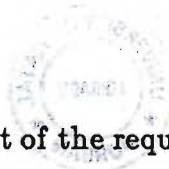


AN INTELLIGENT MULTI-TERMINAL INTERFACE

by

Roger Charles Samuel Peplow



Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Electronic Engineering at the University of Natal, Durban.

Durban

November 1987

Preface

The work described in this thesis was carried out in the Department of Electronic Engineering, University of Natal, Durban, from June 1978 to January 1979, under the supervision of Professor H.L. Nattrass.

This material represents the author's original work except where specific acknowledgement is made, and has not been submitted in part, or in whole, to any other University for degree purposes.

Acknowledgements

My thanks go to my supervisor Professor Lee Nattrass, not only for his guidance during the course of the project, but also for allowing me the time to do the work when the departmental lecturing load has been so high. I also thank him for his patience while waiting for me to complete the thesis which has been awaiting final proof reading for far too long.

Thanks also go to my colleagues Dave Levy and Nils Otte who suffered my ministrations to their computer systems while I was debugging the RMUX interface software. Thanks also for their show of faith in using so many of the final units.

Special thanks must also go to my wife Eleanor for her encouragement, criticism and support and also for her great assistance in the typing of the document.

I would also like to thank Philip Facoline for producing many of the final drawings on his CAD system, Sheila Wright for doing much of the typing, and Tandy Wright for getting much of the final document through the printer.

The document was printed on the KIDRON type setting system in the Department of Electronic Engineering after originally being captured on the HP1000 for the TYPEC text and document processor. The drawings were produced on both a GERBER IDS80 system and a CEADS CAD system and were all plotted on an HP7586 plotter.

Abstract

The document describes the development of a micro-processor based terminal multiplexer to connect four terminals to a standard Hewlett Packard series 1000 mini-computer. The project was required to fulfill the dual roll of both increasing the number of terminals that the HP1000 could support and of reducing the peripheral load on the host CPU.

The final product occupied a standard 200mm square HP size interface card and used an 8085 micro-processor and several 8085 family peripheral chips to provide four full duplex serial channels and a high speed data link with the host.

A multi-tasking executive was written to control the multiplexer software which was finally implemented as 15 independent tasks occupying 8 kilo-bytes of eprom. The software was written to perform all terminal interaction and editing in order to reduce the host CPU involvement to a single interrupt per record.

The resultant interface proved capable of handling an aggregate throughput in excess of 4000 characters per second which was sufficient to cope with all four terminals running at 9600 bits per second, even when all four were transferring in burst mode. The interface also proved to be between five and eighteen times less demanding on the host than the two standard Hewlett Packard interfaces then available. When compared to the low cost HP12531 interface, the multiplexer increased the 9600b/s terminal handling capability of the host from 3 terminals to 52.

Contents

CHAPTER

1.	Introduction	
1.1	Introduction	1
1.2	A Brief look at Peripheral Control	2
2.	An I/O Expander for Terminal Interfaces	5
2.1	Introduction	5
2.2	The HP I/O System	5
2.3	The HP12531 Teletype Interface	7
2.4	The Expander Mainframe	8
2.5	The Terminal Interface	10
2.6	Conclusion	11
3.	An Intelligent Drum Printer Controller	12
3.1	Introduction	12
3.2	Printing on a Drum Printer	13
3.3	The Controller Hardware	14
	The Timing Logic Card	14
	The Processor Module	15
3.4	The Printer Controller Software	16
	The Data Producer	16
	The Data Consumer	16
	Interrupt Tasks	17
3.5	The Message Transfer Protocol	18
3.6	Conclusion	18
4.	The RMUX Terminal Multiplexer— Design Aims	20
4.1	Introduction	20
4.2	A survey of Standard Interface Features	20
4.3	Objectives for an Ideal Interface	22
4.4	The Host—Slave Interface Philosophy	24
4.5	The Slave Micro-processor Requirements	27
	Peripheral Choices	27
	Memory Choice	27
	Memory Address Allocation	28
4.6	Final Hardware Design	28
4.7	Read and Write Signal Decoding	29

