs);
  cputs(" Total link timeouts     :  \x1B[K");
  cprintf(" %d \n\r", timers);
  cputs(" Total data rate (bps.)  :  \x1B[K");
  cprintf(" %ld \n\r", drate);

                                        /* Ask user for next request or after
                                           a few secs update this display. */
  return (next_req(ans));
}




/* Display the system status of the traffic generator. */

system_display (new, ans)
  char new, ans;
{
                                        /* Card activity. */
  int  congest[2], rbuffs[2], tbuffs[2];
                                        /* Receiver error counters. */
  int  state_err[2], seq_err[2], rec_err[2];
                                        /* Program status. */
  int  pstate[2], buff_len[2];
  long prog_flag[2];

  int tactive, mins, secs;

  register int i, j;

                                        /* Get in required information. */
  for (i = 0, j = line; i < 2; i++, j++) {
    lhold(j);
    congest[i] = (*pcstats[j]).system.rcongested +
                    (*pcstats[j]).system.tcongested;
    rbuffs[i] = NO_REC_BUFFS - (*pcstats[j]).system.no_rbuffs;
    tbuffs[i] = NO_TRAN_BUFFS - (*pcstats[j]).system.no_tbuffs;

    state_err[i] = (*pcstats[j]).system.state_err;
    seq_err[i] = (*pcstats[j]).system.seq_err;
    rec_err[i] = (*pcstats[j]).system.rec_err;

    pstate[i] = (*pcstats[j]).system.pstate;
    buff_len[i] = (*pcstats[j]).system.buff_len + 3;
    prog_flag[i] = (*pcstats[j]).system.prog_flag;

    lfree(j);
  }
```

```
                                   /* Compute the time program active for. */
  tactive = time(NULL) - tstart;
  if (tstart == 0)
    tactive = 0;
  mins = tactive / 60;
  secs = tactive % 60;


                           /* If a new display, then write out the headings.*/
  if (new == TRUE)
    title ('s');
                                   /* Position the cursor to start of text. */
  cputs("\x1B[9;1H");

                                                  /* Display all the data. */
  cputs("     BUFFER ACTIVITY: \n\r");
  cputs(" System congestion flag  :          \x1B[K");
  cprintf(" %d   \x1B[10;53H  %d \r\n", congest[0], congest[1]);
  cputs(" Active receiver buffers :          \x1B[K");
  cprintf(" %d   \x1B[11;53H  %d \n\r", rbuffs[0], rbuffs[1]);
  cputs(" Active transmit buffers :          \x1B[K");
  cprintf(" %d   \x1B[12;53H  %d \n\r\n", tbuffs[0], tbuffs[1]);

  cputs("     REC. ERROR COUNTERS: \n\r");
  cputs(" Number of state errors  :          \x1B[K");
  cprintf(" %d   \x1B[15;53H  %d \n\r", state_err[0], state_err[1]);
  cputs(" Number of seq. errors   :          \x1B[K");
  cprintf(" %d   \x1B[16;53H  %d \n\r", seq_err[0], seq_err[1]);
  cputs(" Number of rec. errors   :          \x1B[K");
  cprintf(" %d   \x1B[17;53H  %d \r\n\n", rec_err[0], rec_err[1]);

  cputs("     PROGRAM STATUS: \n\r");
  cputs(" Test data packet length :          \x1B[K");
  cprintf(" %d bytes \n\r", buff_len[0]);
  cputs(" Time traff. gen. active :          \x1B[K");
  cprintf(" %d mins %d secs. \n\r", mins, secs);
  cputs(" Prog. flag (buff_find)  :          \x1B[K");
  cprintf(" %ld   \x1B[22;53H  %d \n\r", prog_flag[0], prog_flag[1]);

                                         /* Ask user for next request or after
                                            a few secs update this display. */
  return (next_req(ans));
}




/* Display packet level statistics. */

packet_display (new, ans)
  char new, ans;
{
                                         /* Receiver activity. */
  long pac_rec[2], pac_sent[2];
  int  no_chans[2];
                                         /* Received control packets. */
  int  dce_clears[2], resets[2], restarts[2];
                                         /* Receiver state. */
  int  c_clears[2], diags[2], timer[2];
  register int i, j;
```

```
                                  /* Get in the required information.*/
for (i = 0, j = line; i < 2; i++, j++) {
  lhold(j);
  pac_rec[i] = (*pcstats[j]).packet.pac_rec;
  pac_sent[i] = (*pcstats[j]).packet.pac_sent;
  no_chans[i] = (*pcstats[j]).system.active;

  dce_clears[i] = (*pcstats[j]).packet.dce_clears;
  resets[i] = (*pcstats[j]).packet.resets;
  restarts[i] = (*pcstats[j]).packet.restarts;

  c_clears[i] = (*pcstats[j]).packet.congest_clears;
  diags[i] = (*pcstats[j]).packet.diagnostics;
  timer[i] = (*pcstats[j]).packet.ptimeouts;
  lfree(j);
}

                        /* If a new display, then write out the headings.*/
if (new == TRUE)
  title ('p');
                                /* Position the cursor to start of text. */
cputs("\x1B[9;1H");

                                              /* Display all the data. */
cputs("      ACTIVITY: \n\r");
cputs(" Total packets received   :          \x1B[K");
cprintf(" %ld    \x1B[10;53H  %ld \n\r", pac_rec[0], pac_rec[1]);
cputs(" Total packets sent       :          \x1B[K");
cprintf(" %ld    \x1B[11;53H  %ld \n\r", pac_sent[0], pac_sent[1]);
cputs(" Number active channels   :          \x1B[K");
cprintf(" %d   \x1B[12;53H  %d \r\n\n", no_chans[0], no_chans[1]);

cputs("      PACKETS RECEIVED: \n\r");
cputs(" Number of DCE clears     :          \x1B[K");
cprintf(" %d   \x1B[15;53H  %d \n\r", dce_clears[0], dce_clears[1]);
cputs(" Number of reset packets :           \x1B[K");
cprintf(" %d   \x1B[16;53H  %d \n\r", resets[0], resets[1]);
cputs(" Number of restarts       :          \x1B[K");
cprintf(" %d   \x1B[17;53H  %d \n\r\n", restarts[0], restarts[1]);

cputs("      STATE: \n\r");
cputs(" No congestion clears     :          \x1B[K");
cprintf(" %d   \x1B[20;53H  %d \r\n", c_clears[0], c_clears[1]);
cputs(" No diagnostics packets   :          \x1B[K");
cprintf(" %d   \x1B[21;53H  %d \r\n", diags[0], diags[1]);
cputs(" Timer expiry counter     :          \x1B[K");
cprintf(" %d   \x1B[22;53H  %d \n\r", timer[0], timer[1]);

                                  /* Ask user for next request or after
                                     a few secs update this display. */
return (next_req(ans));
}
```

```
/* Display the link level statistics. */

link_display (new, ans)
  char new, ans;
{
                                          /* Link activity. */
  int  state[2], timer[2], resets[2];
                                          /* Control frames received. */
  int  rej[2], fcs_err[2], frmr[2];
                                          /* Link state. */
  int  rec_rnr[2], active[2], wd_err[2];
  register int i, j;

                                          /* Get in the required information.*/
  for (i = 0, j = line; i < 2; i++, j++) {
    lhold(j);
    state[i] = (*pcstats[j]).link.lstate;
    timer[i] = (*pcstats[j]).link.t1_timeout;
    resets[i] = (*pcstats[j]).link.resets;

    rej[i] = (*pcstats[j]).link.rej_rec;
    fcs_err[i] = (*pcstats[j]).link.fcs_err;
    frmr[i] = (*pcstats[j]).link.frmr_rec;

    rec_rnr[i] = (*pcstats[j]).link.rec_rnr;
    active[i] = (*pcstats[j]).link.activity;
    wd_err[i] = (*pcstats[j]).link.mem_notrdy +
                    (*pcstats[j]).link.rlook_err;
    lfree(j);
  }

                          /* If a new display, then write out the headings.*/
  if (new == TRUE)
    title ('l');
                                  /* Position the cursor to start of text. */
  cputs("\x1B[9;1H");

                                                  /* Display all the data. */
  cputs("     LINK STATE: \n\r");
  cputs(" Current link state      :          \x1B[K");
  cprintf(" %d   \x1B[10;53H  %d \n\r", state[0], state[1]);
  cputs(" No times T1 expired     :          \x1B[K");
  cprintf(" %d   \x1B[11;53H  %d \r\n", timer[0], timer[1]);
  cputs(" Number of link resets   :          \x1B[K");
  cprintf(" %d   \x1B[12;53H  %d \n\r\n", resets[0], resets[1]);

  cputs("     FRAMES RECEIVED: \n\r");
  cputs(" Number of REJ frames    :          \x1B[K");
  cprintf(" %d   \x1B[15;53H  %d \n\r", rej[0], rej[1]);
  cputs(" No with check errors    :          \x1B[K");
  cprintf(" %d   \x1B[16;53H  %d \n\r", fcs_err[0], fcs_err[1]);
  cputs(" Number of FRMR frames   :          \x1B[K");
  cprintf(" %d   \x1B[17;53H  %d \r\n\n", frmr[0], frmr[1]);
```

```
  cputs("      LINK ACTIVITY: \n\r");
  cputs(" No RNR frames received  :             \x1B[K");
  cprintf(" %d   \x1B[20;53H  %d \n\r", rec_rnr[0], rec_rnr[1]);
  cputs(" % time link rec active  :             \x1B[K");
  cprintf(" %d   \x1B[21;53H  %d \n\r", active[0], active[1]);
  cputs(" WD2511 mem/buffer error :             \x1B[K");
  cprintf(" %d   \x1B[22;53H  %d \n\r", wd_err[0], wd_err[1]);

                                        /* Ask user for next request or after
                                           a few secs update this display. */
  return (next_req(ans));
}




/* This routine is called by all the display functions to build the title
   block.  It does this making use of the IBM extended character set (See IBM
   reference manuals).  As with the other display functions, use is made of
   the IBM extended screen control functions (see DOS technical ref. manual).
   The function is passed a parameter for the screen type and displays a
   title accordingly. */

title (type)
  char type;
{
  int i;
                        /* First clear the screen and home the cursor. */
  cputs("\x1B[2J");

                        /* Print the title block. */
                                        /* Print a left hand corner, */
  cputs("                          \xC9");
  for (i=0; i < 24; i++)                /* followed by a solid line  */
    cputs("\xCD");
  cputs("\xBB \r\n");                   /* and a right hand corner.  */
                        /* Print the title with left and right borders. */
  cputs("                          ");
  cputs("\xBA                        \xBA\r\n");
  cputs("                          ");
  cputs("\xBA X.25 TRAFFIC GENERATOR \xBA\r\n");
  cputs("                          ");
  cputs("\xBA                        \xBA\r\n");
                                /* Print the bottom corners and line. */
  cputs("                          \xC8");
  for (i=0; i < 24; i++)
    cputs("\xCD");
  cputs("\xBC\r\n\n");
```

```
                                  /* Now print the screen specific titles. */
switch  (type)  {
      case 's':
              cputs(" SYSTEM DISPLAY: \n\r");
              cputs(" ==============  ");
              break;
      case 'p':
              cputs(" PACKET LEVEL DISPLAY: \n\r");
              cputs(" ====================  ");
              break;
      case 'l':
              cputs(" LINK LEVEL DISPLAY: \n\r");
              cputs(" ==================  ");
              break;
      case 'c':
              cputs(" CONFIGURATION SCREEN \n\r");
              cputs(" ==================== ");
              break;
      case 'i':
              cputs("\n\r");
              break;
      default:
              cputs(" MAIN DISPLAY: \n\r");
              cputs(" ============  ");
}

                                  /* Display the link title if required. */
if  (type == 's' || type == 'p' || type == 'l')   {
                                  /* Position cursor to row 8, column 33. */
  cputs("\x1B[8;35H");
  cprintf("  Link %d           Link %d \n\r", line, (line+1));
}
else
  cputs("\n\r");

                                  /* Flush the keyboard buffer. */
while (kbhit() != FALSE)
  getche();
}
```

```
/* This function prompts the user to enter what display he would like to view
   next.  It is called by all the display functions and returns the user's
   request.  If the user has not responded, then after a few secs (determined
   by T_UPDATE in the X25TG.CON file) the current display is updated.
   The function also initializes the start time variable and calls the
   traffic generator program if in testing mode. */

next_req (old)
  char old;
{
  static long tlast;
                                /* Initialize system starting time. */
  if (tstart == 0)
    tlast = tstart = time(NULL);

                                        /* Display the prompt. */
  cputs("\n\r");
  cputs(" Display options:  S: system,  P: packet level, ");
  cputs(" L: link level \n\r");
  cputs("                   C: configuration,  X: exit,   ");
  cputs(" other: main  ...>");

                                /* Update display at set intervals. */
  while ( (time(NULL) - tlast) < T_UPDATE ) {
                                /* Call other processes if in test mode. */
    #if TESTING == TRUE
      x25_card();
    #endif
                                /* If user has hit a key, then read it in and
                                   return it's value. */
    if (kbhit() != FALSE) {
      tlast = time(NULL);
      return ((int) getche());
    }
  }
                                /* If no response, then return old value. */
  tlast = time(NULL);
  return((int) old);
}

/*****************************************************************************/
```

```
/*********************
 * CONFIGURATION MENU *
 *********************/

/* This function displays the current traffic generator configuration for a
   line and allows the user to change it.
   The user can also change the number of the line being monitored. */

configure (new)
  char new;
{
  char reply, exit=FALSE;
  int value, i;

                         /* Loop around displaying line configurations. */
  do {
                         /* Get in configuration for the line concerned. */
    lhold (line);
    tg_config.no_chans = (*control[line]).no_chans;
    tg_config.no_pacs  = (*control[line]).no_pacs;
    tg_config.call_len  = (*control[line]).call_len / 10;
    tg_config.cinterval = (*control[line]).cinterval / 10;
    lfree (line);

                                 /* Display the configuration title. */
    title ('c');
    cprintf(" Current configuration : card %d, line %d.", card, line);
    cprintf("   (no. cards: %d)", no_cards);
    cputs("\n\r\n");
                                 /* Display current line configuration. */
    cprintf(" Number of logical channels  : %d \n\r", tg_config.no_chans);
    cprintf(" Number data pacs per call   : %d \n\r", tg_config.no_pacs);
    cprintf(" Call length in seconds      : %d \n\r", tg_config.call_len);
    cprintf(" Interval between calls      : %d \n\r", tg_config.cinterval);
    cputs("\n");
                                 /* Prompt the user to change the line number
                                    being monitored. */
    cprintf("Hit CR to continue, or digit to change line no. >");
    reply = getche();

                                         /* Test the reply for a digit. */
    if (reply >= '0' && reply <= '9') {
                                         /* Digit, then change line number. */
      line = reply - '0';
      card = line / 2;
    }
    else
      exit = TRUE;

  }
  while (exit == FALSE);
```

```
                                        /* See if user wishes to configure the line*/
cputs("\n\n\r");
cputs("Hit C to alter configuration, any other to return >");
reply = getche();
if (reply != 'c' && reply != 'C')
  return;

                                        /* Yes, then prompt for new parameters. */
cputs("\n\r\n");
cputs(" Number of logical chans per link : ");
value = get_reply();
if (value > 500)
  return;
tg_config.no_chans = value;

cputs(" Enter no. data packets per call  : ");
tg_config.no_pacs = get_reply();

cputs(" Enter new call duration in secs. : ");
value = get_reply();
if (value > 3000)
  return;
tg_config.call_len = value;

cputs(" Enter new call interval in secs. : ");
value = get_reply();
if (value > 3000)
  return;
tg_config.cinterval = value;


                                        /* Ask user to confirm entries or abort. */
cputs("\n\r");
cputs("CR to confirm, A for all cards, any other key to abort >");
reply = getch();

                                        /* If reply is A, then update all lines. */
if (reply == 'a' || reply == 'A') {
  for (i = 0; i < (no_cards * 2); i++) {
    lhold(i);
                                                /* Update the master structure. */
    (*control[i]).no_chans = tg_config.no_chans;
    (*control[i]).no_pacs  = tg_config.no_pacs;
    (*control[i]).call_len  = 10 * tg_config.call_len;
    (*control[i]).cinterval = 10 * tg_config.cinterval;
    (*control[i]).stimer = (100 * tg_config.call_len) / tg_config.no_pacs;
    if ((*control[i]).stimer == 0)
      (*control[i]).stimer = 1;
    lfree(i);
  }
  return;
}
```

```
                              /* If a CR, then update line being monitored. */
  if (reply == CR) {
    lhold(line);
                                              /* Update the master structure. */
    (*control[line]).no_chans = tg_config.no_chans;
    (*control[line]).no_pacs  = tg_config.no_pacs;
    (*control[line]).call_len  = 10 * tg_config.call_len;
    (*control[line]).cinterval = 10 * tg_config.cinterval;
    (*control[line]).stimer = (100 * tg_config.call_len) / tg_config.no_pacs;
    if ((*control[line]).stimer == 0)
      (*control[line]).stimer = 1;
    lfree(line);
  }

}



/* This function called by configure().  It reads in the user's reply from
   the keyboard, converting the ASCII digits to decimal and adding them to
   a total.  The total is then returned to configure(). */

get_reply()
{
  char reply;
  int  total=0, i = 0;
                                          /* Loop around reading in digits. */
  do {
    reply = (char) getche();
                                          /* If reply a CR, then exit. */
    if (reply == 0x0D) {
      cputs("\n");
      return (total);
    }

                         /* If the reply is a digit, then update total. */
    if (reply >= '0' && reply <= '9') {
      total = (total * 10) + (reply - '0');
      if (total > 30000)
        return(30000);
    }
                         /* If something else, then reject and backspace. */
    else
      cputs("\b");
  }
  while (1 == 1);
}

/***************************************************************************/
```

APPENDIX C:

Constants and declaration include files.

Files : SYSTEM.CON

ADDRESS.CON

PACKET.CON

LINK.CON

X25TG.CON

DTYPES.C

DVARS.C

VDEFS.C

Contents:

Constants

C-1 : System constants, included in all files.

C-3 : Data structure and I/O address constants.

C-5 : Packet related constants.

C-8 : Link level related constants.

C-10 : X25TG program constants.

Data type declarations

C-11 : The work queue

C-12 : The channel control block

C-13 : WD2511 memory look up tables

C-13 : WD2511 frame buffers and control

C-14 : Master control structure

C-15 : Statistics structures

C-18 : External variable declarations

C-20 : Variable definitions.

```
/* SYSTEM.CON */

/********************/
/* SYSTEM CONSTANTS */
/********************/

#define  TRUE           1
#define  FALSE          0

/* Conditional compilation constant. */
/*==================================*/
/* Setting this constant to TRUE will result in a program which can be run
   on the PC itself for testing purposes.  When FALSE, the program will be
   compiled in a form suitable for downloading onto the X.25 cards. */

#define  TESTING        TRUE


/* Structure parameter values */
/* ========================== */

/* Number of links being used (1 or 2).  Must be 1 if running on PC. */
#if TESTING == TRUE
        #define  NOLINKS        1
#else
        #define  NOLINKS        2
#endif

/* Work que and channel sizes. */
#define  MAX_QUE        120             /* Size of the work que. */
#define  NO_CHANS       1023            /* Max no. channels per link. */
#define  MAX_CHANS      1024            /* Size of the channel status array.
                                           (used for initialization). */

/* These constants define the size of the frame buffer types declared in
   DTYPES.  Buffer size of 256 bytes gives around 240 buffers per link. */

#define  BUFF_SIZE      256             /* Frame buffer size, typically 128,
                                           256 or 512 bytes. */
#define  NO_REC_BUFFS   60              /* Number of packet receive buffers*/
#define  NO_TRAN_BUFFS  60              /* Number of transmitt buffers. */

                        /* For the buffer control structures, length is
                           defined as a factor of 2 for faster access. */
#define  CNO_REC_BUFFS  64
#define  CNO_TRAN_BUFFS 64

/* Minimum number of free buffers on a link before declaring it congested. */
#define  MIN_RFREE      10              /* Minimum free receive buffers. */
#define  MIN_TFREE      10              /* Minimum free transmit buffers. */

/* Define timer updating constants. */
#define  CALL_INT       10              /* Time interval in 10msecs between
                                           successive call requests on a link
                                           (10 gives 10 calls max per sec.)*/
#define  PAC_INT        100             /* Number of 10msec ticks between
                                           updating packet level timers. */

/*************************************************************************/
```

```
/* Process names.
   These are filled into the 'process' element of the work que to indicate
   which process needs to be run. */

#define  PAC_REC         0x00          /* Job for packet receive module. */
#define  PAC_TRANS       0x01          /* Job for packet transmit module. */
#define  LINK_LEV        0x02
#define  UPPER_LEV       0x03
#define  NUL             0x04          /* Job flushed from work queue. */

/* General names. */
#define  RECEIVE         0x00          /* Buffer name constants. */
#define  TRANSMIT        0x01


/* TEST DATA PACKET */
/* ================ */
/* This is the test data packet used by the traffic generator and is about
   120 bytes long.  XXXX is replaced by the transmit channel number. */

#define  TEST_DATA        "\n \
XXXX   TEST PACKET: \n \
This is the test data field used by the X.25 traffic generator \n \
to create data packets. \n"

/****************************************************************************/
```

```
/* ADDRESS.CON */

/*********************************/
/* STRUCTURE ADDRESS CONSTANTS */
/*********************************/

/* These constants are used by the SYSTEMS program which does the required
   initialization of the FAR data structure pointers. */
/* Note that addresses are given in segment notation, this being the Intel
   notation and the way Microsoft C allows initialization of far pointers.
   i.e. SSSS AAAA where S is the segment number and A is the offset within
   the segment. The actual physical address is calculated by shifting S
   four places to the left and adding to A. */


/* The address constants are defined conditionally.  For the normal
   configuration, the memory map for the cards is:
        0       8088 int. table at bottom, rest free for use by DOS
        1       WD1 and WD2 memory look up tables and buffers.
        2       Channels status array.
        3       8088 progs. and common stats and control structures.

   The memory for the PC is then (one card installed, 256k dual ported):
        0,1,2,3 256k private memory for DOS or other applications.
        4       Cards 8088 progs. and stats. and control structures.
        5       WD1 and WD2 memory look up tables and buffers.
        6       Channels status array.
        7       Free for DOS (transient part loaded at top of memory). */

#if TESTING != TRUE
                                        /* WD2511 look up tables. */
        #define  WD1TAB_ADDRESS 0x10000000
        #define  WD2TAB_ADDRESS 0x10008000
                                        /* WD1 buffers. */
        #define  WD1RB_ADDRESS  0x10000200
        #define  WD1TB_ADDRESS  0x10004200
                                        /* WD2 buffers. */
        #define  WD2RB_ADDRESS  0x10008200
        #define  WD2TB_ADDRESS  0x1000C200
                                        /* Channel control block. */
        #define  CHAN_ADDRESS   0x20000000
                        /* Segment address for stats and control structures*/
        #define  BASE           0x30000000
                                        /* Cards 8088 int. table. */
        #define  INT_ADDRESS    0x00000000

#else

/* For PC testing the system memory map (from the PCs point of view)
   is:  0,1,2,3,4,5     All traffic generator programs and codeview debugger.
        top of 5        Stats. and control data structures.
           6            WD2511 processor buffers and look up tables.
         6 to 7         Channels status array.
        top of 7        Used for transient part of DOS.
    The above configuration allows 384k for the codeview debugger, as the
    program was found to use up to 320k. */
```

```
        #define  WD1TAB_ADDRESS 0x60000000
        #define  WD2TAB_ADDRESS 0x60000000

        #define  WD1RB_ADDRESS  0x60000200
        #define  WD1TB_ADDRESS  0x60004200

        #define  WD2RB_ADDRESS  0x60000200
        #define  WD2TB_ADDRESS  0x60004200

                        /* Note that as 64k long, offset must be 0000 ! */
        #define  CHAN_ADDRESS   0x68400000

        #define  BASE           0x50000000
        #define  INT_ADDRESS    0x70000000

#endif


/* Memory map of the top 64k block of the X.25 card:
   Notation used is the starting to end physical address.
        30000 - 3E000   Traffic generator programs (up to 56k).
        3E000 - 3E3FF   1k heap in case used.
        3E400 - 3F7F0   5k program stack (SS, SP setup in startup routine).
        3F800 - 3FAFF   Statistics and control tables.
        3FB00 - 3FFFF   Reserved for assembler startup and int. routines. */


/* Stats and control structure offset addresses. */
/* These are the offsets from the BASE address constant defined above. */
                                  /* Stats. structures reserved memory areas.*/
#define  S1_OFFSET       0xF800           /* 256 bytes per each (128 used). */
#define  S2_OFFSET       0xF900
                                  /* Control structure reserved memory areas.*/
#define  M1_OFFSET       0xFA00           /* 128 bytes reserved, 16 used. */
#define  M2_OFFSET       0xFA80



/* Interrupt service addresses. */
/* Addresses are 32 bits (in segment:offset notation) and are loaded into the
   interrupt table by int_init.  Currently only one defined. */

#define  STDINT          0x3000FF00     /* Default register dump routine. */



/* I/O address constants. */
/* ====================== */

/* Define the base I/O addresses of the WD2511 procesors.
   Value depends on whether PC or 8088 on card is using them. */
#if TESTING == TRUE
        #define  WD1     560
        #define  WD2     560
#else
        #define  WD1     0x0040
        #define  WD2     0x0080
        #define  MUART   0x0000           /* The MUART base address. */
#endif
/****************************************************************************/
```

```
/* PACKET.CON */

/*****************************/
/* PACKET RELATED CONSTANTS */
/*****************************/

                /* PACKET TYPE IDENTIFIERS */
                /* ====== ==== =========== */

/*          From DCE to DTE  , From DTE to DCE   = Identifier    */
/*          ==== === == ===    ==== === == ===     ==========    */

/* Call set-up and clearing packets      */
#define    INCOMING_CALL                          0x0B
#define                       CALL_REQUEST        0x0B
#define    CALL_CONNECTED                         0x0F
#define                       CALL_ACCEPTED       0x0F
#define    CLR_INDICATION                         0x13
#define                       CLR_REQUEST         0x13
#define                 CLR_CONFIRM               0x17

/* Data and interrupt packets    */
#define                 DATA                      0x00   /* XXXXXXX0 */
#define                 INTERRUPT                 0x23
#define                 INT_CONFIRM               0x27

/* Flow control and reset packets        */
#define                 RR                        0x01   /* XXX00001 */
#define                 RNR                       0x05   /* XXX00101 */
#define                       REJ                 0x09   /* XXX01001 */
#define    RES_INDICATION                         0x1B
#define                       RES_REQUEST         0x1B
#define                 RES_CONFIRM               0x1F


/* Restart and diagnostic packets        */
#define    RESTART_IND                            0xFB
#define                       RESTART_REQUEST     0xFB
#define                 RESTART_CONFIRM           0xFF
#define    DIAGNOSTIC                             0xF1


/****************************************************************************/


/* Packet level name constants */
/* =========================== */

/* Assign the packet level states dummy values.
   This allows one to use the state names throughout the program, making it
   easier to read.  Note that hex values are used as the logical channel
   state variable is of type char. */

/* RESTART state : applies to all channels on a link.   */

#define  R1     0x11            /* Packet level ready    */
#define  R2     0x12            /* DTE restart sequence */
#define  R3     0x13            /* DCE restart sequence */
```

```
/* States for each logical channel :    */

/* CALL SET-UP states   */

#define  P1      0x21              /* Packet level ready, call set-up phase. */
#define  P2      0x22              /* DTE waiting for DCE to connect the call.*/
#define  P3      0x23              /* DCE waiting for DTE to accept the call. */
#define  P4      0x24              /* Data transfer state. */
#define  P5      0x25              /* Call collision.      */

/* CALL CLEARING states */
#define  P6      0x26              /* DTE clear request */
#define  P7      0x27              /* DCE clear indication */


/* Flow control states - transfer of reset packets within the p4 state  */

#define  D1      0x31              /* Flow control ready, the same as p4   */
#define  D2      0x32              /* DTE reset request    */
#define  D3      0x33              /* DCE reset indication */


/* Coding of resetting cause field in reset indication packets.*/

#define  R_DTE_ORIGINATED          0x00

#define  R_REMOTE_PROC_ERR         0x03
#define  R_LOCAL_PROC_ERR          0x05
#define  R_NETWORK_CONGEST         0x07
#define  R_INCOMP_DEST             0x11


/* Coding of clearing cause field in clear indication packet. */
/* Only those used are listed. */
#define  C_DTE_ORIGINATED          0x00

#define  C_NUMBER_BUSY             0x01
#define  C_OUT_OF_ORDER            0x09
#define  C_REMOTE_PERROR           0x11
#define  C_INCOMPAT_DEST           0x21
#define  C_LOCAL_PERROR            0x23
#define  C_NET_CONGESTION          0x05
#define  C_NOT_OBTAINABLE          0x0D


/* Coding of restarting cause field in restart packets. */

#define  LOCAL_PROC_ERR            0x01
#define  NET_CONGESTION            0x03
#define  NET_OPERATIONAL           0x07

/****************************************************************************/
```

```
/* General packet level constants. */
/* =============================== */

#define  GFI_CONST      0x10    /* Qualifier bit =0, delivery confirmation =0
                                   sequence number =01 for mod 8 sequencing*/
#define  GFI_CALL       0x50    /* As above but with the delivery bit set. */
#define  GFI_DATA       0x50    /* Delivery bit also set for data packets. */

#define  PWINDOW        4       /* Standard packet level window size. */

                        /* The called address field for call request packets.
                           Call sub-addressing fields are also to be included
                           here.  Max length is 15 decimal digits. */
#define  CALLED         1,1,1,1,9,9

                        /* Calling address field for call request packets. */
#define  CALLING        1,2,3,4


/* Offsets from the start of the buffers. */
#define  GFI            0
#define  LCN            1
#define  PTYPE          2
#define  BYTE4          3       /* Meaning if any depends on packet type. */
#define  BYTE5          4


/* DTE time limits      */

#define  T20     180            /* State r2            */
#define  T21     200    /* State p2 : waiting for call connected packet */
#define  T22     180    /* State d2 : waiting for reset confirm packet  */
#define  T23     180    /* State p6 : waiting for clear confirm packet  */

/* DCE time outs. None are used by this program as only for a DTE      */
#define  T10     60             /* State r3     */
#define  T11     180            /* State p3     */
#define  T12     60             /* State d3     */
#define  T13     60             /* State p7     */

#define  NO_CLEARS      2       /* Number of times a clear request timer may
                                   expire before sending a restart packet. */
#define  NO_RESTARTS    2       /* Number of times restart may be sent before
                                   declaring packet level out of order. */

/***************************************************************************/
```

```
/* LINK.CON */

/************************/
/* LINK LEVEL CONSTANTS */
/************************/

/* Link level register constants. */

#define  LINK_T1        0x40            /* Link level timer */
#define  LINK_N2T1      0x11            /* and retransmit counter. */

#define  LINK_A1        0x01            /* Command A field. */
#define  LINK_A2        0x03            /* Response A field. */

#define  WD_BSIZE       0x04            /*Buffer size. Set to 08 for WD chips
                                           and vary the maximum buffer size
                                           using the BUFF_SIZE constant. */

#define  CR0_DEF        0x12            /* DISC before SABM, full duplex, WD
                                           initiates link up, rec ready. */
#define  LDOWN          0x33            /* WD to disconnect link. */

#define  CR1_DEF        0x10            /* Std. mode, addr. not tri-stated.*/
#define  SEND           0x11            /* WD to send next buff if BRDY set*/


/* Link level processor register masks. */

#define  INT_MSK        0xE0            /* Interrupt masks (for SR1). */
#define  PKR_MSK        0x80
#define  XBA_MSK        0x40
#define  ERR_MSK        0x20

#define  ACKED          0x80            /* Transmit table control masks. */
#define  BRDY           0x01

#define  FRCML          0x80            /* Receive table control masks. */
#define  RECRDY         0x01


/* MUART register values. */

#define  C1             0x07            /* 1Khz timer input, 8086, P17 int,
                                           break disab, 1 stop, 8 bit char.*/
#define  C2             0x94            /* 9600x64 baud rate, clock scale of
                                           3, odd parity, parity enabled. */
#define  C3             0xA2            /* Dissab trans, normal int, int ack
                                           enable, rec dissable. */
#define  M1             0xC0            /* Baud rate output, port 2 input,
                                           timer mode, cascade timers. */
#define  CP1            0x7E            /* Port 1 configuration. */
#define  IE             0x00            /* Dissable interrupts. */
#define  P1VAL          0XFF            /* Port 1 outputs. */

#define  TSTART         0x20            /* Initial timer values. */
#define  TINTERVAL      10              /* No. of 1 msec. timer ticks between
                                           runs of the traffic generator. */
#define  WDOG           100             /* Watch dog timer in msecs. */

/***************************************************************************/
```

```
/* Define the register offsets from the base addresses in ADDRESS.CON. */

#define  CR0          0      /* Control and monitoring registers. */
#define  CR1          1
#define  SR0          2              /* Status registers. */
#define  SR1          3
#define  SR2          4
#define  ER0          5
#define  CHAIN_MON    6              /* Receiver monitor. */
#define  REC_CFIELD   7
#define  T1REG        8              /* Frame level timer and counter. */
#define  N2T1         9
#define  TLOOK_HI     10             /* DMA table set-up. */
#define  TLOOK_LO     11
#define  SIZE_REG     12             /* Buffer size. */
#define  XMT_COMM     14             /* 'A' field registers. */
#define  XMT_RESP     15


/* MUART addresses and constants. */

#define  COMM1        0              /* Address offsets from the base. */
#define  COMM2        1
#define  COMM3        2
#define  MODE         3
#define  CONTP1       4
#define  INTEN        5
#define  INTADD       6
#define  RECBUFF      7
#define  PORT1        8
#define  PORT2        9
#define  TIMER1       10
#define  TIMER2       11
#define  TIMER3       12
#define  TIMER4       13
#define  TIMER5       14
#define  STATUS       15


/****************************************************************************/
```

```
/* X25TG.CON */

/***************************
 * X25TG PROGRAM CONSTANTS *
 ***************************/

/* Default system configuration. */

#define  CALL_LEN        10            /* Default call length in seconds. */
#define  CINTERVAL       2             /* Time interval (in secs) between a
                                          clear and next call on channel. */
#define  NO_PACS         30            /* Number of data packets to try and
                                          send per call made. */
#define  C_CHANS         10            /* Number of logical chans to use. */


/* Card I/O addresses. */

#define  PCIOBASE        0x220         /* The base address of the cards in
                                          the PC's I/O map (544 decimal). */
#define  PCIOLEN         0x20          /* The lenght of each I/O range. */
                                   /* Each 32 bytes long and 6 cards selectable
                                      i.e. from 0x220 (544) to 0x2C0 (736).  */


/* I/O control addresses relative to the base address. */

#define  RESET           0
#define  NMI             1


/* Character constants. */

#define  CR              0x0D
#define  ESC             0x1B

                         /* Display update period in seconds. */
#define  T_UPDATE        2

/****************************************************************************/
```

```
/* DTYPES.C */

/****************************/
/* DATA TYPE DECLARATIONS */
/****************************/

/* This module declares all the data types used by all the X.25 programs.
   It should be included in all program files prior to the variable
   declarations. */

/* Several structures are referenced via far pointers, to allow them to be
   easily located at specific far addresses.  The structures are located
   by systems, using pointer addresses from the ADDRESS.CON file.
   Where possible structures have a size in powers of 2 to allow the compiler
   to use a shift statement instead of a multiply when accessing elements. */




/*******************
 * The work_queue *
 *****************/
/* This queue consists of a list of jobs to be done.
   Jobs can be added to the queue at any time by the following modules :
      traffgen; pacrec; pactrans; linklev; systems
   The systems program goes through the queue, dispatching the appropriate
   modules to the processor for execution.
   i.e. the work queue is basically a job FIFO.
   work_in points to the last job entered and
   work_out points to the last job processed.    */

/* The queue size is determined by the constant MAX_QUE, which is normally
   set to 100.  The queue is located in near memory and is accessed directly.
   The structure is 8 bytes long to ensure fastest access. */

struct queue {
        char process;           /* Which processing module the job is for. */
        char ptype;             /* The packet type or event code. */
        int  link;              /* Link level processor no. 1 or 2. */
        int  channel;           /* The logical channel number. */
        int  buffer;            /* The buffer number. */
};
```

```
/******************************
 * The channel control block *
 ******************************/
/* This is an array of logical channels structures.
   The array size is determined by MAX_CHANS, and is usually 1024.
   The array is two dimensional i.e. there is one array of logical channels
   for each X.25 link.  */

/* The array is usually located in the third 64k block of the X.25 card's
   memory and is typically 64K long.  (The structure length is adjusted  to
   be 32 bytes long for the fastest access time.) */

struct chan_state {
        char state;                 /* Stores the state of the logical channel.*/
        char trans_state;           /* Stores the data transfer sub state. */
        char dce_rdy;               /* False if DCE is not ready. */

                                    /* Receive and send packet sequence nos. */
        char dte_vs;                /* Send state variable. */
        char dte_vr;                /* Receive state variable. */
        char K                      /* Number of outstanding packets (no. packs.
                                       sent but not yet acknowledged by DCE). */
        char R                      /* Number of packets received for which
                                       acknowledgement is still to be sent. */
        char mbit;                  /* More data bit. */

        char add_len;               /* Calling and called DTE address lengths. */
        char dte_adds[7];           /* Calling DTE addresses.  SAPONET uses
                                       recommendation X121 which specifies a max
                                       of 14 BCD digits in an address. */

        char tactive;               /* TRUE: timer for channel is active. */
        int  timer;                 /* Reply time-out timer for the channel. */
        char tcount;                /* Number of times the timer may expire. */

        char window;                /* Packet level window size for send and
                                       receive channels. */
        char no_qued;               /* Number of qued packets on the channel. */
        int  buffno;                /* The number of the last packet sent. */

        int  send_timer;            /* Test packet send timer, decremented every
                                       10 msecs.  Test packet sent when zero. */
        int  clear_timer;           /* Remaining call duration in 10msecs. units
                                       Call cleared when timer expires. */
        int  call_timer;            /* Remaining delay before a call request. */

        char spare;