5.	The Multi-plexer Interface Operating Software	33
5.1	Introduction	33
5.2	The A8085 Cross Assembler	33
5.3	The Requirements of a Multi-Tasking Executive	35
5.4	Semaphores for Task Control	36
5.5	The MTX Multi Tasking Executive Kernal	37
	The Task Control Block	37
	The Exchange	38
	The Use of Exchanges	39
	Event Calls	39
	Resource Calls	39
5.6	Memory Management Facilities	40
5.7	Task Control Facilities	42
5.8	MTX—A Summary	42
	Interrupt Handling	42
	Coding the System	43
5.9	Time List Handling	43
5.10	Application Task Structure	44
5.11	Host Interaction Tasks	45
	TO__HOST	45
	FROM__HOST	46
	STC__INT	49
5.12	Port Handler Tasks — An Overall View	50
5.13	TX__TASK, the Port Transmit Handler	52
5.14	RX__TASK, the Port Receive Handler	54
	Receive Interrupt Handler	55
5.15	BREAK__TASK, the Unsolicited Interrupt Handler	57
6.	The Host Software Driver Routine — DVX05	59
6.1	Introduction to DVX05	59
6.2	The normal Interrupt Response Procedure	60
6.3	Operating System feature changes — The Philosophy	61
6.4	Mapping System changes — \$DVM5	63
6.5	The layout of the Main Driver — DVX05	65
	The Initiation Section	66
	Configuration of I/O Instructions	67
	Cold Start Initialization	67
	Power Fail Handling	67
	Minimizing Map Changes	67
	Request Processing	68
	The Completion Section	69
	Timeout Processing	69

	Normal Interrupt Processing	71
6.6	Timing considerations	71
	Cold Start Delay	71
	Driver Exit Delay	72
	CLC to driver-output delay	72
7.	Performance Measurements and Results	74
7.1	Introduction	74
7.2	Interface Timing Measurements	74
7.3	Host Throughput Measurements	76
8.	Conclusion	80
	APPENDICES	82
	Appendix A. A terminal I/O Extender for the HP1000	82
	Appendix B. Honeywell Line Printer Controller – Hardware Description	92
	Appendix C. Honeywell Line Printer Controller – Software Description	99
	Appendix D. Honeywell Line Printer Controller – Host Software	105
	Appendix E. RMUX Hardware Description	108
	Appendix F. MTX Software Description and User Manual	126
	Appendix G. RMUX System Software	143
	Appendix H. The RTE IVB Input/Output System	162
	Appendix I. The host driver to control the RMUX Interface	168
	Appendix J. The 8085 Assembler-A8085	183
	Appendix K. The RMUX Users Manual	196

FIGURES

2.1	Standard HP I/O interface control logic	6
2.2	Master control card schematic	9
4.1	Host—Slave Interaction Policy	25
4.2	The Ideal Slave System Interface	26
4.3	Recommended 8085—8237 Connection Scheme	30
4.4	74LS257 Quad Multiplexer construction	31
4.5	Read and Write Signal Routing	31
5.1	TO__HOST Task Layout	46
5.2	FROM__HOST Task Layout	47
5.3	STC__INT—The Interrupt Handler	49
5.4	Partial Listing of &RXDRV	51
5.5	An Overview of TX__TASK, the Output Handler	53
5.6	Long Mode Block Transfer Sequence	56
6.1	The Driver Mapping Table (DMT) format.	63
6.2	Mapping System Changes for Read Requests	64
6.3	The Initiation section of DVX05	66
6.4	The Completion section of DVX05	70
A.1	Timing of flag, Control and Interrupt Logic	82
A.2	Standard HP I/O Interface Logic	83
A.3	Priority and Interrupt Logic in the I/O	84
A.4	Master Control Board Logic	86
A.4	Terminal Interface Logic	88
A.6	Terminal Interface board Layout	91
B.1	Timing Diagram for the Timing Generator	93
B.2	Timing Generator Module Schematic	94
B.3	Printer Controller Module Schematic	97
C.1	Printer control programme. Module Layout	99
C.2	Print Request Record Format	102
D.1	The Hexadecimal to ASCII coding scheme	106
E.1	RMUX Hardware Block Diagram	109
E.2	RMUX Address Map	113
E.3	Backplane Data Input Timing	116
E.4	RMUX Multiplexer Schematic	123
E.5	Component Side Track Layout	124
E.6	Solder Side Track Layout	125
F.1	Exchange Format	127
F.2	Format of an Exchange Table Entry	129
F.3	Task Control Block Format(TCB)	130
F.4	Message Queue Header Table	131
F.5	Memory Buffer Format	132

F.6	Sample Master Control File	142
G.1	RMUX Multiplexer Message Flow Diagram	144
G.2	Flow Chart for ^STC INT Interrupt Routine	146
G.3	Flow Chart for the ^FROM HOST Routine	148
G.4	Flow Chart for the ^TO HOST Routine	149
G.5	Flow Chart for Terminal Task TX TASK	151
G.6	Flow Chart for TX CONT Interrupt Handler	152
G.7	Message Buffer Format	153
G.8	Receive Interrupt Handler Flow Chart	156
G.9	Flow Chart for BREAK Task and Interrupt	158
H.1	Equipment Table Entry Format	163
I.1	Read/Write request message format	170
I.2	Control request message format	171
I.3	Message packet input flow	177
I.4	Message packet output flow	178
I.5	EQT entry usage by DVX05	179
I.6	Status bits in EQT5	179
I.7	Driver return error codes (B5-3 of EQT5)	180
I.8	Read/Write Conword bit definition	180
I.9	Control request conword bit definition	180
K.1	Control 30B call configuration parameter	198
K.2	XON/XOFF style handshake protocol	199
K.3	HP style handshake protocol	199
K.4	QUME style handshake protocol	200
K.5	Special character processing for normal	210
K.6	Control calls for the RMUX interface	202
K.7	Read and write calls for the RMUX interface	202

TABLES

7.1	Interface Timing Results	75
7.2	Host Efficiency with different Interfaces	78
A.1	48 pin Terminal Connector Assignments	89
A.2	116 pin Interface backplane signals	90
B.1	Printer Signal Definitions	98
C.1	Print Record Control Characters	101
E.1	Serial Port Address and Pin Assignments	115
E.2	48 pin Peripheral Connector Assignments	120
E.3	86 way Backplane Connector Assignments	121
E.4	Parts list for RMUX Multiplexer	122
G.1	Conword Option bits for Read and Write Calls	159
G.2	Control Request Options	160
G.3	Configure Request Format (control 30B call)	160
G.4	Special Character Processing for Read Request	161
K.1	Signal pinouts on terminal interface	204
K.2	DTE connections according to RS232C	206

Glossary

ACK	ASCII character for acknowledge. Value 05H. (CNTRL F).
ADSTB	Address strobe. A signal strobe generated by the INTEL 8237 DMA controller.
AEN	Address enable. A signal strobe generated by the INTEL 8085 microprocessor.
ASCII	American Standard Code for Information Interchange.
ASMI	The original 8085 cross assembler that ran on the HP1000 computer.
AUTOR	The power fail – auto re-start programme used by RTE-IVB.
BACI	Buffered Asynchronous Communications Interface for HP1000 series mini-computers.
BS	ASCII character for back space. Value 08H. (CNTRL H).
CCZ	Character Count Zero. A logic signal used in the Honeywell line printer controller.
CHS	Character Strobe. A logic signal used in the Honeywell line printer controller.
CIC	Central Interrupt Control. The interrupt control programme used by RTE-IVB.
CLC	Clear Control. A logic signal and an assembler instruction used in the HP1000.
CLF	Clear Flag. A logic signal and an assembler instruction used in the HP1000.
CMOS	Complementary Metal Oxide Semi-conductor.
CNTL	Abbreviation for control. Applied to the control key of an ASCII keyboard
CNTRL	see CNTL.
CNWRD	Abbreviation for the Control Word in an EXEC call as used in RTE-IVB.
CONWORD	see CNWRD. (alternate abbreviation)
CONWRD	see CNWRD. (alternate abbreviation)
CR	ASCII character for Carriage Return. Value 0DH. (CNTL M)
CRS	Controlled Reset. A logic signal used in the HP1000 I/O device interfaces.
CTCBAD	Current Task Control Block Address. A variable used in the MTX executive.
CTCBID	Current Task Control Block Identifier. A variable used in the MTX executive.
CTS	Clear To Send. A logic signal defined for the RS-232-C serial interface.
DCPC	Dual Channel Port Controller. The DMA logic card in the HP1000.
DEL	The ASCII character for Delete. Value 7FH.
DMA	Direct Memory Access.
DMAC	Direct Memory Access Controller. The mnemonic used by INTEL for their 8237 controller IC.
DRQ	DMA Request. A logic signal.
DSR	Data Set Ready. A logic signal defined for the RS-232-C serial interface.
DTE	Data Terminal Equipment. An RS-232-C term used to refer to the terminal devices in a terminal to modem link.
DTR	Data Terminal Ready. A logic signal defined for the RS-232-C serial interface.
EIA	Electronic Industries Association.
ENF	Enable Flag. A logic timing signal (T2) used in the HP1000 I/O system.
ENQ	The ASCII character for Enquire. Value 05H. (CNTL E)
EOB	End Of Block. A logic signal generated by the DMAC.
EOT	The ASCII character for End Of Tape. Value 04H. (CNTL D).
EPROM	Electrically Programmable Read Only Memory. A term applied to a popular class of memory IC's.
EQT	Equipment Table. A major table in the RTE-IVB system.
ESC	The ASCII character for Escape. Value 1BH. (CNTL []).
ETX	The ASCII character for End Of Text. Value 03H. (CNTL C).