};
```

```
/*********************************
 * WD2511 memory look up tables *
 *********************************/
/* This data type is for the transmit and receive memory look up tables
   used by the WD2511 link level processors. */

struct buff_table {
        char state;             /* The state of the memory buffer. */

        char sadr_hi;           /*The memory segment address of the buffer.*/
        char sadr_lo;           /* (WD2511 stores high byte first.) */

        char count_hi;          /* Counter for the size of the frame. */
        char count_lo;          /* (Again, high byte comes first.) */

        int  buffer;            /* Number of the buffer segment being used.*/
        char spare;
};


/* WD2511 error counters:
   ======================
  The six error counters follow contiguously after the rlook table in memory.
  The link level program occasionally reads them, transferring the values to
  the statistics data structure. */

struct errtable {
        char fcs_err;           /*Frames received with bad check sequences.*/
        char sframes;           /* Short frames received. */
        char tl_out;            /* Number of times T1 ran out. */
        char spare;
        char rej_rec;           /* Number of reject frames received. */
        char rej_trans;         /* Number of reject frames transmitted. */
        int  filler;
};


/************************
 * WD2511 frame buffers *
 ************************
 These consist of two large arrays per link level processor, one for transmit
 and one for receive buffers.  The transmitt and receive buffers are each
 typically 16K long, giving a total of 120 buffers per link (for a 256 byte
 data field). Thus the traffic generator can easily cope with high peak data
 rates and support a large number of logical channels. */

struct buff_struc {
        char gfi;               /* Packet general format identifier. */
        char lcn;               /* The logical channel number. */
        char ptype;             /* The packet type identifier. */

        char data[BUFF_SIZE];           /* The user data field
                                           (typically 256 bytes long). */
};
```

```
/* WD2511 buffer control:
   ====================
   This data type defines all the control information for the data buffers
   defined above.  There is a one to one correspondence between elements of
   the control array and the actual buffers. */

struct buff_control {
        char busy;                /* True if buffer contains a frame which is
                                     still to be processed.  ie. True if buffer
                                     is reserved by packet or upper levels. */
        char send;                /* True if buffer contains a packet to be
                                     transmitted. Cleared once acked by the DCE
                                     or if flushed after a channel reset. */
        int  length;              /* Contains the length of the frame.      */
        int  channel;             /* The number of the logical channel
                                     associated with the buffer. */

        char source;              /* True if packet has a data field, from or
                                     intended for the upper level (traffgen).*/
        char spare;
};



/*************************
 * Master control type *
 *************************
 This is the main control structure of the program and generally determins
 how the program is run.  Parameters are filled in by the PC, thus providing
 a means whereby program characteristics (test call duration, number of test
 packets to be sent, etc.) may be changed during run time. */

struct mcontrol {
                        /* Main control from the PC. */
        char lback;               /* TRUE: receiver channel configured as if
                                     receiving from a DCE. (lcn = NO_CHANS -lcn
                                     +1).  Set during loop back tests. */
        char pad;                 /* TRUE: pac_rec module to pass data items up
                                     to the upper level (used for pad mode). */

        char exit;                /* TRUE: clear all calls and then the link.*/
        char pcread;              /* True if PC program wishes to read the
                                     card's statistics. */

                        /* User configurable parameters. */
        int  no_chans;            /* Max number of logical channels to use. */
        int  no_pacs;             /* Number of test packets to try and send
                                     per call. */
        int  call_len;            /* Call length in 0.1 sec units. */
                                  /* (User can enter from 1 sec to 50 mins.) */
        int  cinterval;           /* Time in 0.1 sec units between a clear and
                                     the next call requests on a channel. */

        int  stimer;              /* Calculated time 10msecs units between test
                                     data packets sent on a channel. */
        int  cspace;              /* Spacing in 10 msec units between call
                                     request packets on a link. */
};
```

```
/*************************
 * Statistics data types *
 *************************/
/* This structure contains the complete statistics of the traffic generator.
   The three component types are defined first.
   This structure is accessed by the X25PC program, which displays the
   statistics on the console. */


/* System status data:
   ===================
   This two dimensional data structure contains the status of each link.
   In particular, it contains data about the status and activity of each of
   packet levels. */

struct  sysstats  {
                         /* General status. */
        int  pstate;             /* State of the packet level ie. r1,2 or 3 */
        int  active;             /* Number of active channels on the link.  */
                                 /* ie. No. chans. in data transfer state P4*/

        int  stopped;            /* True when stopped so PC can read stats. */
        int  init;               /* TRUE:  card has finished doing system
                                           initialization. */

                         /* Buffer status. */
        int  rcongested;         /* Set true when receive buffer getting
                                    congested  (less than MIN_RBUFFS buffers
                                    still unallocated). */
        int  tcongested;         /* True when transmit buffer congested. */

        int  no_rbuffs;          /* Number of available (free) receive buffers
                                    per link. */
        int  no_tbuffs;          /* Number free transmit buffers per link. */

                         /* Packet received error counters. */
        int  state_err;          /* Channel found to be in wrong state. */
        int  seq_err;            /* Number of packets received with invalid
                                    sequence numbers. */
        int  rec_err;            /* Counter for invalid packets received. */
                                 /* e.g. packets with an invalid gfi or type
                                    code in the header. */

        long prog_flag;          /* Program flag to monitor function calling
                                    during program testing. */
        int  buff_len;           /* Length of test packet data field. */

        int  spare[2];           /* Make structure 32 bytes for fast access.*/
};
```

```
/* Packet received status data:
   ============================
   This structure keeps the totals of packets received on the link.
   (i.e. packets received from the device being tested.)
   These totals indicate the packet level activity. */

struct  pacstats  {

                        /* Packet level totals. */
        long pac_rec;           /* Total number of packets received. */
        long pac_sent;          /* Total number of packets sent. */

        long calls;     /* Total number of calls set up by the traffic
                           generator. (i.e. total call connected packets.) */
        long data;              /* Total valid data packets received. */

                        /* Total received control packets. */
        int  rr;                /* Total receiver ready packets. */
        int  resets;            /* Total logical channel resets. */
        int  restarts;          /* Total packet level restarts. */
        int  int_pacs;          /* Number of interrupt packets received. */
        int  diagnostics;       /* Number of diagnostic packets received */

                        /* Total clears received. */
        int  dte_clears;        /* Total clears initiated by the traffic
                                   generator. (i.e. DTE originated)   */
        int  dce_clears;        /* Total clears initiated by the DCE. */
        int  congest_clears;    /* Number of  calls cleared by DCE due to
                                   system congestion. */
        int  dest_clears;       /* Number clears due to destination probs.*/

                /* Diagnostic causes in clear, reset, restart and diagnostic
                   packets received from the DCE. */
        int  dseq_err;          /* Invalid send or receive sequence nos. */
        int  dstate_err;        /* Packet type invalid for state. */
        int  drec_err;          /* Packet structure problems. */
        int  d_texp;            /* DCE timer expired. */
        int  d_callprob;        /* Call set up problems. */

        int  ptimeouts;         /* Number of times packet level timers
                                   expired. */

        int  spare[9];          /* Make structure 64 bytes long. */
};
```

```
/* Link level status data:
   ======================
   This data type keeps statistics of activity on the two link levels. */

struct  lstats  {
                                        /* State. */
        int  lstate;                /* True if the link is up. (SR20) */
        int  resets;                /* Number of resets (SABMs received). */
        int  t1_timeout;            /* Number of times T1 timer expired. */

                                        /* Frames received. */
        int  rec_rnr;               /* Number of RNR frames from DCE. */
        int  rej_rec;               /* Number of reject frames received. */
        int  fcs_err;               /* Frame check sequence error. */
        int  shortf;                /* Received short frames (<32 bits). */
        int  frmr_rec;

                                        /* Frames transmitted. */
        long byte_tot;              /* Total bytes transmitted. */
        int  rej_trans;             /* Number of reject frames transmitted. */
        int  frmr_trans;            /* Number of frame rejects sent. */

                                        /* WD2511 problems. */
        int  mem_notrdy;            /* Receiver overrun or trans. underruns. */
        int  rlook_err;             /* Receiver look up table not ready. */
        int  activity;              /* Percentage of time link (rec) is active.*/

        int  spare;
};


struct statistics {
                /* The structure is made up of the statistics structures
                   declared above.  (Total length is 128 bytes.). */
        struct sysstats  system;
        struct pacstats  packet;
        struct lstats    link;
};

/****************************************************************************/
```

```
/* DVARS.C */

/*********************************/
/* EXTERNAL VARIABLE DECLARATIONS */
/*********************************/

/* This module declares all the variables used by the X.25 programs
   running on the X.25 card.  (The program for the PC, X25TG.C, has its
   own variable declarations.)  All variables are declared as external. */


/* The work queue */
/* ============== */
                                          /* The work queue. */
extern struct queue  work_que[MAX_QUE];

extern int work_out;              /* Points to last job processed. */
extern int work_in;               /* Points to last job entered. */


/* The channel control block */
/* ========================= */

extern struct chan_state  (far *chans)[2][MAX_CHANS];


/* WD2511 memory look up tables */
/* ============================ */

                          /* For two links, an array of pointers to an array of
                             buff_tables is required. */
extern struct  buff_table  (far *tlook[2])[8];
extern struct  buff_table  (far *rlook[2])[8];

extern int tnext[2];              /* Next expected tlook segment number. */
extern int rnext[2];              /* Next expected rlook segment number.
                                     ie.segment into which next received packet
                                          will be filled in by the WD2511. */

/* WD2511 error table */
/* ================== */
                                  /* One structure per link level processor. */
extern struct errtable  (far *wd_err[2]);


/* WD2511 frame buffers */
/* ==================== */

                          /* Declare an array of far pointers to an array of
                             buffers with structure type buff_struc. */
extern struct buff_struc (far *tbuff[2])[NO_TRAN_BUFFS];
extern struct buff_struc (far *rbuff[2])[NO_REC_BUFFS];
```

```
                         /* WD2511 buffer control.  Contains the control data
                            for the frame buffers above. */
extern struct buff_control  tcontrol[2][CNO_TRAN_BUFFS];
extern struct buff_control  rcontrol[2][CNO_REC_BUFFS];

extern int tbuff_point[2];      /* Pointer to last free transmit buffer. */
extern int rbuff_point[2];      /* Pointer to the last free receive buffer.*/


/* WD2511 send queue */
/* ================= */
/* This is a queue of frames ready to be transmitted by the link level
   processors.  The que is basically a FIFO, with data frames being added by
   the packet level and removed by the link level once acked by the DCE. */

extern int send_que[2][CNO_TRAN_BUFFS];         /* List of buffer numbers. */

extern int send_in[2];  /* Points to the last frame added to the que. */
extern int sending[2];  /* Points to last frame given to WD2511 processor. */



/* Master control structure */
/* ======================== */
                                /* One structure per link. i.e. each link can
                                   be independently configured. */
extern struct mcontrol  (far *master[2]);


/* Statistics structure */
/* ==================== */
                                /* One stats structure for each link level.*/
extern struct statistics  (far *stats[2]);


/* 8088 interrupt table */
/* ==================== */
                                /* An array of 32 bit int address pointers.*/
extern long (far *int88)[64];


/* Test packet */
/* =========== */
                                /* Test packet array declaration. */
extern char test_packet[BUFF_SIZE];


/***************************************************************************/
```

```
/* VDEFS.C */
/************************/
/* VARIABLE DEFINITIONS */
/************************/

/* This file should be included in one of the program files (usually SYSTEMS)
  to define the extern variables. */

/* The work queue */
/* ============== */
struct queue work_que[MAX_QUE];
int  work_in;
int  work_out;

/* The channel control block */
/* ========================= */
struct chan_state (far *chans)[2][MAX_CHANS];


/* WD2511 memory look up tables */
/* ============================ */
struct buff_table  (far *tlook[2])[8];
struct buff_table  (far *rlook[2])[8];

int  tnext[2], rnext[2];

struct errtable   (far *wd_err[2]);

/* WD2511 frame buffers and control */
/* ================================ */
struct buff_struc  (far *tbuff[2])[NO_TRAN_BUFFS];
struct buff_struc  (far *rbuff[2])[NO_REC_BUFFS];

struct buff_control  tcontrol[2][CNO_TRAN_BUFFS];
struct buff_control  rcontrol[2][CNO_REC_BUFFS];

int rbuff_point[2], tbuff_point[2];

/* WD2511 send queue */
/* ================= */
int  send_que[2][CNO_TRAN_BUFFS];
int  send_in[2], sending[2];


/* Control and statistics structures */
/* ================================= */
struct mcontrol  (far *master[2]);
struct statistics   (far *stats[2]);

/* 8088 interrupt table */
long  (far *int88)[64];

/* Test packet */
char temp_pac[] = {TEST_DATA};
char test_packet[BUFF_SIZE];

/*****************************************************************************/
```

APPENDIX D:

The executive and traffic generator programs.

Files : SYSTEMS.C

TRAFFGEN.C

Contents:

SYSTEMS

D-1 : main — The main function of the executive and hence of the program as a whole.

D-3 : Initialization functions

D-8 : Buffer control functions

Functions to allocate free and flush the buffers.

D-12 : System and packet level timers

TRAFFGEN

D-17 : traffic_gen

D-17 : Call, clear and send checking functions

Functions to update traffic generator timers and send the appropriate packets.

```
/* SYSTEMS.C */

/*****************************************/
/* X.25 TRAFFIC GENERATOR - MAIN PROGRAM */
/*****************************************/

/* This module is the main program or executive of the traffic generator.
   It consists of the following sections:
               The main function which acts as a simple job dispatcher.
               All the system initialization functions.
               All the buffer control functions.
               All the system and packet level timer functions.

   At start up the far pointers are initialized, as are the buffers, WD2511
   tables and the channel control block.  From then on it loops around
   continuously calling other program modules as required.
   The module is the only one to manipulate the frame buffers, allocating and
   freeing them as requested.  It also looks after all the system timers and
   communications with the PC. */

#include "SYSTEM.CON"
#include "ADDRESS.CON"
#include "PACKET.CON"
#include "LINK.CON"

#include "DTYPES.C"
                        /* Variable definitions included in this module. */
#include "VDEFS.C"

#if TESTING != TRUE
  #include "conio.h"
#endif



/* The main program.  Note that it is given a different name if called from
   X25TG as only one main() allowed per program. */

#if TESTING == TRUE
  x25_card()
  {
#else
  main()
  {
                                /* Perform system initialization. */
  initialize();
                                /* Loop around continuously doing jobs. */
  while  ((*master[1]).exit == FALSE)  {
#endif

    link_level();       /* See if anything is happening on the link level. */

    sys_timers();       /* Check the system timer and call the traffic
                           generator or packet level timers if required. */

    pc_signal();        /* See if the PC wants to read the stats structure.*/
```

```
                              /* See if there are any entries in the work que. */
    if (work_out != work_in) {
                                      /* First point to job to be done. */
      work_out = (++work_out == MAX_QUE) ? 0 : work_out;
                                              /* Do the particular job. */
      switch  ( (int) work_que[work_out].process)  {
                case PAC_REC:
                        pac_received();
                        break;
                case PAC_TRANS:
                        pac_transmit();
                        break;
                        /* Upper level- not used in traffic gen. */
                case UPPER_LEV:
                        upper_lev();
                        break;
                        /* NULL: entry scrubed due to queue flushing. */
                case NUL:
                        break;
      }
    }

#if TESTING != TRUE
  }
                        /* If PC control program wants the card to stop, then
                           shut down the links in an orderly manner. */
  shut_down();
#endif

}

/***************************************************************************/
```

```
/****************************
 * INITIALIZATION FUNCTIONS *
 ****************************/

/* This function performs a complete system initialization.
   (Note that the order in which the functions are called is important). */

initialize()
{
  int  linkno, i, no;

  mem_init();                 /* Initialize memory for parity checking.
                                 (memory may not all be dual ported.) */
#if TESTING != TRUE
  int_init();                 /* Set up the 8088 interrupt table. */
#endif

  locate();                   /* Locate all the far data structures. */

  rbuff_init();               /* Initialize the receiver buffers. */
  tbuff_init();               /* Initialize the transmit buffers and load
                                 in the test data packet. */

                              /* Initialize all chans status structures. */
  for (linkno = 0; linkno < NOLINKS; linkno++) {
    no = (*master[linkno]).no_chans + 40;
    for (i = NO_CHANS; i > (NO_CHANS - no); i--)
      chan_init (linkno, i);
    for (i = 0; i < no; i++)
      chan_init (linkno, i);
  }

  work_in = work_out = 0;     /* Initialize work queue pointers. */

  muart_init();               /* Set up traffic generator and packet level
                                 timers, as well as physical interface. */

  wd_table();                 /* Initialize the WD2511 look up tables. */
  wd_init();                  /* Program the WD2511 registers. */
                              /* Returns once the links are up. */
  (*stats[0]).system.pstate = R1;
  (*stats[1]).system.pstate = R1;

                              /* Tell the PC that finished initialization*/
  (*stats[0]).system.init = TRUE;
}
```

```
/* This function locates the look up tables and buffer
   arrays used by the WD2511 processors.  It also locates
   the channels array and the 8088 interrupt array. */

locate()
{
                                          /* Locate the channel status array.*/
  chans = (struct chan_state far *) CHAN_ADDRESS;

                                          /* Locate look-up tables for WD1. */
  tlook[0] = (struct buff_table far *) WD1TAB_ADDRESS;
  rlook[0] = (struct buff_table far *) (WD1TAB_ADDRESS + 64);
  wd_err[0] = (struct errtable far *) (WD1TAB_ADDRESS + 128);

                                          /* Locate look-up tables for WD2. */
  tlook[1] = (struct buff_table far *) WD2TAB_ADDRESS;
  rlook[1] = (struct buff_table far *) (WD2TAB_ADDRESS + 64);
  wd_err[1] = (struct errtable far *) (WD2TAB_ADDRESS + 128);

                                          /* Locate the buffers for WD1. */
  rbuff[0] = (struct buff_struc far *) WD1RB_ADDRESS;
  tbuff[0] = (struct buff_struc far *) WD1TB_ADDRESS;

                                          /* Locate the buffers for WD2. */
  rbuff[1] = (struct buff_struc far *) WD2RB_ADDRESS;
  tbuff[1] = (struct buff_struc far *) WD2TB_ADDRESS;

                                  /* Locate the master control structure. */
  master[0] = (struct mcontrol far *) (BASE + M1_OFFSET);
  master[1] = (struct mcontrol far *) (BASE + M2_OFFSET);

                          /* Locate the system system statistics structure.*/
  stats[0] = (struct statistics far *) (BASE + S1_OFFSET);
  stats[1] = (struct statistics far *) (BASE + S2_OFFSET);

                                  /* Locate 8088 interrupt jump table. */
  int88 = (struct cpu_int far *)INT_ADDRESS;

}
```

```
/* This function intializes the receiver buffer and it's control structure.*/
rbuff_init()
{
  register int  linkno, buffno;

                                /* Loop through all the receive buffers. */
  for (linkno = 0; linkno < NOLINKS; linkno++) {
    for (buffno = 0; buffno < NO_REC_BUFFS; buffno++) {

                                /* Initialize the buffer control data. */
      rcontrol[linkno][buffno].busy = FALSE;     /* Buffer is free. */
      rcontrol[linkno][buffno].send = FALSE;
      rcontrol[linkno][buffno].channel = 0;      /* 0 for channel unassigned*/
      rcontrol[linkno][buffno].length = 0;

    }
                                /* Initialize the buffer pointer. */
    rbuff_point[linkno] = 0;
    (*stats[0]).system.no_rbuffs = NO_REC_BUFFS;
    (*stats[1]).system.no_rbuffs = NO_REC_BUFFS;
  }
}


/* This function loads the test data packet into all the transmit buffers.
   It also initializes the transmit buffer control structure. */

tbuff_init()
{
  register int  linkno, buffno;
  int  len, i;
                                /* Initialize extern test packet structure.*/
  len = sizeof(temp_pac);
  for  (i = 0; i < len; i++)
    test_packet[i] = temp_pac[i];

                                /* Loop through all the transmit buffers. */
  for (linkno = 0; linkno < NOLINKS; linkno++) {
    for (buffno = 0; buffno < NO_TRAN_BUFFS; buffno++)  {

                                /* Initialize the buffer control data. */
      tcontrol[linkno][buffno].busy = FALSE;     /* Buffer is free. */
      tcontrol[linkno][buffno].send = FALSE;
      tcontrol[linkno][buffno].channel = 0;      /* 0 for channel unassigned*/
      tcontrol[linkno][buffno].length = 0;

                                /* Now load in the test data packet. */
      for  (i = 0; (i < BUFF_SIZE) && (i < len); i++)
        (*tbuff[linkno])[buffno].data[i] = test_packet[i];
    }
                                        /* Initialize the buffer pointer. */
    tbuff_point[linkno] = 0;
    (*stats[0]).system.no_tbuffs = NO_TRAN_BUFFS;
    (*stats[1]).system.no_tbuffs = NO_TRAN_BUFFS;
    (*stats[linkno]).system.buff_len = len;
  }
}
```

```
/* This function does a complete initialization of the chans
   status array for a particular channel. */

chan_init (linkno, channo)
  register int linkno, channo;
{
                                   /* Set state to P1, transfer state to D1. */
  (*chans)[linkno][channo].state = P1;
  (*chans)[linkno][channo].trans_state = D1;
  (*chans)[linkno][channo].dce_rdy = TRUE;
                                   /* Reset all the sequence counters. */
  (*chans)[linkno][channo].dte_pr = 0;
  (*chans)[linkno][channo].dte_ps = 0;
  (*chans)[linkno][channo].old_dce_pr = 0;
  (*chans)[linkno][channo].old_dce_ps = 0;
  (*chans)[linkno][channo].mbit = 0;

                                   /* Deactivate all the timers. */
  (*chans)[linkno][channo].tactive = FALSE;
                                   /* Reset the queueing indicators. */
  (*chans)[linkno][channo].window = PWINDOW;
  (*chans)[linkno][channo].no_qued = 0;
  (*chans)[linkno][channo].buffno = 0;
                                   /* Initialize the send and clear timers. */
  (*chans)[linkno][channo].send_timer = 1;
  (*chans)[linkno][channo].clear_timer = (*master[linkno]).call_len;
  (*chans)[linkno][channo].call_timer = 1;      /* Start immediatly. */
}


mem_init()
{
}


/* This function loads the interrupt service routine addresses
   into the look up tables used by the 8088. */
int_init()
{
  register i;
                         /* Fill all segments with the address of the default
                            service routine.  (STDINT does a register dump).*/
  for (i = 0; i < 64; i++)
    (*int88)[i] = STDINT;
}
```

```
/* This function programs the MUART, setting up the system and packet level
   timers.  The physical interface lines are also initialized. */

muart_init()
{
#if TESTING != TRUE
                                          /* Program the command registers. */
  outp ( MUART + COMM1 , C1);
  outp ( MUART + COMM2 , C2);
  outp ( MUART + COMM3 , C3);
  outp ( MUART + MODE , M1);
  outp ( MUART + INTEN , IE);             /* Interrupt mask. */
  outp ( MUART + CONTP1 , CP1);           /* Port 1 configuration. */
  outp ( MUART + PORT1 , P1VAL);
                                                  /* Timer programming. */
  outp ( MUART + TIMER1 , WDOG);          /* Watch dog timer. */
  outp ( MUART + TIMER3 , TSTART);        /* System timer. */
                                          /* (Timers 2, 4 and 5 are spare.) */
#endif
}


#if TESTING != TRUE
  _setargv()     /* This nul function is used to suppress the library command
                    line processing function, which is not required here. */
  {
  }

  _setenvp()     /* This nul function is used to suppress the library
                    envirnoment processing function. */
  {
  }
#endif

/***************************************************************************/
```

```
/******************
 * BUFFER CONTROL *
 ******************/

/* These functions manage the memory buffers via the buffer control
   structures.  They are the only routines to manipulate the tcontrol
   and rcontrol structures. */


/* This function finds the next free transmit or receive buffer and returns
   its buffer number.  It keeps track of the tbuff_point and rbuff_point
   pointers and is the only function to use them.  It is also the only
   function to allocate buffers. */

buff_find (type,linkno, channo)
  char type;
  register int linkno;
  int  channo;
{
  register pointer;
  int temp;
                        /* Program flag used to monitor buffer allocation. */
  (*stats[0]).system.prog_flag += 1;

                        /* Type determines which buffers we are looking at.*/
  if (type == TRANSMIT) {

    pointer = tbuff_point[linkno];
    do
                /* Increment the pointer to look at next transmit buffer. */
      pointer = (++pointer == NO_TRAN_BUFFS) ? 0 : pointer;
                /* Repeat until find a buffer which is not busy. */
    while  (tcontrol[linkno][pointer].busy == TRUE);

                                /* Having found the next free transmit
                                   buffer, declare it as being busy. */
    tcontrol[linkno][pointer].busy = TRUE;
                                /* Fill in the channel number. */
    tcontrol[linkno][pointer].channel = channo;

                                /* Update the number of free buffers left and
                                   see if link level has a backlog of packets
                                   to be sent. */
    temp = --(*stats[linkno]).system.no_tbuffs;
                                /*Determine if buffer is getting congested.*/
    if (temp <= MIN_TFREE)
      (*stats[linkno]).system.tcongested = TRUE;

    tbuff_point[linkno] = pointer;
  }
```

```
  else {        /* Similar do loop to find the next receive buffer. */

    pointer = rbuff_point[linkno];
    do
                /* Increment the pointer to look at the next receive buffer*/
      pointer = (++pointer == NO_REC_BUFFS) ? 0 : pointer;
                /* Repeat until find a buffer which is not busy. */
    while  (rcontrol[linkno][pointer].busy == TRUE);

                                /* Having found the next free receive buffer,
                                   declare it as being busy. */
    rcontrol[linkno][pointer].busy = TRUE;
                                /* Fill in the channel number. */
    rcontrol[linkno][pointer].channel = channo;

                                /* Update the number of free buffers left and
                                   see if receiver is still ready. */
    temp = --(*stats[linkno]).system.no_rbuffs;
                                /* Determin if buffer is getting congested.*/
    if (temp <= MIN_RFREE)
      (*stats[linkno]).system.rcongested = TRUE;

    rbuff_point[linkno] = pointer;
  }

                          /* Finally, return the buffer pointer. */
  return(pointer);
}



/* This function frees a buffer after it has been processed.  The system
   status structure is updated. */

buff_free (type, linkno, pointer)
  char type;
  register int linkno, pointer;
{
  int temp;

  if  (type == TRANSMIT)  {

    if (tcontrol[linkno][pointer].busy != FALSE) {
                                /* Free the buffer. */
      tcontrol[linkno][pointer].busy = FALSE;
                                /* Clear the other buffer control entries. */
      tcontrol[linkno][pointer].send = FALSE;
      tcontrol[linkno][pointer].length = 0;
      tcontrol[linkno][pointer].channel = 0;
                                /* Update the number of free buffers left and
                                   see if link level still has a backlog of
                                   packets to be sent. */
      temp = ++(*stats[linkno]).system.no_tbuffs;
                                /* See if buffer is now uncongested. */
      if (temp > MIN_TFREE)
        (*stats[linkno]).system.tcongested = FALSE;
    }
  }
```

```
  else {                            /* Similarly for freeing a receive buffer. */

    if (rcontrol[linkno][pointer].busy != FALSE) {
                                    /* Free the buffer. */
      rcontrol[linkno][pointer].busy = FALSE;
                                    /* Clear the other buffer control entries. */
      rcontrol[linkno][pointer].send = FALSE;
      rcontrol[linkno][pointer].length = 0;
      rcontrol[linkno][pointer].channel = 0;
                                    /* Update the number of free buffers left and
                                       see if receiver is ready. */
      temp = ++(*stats[linkno]).system.no_rbuffs;
                                    /* See if buffer is now uncongested. */
      if (temp > MIN_RFREE)
        (*stats[linkno]).system.rcongested = FALSE;
    }
  }

}


/* This function flushes all packets from the transmit and send ques.
   It is called after a call has been cleared. */

buff_flush (linkno, channo)
  register int linkno;
  int channo;
{
  register int buffno;
  int pointer;

  link_level();
                                    /* Look through all the transmit buffers. */
  for (buffno = 0; buffno < NO_TRAN_BUFFS; buffno++) {
                                    /* Checking for this channel number. */
    if  ( (tcontrol[linkno][buffno].channel == channo) &&
                (tcontrol[linkno][buffno].busy == TRUE) )  {

                                                    /* Free the buffer. */
      if (tcontrol[linkno][buffno].send == FALSE)
        buff_free (TRANSMIT, linkno, buffno);

                                    /* Buffer in send que, clear the send flag.
                                       (Link level checks this flag and will free
                                       the buffer when it processes it.) */
      if (tcontrol[linkno][buffno].send == TRUE)
        tcontrol[linkno][buffno].send = FALSE;
    }
  }
```

```
                                        /* Must also flush the packet transmit work
                                           que so that channel completely cleared. */
  pointer = work_out;
  while (pointer != work_in) {
                                        /* Point to next work queue entry. */
    pointer = (++pointer == MAX_QUE) ? 0 : pointer;
                                        /* Flush if a entry for this channel. */
    if ( (work_que[pointer].process == PAC_TRANS) &&
                (work_que[pointer].link == linkno) &&
                        (work_que[pointer].channel == channo) )
      work_que[pointer].process == NUL;
  }

}


/* This function does a complete flushing of all logical channels on a link.
   It is called by the sys_restart function.  As well as flushing the
   transmit buffers, the work ques are flushed. */

flush_all (linkno)
  register int linkno;
{
  register int buffno;
  int pointer;
                                        /* Look through all the transmit buffers. */
  for (buffno = 0; buffno < NO_TRAN_BUFFS; buffno++) {
                                                /* Free the buffer. */
      buff_free (TRANSMIT, linkno, buffno);
  }
                                        /* Scrap all entries on the link send queue.
                                           (The buffers having been freed above.) */
  send_in[linkno] = sending[linkno];

                                        /* Flush packet transmit work que. */
  pointer = work_out;
  while (pointer != work_in) {
                                                /* Point to next work que entry. */
    pointer = (++pointer == MAX_QUE) ? 0 : pointer;
                                                /* See if a entry for this channel.*/
    if ( (work_que[pointer].process == PAC_TRANS) &&
                (work_que[pointer].link == linkno) )   {
      work_que[pointer].process == NUL;
    }
  }

}

/****************************************************************************/
```

```
/*****************************
 * SYSTEM TIMERS AND CONTROL *
 *****************************/


/* This function gets the time in from the MUART and calls the traffic
   generator module if required. */

sys_timers()
{
                          /* Use static variables to provide private, permanent
                             storage of timers. */
  static int call_timer = CALL_INT;
  static int packet_timer = PAC_INT;
  static int old_time = 0;
  int time;
  char call = FALSE;


#if TESTING != TRUE
                                  /* Check if 10msecs has passed. */
  time = TSTART - inp(MUART + TIMER3);
  if (time >= TINTERVAL) {
                                  /* Reset the system timer. */
    time = time - TINTERVAL;
    time = (time > 10) ? 10 : time;
    outp (MUART + TIMER3 , (TSTART - time));
#endif

                          /* See if should check for call timers. */
    if (--call_timer == 0) {
      call_timer = CALL_INT;
      call = TRUE;
    }
                                  /* Call the traffic generator. */
    traffic_gen (call);

                                  /* Update packet level timers once a sec. */
    if (--packet_timer == 0) {
      packet_timer = PAC_INT;
      ptimer_check();
    }

#if TESTING != TRUE
  }                               /* Reset the watch dog timer. */
  outp (MUART + TIMER1 , WDOG);
#endif

}
```

```
/* Halts the program to allow PC to update display. */
pc_signal()
{
#if TESTING != TRUE
  if (((*master[0]).pcread == TRUE) || ((*master[1]).pcread == TRUE)) {
                                          /* Signal confirmation to PC. */
    (*stats[0]).system.stopped = TRUE;
    (*stats[1]).system.stopped = TRUE;
                                          /* Wait until PC has finished. */
    while (((*master[0]).pcread == TRUE) || ((*master[1]).pcread == TRUE)) {
    }
    (*stats[0]).system.stopped = FALSE;
    (*stats[1]).system.stopped = FALSE;
  }
#endif
}

/* Clear all calls on all active channels and, after waiting
   two seconds, disconnect the link. */
shut_down()
{
  int  linkno, channo;
  int  min_lcn, i;
#if TESTING != TRUE
                                          /* Loop through all channels used. */
  for (linkno = 0; linkno < 2; linkno++) {
    min_lcn = NO_CHANS - (*master[linkno]).no_chans;
    for (channo = min_lcn; channo <= NO_CHANS; channo++) {
      if ((*chans)[linkno][channo].state != P1) {
                                          /* If channel active, then clear it*/
        chan_clear (linkno, channo);
      }
    }
  }
                                  /* Loop round for two seconds. */
  for (i = 0; i < 2000; i++) {
    outp (MUART + TIMER1 , TSTART);
    while ( (inp(MUART + TIMER1)) > (TSTART - 1)) {
    }
                                  /* Keep processing link level and work que.*/
    link_level();
    if (work_out != work_in) {
      switch  ( (int) work_que[work_out].process)  {
                case PAC_REC:
                        pac_received();
                        break;
                case PAC_TRANS:
                        pac_transmit();
                        break;
                default:
                        work_out = (++work_out == MAX_QUE) ? 0 : work_out;
      }
    }
  }
                                  /* Send a DISC on the link level. */
  outp (WD1 + CR0 , 0);
  outp (WD2 + CR0 , 0);
#endif
}
/*************************************************************************/
```

```
/*********************************
 * Packet level timer functions. *
 *********************************/

/* This function allocates and starts a packet timer. */
/* (Note: timers are cleared in pac_received via
          (*chans)[linkno][channo].tactive = FALSE */
timer_req (linkno, channo, type)
int  linkno, channo;
char type;
{
  switch  ( (int) type)  {
        case CALL_REQUEST:
                (*chans)[linkno][channo].timer  = T21;
                (*chans)[linkno][channo].tcount = 1;
                break;
        case CLR_REQUEST:
                (*chans)[linkno][channo].timer  = T23;
                (*chans)[linkno][channo].tcount = NO_CLEARS;
                break;
        case RES_REQUEST:
                (*chans)[linkno][channo].timer  = T22;
                (*chans)[linkno][channo].tcount = 1;
                break;
        case RESTART_REQUEST:
                (*chans)[linkno][channo].timer  = T20;
                (*chans)[linkno][channo].tcount = NO_RESTARTS;
                break;
  }
                                  /* Start the timer. */
  (*chans)[linkno][channo].tactive = TRUE;
}



/* This function is called once a second to update the
   packet level timers of all the active channels. */
ptimer_check ()
{
  register int  linkno, i;
  int  min_lcn;
  char expired, restart;
                                  /* Loop around checking all used channels. */
  for  (linkno = 0; linkno < NOLINKS; linkno++)  {

    min_lcn = NO_CHANS - (*master[linkno]).no_chans;
    for  (i = NO_CHANS; i > min_lcn ; i--)  {
      expired = restart = FALSE;
                                  /* If timer active, decrement it. */
      if  ((*chans)[linkno][i].tactive == TRUE)
        if  (--(*chans)[linkno][i].timer == 0)
                                  /* If timer expired, decrement the counter.*/
          if  (--(*chans)[linkno][i].tcount == 0)
            expired = TRUE;
          else
            restart = TRUE;
```

```
                                    /* Update counter if timer expired. */
  if (expired || restart)
    (*stats[linkno]).packet.ptimeouts++;

                                    /* Take action if timer has expired, but the
                                       counter has still to expire. */
  if (restart == TRUE)
    switch  ( (int) (*chans)[linkno][i].state)  {

                                    /* If in restart request state: */
      case R2:
                                            /* Send another restart packet. */
              work_in = (++work_in == MAX_QUE) ? 0 : work_in;
              work_que[work_in].process = PAC_TRANS;
              work_que[work_in].link = linkno;
              work_que[work_in].channel = i;
              work_que[work_in].ptype = RESTART_REQUEST;
                                            /* Restart the timer. */
              (*chans)[linkno][i].timer = T20;
              break;
                                    /* If in clear request state: */
      case P2:
                                            /* Send a clear request packet. */
              work_in = (++work_in == MAX_QUE) ? 0 : work_in;
              work_que[work_in].process = PAC_TRANS;
              work_que[work_in].link = linkno;
              work_que[work_in].channel = i;
              work_que[work_in].ptype = CLR_REQUEST;
                                            /* Restart the timer. */
              (*chans)[linkno][i].timer = T23;
              break;
      default:
              break;
  }


  if (expired == TRUE)
    switch  ( (int) (*chans)[linkno][i].state)  {
                                    /* If in restart request state: */
      case R2:
                                    /* Declare the packet level as being down. */
              (*stats[linkno]).system.pstate = FALSE;
              break;
                                    /* If in call request state: */
      case P2:
                                            /* Send a clear request. */
              work_in = (++work_in == MAX_QUE) ? 0 : work_in;
              work_que[work_in].process = PAC_TRANS;
              work_que[work_in].link = linkno;
              work_que[work_in].channel = i;
              work_que[work_in].ptype = CLR_REQUEST;
                                    /* Put channel into clear request state. */
              (*chans)[linkno][i].state = P6;
              break;
```

```
                                /* If in reset request state: */
        case D2:
                                /* Send a clear request packet. */
                work_in = (++work_in == MAX_QUE) ? 0 : work_in;
                work_que[work_in].process = PAC_TRANS;
                work_que[work_in].link = linkno;
                work_que[work_in].channel = i;
                work_que[work_in].ptype = CLR_REQUEST;
                                /* Put channel into clear request state. */
                (*chans)[linkno][i].state = P6;
                break;
                                /* If in clear request state: */
        case P6:
                                /* Send a restart request packet. */
                work_in = (++work_in == MAX_QUE) ? 0 : work_in;
                work_que[work_in].process = PAC_TRANS;
                work_que[work_in].link = linkno;
                work_que[work_in].channel = i;
                work_que[work_in].ptype = RESTART_REQUEST;
                                /* Put channel into restart request state. */
                (*stats[linkno]).system.pstate = R2;
                break;
        default:
                                /* Increment error counter if invalid. */
                (*stats[linkno]).system.state_err++;
      }
                                        /* Finally, update expired counter.*/
    (*chans)[linkno][i].tcount++;

    }
  }

}

/**************************************************************************/
```

```
/* TRAFFGEN.C */

/*********************
 * TRAFFIC GENERATOR *
 *********************/

/*This is the actual traffic generator module, which sets up and clears calls
  as well as passing the required number of test data fields to the packet
  transmit module.  The program is called regularly by the systems module. */