EXCHG	Exchange. A major data structure in the MTX executive.
EXEC	The subroutine call used to access all RTE-IVB system features.
FF	The ASCII character for Form Feed. Value 0CH. (CNTL L).
FIFO	First In – First Out memory system.
FLBF	Flag Buffer Flip Flop. A logic flip flop on the HP1000 interface subsystem.
FLG	Flag Flip Flop. A logic flip flop on the HP1000 Interface subsystem.
FMGR	File Manager. The name given to the RTE-IVB command line interpreter and file management subsystem.
GLITCHES	A slang term commonly used to refer to brief unwanted voltage spikes on logic signals.
GND	A term applied to the logic signal ground or zero volts.
HASHING	A technique for storing symbols in a table by computing the storage location from the symbol itself.
HBEN	High Byte Enable. A logic signal used on the RMUX.
HED	The 8085 cross assembler directive code to specify a heading.
HOLDA	Hold Acknowledge. A logic signal used by the 8085 processor.
HOLDR	Hold Request. A logic signal used by the 8085 processor.
HP	A registered abbreviation for Hewlett Packard.
IAK	Interrupt Acknowledge. A logic signal used in the HP1000 I/O system.
IDS80	The model name for the main CAD system produced by GERBER SYSTEMS TECHNOLOGY prior to 1984.
IEN	Interrupt Enable. A logic signal used in the HP1000 I/O system.
IFNZ	If Not Zero. An assembler directive used in the A8085 cross assembler.
IFX	If 'X'. An assembler directive used in the HP1000 RTE-IVB assembler.
IFZ	If Zero. An assembler directive used in the A8085 cross assembler.
INTA	Interrupt Acknowledge. A logic signal used by the 8085 processor.
INTR	Interrupt Request. A logic signal used by the 8085 processor.
IOG	I/O group. A logic signal used in the HP1000 I/O system.
IOI	I/O Input. A logic signal used in the HP1000 I/O system.
IOO	I/O Output. A logic signal used in the HP1000 I/O system.
IOR	I/O Read. A logic signal generated by the 8085 processor.
IOW	I/O Write. A logic signal generated by the 8085 processor.
IRQ	Interrupt Request. A logic signal used in the HP1000 I/O system.
JMP	Jump. An assembler instruction for the 8085.
JSB	Jump Subroutine. An assembler instruction for the 8085.
KBAUD	Abbreviation for Kilo-Baud, the signal frequency of a data line. Commonly used (incorrectly) in place of bits per second.
KBIT	Kilo-bit. 1024 bits of data.
KBYTE	Kilo-byte. 1024 bytes (8 bits) of data.
KHZ	Kilo-hertz. 1000 Hertz or cycles per second. A unit of frequency.
LBEN	Low Byte Enable. A logic signal used in the RMUX.
LDA	An assembler mnemonic for load accumulator used in both 8085 and HP1000 assemblers.
LF	The ASCII character for Line Feed. Value 0AH. (CNTL J)
LIA	Load into A. An assembler instruction for the HP1000.
LSB	An abbreviation for the Least significant bit/byte.
LSI	Large Scale Integration.