#include "SYSTEM.CON"
#include "PACKET.CON"

#include "DTYPES.C"
#include "DVARS.C"



/* This function is the actual traffic generator.  It is called every 10 msec
   and sends out test data packets as required.  Call and clear timers are
   updated once every 100 msec.  Constants are obtained from the PC via the
   master control structure. */

traffic_gen (call)
  char call;
{
  register int linkno;

  for (linkno = 0; linkno < NOLINKS; linkno++) {

                          /* Check if time to send a test data packet. */
    send_check(linkno);
                  /* Check if a call request or clear packet needs to be sent*/
    if (call == TRUE) {
      call_check(linkno);
      clear_check(linkno);
    }
  }

}

/****************************************************************************/


/********************************************
 * Call, clear and send checking functions *
 ********************************************/


/* For all inactive channels, decrement the call timer and if
   expired check system usage and send a call request. */

call_check (linkno)
  register int linkno;
{
  register int channo;
  int  min_lcn, timer;
```

```
                                        /* Only proceed if system is ready.*/
if  ( ((*stats[linkno]).system.tcongested == FALSE) &&
            ((*stats[linkno]).system.rcongested == FALSE) )  {

  min_lcn = NO_CHANS - (*master[linkno]).no_chans;

                                /* Loop through all channels decrementing the
                                   call request timer. */
  for (channo = NO_CHANS; channo > min_lcn; channo--)
    if ( (*chans)[linkno][channo].call_timer != 0 )
            (*chans)[linkno][channo].call_timer--;

                        /* Now loop through all channels and, if required,
                           send a call request on highest logical channel
                           with an expired timer. */
  for (channo = NO_CHANS; channo > min_lcn; channo--) {
    if ((*chans)[linkno][channo].call_timer == 0 &&
            (*chans)[linkno][channo].state == P1)   {

                        /* Add a call request packet to the work que. */
      work_in = (++work_in == MAX_QUE) ? 0 : work_in;
      work_que[work_in].process = PAC_TRANS;
      work_que[work_in].link = linkno;
      work_que[work_in].channel = channo;
      work_que[work_in].ptype = CALL_REQUEST;

                                /* Put channel into call request state. */
      (*chans)[linkno][channo].state = P2;

                                        /* Start the test packet timer. */
      timer = (*master[linkno]).stimer;
      (*chans)[linkno][channo].send_timer = timer;
                                /* Start the clear timer. */
      timer = (*master[linkno]).call_len;
      (*chans)[linkno][channo].clear_timer = timer;
                                        /* Restart the call timer. */
      timer = (*master[linkno]).call_len + (*master[linkno]).cinterval;
      (*chans)[linkno][channo].call_timer = timer;

                        /* Update the link activity counter to highest no.
                           chans. used (Note that pre decrement first). */
      if ((NO_CHANS - channo + 1) > --(*stats[linkno]).system.active)
        (*stats[linkno]).system.active = (NO_CHANS - channo + 1);

                                /* Break out of the for loop.
                                   (At most one call request per 0.1 secs.)*/
      break;
    }
  }

 }

}
```

```
/* Decrement channel clear timers and clear calls if required. */

clear_check (linkno)
  register int linkno;
{
  register int channo;
  int  min_lcn, timer;
                                  /* Loop through all channels decrementing the
                                     clear timer. */
  min_lcn = NO_CHANS - (*master[linkno]).no_chans;
  for (channo = NO_CHANS; channo > min_lcn; channo--) {

    if ((*chans)[linkno][channo].state != P1)  {
      if  (--(*chans)[linkno][channo].clear_timer == 0) {

                          /* If required, add a request to the work que. */
                          /* Do this via a function defined in PACREC module.*/
        chan_clear (linkno, channo);
                                          /* Start the call timer. */
        timer = (*master[linkno]).cinterval;
        (*chans)[linkno][channo].call_timer = timer;
      }
    }

  }

}



/* Decrement send timers of all active channels.  If expired and buffers are
   available, build a test buffer and pass to packet level for transmission
   to the DCE.  The receiver status and packet level window are also
   checked first. */

send_check (linkno)
  register int linkno;
{
  register int channo;
  int  min_lcn, timer;
                          /* If the link is not in the twaiting state, loop
                             round all active channels and send test packets
                             if required. */
  if ((*stats[linkno]).system.tcongested == FALSE)  {

    min_lcn = NO_CHANS - (*master[linkno]).no_chans;
    timer = (*master[linkno]).stimer;
                                  /* Loop through all the high channels. */
    for (channo = NO_CHANS; channo > min_lcn; channo--) {

                          /* If the logical channel is in data transfer state,
                             decrement it's packet send_timer and see if time
                             to send a test packet.*/
      if ( ((*chans)[linkno][channo].state == P4) &&
                          ((*chans)[linkno][channo].trans_state == D1) )  {
        if  (--(*chans)[linkno][channo].send_timer == 0) {
```

```
                        /* Must check if packet level can cope with another
                           packet on this channel before actually calling the
                           function to add the message to the queue. */
        if ( (*chans)[linkno][channo].no_qued <
                                         (*chans)[linkno][channo].window )
          testpac (linkno, channo);

                                        /* Restart the send timer. */
        (*chans)[linkno][channo].send_timer = timer;

        }
      }

    }
  }

}



/* This function is called by send_check and passes the standard test message
   to the packet level via the work queue.  For monitoring purposes, the
   function fills in the channel number (in decimal digits) at the start of
   the test message. */

testpac (linkno, channo)
  int  linkno, channo;
{
  register int buffno, i;
  int temp;
  char digit[4];

                        /* Find and reserve the next free transmit buffer. */
  buffno = buff_find (TRANSMIT, linkno, channo);

                        /* Copy in the first 32 bytes of the test packet.
                           The rest will already be in the buffer as all
                           control packets used are less than 32 bytes. */
  for  (i = 0; (i < 32) && (test_packet[i] != '\0'); i++)
    (*tbuff[linkno])[buffno].data[i] = test_packet[i];

                        /* Convert the channel number to ASCII digits
                           and fill it into the test packet. */
  temp = channo;
  for (i = 0; i < 3; i++)
    digit[i] = 0;
                                /* Break hex number into decimal digits. */
  while (temp >= 1000) {
    digit[3]++;
    temp = temp - 1000;
  }
  while (temp >= 100) {
    digit[2]++;
    temp = temp - 100;
  }
```

```
  while (temp >= 10) {
    digit[1]++;
    temp = temp - 10;
  }
  digit[0] = temp;
                                  /* Convert decimal digits to ASCII code. */
  for (i = 0; i <= 3; i++) {
    digit[i] = digit[i] + 0x11;
                                  /* Fill the channel number into the buffer.*/
    (*tbuff[linkno])[buffno].data[8 - i] = digit[i];
  }

                                  /* Fill in the packet data field length into
                                     the buffer control structure. */
  tcontrol[linkno][buffno].length = (*stats[linkno]).system.buff_len;

                                          /* Add the buffer to the work que. */
  work_in = (++work_in == MAX_QUE) ? 0 : work_in;
  work_que[work_in].process = PAC_TRANS;
  work_que[work_in].link = linkno;
  work_que[work_in].channel = channo;
  work_que[work_in].ptype = DATA;
  work_que[work_in].buffer = buffno;

                                  /* Increment the traffgen/packet level flow
                                     control counter. */
  (*chans)[linkno][channo].no_qued++;

}



/* Future expansion: Pac rec module passes up data packets.
   Not applicable to traffic generator. */
upper_lev()
{
  int linkno, buffno;
                          /* Get in buffer details and free the buffer. */
  linkno = work_que[work_out].link;
  buffno = work_que[work_out].buffer;
  buff_free (RECEIVE, linkno, buffno);
}

/****************************************************************************/
```

APPENDIX E:

Packet level programs.

Files : PACTRANS.C

PACREC.C

LINKLEV.C

Contents:

PACTRANS

E-1 : pac_transmit

E-4 : Packet building functions.

Listed in standard order are the functions which actually create the packets.

PACREC

E-12 : pac_received

E-15 : Packet processing functions.

A list of functions to process a received packet and update the statistics.

E-25 : Utility functions.

LINKLEV

E-30 : link_level

E-31 : WD2511 service functions

E-34 : WD2511 initialization functions.

```
/* PACTRANS.C */

/****************************/
/* PACKET TRANSMIT PROGRAM */
/****************************/

/* This program is called by SYSTEMS when a packet needs to be transmitted.
   The main function, PAC_TRANSMIT, examines the work que to determin what
   type of packet is required.  The appropriate function is then called to
   build the packet.  Finally, the buffer is added to the send queue for
   transmission by the link level processors. */


#include "SYSTEM.CON"
#include "PACKET.CON"

#include "DTYPES.C"
#include "DVARS.C"



/* The main function of this module and the only entry point.  The function
   reads in details from the work queue and calls the appropriate functions
   to build the packet.  The frame is then added to the link level queue. */

pac_transmit ()
{
  register int linkno, channo;
  int  buffno, len;
  unsigned char cjob;
  unsigned short job;

                                 /* Read in details from the work que. */
  linkno = (int) work_que[work_out].link;
  cjob = work_que[work_out].ptype;
  job = (unsigned short) cjob;
  channo = work_que[work_out].channel;


                                 /* Find the next free transmit buffer. */
  if  ((job != DATA) && (job != INTERRUPT))
    buffno = buff_find (TRANSMIT, linkno, channo);
  else
                         /* For data packets the traffgen module passes a
                            message to the packet level via the work queue. */
    buffno = work_que[work_out].buffer;
```

```
                        /* Call the appropriate function to build the packet.
                           Also, call a timer function if required and update
                           the logical channel state. */
switch (job) {
                                     /* Call set up and clearing. */
                                     /* ========================= */
      case CALL_REQUEST:
              len = t_call_request (linkno, channo, buffno);
              timer_req (linkno, channo, CALL_REQUEST);
              (*chans)[linkno][channo].state = P2;
              break;

      case CALL_ACCEPTED:
              len = t_c_accepted (linkno, channo, buffno);
              (*chans)[linkno][channo].state = P4;
              break;

      case CLR_REQUEST:
              len = t_clr_request (linkno, channo, buffno);
              timer_req (linkno, channo, CLR_REQUEST);
              (*chans)[linkno][channo].state = P6;
              break;

      case CLR_CONFIRM:
              len = t_clr_confirm (linkno, channo, buffno);
              (*chans)[linkno][channo].state = P1;
              break;

                                     /* Data and interrupt packets. */
                                     /* =========================== */
      case DATA:
              len = t_data (linkno, channo, buffno);
              if ((*chans)[0][channo].state != P4) {
                buff_free (TRANSMIT, 0, channo);
                channo = (*chans)[0][channo].state;
                return;
              }
              break;

      case INTERRUPT:      /* Note: interrupt packets currently not used.*/
              len = t_interrupt (linkno, channo, buffno);
              break;
      case INT_CONFIRM:
              len = t_int_confirm (linkno, channo, buffno);
              break;

                                     /* Flow control and reset packets. */
                                     /* =============================== */
      case RR:
              len = t_rr (linkno, channo, buffno);
              break;
      case RNR:          /* Note: RNR packets not used by traffic generator.*/
              len = t_rnr (linkno, channo, buffno);
              break;
```

```
        case RES_REQUEST:
                len = t_res_request (linkno, channo, buffno);
                timer_req (linkno, channo, RES_REQUEST);
                (*chans)[linkno][channo].trans_state = D2;
                break;
        case RES_CONFIRM:
                len = t_res_confirm (linkno, channo, buffno);
                (*chans)[linkno][channo].trans_state = D1;
                break;

                                        /* Restart packets. */
                                        /* ================ */
        case RESTART_REQUEST:
                len = t_req_restart (linkno, buffno);
                timer_req (linkno, channo, RESTART_REQUEST);
                (*stats[linkno]).system.pstate = R2;
                break;
        case RESTART_CONFIRM:
                len = t_conf_restart (linkno, channo, buffno);
                break;

                                /* If none of the above, then update error
                                   counter (displayed on systems sceen). */
        default:
                (*stats[linkno]).system.rec_err++;
                break;
    }

                                /* Fill in details of the buffer into the
                                   buffer control structure. */
    tcontrol[linkno][buffno].channel = channo;          /* Channel number.*/
    tcontrol[linkno][buffno].length  = len;             /* Buffer length. */

                        /* Declare the buffer as being ready to send.*/
    tcontrol[linkno][buffno].send = TRUE;

                                /* Add the frame to the link level queue.*/
    send_in[linkno] = (++send_in[linkno]==NO_TRAN_BUFFS) ? 0 : send_in[linkno];
    send_que[linkno][send_in[linkno]] = buffno;

}

/****************************************************************************/
```

```
/******************************/
/* PACKET BUILDING FUNCTIONS */
/******************************/

/* This function builds a call request packet.  The address field is built
   from constants defined in the PACKET.CON file.  Currently call facility
   fields are not supported.  Call subaddressing is implemented by using
   the DTE address field. */

                /* Initialize the address arrays (cant inside a function). */
int calling[] = { CALLING };
int called [] = { CALLED };

t_call_request (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register  temp;
  char far *address;
  int len1, len2;
  int byteno, nibno;
                                /* Determin the address lengths. */
  len1 = sizeof (calling) / 2;
  len2 = sizeof (called) / 2;

                                /* Calculate the base address of the buffer
                                   to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );


                        /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = CALL_REQUEST;

                /* Fill in the calling and called DTE address lengths. */
  temp = len1;
  *(address + 3) = (char) (len2 + (temp << 4));


  byteno = 4;
  nibno  = 1;
                                /* Fill in the called address field. */
  for (temp = 0; temp < len1; temp++) {

    if (nibno == 1) {
      *(address + byteno) = (called[temp] << 4);
      nibno = 0;
    }
    else {
      *(address + byteno) += called[temp];
      byteno++;
      nibno = 1;
    }
  }
```

```
                                /* Fill in the calling address field. */
  for (temp = 0; temp < len2; temp++) {

    if (nibno == 1) {
      *(address + byteno) = (calling[temp] << 4);
      nibno = 0;
    }
    else {
      *(address + byteno) += calling[temp];
      byteno++;
      nibno = 1;
    }
  }

                                /* Use an integral number of octets. */
  if (nibno == 0) {
    byteno++;
    nibno = 1;
  }
                                /* Call facilities length set to 0. */
  *(address + byteno) = 0;
  byteno++;

  return (byteno);              /* Return the packet length. */

}


/* This function builds a three byte call accept packet. */

t_c_accepted (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register  temp;
  char far *address;
                                /* Calculate the base address of the buffer
                                   to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );

                        /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = CALL_ACCEPTED;

  return (3);                   /* Return the packet length. */
}
```

```
/* Build a clear request packet, with clearing cause being DTE_ORIGINATED. */

t_clr_request (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;
                                /* Calculate the base address of the buffer
                                   to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );

                        /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = CLR_REQUEST;

                                /* Byte 4 is the clearing cause. */
  *(address + BYTE4) = C_DTE_ORIGINATED;

  return (4);                   /* Return the packet length. */
}


/* This function builds a clear confirmation packet. */

t_clr_confirm (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;
                                /* Calculate the base address of the buffer
                                   to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );

                        /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = CLR_CONFIRM;

  return (3);                   /* Return the packet length. */
}
```

```
/* This function adds the data packet header to the message passed from the
   traffic generator module.  The channel send sequence number is updated.
   Modulo 8 sequence numbering is used. */

t_data (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char byte3;
  char far *address;
                                /* Calculate the base address of the buffer
                                   to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );

                                /* Format identifier and group number. */
  temp = channo;
  *(address + GFI) = GFI_DATA | ((temp >> 8) & 0x0F);
                                        /* Logical channel number. */
  *(address + LCN) = (char) temp;

                        /* Set the packet's receive sequence number to the
                           receive state variable dte_vr. */
  temp = (*chans)[linkno][channo].dte_vr;
                        /* Reset the receive acknowledgement counter. */
  (*chans)[linkno][channo].R = 0;
                /* Fill the number into top 3 bits of packet identifier. */
  byte3 = (char) (temp << 5);

                      /* Bits 2, 3 and 4 used for packet send sequence no. */
  temp = (*chans)[linkno][channo].dte_vs;
  byte3 |= (char) (temp << 1);
                                /* Update the channel send state variable.*/
  temp = (*chans)[linkno][channo].dte_vs;
  temp = (++temp == 8) ? 0 : temp;
  (*chans)[linkno][channo].dte_vs = temp;
                        /* Also update the outstanding packet counter K. */
  (*chans)[linkno][channo].K++;


  temp = (*chans)[linkno][channo].mbit;
  byte3 |= (char) (temp << 4);          /* Mbit is in bit 5 of the byte. */

  *(address + PTYPE) = byte3;

                        /* Return the total frame lenght, the original data
                           field length having been filled in by traff_gen.*/
  return (tcontrol[linkno][buffno].length + 3);
}
```

```
/* Build an interrupt packet.  These are currently not used by the traffic
   generator.  The fourth byte of the packet would contain the user data. */

t_interrupt (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;

  address = &( (*tbuff[linkno])[buffno].gfi );

                            /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                    /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                    /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = INTERRUPT;
                                    /* User data field, just set to 0. */
  *(address + 3) = 0;

  return (3);                       /* Return the packet length. */
}


/* Build the three byte interrupt confirmation packet. */

t_int_confirm (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;

  address = &( (*tbuff[linkno])[buffno].gfi );

                            /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                    /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                    /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = INT_CONFIRM;

  return (3);                       /* Return the packet length. */
}
```

```
/* Build a receive ready packet to acknowledge received data packets. */

t_rr (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;
                                   /* Calculate the base address of the buffer
                                      to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );

                           /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = (char) (GFI_CALL | ((temp >> 8) & 0x0F));
                                   /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;

                           /* Set the receive sequence number in the packet to
                              the value of the receive state variable dte_vr. */
  temp = (*chans)[linkno][channo].dte_vr
                                   /* Byte 3 is the packet type identifier and
                                      the receive sequence number. */
  *(address + PTYPE) = (char) (RR | (temp << 5));
                           /* Reset the receive acknowledgement counter. */
  (*chans)[linkno][channo].R = 0;
  return (3);                      /* Return the packet length. */
}



/* Build an RNR packet.  Note that as these packets are not supported on all
   X.25 equipment, the traffic generator currently does not send them.
   If the receive frame buffers are getting congested, then an RNR frame will
   be sent to the DCE. */

t_rnr (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;
                                   /* Calculate the base address of the buffer
                                      to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );
                           /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = (char) (GFI_CALL | ((temp >> 8) & 0x0F));
                                   /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;

                           /* Set receive sequence number in packet to value of
                              the channel's receive state variable dte_vr. */
  temp = (*chans)[linkno][channo].dte_vr;
                                   /* Byte 3 is the packet type identifier and
                                      the receive sequence number. */
  *(address + PTYPE) = (char) (RNR | (temp << 5));
                           /* Reset the receive acknowledgement counter. */
  (*chans)[linkno][channo].R = 0;
  return (3);
}
```

```
/* Build a reset request packet.  The resetting cause is filled in as DTE_
   ORIGINATED.  These packets are sent as a result of received data packet
   sequence errors.  Note that the channel sequence numbers are reset by the
   chan_reset utility function in PACREC. */

t_res_request (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;
                                /* Calculate the base address of the buffer
                                   to speed up the access time. */
  address = &( (*tbuff[linkno])[buffno].gfi );

                        /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = RES_REQUEST;

                                /* Byte 4 is the resetting cause field. */
  *(address + BYTE5) = R_DTE_ORIGINATED;

  return (4);                   /* Return the packet length. */
}


/* Build a reset confirmation packet. */

t_res_confirm (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;

  address = &( (*tbuff[linkno])[buffno].gfi );

                        /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = RES_CONFIRM;

  return (3);                   /* Return the packet length. */
}
```

```
/* Build a packet level restart packet.  The restarting cause is entered
   as being a "local procedure error".  Note that the required packet level
   initialization is done by the sys_restart utility function in PACREC. */

t_req_restart (linkno, buffno)
int  linkno, buffno;
{
  register temp;
  char far *address;

  address = &( (*tbuff[linkno])[buffno].gfi );

                          /* Byte 1 is the format identifier and group number*/
  *(address + GFI) = GFI_CALL;
                                  /* Byte 2 is the logical channel number. */
  *(address + LCN) = 0;
                                  /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = RESTART_REQUEST;

                                  /* Byte 4 is the restarting cause field. */
  *(address + BYTE4) = LOCAL_PROC_ERR;

  return (3);                     /* Return the packet length. */
}


/* Build a restart confirmation packet. */

t_conf_restart (linkno, channo, buffno)
int  linkno, channo, buffno;
{
  register temp;
  char far *address;

  address = &( (*tbuff[linkno])[buffno].gfi );

                          /* Byte 1 is the format identifier and group number*/
  temp = channo;
  *(address + GFI) = GFI_CALL | ((temp >> 8) & 0x0F);
                                  /* Byte 2 is the logical channel number. */
  *(address + LCN) = (char) channo;
                                  /* Byte 3 is the packet type identifier. */
  *(address + PTYPE) = RESTART_CONFIRM;

  return (3);                     /* Return the packet length. */
}

/****************************************************************************/
```

```
/* PACREC.C */

/****************************/
/* PACKET RECEIVED PROGRAM */
/****************************/

/* The packet received program is called by SYSTEMS after a packet has been
   received from the DCE.  The main function reads in the packet header and
   decodes it, checking for a valid header.  A function is then called to
   process the particular packet. */

#include "SYSTEM.CON"
#include "PACKET.CON"

#include "DTYPES.C"
#include "DVARS.C"


/* The modules main function. */

pac_received()
{
  register int  linkno, lcn;
  int  buffno, length, temp;
  unsigned char gfi, ptype, byte4, byte5;
  char error=0;
  char far *address;
  unsigned short iptype;

                                  /* Read in the work que information. */
  linkno = work_que[work_out].link;
  buffno = work_que[work_out].buffer;
                          /* Get in length from buffer control structure. */
  length = rcontrol[linkno][buffno].length;

                /* Now access the buffer itself, reading in the first three
                   bytes of the packet  i.e. the packet header. */

                          /* For extra speed, calculate the address first. */
  address = &( (*rbuff[linkno])[buffno].gfi);

  gfi = *(address + GFI);                  /* General format identifier. */

                /* First four bits of gfi are part of the channel number. */
  temp = gfi & 0x000F;
                          /* Read in the rest of the logical channel number.
                             paying particular attention to the sign bit. */
  lcn = (*(address + LCN) & 0x00FF) | (temp << 8);

  ptype = *(address + PTYPE);                     /* Packet type identifier. */

                                  /* Read in the fourth byte of the packet
                                     (if available). e.g. for clearing cause.*/
  byte4 = *(address + 3);
                                  /* Read in the fifth byte of the packet
                                     (if available). e.g. diagnostic codes. */
  byte5 = *(address + 4);
```

```
                                /* If in loop back test mode, calculate the
                                   new lcn i.e. receive with low lcn nos. */
if ((*master[linkno]).lback == TRUE && lcn != 0)
  lcn = NO_CHANS - lcn + 1;


                /* Check that have a valid GFI : Qualifier bit set to 0.
                   (i.e. datagrams are not supported.) */
if ((gfi & 0x80) != 0)
  error = TRUE;
                                /* Mod 8 sequencing must be used.
                                   (SAPONET does not support mod 128)   */
if ((gfi & 0x30) != 0x10)
  error = TRUE;
                        /* Check that have a valid logical channel number.
                           (i.e. between 1 and NO_CHANS inclusive.)   */
if ((lcn < 0) || (lcn > NO_CHANS))
  error = TRUE;

if (error == TRUE) {
                                        /* If an error, update the counter.*/
  (*stats[linkno]).system.rec_err++;
                                        /* Can then only free the buffer. */
  buff_free (RECEIVE, linkno, buffno);
                                        /* Return to main program. */
  return;
}
                        /* Examine ptype and call the appropriate function
                           to process the received packet. */

                                /* Data packet if bit 1 of ptype is 0.*/
if ((ptype & 0x01) == DATA)
  data_packet (linkno, lcn, buffno, ptype);
                                        /* Test for flow control packets. */
else  if ((ptype & 0x1F) == RR)
  rr (linkno, lcn, ptype);
else  if ((ptype & 0x1F) == RNR)
  rnr (linkno, lcn, ptype);
else {

iptype = (unsigned short) ptype;

                                /* For the others, use a switch statement. */
switch (iptype) {
                                        /* Call set-up and clearing. */
     case INCOMING_CALL:
             incoming_call (linkno, lcn, address);
             break;
     case CALL_CONNECTED:
             call_connected (linkno, lcn);
             break;
     case CLR_INDICATION:
             clr_indication (linkno, lcn, byte4);
             if (length == 5)
               diag_decode (byte5);
             break;
     case CLR_CONFIRM:
             clr_confirm (linkno, lcn);
             break;
```

```
        case INTERRUPT:
                interrupt (linkno, lcn, buffno);
                break;
        case INT_CONFIRM:
                int_confirm (linkno, lcn);
                break;
        case RES_INDICATION:
                reset_indication (linkno, lcn, byte4);
                if (length == 5)
                  diag_decode (byte5);
                break;
        case RES_CONFIRM:
                reset_confirm (linkno, lcn);
                break;

        case RESTART_IND:
                rest_indication (linkno, lcn, byte4);
                if (length == 5)
                  diag_decode (byte5);
                break;
        case RESTART_CONFIRM:
                restart_confirm (linkno, lcn);
                break;

        case DIAGNOSTIC:
                (*stats[linkno]).packet.diagnostics++;
                diag_decode (linkno, lcn, byte4);
                break;
        default:
                                /* Update counter for invalid codes. */
                (*stats[linkno]).system.rec_err++;
                break;
    }
  }

   buff_free (RECEIVE, linkno, buffno);

}

/****************************************************************************/
```

```
/*******************************/
/* PACKET PROCESSING FUNCTIONS */
/*******************************/


/* DCE requests the setting up of a new channel. . If in ready state P1,
   then send a call accept packet and go to data transfer state P4.  Also
   for P2 and P6.  For any other state, send a clear request packet. */

incoming_call (linkno, channo, address)
int  linkno, channo;
char far *address;
{
  if ((*chans)[linkno][channo].state == P1 ||
        (*chans)[linkno][channo].state == P2 ||
                (*chans)[linkno][channo].state == P6)  {
                                        /* Channel in data transfer state. */
    (*chans)[linkno][channo].state = P4;
                                        /* Data transfer sub-state ready. */
    (*chans)[linkno][channo].trans_state = D1;

                                /* Add a call accept packet to the work que*/
    work_in = (++work_in == MAX_QUE) ? 0 : work_in;
    work_que[work_in].process = PAC_TRANS;
    work_que[work_in].link = linkno;
    work_que[work_in].channel = channo;
    work_que[work_in].ptype = CALL_ACCEPTED;

                                        /* Read in DTE address. */
    header_address (linkno, channo, address);
                                        /* Call user data not supported. */
    return;
  }

                          /* If channel in some other state , then send back a
                             clear request packet and put the channel into the
                             clear request state. (See end of this module.) */
  chan_clear (linkno, channo);
                                        /* Increment the error counter. */
  (*stats[linkno]).system.state_err++;
}


/* This function is called by the call request and incoming call functions.
   At present it just reads the 14 digit (7 byte) address (defined in X121)
   into the chans array (Address processing is not required). */
header_address (linkno, channo, address)
int  linkno, channo;
char far *address;
{
  int i;
                                        /* Address length field. */
  (*chans)[linkno][channo].add_len = *(address + 3);
                                        /* Read in called DTE address. */
  for (i = 0; i < 7; i++)
    (*chans)[linkno][channo].dte_adds[i] = *(address + 4 + i);
}
```

```
/* DCE has acknowledged a call request and set up a logical channel.
   Check for correct state, then update the channel's status array. */

call_connected (linkno, channo)
int  linkno, channo;
{
        /* Check that the logical channel is in the DTE waiting state.  */
  if ((*chans)[linkno][channo].state == P2) {

                                          /* Reset the call request timer. */
    (*chans)[linkno][channo].tactive = FALSE;
                                  /* Put channel into data transfer state. */
    (*chans)[linkno][channo].state = P4;
                                          /* Flow control ready sub-state. */
    (*chans)[linkno][channo].trans_state = D1;
                                          /* Increment call counter. */
    (*stats[linkno]).packet.calls++;
    return;
  }
                          /* If channel in DTE clear request state, then
                             discard packet and increment the error counter.*/
  if ((*chans)[linkno][channo].state == P6) {
    (*stats[linkno]).system.state_err++;
    return;
  }
                          /* If in some other state, then there is an error.
                             Call a function to clear the channel and put it
                             into state P6 (see end of this module). */
  chan_clear (linkno, channo);
                                          /* Also, increment error counter. */
  (*stats[linkno]).system.state_err++;
}


/* DCE would like to clear the call.  First update the statistics.
   Unless in DTE clear request state, clear the logical channel, flush the
   buffers and add a clear confirm entry to the work queue. */

clr_indication (linkno, channo, cause)
int  linkno, channo;
char cause;
{
                          /* Using the cause byte of the packet, increment the
                             appropriate clear counter. */
  if (cause == C_DTE_ORIGINATED)
    (*stats[linkno]).packet.dte_clears++;
  else
    (*stats[linkno]).packet.dce_clears++;
  switch  (cause)  {
    case C_NET_CONGESTION:
        (*stats[linkno]).packet.congest_clears++;
        break;
    case C_NUMBER_BUSY:
    case C_INCOMPAT_DEST:
    case C_NOT_OBTAINABLE:
        (*stats[linkno]).packet.dest_clears++;
        break;
  }
```

```
                              /* DTE clear request state: stop the clear timer
                                 and go direct to ready state. */
  if ((*chans)[linkno][channo].state == P6) {
    (*chans)[linkno][channo].tactive = FALSE;
    (*chans)[linkno][channo].state = P1;
    return;
  }
                      /* Any other state, re-initialize chans status array.*/
  chan_init (linkno, channo);
  (*chans)[linkno][channo].state = P1;
                                             /* Flush all transmit buffers. */
  buff_flush (linkno, channo);

                      /* Add a clear confirmation packet to the work que. */
  work_in = (++work_in == MAX_QUE) ? 0 : work_in;
  work_que[work_in].process = PAC_TRANS;
  work_que[work_in].link = linkno;
  work_que[work_in].channel = channo;
  work_que[work_in].ptype = CLR_CONFIRM;
}



/* Remote DTE has confirmed a clear which originated at this DTE.
   If correct, clear the timer and update the channels state.
   If in wrong state, call a function to clear the channel. */

clr_confirm (linkno, channo)
int  linkno, channo;
{
                                  /* Check that the current state is P6. */
  if ((*chans)[linkno][channo].state == P6) {
                                  /* Yes, stop the clear timer. */
    (*chans)[linkno][channo].tactive = FALSE;
                                  /* Put the channel into ready state. */
    (*chans)[linkno][channo].state = P1;
    return;
  }
                      /* If the channel was not in the DTE waiting state then there
                         is an error.  First clear the channel, put it into state
                         P6 and send a clear request. */
  chan_clear (linkno, channo);
                                             /* Update state error counter. */
  (*stats[linkno]).system.state_err++;
}
```

```
/* This function processes data packets, checking for correct
   state and sequence numbers. */

data_packet (linkno, channo, buffno, ptype)
int  linkno, channo, buffno;
char ptype;
{
  char temp, dce_pr, dce_ps;

                  /* Check that the logical channel is in the data
                     transfer state and that the flow control is ready. */
  if ( ((*chans)[linkno][channo].state == P4) &&
                  ((*chans)[linkno][channo].trans_state == D1) )  {

                          /* If okay, check for correct sequence numbers. */
                  /* P(R) occupies the three M.S.Bits of the type identifier.*/
    temp = ptype;
    dce_pr = (temp >> 5) & 0x07;
                                          /* P(S) occupies bits 1,2 and 3. */
    temp = ptype;
    dce_ps = (temp >> 1) & 0x07;

                          /* Call a function to check the receive sequence no.
                             and update the chans. status array. */
    if  (pr_seq_check(linkno, channo, dce_pr) == FALSE)  {
                                          /* If wrong, reset the channel. */
      chan_reset (linkno, channo);
      return;
    }
                          /* Do a similar check on the send sequence number. */
    if  (ps_seq_check(linkno, channo, dce_ps)  == FALSE)  {
      chan_reset (linkno, channo);
      return;
    }
                          /* Check window and send an RR packet if required. */
    win_check (linkno, channo, dce_ps);

                          /* Increment the data packet received counter. */
    (*stats[linkno]).packet.data++;

    return;
  }

                          /* Take action if channel is in wrong state. */
                          /* First increment the error counter. */
  (*stats[linkno]).system.state_err++;

                          /* Discard packet if in DTE reset request state. */
  if ( ((*chans)[linkno][channo].state == P4) &&
                  ((*chans)[linkno][channo].trans_state == D2) )  {
    return;
  }
                          /* If channel in DCE reset indication state, */
  if ((*chans)[linkno][channo].state == P4 &&
                  (*chans)[linkno][channo].trans_state == D3)  {
                          /* then discard packet and reset the channel. */
    chan_reset (linkno, channo);
    return;
  }
```

```
                              /* Discard if channel in DTE clear request state. */
  if ((*chans)[linkno][channo].state == P6)
    return;
                              /* Clear channel if in some other state. */
  chan_clear (linkno, channo);
}



/* If channel in correct state, add a interrupt confirmation packet to
   the work queue, increment counter and return. */
interrupt (linkno, channo, buffno)
int  linkno, channo, buffno;
{
        /* Check that logical channel in data transfer state and ready. */
  if   ((*chans)[linkno][channo].state == P4   &&
                 (*chans)[linkno][channo].trans_state == D1)   {

                                        /* Send a interrupt confirm packet. */
    work_in = (++work_in == MAX_QUE) ? 0 : work_in;
    work_que[work_in].process = PAC_TRANS;
    work_que[work_in].link = linkno;
    work_que[work_in].channel = channo;
    work_que[work_in].ptype = INT_CONFIRM;
                                                 /* Increment the counter. */
    (*stats[linkno]).packet.int_pacs++;
    return;
  }
                              /* Take action if channel is in wrong state. */
                              /* First increment the error counter. */
  (*stats[linkno]).system.state_err++;

                              /* Discard packet if in DTE reset request state. */
  if ((*chans)[linkno][channo].state == P4 &&
                 (*chans)[linkno][channo].trans_state == D2)   {
    return;
  }
                              /* If channel in DCE reset indication state, */
  if ((*chans)[linkno][channo].state == P4 &&
                 (*chans)[linkno][channo].trans_state == D3)   {
                              /* then discard packet and reset the channel. */
    chan_reset (linkno, channo);
    return;
  }
                              /* Discard if channel in DTE clear request state. */
  if ((*chans)[linkno][channo].state == P6)
    return;
                              /* Clear channel if in some other state. */
  chan_clear (linkno, channo);
}
```

```
/* Interrupt confirmation function.  Only take action if the channel is
   in the wrong state. */
int_confirm (linkno, channo)
int  linkno, channo;
{
        /* Check that logical channel in data transfer state and ready. */
  if  ((*chans)[linkno][channo].state == P4  &&
                (*chans)[linkno][channo].trans_state == D1)   {
                                /* Indicate acknowledgement to upper level.*/
    (*chans)[linkno][channo].int_pac = FALSE;
    return;
  }
                        /* Take action if channel is in wrong state. */
                        /* First increment the error counter. */
  (*stats[linkno]).system.state_err++;
                /* Discard packet if in DTE reset request state. */
  if ((*chans)[linkno][channo].state == P4 &&
                (*chans)[linkno][channo].trans_state == D2)   {
    return;
  }
                        /* If channel in DCE reset indication state, */
  if ((*chans)[linkno][channo].state == P4 &&
                (*chans)[linkno][channo].trans_state == D3)   {
                        /* then discard packet and reset the channel. */
    chan_reset (linkno, channo);
    return;
  }
                        /* Discard if channel in DTE clear request state. */
  if ((*chans)[linkno][channo].state == P6)
    return;
                        /* Clear channel if in some other state. */
  chan_clear (linkno, channo);
}



/* Check that channel is in correct state and has a valid
   receive sequence number.  If not, reset the channel.
   Takes appropriate action if channel in wrong state. */

rr (linkno, channo, byte3)
int  linkno, channo;
char byte3;
{
  char dce_pr;
  int  temp;
                /* Check for data transfer state, flow control ready. */
  if ((*chans)[linkno][channo].state == P4  &&
        (*chans)[linkno][channo].trans_state == D1)   {
```

```
                         /* Check the P(R) sequence number and update the
                            counters in chans accordingly. */
    dce_pr = (byte3 >> 5) & 0x07;
    if  (pr_seq_check (linkno, channo, dce_pr) == TRUE)   {
                                 /* If alright, change the channel status. */
      (*chans)[linkno][channo].dce_rdy = TRUE;
                                 /* Update the statistics. */
      (*stats[linkno]).packet.rr++;
    }
    else
                                 /* If invalid, reset the channel. */
      chan_reset (linkno, channo);

    return;
  }

                         /* Discard packet if in DTE reset request state. */
  if ((*chans)[linkno][channo].state == P4 &&
              (*chans)[linkno][channo].trans_state == D2)   {
    return;
  }
                         /* Discard if channel in clear request state. */
  if ((*chans)[linkno][channo].state == P6 ||
              (*chans)[linkno][channo].state == P7)
    return;
                         /* Clear channel if in some other state. */
  chan_clear (linkno, channo);
  (*stats[linkno]).system.state_err++;
}


/* Check for correct channel state and a valid receive sequence number.
   If invalid reset the channel, otherwise change channel status. */

rnr (linkno, channo, byte3)
int  linkno, channo;
char byte3;
{
  char dce_pr;
                         /* Check that channel is in the correct state. */
  if  ((*chans)[linkno][channo].state == P4  &&
        (*chans)[linkno][channo].trans_state == D1)   {