LU	Logical Unit. A number used in the RTE-IVB system to describe any addressable device.
LXI	An assembler instruction for the 8085 processor to load a double register with a constant.
MEMR	Memory Read. A logic signal used by the 8237 DMA controller.
MEMW	Memory Write. A logic signal used by the 8237 DMA controller.
MERG	An assembler directive defined for the A8085 cross assembler to merge files.
MHF	Manual Head of Form. A logic signal used by the Honeywell line printer.
MHZ	Mega-Hertz. 1 000 000 hertz or cycles per second. A unit of frequency.
MIA	Merge into A. An assembler instruction for the HP1000.
MPU	Micro-processor Unit.
MSB	An abbreviation for the Most significant bit/byte.
MSI	Medium Scale Integration.
MSS	Manual single space. A logic signal used in the Honeywell line printer.
MTX	The name given to the small multi-tasking executive written for the RMUX.
MVI	Move Immediate. An assembler instruction for the 8085.
NAMR	The term used to fully describe a file in RTE-IVB which includes several sub-fields.
NAND	Negative AND. A standard boolean logic gate.
NS	Nano-seconds. A unit of time.
NSEC	see NS.
NSEG	New segment. An assembler directive defined for the A8085 assembler.
NYBBLES	A term used to describe 4 bits of data.
OE	Output Enable. A logic signal used by many logic devices.
OOF	Out Of Forms. A logic signal used in the Honeywell line printer.
OPCODE	A term used for the operation code in any assembler instruction.
ORG	Origin. An assembler directive used to set the value of the programme counter.
OTA	Output A. An assembler instruction used in the HP1000.
PAA	Print Address Advance. A logic signal used in the Honeywell line printer.
PAC	Print and Compare. A logic signal used in the Honeywell line printer.
PCB	Printed Circuit Board.
PES	Printer Emergency Stop. A logic signal used in the Honeywell line printer.
PIC	Programmable Interrupt Controller. The Intel 8259 IC.
PLP	Paper Line Pulse. A logic signal used in the Honeywell line printer to signal the movement of the paper.
POPIO	Power On Preset for I/O. A logic signal used in the HP1000 I/O section.
PPLS	Printed circuit Pattern Layout System. A Sperry Univac programme for the layout of pcb's.
PRH	Priority High. A daisy chained logic signal used in the HP1000 I/O section.
PRL	Priority Low. see PRH.
PRMPT	The unsolicited input handler in the RTE-IVB system.
PSW	Programme status word. The flags and accumulator can be combined on the 8085 into a single 16 bit register for stack pushes and pops.
PVH	Paper Velocity High. A logic signal used in the Honeywell line printer to drive the paper feed moter at high speed.
PVL	Paper Velocity Low. (see PVH).
RAM	Random Access Memory. A term usually applied to read and write memory.

RESNET	Resistor Network. A network of thick film resistors which are usually very closely matched thermally.
RET	Return. An assembler instruction to return from a subroutine call.
RMX	The name of the Intel multi-tasking executive for use on 8085 processors.
ROM	Read Only Memory.
RSEG	Replace Segment. A special assembler directive defined in the A8085 cross assembler. (see also NSEG).
RTE	Real Time Executive. The name of the Hewlett Packard real time operating system for HP1000 series computers.
RTIOC	Real Time I/O Controller. The name of the RTE processor used to handle all I/O requests in RTE.
RTS	Request To Send. A logic signal defined in the RS-232-C standard for serial communication.
RXD	Receive Data. A negative true logic signal defined in the RS-232-C standard for serial communication.
RXRDY	Receiver Ready. A logic signal generated by the Intel 8251 serial communication chips.
SBD	Sentinal Bit Detect. A logic signal used in the Honeywell line printer controller to indicate the end of a print buffer.
SC	Select Code. A term applied to the I/O address in the HP1000 system.
SCL	Select Code Low. A logic signal used in the HP1000 to select devices that match the low digit of the octal I/O address.
SCM	Select Code Most. A logic signal used in the HP0100 to select devices that match the high digit of the octal I/O address.
SEL	Select. A logic signal generated in the RMUX when both the SCL and SCM addresses match the interface address.
SFC	Skip if Flag Clear. A logic signal and assembler instruction of the HP1000.
SFS	Skip if Flag Set. (opposite effect to SFC).
SHLD	Store HL direct. An assembled instruction for the 8085.
SID	Serial In Data. The single bit input of the 8085 processor.
SKF	Skip on Flag. A logic signal used in the HP1000 I/O system.
SKP	Skip. An assembler directive to skip a page on the list device.
SPC	Space. An assembler directive to skip lines on the list device.
SRQ	Service Request. A logic signal used in the HP1000 I/O system.
SSI	Small Scale Integration.
STA	Store Accumulator. An assembler instruction for both the 8085 and the HP1000.
STC	Set Control. A logic signal and an assembler instruction in the HP1000.
STF	Set Flag. A logic signal and an assembler instruction in th HP1000.
SVPC	Save Programme Counter. An assembler directive defined in the A8085 assembler for segment manipulation.
SYN	The ASCII character for Sync. Value 16H. (CNTL V).
TCB	Task Control Block. A 72 byte data structure used in the MTX executive.
TCP	True Compare. A logic signal used in the Honeywell line printer.
TLOG	Transmission Log. A count of the characters sent. Used in the RTE-IVB I/O system.
TTL	Transistor transistor logic.