                         /* Check the P(R) sequence number and update the
                            counters in chans accordingly. */
    dce_pr = (byte3 >> 5) & 0x07;
    if  (pr_seq_check (linkno, channo, dce_pr) == TRUE)   {
                                 /* If alright, change the channel status. */
      (*chans)[linkno][channo].dce_rdy = FALSE;
    }
    else
                                 /* If invalid, reset the channel. */
      chan_reset (linkno, channo);

    return;
  }
```

```
                          /* Discard packet if in DTE reset request state. */
  if ((*chans)[linkno][channo].state == P4 &&
                (*chans)[linkno][channo].trans_state == D2)   {
    return;
  }
                          /* If channel in DCE reset indication state, */
  if ((*chans)[linkno][channo].state == P4 &&
                (*chans)[linkno][channo].trans_state == D3)   {
                          /* then discard packet and reset the channel. */
    chan_reset (linkno, channo);
    return;
  }
                          /* Discard if channel in DTE clear request state. */
  if ((*chans)[linkno][channo].state == P6)
    return;
                          /* Clear channel if in some other state. */
  chan_clear (linkno, channo);
  (*stats[linkno]).system.state_err++;
}



/* DCE wishes to reset the channel.  If in ready state D1, reset the sequence
   numbers, flush the buffers and add a reset confirm packet to the work que.
   If in DTE reset request state, then just go directly to state D1. */

reset_indication (linkno, channo)
int  linkno, channo;
{
                  /* Only take action if in the flow control ready state D1. */
  if   ((*chans)[linkno][channo].state == P4   &&
         (*chans)[linkno][channo].trans_state == D1)   {

                                  /* Add a reset confirm to the work que. */
    work_in = (++work_in == MAX_QUE) ? 0 : work_in;
    work_que[work_in].process = PAC_TRANS;
    work_que[work_in].link = linkno;
    work_que[work_in].channel = channo;
    work_que[work_in].ptype = RES_CONFIRM;
                                  /* Put channel into DCE reset ind. state. */
    (*chans)[linkno][channo].trans_state = D3;
                                  /* Reset all channel sequence counters. */
    (*chans)[linkno][channo].dte_pr = 0;
    (*chans)[linkno][channo].dte_ps = 0;
    (*chans)[linkno][channo].old_dce_pr = 0;
    (*chans)[linkno][channo].old_dce_pr = 0;
                                  /* Flush any qued buffers on the channel. */
    buff_flush (linkno, channo);
  }

                          /* Discard packet if channel alreay in DCE reset
                             indication state. */
  if ((*chans)[linkno][channo].state == P4 &&
                (*chans)[linkno][channo].trans_state == D3)
    return;
```

```
                         /* Go direct to flow control ready state D1 if chan.
                            was in DTE reset request state. */
  if ((*chans)[linkno][channo].state == P4 &&
              (*chans)[linkno][channo].trans_state == D2)  {
    (*chans)[linkno][channo].trans_state = D1;
    return;
  }
                         /* Discard if channel in DTE clear request state. */
  if ((*chans)[linkno][channo].state == P6)
    return;
                         /* Clear channel if in some other state. */
  chan_clear (linkno, channo);
  (*stats[linkno]).system.state_err++;
}




/* DCE has confirmed a reset.  If in correct state, go to flow control ready
   state D1.  If in wrong data state, then reset the channel.
   Send a clear if not in data transfer state P4. */

reset_confirm (linkno, channo)
  int linkno, channo;
{
                                 /* Check for DTE reset request state D2. */
  if  ((*chans)[linkno][channo].state == P4 &&
        (*chans)[linkno][channo].trans_state == D1)  {

                                         /* Clear the reset timer. */
    (*chans)[linkno][channo].tactive = FALSE;
                                         /* Data transfer ready state. */
    (*chans)[linkno][channo].trans_state = D1;
    return;
  }

                         /* Reset channel if in flow control ready state. */
  if ((*chans)[linkno][channo].state == P4 &&
              (*chans)[linkno][channo].trans_state == D1)  {
    chan_reset (linkno, channo);
    return;
  }
                         /* Reset channel if in DCE reset indication state. */
  if ((*chans)[linkno][channo].state == P4 &&
              (*chans)[linkno][channo].trans_state == D3)  {
    chan_reset (linkno, channo);
    return;
  }
                         /* Discard if channel in DTE clear request state. */
  if ((*chans)[linkno][channo].state == P6)
    return;
                         /* Clear channel if in some other state. */
  chan_clear (linkno, channo);
  (*stats[linkno]).system.state_err++;
}
```

```
/* DCE wishes to restart the packet level DTE/DCE interface.  If in ready
   state R1, then do a complete initialization the packet level.
   Send a restart confirmation packet. */

rest_indication (linkno, channo)
int  linkno, channo;
{
                                /* Go to ready state if already in state R2. */
  if  ((*stats[linkno]).system.pstate == R2)   {
    (*stats[linkno]).system.pstate = R1;
    (*stats[linkno]).system.state_err++;
    return;
  }
                                        /* Otherwise restart the channel. */
  sys_restart (linkno);
                                /* Add a restart confirm packet to the queue. */
  work_in = (++work_in == MAX_QUE) ? 0 : work_in;
  work_que[work_in].process = PAC_TRANS;
  work_que[work_in].link = linkno;
  work_que[work_in].channel = channo;
  work_que[work_in].ptype = RESTART_CONFIRM;
                                        /* Increment the restart counter. */
  (*stats[linkno]).packet.restarts++;
}


/* If in DTE restart request state, then go to state R1.
   If in other states then there is an error, so do a system
   restart here and send a restart request packet. */

restart_confirm (linkno, channo)
int  linkno, channo;
{
                                        /* Go to ready state if DTE requested
                                           the packet level restart. */
  if  ((*stats[linkno]).system.pstate == R2)   {
    (*stats[linkno]).system.pstate = R1;
    return;
  }
                                        /* Otherwise increment error counter. */
  (*stats[linkno]).system.state_err++;
                                /* Put channel into DTE restart indication state. */
  (*stats[linkno]).system.pstate = R3;
                                        /* Call a function to do a complete packet
                                           level re-initialization. */
  sys_restart (linkno);
                                        /* Increment the restart counter. */
  (*stats[linkno]).packet.restarts++;
}
```

```
/* This function processes a diagnostics packet, calling a function to
   update the packet level statistics. */
diagnostic (linkno, channo, code)
int  linkno, channo;
char code;
{
                /* Increment the counter for diagnostic packets. */
  (*stats[linkno]).packet.diagnostics++;
                                /* Call function to interpret the diagnostic
                                   code and update the required counter. */
  diag_decode (linkno, channo, code);
}

/***************************************************************************/


/*********************/
/* UTILITY FUNCTIONS */
/*********************/


/* Called in the event of a sequence error, this function adds
   a reset request to the work queue for the pac. trans. module.
   It also resets the channel sequence nos. and flushes the queues. */

chan_reset (linkno, channo)
  int  linkno, channo;
{
                        /* Flush any queued buffers on the channel. */
  buff_flush (linkno, channo);
                                /* Put the channel into reset request state*/
  (*chans)[linkno][channo].trans_state = D2;
                                /* Reset all channel state variables. */
  (*chans)[linkno][channo].dte_vs = 0;
  (*chans)[linkno][channo].dte_vr = 0;
  (*chans)[linkno][channo].K = 0;
  (*chans)[linkno][channo].R = 0;
                        /* Initialize the traffgen flow control counter. */
  (*chans)[linkno][channo].no_qued = 0;
                                /* Add a reset request to the work que. */
  work_in = (++work_in == MAX_QUE) ? 0 : work_in;
  work_que[work_in].process = PAC_TRANS;
  work_que[work_in].link = linkno;
  work_que[work_in].channel = channo;
  work_que[work_in].ptype = RES_REQUEST;
                                /* Increment the reset counter. */
  (*stats[linkno]).packet.resets++;
}
```

```
/* Called in the event of a invalid packet level state or the
   reception of a invalid packet, this function adds a clear
   request to the work que for the pac. trans. module.
   It also clears the logical channel and flushes the queues. */

chan_clear (linkno, channo)
  int  linkno, channo;
{
                                /* Flush any qued buffers on the channel. */
  buff_flush (linkno, channo);
                                /* Initialize the channel sequence nos. */
  (*chans)[linkno][channo].dte_pr = 0;
  (*chans)[linkno][channo].dte_ps = 0;
  (*chans)[linkno][channo].old_dce_pr = 0;
  (*chans)[linkno][channo].old_dce_ps = 0;
  (*chans)[linkno][channo].tactive = FALSE;
                        /* Initialize the traffgen flow control counter. */
  (*chans)[linkno][channo].no_qued = 0;
                                /* Put the channel into clear request state*/
  (*chans)[linkno][channo].state = P6;
  (*chans)[linkno][channo].trans_state = D1;
  (*chans)[linkno][channo].dce_rdy = TRUE;
                                /* Add a clear request to the work que. */
  work_in = (++work_in == MAX_QUE) ? 0 : work_in;
  work_que[work_in].process = PAC_TRANS;
  work_que[work_in].link = linkno;
  work_que[work_in].channel = channo;
  work_que[work_in].ptype = CLR_REQUEST;
}


/* This function performs a complete packet level restart. */

sys_restart (linkno)
  int  linkno;
{
  register int i;
                        /* Call operating system to clear all the buffers
                           and flush all the transmit work que entries. */
  flush_all (linkno);
                        /* Re-initialize all the channels' status arrays. */
  for (i = 0; i <= NO_CHANS; i++)
    chan_init (linkno, i);
                        /* Put link into packet level ready state. */
  (*stats[linkno]).system.pstate = R1;
}
```

```
/* This procedure decodes the diagnostic field in packets and
   increments the appropriate statistics counter. */

diag_decode (linkno, channo, code)
int  linkno, channo;
char code;
{
                                        /* Invalid sequence numbers. */
  if  ((code == 1) || (code == 2))
    (*stats[linkno]).packet.dseq_err++;
                                        /* Invalid packet levle state. */
  if  ((code >= 16) && (code <= 29))
    (*stats[linkno]).packet.dstate_err++;
                                        /* Packet structure problems. */
  if ( (code >=32) && (code <= 44))
    (*stats[linkno]).packet.drec_err++;
                                        /* Timer expired. */
  if ( (code >=48) && (code <= 52))
    (*stats[linkno]).packet.d_texp++;
                                        /* Call set up problems. */
  if ( (code >= 64) && (code <=68))
    (*stats[linkno]).packet.d_callprob++;
}



/*This function is called on reception of a data, rr or rnr packet and checks
  that the receive sequence number is within the required window of
  (dte_vs - K) < dce_pr <= dte_vs  using mod8 sequencing.
  If correct, then the sequence numbers in the channel control block are
  updated and the function returns true.  The flow control mechanism with the
  traffgen module is also updated. */

pr_seq_check (linkno, channo, dce_pr)
  int  linkno, channo;
  char dce_pr;
{
  char  dte_vs;
  char  mod8_dte_vs;

  dte_vs = (*chans)[linkno][channo].dte_vs;
  K = (*chans)[linkno][channo].K
                           /* First correct for the mod 8 sequencing, taking
                              the dce_pr as the reference. */
  mod8_dte_vs = (dte_vs < dce_pr) ? (dte_vs + 8) : dte_vs;

                           /* Check that dce_pr is within the required window.*/
  if (dce_pr > mod8_dte_vs) || (dce_pr <= (mod8_dte_vs - K)) {
    (*stats[linkno]).system.seq_err++;
    return (FALSE);
  }
```

```
                            /* If the receive sequence numbers are correct, then
                               update the receive state variable. */
  (*chans)[linkno][channo].dte_vr = dce_pr;
  (*chans)[linkno][channo].K = mod8_dte_vs - dce_pr;
                            /* Update the traffgen/packet level flow control
                               mechanism. */
  (*chans)[linkno][channo].no_qued = mod8_dte_vs - dce_pr;

  return (TRUE);
}



/* This function is called after the reception of a data packet and
   checks for a valid packet send sequence number (dte_vr = dce_ps).
   If correct chans is updated and the function returns true. */

ps_seq_check (linkno, channo, dce_ps)
  int  linkno, channo;
  char dce_ps;
{
  char dte_vr;
  int temp;

  dte_vr = (*chans)[linkno][channo].dte_vr;

                   /* For ps, must check that the packet is the next in the
                      sequence.  i.e. dce_ps = dte_vr. */
  if  (dce_ps != dte_vr)  {
    (*stats[linkno]).system.seq_err++;
    return (FALSE);
  }

                            /* If the send sequence number is  correct, then
                               update the receive state variable. */
  (*chans)[linkno][channo].dte_vr = (++dte_vr = 8) ? 0 : dte_vr;
                /* Update the packet received (but not yet acked.) counter. */
  (*chans)[linkno][channo].R++;

return (TRUE);
}



/* This function is called by the data_packet function after the sequence
   numbers have been checked.  It adds an RR packet to the work queue if
   the packet window limit has been reached. */

win_check (linkno, channo, dce_ps)
  int  linkno, channo;
  char dce_ps;
{
  int R;
  R = (*chans)[linkno][channo].R;
                            /* Test if we have reached the window limit.
                               i.e. DCE cant transmit another packet until it
                               receives an acknowledgement. */
```

```
  if ( R == (*chans)[linkno][channo].window ) {
                              /* Yes, then send an RR packet to the DCE.*/
    work_in = (++work_in == MAX_QUE) ? 0 : work_in;
    work_que[work_in].process = PAC_TRANS;
    work_que[work_in].link = linkno;
    work_que[work_in].channel = channo;
    work_que[work_in].ptype = RR;
  }
}

/****************************************************************************/
```

```
/* LINKLEV.C */

/***********************************/
/* Packet to link level interface. */
/***********************************/

/* This program interfaces the WD2511 link level processors to the packet
   level.  It contains all functions relating to the WD2511 processors. */

#include "SYSTEM.CON"
#include "ADDRESS.CON"
#include "LINK.CON"

#include "DTYPES.C"
#include "DVARS.C"


/* The main function of the link level interface.  It is called regularly by
   systems to monitor activity on the link level.  Functions are called to
   service the WD devices if, for example, a frame has been received. */

link_level()
{
  register int linkno;
  char state;
  static long total=1;
  static long idle[2]={0,0};
  static int prev_rnr[2]={0,0};
  int  sr0, sr1, sr2;

                                  /* Process both links. */
  for (linkno = 0; linkno < NOLINKS; linkno++) {
                                  /* Read in the WD status registers. */
    if (linkno == 0) {
      sr0 = inp (WD1 + SR0);
      sr1 = inp (WD1 + SR1);
      sr2 = inp (WD1 + SR2);
    }
    else {
      sr0 = inp (WD2 + SR0);
      sr1 = inp (WD2 + SR1);
      sr2 = inp (WD2 + SR2);
    }
                                  /* Update the link state. */
    (*stats[linkno]).link.lstate = ((sr2 & 0x01) == 0) ? TRUE : FALSE;
                                  /* Compute link receive activity. */
    total += 1;
    if ((sr2 & 0x20) != 0) {
      idle[linkno] += 1;
                                  /* Calculate % time receiver active. */
      (*stats[linkno]).link.activity = (100 * (total - idle[linkno])) /total;
    }
                                  /* Update the RNR frame count. */
    if ((sr0 & 0x01) == 0)
      prev_rnr[linkno] = FALSE;
    else {
      if (prev_rnr[linkno] == FALSE)
        (*stats[linkno]).link.rec_rnr++;
      prev_rnr[linkno] = TRUE;
    }
```

```
                                  /* Check if a packet has been received. */
    while  ( ((*rlook[linkno])[rnext[linkno]].state & FRCML ) != 0 )
      wd_rec (linkno);

                                  /* Check that WD has packets to send. */
    while  ( ((*tlook[linkno])[tnext[linkno]].state & ACKED) != 0  &&
                          send_in[linkno] != sending[linkno] )
      wd_send (linkno);

                                  /* Call error stats routine if required. */
    if ((srl & ERR_MSK) != FALSE)
      linkerr (linkno);

                                  /* Set the SEND bit of the WD processor. */
    if (linkno == 0)
      outp ( WD1 + CR1 , SEND);
    else
      outp ( WD2 + CR1 , SEND);
  }
}

/****************************************************************************/


/****************************
 * WD2511 service functions *
 ****************************/


/* A packet has been correctly received.  This function adds the packet to
   the work que for processing by the PAC_REC module.  It then updates the
   rlook table, filling in the address of the next free buffer.
   Thus packets are easily queued for later processing by the packet level.
   Also, messages can easily be transferred to higher software layers simply
   by passing up the buffer number. */

wd_rec (linkno)
  register linkno;
{
  int  buffno, address, temp;
                                      /* Increment the work que pointer. */
  work_in = (++work_in == MAX_QUE) ? 0 : work_in;

                          /* Add the packet to the work que. */
                                      /* Fill in the process and linkno. */
  work_que[work_in].process = PAC_REC;
  work_que[work_in].link    = (char) linkno;
                                  /* Fill in the buffer number, stored in two
                                     spare bytes of the rlook table. */
  buffno =  (*rlook[linkno])[rnext[linkno]].buffer;
  work_que[work_in].buffer = buffno;

                                  /* Get in the packet length and enter into
                                     the buffer control structure. */
  temp = ( ((*rlook[linkno])[rnext[linkno]].count_hi & 0x0F) << 8);
  temp += (*rlook[linkno])[rnext[linkno]].count_lo;
  rcontrol[linkno][buffno].length = temp;
```

```
                              /* Now update the rlook table, filling in the address
                                 of the next free receive buffer. */
  buffno = buff_find (RECEIVE, linkno, 0);

                                      /* Get the address via the '&' operator. */
  address = (short)  &((*rbuff[linkno])[buffno].gfi);
                                      /* Store the address high order byte first.*/
  (*rlook[linkno])[rnext[linkno]].sadr_lo = (char) address;
  (*rlook[linkno])[rnext[linkno]].sadr_hi = (char) (address >> 8);
                                      /* Also store in the buffer number. */
  (*rlook[linkno])[rnext[linkno]].buffer = buffno;

                                      /* Declare the segment as ready to receive.*/
  (*rlook[linkno])[rnext[linkno]].state = RECRDY;

                              /* Finally, increment the receive table pointer to
                                 point to the next expected receive packet. */
  rnext[linkno] = (++rnext[linkno] == 8) ? 0 : rnext[linkno];

                                      /* Update the packet receive counter. */
  (*stats[linkno]).packet.pac_rec++;
}



/* When ready to be transmitted, the send flag of a buffer is set true and
   the buffer is added to the send que.  This function takes a buffer from
   the send que and passes it to the WD link processor via the tlook table.*/

wd_send (linkno)
  register linkno;
{
  int  buffno, length, address;
                                              /* First free the acked buffer. */
  buffno = (*tlook[linkno])[tnext[linkno]].buffer;
  buff_free (TRANSMIT, linkno, buffno);

                                      /* Point to the next buffer to be sent. */
 sending[linkno] = (++sending[linkno] == NO_TRAN_BUFFS) ? 0 :sending[linkno];
                                      /* Get in the buffer number from send que. */
  buffno = send_que[linkno][sending[linkno]];

                                      /* Check that buffer still to be sent.
                                         (ie. check that it has not been flushed)*/
  if (tcontrol[linkno][buffno].send == FALSE) {
    buff_free (TRANSMIT, linkno, buffno);
    return;
  }
                                      /* Fill buffer number into the tlook table.*/
  (*tlook[linkno])[tnext[linkno]].buffer = buffno;

                                      /* Get the packet length from the buffer
                                         control structure. */
  length = tcontrol[linkno][buffno].length;
                                      /* Fill it in, high order byte first. */
  (*tlook[linkno])[tnext[linkno]].count_lo = (char) length;
  (*tlook[linkno])[tnext[linkno]].count_hi = (char) (length >> 8);
```