TTY	Teletype. An abbreviation for a simple terminal as typified by the products of the Teletype Corporation of America.
TXD	Transmit Data. A negative true logic signal defined in the RS-232-C standard for serial communication.
UART	Universal Asynchronous Receiver Transmitter.
UNL	Unlist. An assembler directive used to control the listing.
USA	User save. An assembler instruction used in the HP1000 to save the user mapping registers.
USART	Universal Asynchronous/Synchronous Receiver Transmitter.
WR	Write. A logic signal used in the RMUX.
WVA	The location prefix used in the Honeywell line printer to specify the printer logic frame.
XOFF	Transmit off. The ASCII DC3 signal (CNTL S) is usually used for this in band flow control method.
XON	Transmit on. The ASCII DC1 signal (CNTL Q) is usually used for this in band flow control method. (see XOFF).

Chapter 1

1.1 Introduction

Three separate developments all aimed at improving or extending the Input/Output facilities offered on the Hewlett Packard HP1000 series minicomputers are described.

The first project, to extend the number of terminal ports, was developed using discrete logic for all functions. It consisted of a specialized backplane extender to hold up to sixteen interface cards. These interfaces were designed to emulate exactly the standard HP12531 teletype card, but made use of modern MSI and LSI devices to reduce component count and centralize control.

The second project, used as a test bed project for incorporating a microprocessor peripheral controller into a larger system, was the development of a controller for a 300 line per minute line printer. This project illustrated the potential of the microprocessor, as the resulting controller handled nearly every aspect of the printer control, allowing much of the original discrete logic in the printer itself to be removed.

The third and major project was to produce an intelligent, single board terminal multiplexer to connect to four independent asynchronous terminals. A microprocessor and several sophisticated microprocessor peripheral controller chips were used in the design, resulting in a terminal interface that was not only four times as dense as the standard teletype interface but also some thirty times more frugal in its demands on the host machine's time, and offered a far wider range of features.

Chapters five and six describe the software developed for this multiplexer, which consisted of an Intel 8085 cross assembler to run on the HP1000 computer, a semaphore driven multitasking operating system for the 8085, the fifteen communication tasks used to control all the interface functions, and finally, the special two part driver needed to enable the Hewlett Packard RTE-4B operating system to accept four fully independent terminal ports on one interface card.

The report finally describes some of the tests that were carried out on the interface performance. The results obtained show that this interface offers a significant reduction in host machine overhead, even when compared to Hewlett Packard's more sophisticated and expensive interfaces.

As with most development projects, hindsight can produce many new ideas and better ways of implementing something and this project was no exception. Thus, the concluding chapter describes some of the possible options that could be incorporated in the interface, as well as some improvements that could be made in the initial design, now that more sophisticated devices are available.

The first model of this interface has been installed for some five years, and at time of writing, there were more than twenty production versions in operation, indicating that the philosophy of an intelligent multi-terminal interface has produced an effective product.

The remainder of this chapter gives a brief outline of the history of computers with specific reference to the use of intelligent controllers to reduce CPU overhead in peripheral transactions. It is on the basis of this history that this particular project evolved.

1.2 A Brief look at Peripheral Control in Computers

Computers are generally accepted as belonging to one of three categories, namely, mainframe, mini-, and micro-computers. This classification is based roughly on the size of the machine, but is rather vague with many areas of overlap.

Mainframes evolved from the original computers and still tend to occupy several large cabinets of electronics, consume large amounts of power, and require special computer rooms with carefully controlled environments. Their operating systems are usually suited to batch mode processing where the CPU can be kept busy, while their I/O system is suited to moving large quantities of data rapidly between a very small range of peripherals such as discs, magnetic tapes, and line printers.

As computerised control of machinery became a reality, the high cost of mainframe computers when coupled with their high speed and their limited variety of I/O systems made them unsuitable for most control applications. As a result, the minicomputer was born, initially to act as a slave I/O controller for a mainframe, and ultimately as a computer in its own right.

These minicomputers were characterised by small size (typically only one small cabinet), low cost and very versatile I/O systems. They were designed to run real time control programmes using their versatile interrupt systems to capture data which was then passed onto the host mainframe for processing.

In time, these real time programmes developed into real time operating systems, and the minicomputer developed its own market for computing, complementing rather than competing with the mainframe market.

While minicomputers grew from dedicated controllers into fully fledged computer systems, following the same growth path that mainframes had taken, the integrated circuit revolution gave birth to the microprocessor, a small, limited function computer on a single chip of silicon. Initially developed without a goal, the microprocessor was soon recognised to be an ideal substitute for a minicomputer in many small control applications.

The situation had now evolved where mainframes were too large for control supervision activities, being more suited to running accounts, orders and all other administrative functions of an organization, while the minicomputer had grown sufficiently to be able to control large sections of a plant. The microprocessor then became the ideal candidate for small control problems, and in this field the microprocessor has flourished.

It is interesting to note that while mainframe manufacturers were quick to incorporate minicomputers into their mainframes as dedicated I/O controllers, the minicomputer manufacturers, traditionally loathe to accept any ideas from their big brothers, were slow to incorporate microprocessors into their machines. In fact the microprocessor manufacturers themselves were far quicker to apply the multiprocessing concept and have produced some very sophisticated peripheral controller chips each incorporating its own microprocessor. The availability of these easy to use, sophisticated support chips has made the microprocessor into an extremely powerful device; now, a scant ten years after its invention, the microprocessor is being used in every form of computing imaginable. It is no longer merely looked upon as a useful I/O controller.

While it was stated that minicomputer manufacturers themselves were slow to adopt the microprocessor as a peripheral controller, many independent manufacturers of minicomputer peripheral controllers have incorporated microprocessors into their products, often with dramatic improvements in cost, speed, size, and ease of use.

The development of the four-terminal intelligent multiplexer was undertaken in order to produce a product which was not available from Hewlett Packard and which would offer improved features due to its programmability.

Chapter 2

An I/O Expander for Terminal Interfaces

2.1 Introduction

The Hewlett Packard HP1000 series minicomputer used by the Department of Electronic Engineering of the University of Natal (Durban), has a maximum I/O addressing capability of 56 device interfaces, of which only 14 may be installed in the computer cabinet itself—any further interfaces requiring an extender cabinet. With the advent of the multi-user operating systems of RTE-3 and RTE-4B, the need for terminals quickly exhausted the available slots in the mainframe. The solutions were either to buy an extender and the required terminal interfaces from Hewlett Packard, or to design a suitable extender and build it. To save funds, and also to provide a medium for an in depth study of the HP1000 I/O system, the latter approach was adopted and a 16 card terminal extender system was developed.

The interface cards were designed to emulate the standard HP12531 teletype interface from both the computer's and the terminal's point of view. Although this saved writing driver software, the resultant card which used an LSI UART (Large Scale Integration Universal Asynchronous Receiver/Transmitter) required that several UART features had to be ignored or even overridden to produce the desired compatibility. In retrospect, this was not the wisest solution to the I/O extension as the extender became obsolete after only four years due partly to its poor performance forced by the emulation.

The remainder of this chapter describes briefly the development of this extender with the principal emphasis on those aspects which influenced the final multiplexer project.

2.2 The HP I/O System

This section describes only those aspects of the HP I/O system that are pertinent to the peripheral controller itself: in this case the serial I/O device. For a more detailed description of the I/O system, see Appendices A and H and the HP reference manual.^[1]

Any HP1000 series interface can be divided into two major sections: the section which contains the specific I/O controller, and the section needed to implement the CPU's interrupt and priority structure which is distributed over all interface cards. This latter section (shown in Fig.2.1) uses some 30 logic gates and typically requires 8 or 9 SSI TTL packages to implement. From this interrupt logic block, an STC strobe, an output strobe (IOO) and an input strobe (IOI) emanate, while a single interrupt input (STF) exists. These are the only signals available for controlling the peripheral section which also has access to a 16 bit bi-directional data bus. The function of the STC strobe is to enable the CPU to command the peripheral section to perform some action without requiring any data transfer.