```
                                    /* Get buffer address via the & operator. */
  address = (short) &((*tbuff[linkno])[buffno].gfi);
                                    /* Fill the buffer address into the tlook
                                       table high order byte first. */
  (*tlook[linkno])[tnext[linkno]].sadr_lo = (char) address;
  (*tlook[linkno])[tnext[linkno]].sadr_hi = (char) (address >> 8);

                                    /* Clear the ACKED bit and set the BRDY bit
                                       i.e. buffer is ready to be sent. */
  (*tlook[linkno])[tnext[linkno]].state = BRDY;

                                    /* Increment the transmit table pointer
                                       (point to next buffer to be acked). */
  tnext[linkno] = (++tnext[linkno] == 8) ? 0 : tnext[linkno];

                                    /* Increment packet send counter. */
  (*stats[linkno]).packet.pac_sent++;

                            /* Update the byte transmit total, with 6 bytes for
                               frame fields (flags, address, control and FCS).
                               Link control frames (eg. RR) are approximated as
                               1 byte per packet sent. */
  length = tcontrol[linkno][buffno].length;
  (*stats[linkno]).link.byte_tot += length + 6 + 1;
}


/* This function reads in the cause of the error/event from
   the error register and updates the statistics counters. */

linkerr (linkno)
  int  linkno;
{
  char errval, value;
                                            /* Read in the event code. */
  if (linkno == 0)
    errval = inp (WD1 + ER0);
  else
    errval = inp (WD2 + ER0);
                                            /* Find out what the event was. */
  switch  ( (int) errval)  {
        case 0x02:
        case 0x04:
          (*stats[linkno]).link.mem_notrdy++;
          break;
        case 0x10:
          (*stats[linkno]).link.rlook_err++;
          break;
        case 0x80:
          (*stats[linkno]).link.resets++;
          break;
        case 0xC0:
          (*stats[linkno]).link.frmr_rec++;
          break;
```

```
        case 0xC1:
        case 0xC3:
        case 0xC4:
        case 0xC8:
          (*stats[linkno]).link.frmr_trans++;
          break;
  }
                            /* Update the WD2511 error counters, resetting the
                               WD error table when finished. */
  (*stats[linkno]).link.fcs_err += (*wd_err[linkno]).fcs_err;
  (*wd_err[linkno]).fcs_err = 0;
  (*stats[linkno]).link.shortf += (*wd_err[linkno]).sframes;
  (*wd_err[linkno]).sframes = 0;
  (*stats[linkno]).link.t1_timeout += (*wd_err[linkno]).t1_out;
  (*wd_err[linkno]).t1_out = 0;
  (*stats[linkno]).link.rej_rec += (*wd_err[linkno]).rej_rec;
  (*wd_err[linkno]).rej_rec = 0;
  (*stats[linkno]).link.rej_trans += (*wd_err[linkno]).rej_trans;
  (*wd_err[linkno]).rej_trans = 0;
}

/***************************************************************************/


/*************************
 * WD2511 initialization *
 *************************/


/* This function sets up the transmit and receive buffer tables of the WD
   devices.  The error counters and frame queue pointers are initialized. */

wd_table()
{
  register  linkno, segno;
  int buffno, address;
                                    /* Loop round setting up all tables. */
  for  (linkno = 0; linkno < NOLINKS; linkno++)  {
    for  (segno = 0; segno < 8; segno++)  {

                                    /* First set up the receive table. */
                                    /* Get in a  free buffer. */
      buffno = buff_find (RECEIVE, linkno, 0);

                                    /* Calculate it's address via & operator. */
      address = (short)  &((*rbuff[linkno])[buffno].gfi);
                                    /* Store the address. */
      (*rlook[linkno])[segno].sadr_lo = (char) address;
      (*rlook[linkno])[segno].sadr_hi = (char) (address >> 8);
                                    /* Also store the buffer number in table. */
      (*rlook[linkno])[segno].buffer = buffno;

                                    /* Declare segment as ready to receive. */
      (*rlook[linkno])[segno].state = RECRDY;
```

```
                                        /* Set up the transmit table. */
                                        /* We just set all acked bits (in order to
                                           avoid the 0 - 0 state). */
      (*tlook[linkno])[segno].state = ACKED;
                              /* Fill in buffno as WD will free these buffers. */
      buffno = buff_find (TRANSMIT, linkno, 0);
      (*tlook[linkno])[segno].buffer = buffno;
    }
  }

                                                /* Initialize the error table. */
  for (linkno = 0; linkno < NOLINKS; linkno++) {
    (*wd_err[linkno]).fcs_err = 0;
    (*wd_err[linkno]).sframes = 0;
    (*wd_err[linkno]).t1_out = 0;
    (*wd_err[linkno]).rej_rec = 0;
    (*wd_err[linkno]).rej_trans = 0;
  }

                                        /* Initialize the frame queue pointers. */
  for (linkno = 0; linkno < NOLINKS; linkno++)
    send_in[linkno] = sending[linkno] = 0;
}



/* This function programs the registers of the link level processors and
   starts up the link level. */

wd_init()
{
  int  address;
  char temp;
  int i;
                                        /* First set up link processor 1. */

                                                /* Disconnect the link.   */
  outp ( WD1 + CR0 , LDOWN);                    /* MDISC bit of CR0 set. */

  outp ( WD1 + CR1 , CR1_DEF);                  /* Program control register 1. */
                                        /* Set up the link  timer and counter. */
  outp ( WD1 + T1REG , LINK_T1);
  outp ( WD1 + N2T1 , LINK_N2T1);
                                                /* Set up the link level A field. */
  outp ( WD1 + XMT_COMM , LINK_A1);
  if ((*master[0]).lback == TRUE)
    outp ( WD1 + XMT_RESP , LINK_A1);
  else
    outp ( WD1 + XMT_RESP , LINK_A2);

                  /* Tell the link processor the address of the memory look up
                     tables, first casting the 32 address to 16 bits. */
  address = (short) WD1TAB_ADDRESS;
                                                /* Now store the address. */
  outp ( WD1 + TLOOK_LO , (char) address );
  outp ( WD1 + TLOOK_HI , (char) (address >> 8) );
                                                /* Set the maximum buffer size. */
  outp ( WD1 + SIZE_REG , WD_BSIZE);
```

```
                                        /* Initialize the segment pointers.*/
rnext[0] = 0;
tnext[0] = 0;


                /* Program the first control register and set up the link. */
outp ( WD1 + CR0 , CR0_DEF);
                                /* Wait until the link is up. */
while ((inp(WD1 + SR2) & 1) != 0) {
  temp = inp(WD1 + SR1);
  for  (i = 0; i < 1000; i++)
    temp = 0;
}
if (NOLINKS == 1)
  return;

                                /* Now set up link processor 2. */

                                        /* Disconnect the link.   */
outp ( WD2 + CR0 , LDOWN);              /* MDISC bit of CR0 set. */

outp ( WD2 + CR1 , CR1_DEF);            /* Control register 1. */
                                /* Set up the link level timer and counter.*/
outp ( WD2 + T1REG , LINK_T1);
outp ( WD2 + N2T1 , LINK_N2T1);
                                        /* Set up the link level A field. */
outp ( WD2 + XMT_COMM , LINK_A1);
if  ((*master[1]).lback == TRUE)
  outp ( WD2 + XMT_RESP , LINK_A1);
else
  outp ( WD2 + XMT_RESP , LINK_A2);


                /* Tell the link processor the address of the memory look up
                   tables, first casting the 32 address to 16 bits. */
address = (short) WD2TAB_ADDRESS;
                                        /* Now store the address. */
outp ( WD2 + TLOOK_LO , (char) address );
outp ( WD2 + TLOOK_HI , (char) (address >> 8) );
                                        /* Set the maximum buffer size. */
outp ( WD2 + SIZE_REG , WD_BSIZE);
                                        /* Initialize the segment pointers.*/
rnext[1] = 0;
tnext[1] = 0;


                /* Program the first control register and set up the link. */
outp ( WD2 + CR0 , CR0_DEF);
                                /* Wait until the link is up. */
while ((inp(WD2 + SR2) & 1) != 0) {
  temp = inp(WD2 + SR1);
  for  (i = 0; i < 1000; i++)
    temp = 0;
 }
}

/*****************************************************************************/
```

SOFTWARE NOTES

F.1 Program compiling:

The traffic generator programs are compiled and linked by means of the COMPILE batch file. Programs should be compiled on an IBM PC or compatible, equiped with a hard disk drive, using Microsoft C version 3.0 or higher. Compiler options used are noted as remarks in the batch file listing. The Microsoft link utility is used to link the object files into the required executible ones (X25TG and CARD). Note that the /CO option should be selected if the Codeview debugger is to be used.

The COMPILE.BAT listing follows:

(For more details on compiling and linking refer to the Microsfot C user's manual).

```
rem  COMPILE.BAT
echo off
cls
break on

echo .
echo This command file compiles and links all the traffic generator progs.
echo (Will take about 15 mins.)                              update: 18/10/86.
echo .

rem  To run programs on the PC's 8088 processor instead of the X.25 card's,
rem  set the TESTING constant true (first page of constant definitions in
rem  the SYSTEM.CON include file).

rem  Commands for Microsoft C verion 3.0 or higher.
rem  (Assumes programs compiled on a PC equiped with a hard disk drive.)
rem  Compiler options used are the following:
rem              Ze  Enable the FAR keyword.
rem              Zi  Allows the codeview debugger to be run.
rem              Zp  Packs the data structures.
rem              Gs  Removes stack probes for fastest execution.
rem              Ot  Compile for optomized code execution time.
rem  For testing, ommit Gs and use the Od option to disable optiomization.
rem  (Zi option only available on C compiler version 4.)
rem  Program may also be individually compiled using the MSC command.
```

```
echo  Compiling X25TG
MSC /Ze /Zi /Zp /Od  X25TG.C;
IF ERRORLEVEL 2 GOTO EXIT

echo  Compiling SYSTEMS
MSC /Ze /Zi /Zp /Od  SYSTEMS.C;
IF ERRORLEVEL 2 GOTO EXIT

echo  Compiling TRAFFGEN
MSC /Ze /Zi /Zp /Od  TRAFFGEN.C;
IF ERRORLEVEL 2 GOTO EXIT

echo  Compiling PACREC
MSC /Ze /Zi /Zp /Od  PACREC.C;
IF ERRORLEVEL 2 GOTO EXIT

echo  Compiling PACTRANS
MSC /Ze /Zi /Zp /Od  PACTRANS.C;
IF ERRORLEVEL 2 GOTO EXIT

echo  Compiling LINKLEV
MSC /Ze /Zi /Zp /Od  LINKLEV.C;
IF ERRORLEVEL 2 GOTO EXIT


rem  If all successfully compiled, then link the object modules.
rem  The following library modules will be required (small memory model):
rem             EM.LIB, SLIBC.LIB, SLIBFP.LIB
rem  The following include files are required:
rem             STDIO.H, CONIO.H, TIME.H

rem  Two different link options.  The first two link commands produce two
rem  separate programs, one for the PC (X25TG) and one for the card (CARD).
rem  The third command line, used in testing, produces a single program
rem  X25TG.  The CO option allows the codeview debugger to be used.

rem  LINK  X25TG;
rem  LINK  SYSTEMS+TRAFFGEN+PACREC+PACTRANS+LINKLEV, CARD;

LINK  X25TG+SYSTEMS+TRAFFGEN+PACREC+PACTRANS+LINKLEV, , , /CO
goto END


:EXIT
echo  Compilation failed.

:END
```

F.2 Program loading and TG.BAT

The TG batch file loads the traffic generator programs and passes control to X25TG. It contains all the hardware configuration information, passing the configuration parameters to X25TG on startup. Note that two loading options are availabe, with the unused commands being commented out.

The TG.BAT listing follows:

(Refer to chapter 7 for more details on program loading).

```
rem  TG.BAT
echo off
break on
cls

echo *
echo *** Loading traffic generator programs.
echo *

rem  Use DOS debug utility to load traffic gen. programs onto the cards.
rem  The startup file contains the required instructions and addresses.
rem  Programs are loaded into memory and copied to the X.25 cards installed.
rem  The startup routine for the card's 8088 is also loaded.
rem  debug < STARTUP.TXT > NULL

rem  When running programs on the PC's 8088, halt the card's 8088 processor.
debug < CPUSTOP.TXT > NULL


rem  Now load control program, passing it the configuration variables.
rem  (Load the codeview debugger first if required.)
rem  CV X25TG 1 5 0

X25TG  1  5 0

rem  First parameter passed is defined as follows:
rem             0 : traffic generator connected to DCE.
rem             1 : system configured for loop back testing.
rem  The following digits consist of a list of address block numbers followed
rem  by a list of I/O bank nos. (one pair per card installed).

rem  Address block nos. are the memory bank numbers (in 64k units) of the
rem  card's program block.  e.g. 4 means programs will be loaded into the
rem  the fifth 64k block in the PC's memory map.
rem  I/O bank nos. (from 0 to 5) select which of the six I/O locations is
rem  selected by the card's jumper setting (J6).

rem  cls
```

F.2.1 Startup text files:

The TG command file uses the debug utility to perform the necessary program loading. The debug utility gets its commands from either the "cpustop" or "startup" text files. If programs are being run on the PC's 8088 processor, then the cpustop file should be used. The installed X.25 card will then be reset (by reading from I/O port 220 hex), a halt instruction loaded into memory and the card enabled. The card's 8088 will execute the halt instruction and stop (otherwise it could corrupt the memory). If using the card's processor, then the "startup" file is used to load the traffic generator programs and a short startup routine into memory.

```
X.25 TRAFFIC GENERATOR

Welcome to the X.25 traffic generator on the IBM PC.

One X.25 card installed.

Configured for loop back testing (DTE to DTE).

Number of logical channels : 10 per link.
Number data pacs per call  : 60
Average call length        : 10 secs.
Interval between calls     : 2 secs.

Hit C to alter default test configuration, any other key to start >

System initialization, please wait...
```

Fig G.1 The introductory display.

F.3 Program running:

Having compiled and linked the traffic generator programs and edited TG.BAT for the required hardware configuration, the programs may be run by simply typing TG. The introductory display should appear, asking if the user wishes to configure the traffic generator (see fig F.1). If the program fails, check that the ANSI.SYS device driver is included in the DOS CONFIG.SYS file (refer to DOS manual).

Once started, the program is menu driven with the user hitting a key to view the required display (see fig F.2). Hitting an invalid key results in the main screen being displayed, while hitting "x" returns the user to DOS.

```
B>
                        X.25 TRAFFIC GENERATOR

 PACKET LEVEL DISPLAY:
 ====================          Link 0          Link 1
     ACTIVITY:
 Total packets received  :      3708            3706
 Total packets sent      :      3709            3709
 Number active channels  :      10              10

     PACKETS RECEIVED:
 Number of DCE clears    :      0               0
 Number of reset packets :      0               0
 Number of restarts      :      0               0

     STATE:
 No congestion clears    :      0               0
 No diagnostics packets  :      0               0
 Timer expiry counter    :      0               0

 Display options:  S: system,  P: packet level,  L: link level
                   C: configuration,  X: exit,   other: main   ...>
```

Fig G.2 The packet level display.

DEPUTY DIRECTOR
SAPONET (3C23A)
PRIVATE BAG X74
PRETORIA

Thesis terms of reference.

20 AUGUST 1984

STUART ASPIN
15 SEDGEMOOR ROAD
CAMPS BAY
CAPE TOWN
8001

Dear Stuart,

In reply to your letter dated 8 August 1984.

Yes we very definitely require some way of exercising our packet-switching processors with relatively high volumes of traffic. The processors we wish to observe under heavy traffic load conditions have a thruput ranging from 300Kbps to 500Kbps. The exerciser would therefore have to generate at least 250Kbps so that one could be used on the 300Kbps processors and two on the 500Kbps processors. This thruput can be achieved using four 64Kbps links from the 'exerciser'.

It is not particularly important what sort of packets are sent at level 3. It is, however, required that each link maintain at least 6 channels and that twelve calls be established every 30 seconds within the same 1 second interval and then disconnect after 25 seconds. This process is to be repeated for the duration of the exercise.

The exerciser must report the following information at level 2.
(a) Total number of frames sent and received;
(b) Total number of bad FCSs received from the network;
(c) Total number of REJECT frames received from the network;
(d) Total number of aborted frames from network.

It is suggested that the 6 channels on one link are connected to the 6 channels on the second link, similarly for the third and fourth links.

Level 3 information to be reported should be:
(a) Total number of data packets sent per link;
(b) Total number of calls set up per link;
(c) Total number of calls cleared by the network.

These are the brief guidelines for the 'exerciser'. If you have any other specific queries you can direct them either to myself or to Mr A. Da Silva in Cape Town at 460340.

Yours sincerely,

ALAN KNOTT-CRAIG

WESTERN DIGITAL
C O R P O R A T I O N

TECHNICAL MEMO

2445 McCabe Way
Irvine, California 92714
(714) 863-0102 TWX 910-595-1139

MEMO: 192

DEVICE: WD2511A
TITLE: Addendum to WD2511 Data Sheet
DATE: February 15, 1985

The purpose of this Technical Memo is to facilitate your design using the WD2511A.This supercedes all previous Technical Memos (specifically 174 and 189). The user interface of the WD2511 and the WD2511A has not been changed. Therefore, the WD2511A will be replacing the WD2511 in the near future.

We expect to offer the WD2511A in a 68 pin Ceramic chip carrier, suitable for surface mounting.

Please note the following deviations between the WD2511A and the published specifications. We are planning an update to the data book in mid-85.

1. The maximum time for TDAH and TDMW is 125 nanoseconds; the minimum time for THRD is 350 nS. The value of CLK must be 2.0 MHz +/- 1%.

2. The transmit buffer chaining feature is limited to transmission speeds of 100 Kbps or less. Receive chaining, however, does work across the full speed range of the part.

3. The value of the receiver chain address pointer must be greater than OOFF (HEX). Values of 0000 through OOFF are treated as if no buffer is available.

4. When using the 2511A internal loopback mode, CR07 and CR04 should should be set. Furthermore, CTS no longer has to be externally strapped.

5. When reading SR1 to clear an interrupt, reread the register (OR'ing the values of each read) until SR17, SR16 and SR15 are all clear. Remember to read ERO each time SR15 (ERROR) is found to be set.

   When reading the other values in one of the status registers, read the value until the results of two consecutive reads are identical.

Technical Memo
No: 192
Page 2

6. It is suggested that the host implement a timer when presenting buffers to the chip for transmitting so that if an acknowledgement is overdue, the SEND bit can be set again, or the link reset.

7. The 2511A does not expect to receive an unsolicited RR response to clear a remote busy condition; therefore, it polls for the status of the remote station at T1 intervals.

   Conversely the 2511A does not send an unsolicited RR response when its local busy condition clears. It expects the remote station to poll for its status. If this causes a problem, the user can build an RR response and send it using transparent mode.

8. The self test feature of the 2511A still exists, but is subject to removal in future revisions.

9. If the 2511A receives a DISC after sending a SABM, it responds UA in order to comply with an early Telenet specification.

10. There must be at least 500 nS between the rising edge of DACK to the next falling edge of DACK.

11. Changes to CR07 and CR04 must be done while the chip is in the disconnected state (i.e. CR07=MDISC=1.)

15 DEC 1988