The limitation of only having one interrupt per device, and no extra signal bits for interrupt source identification, results in the I/O system limiting all I/O to half duplex communication, irrespective of the peripheral section requirements.

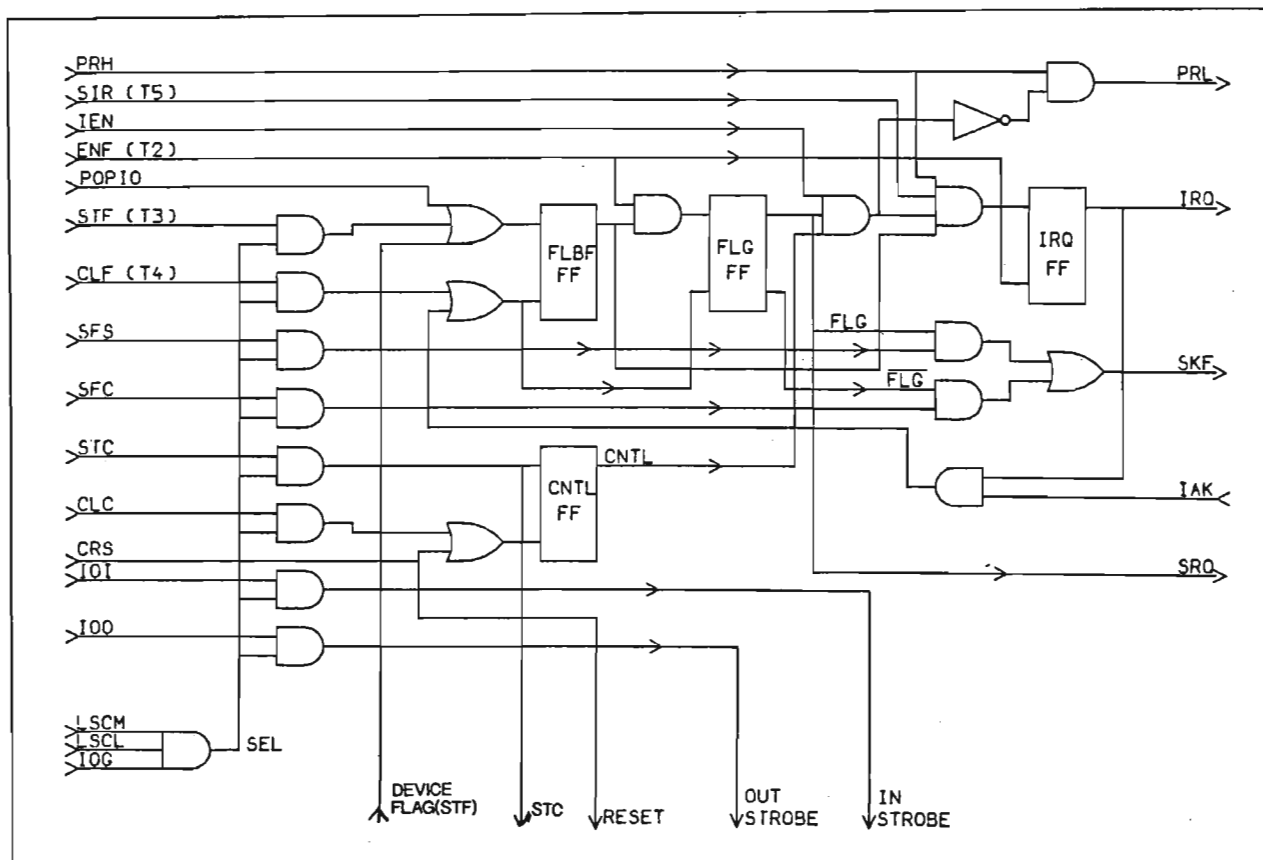


Figure 2.1 Standard HP I/O interface control logic

2.3 The HP12531 Teletype Interface

This interface, driven by the driver DVR00, was, at the time of the extender development, the most common serial interface for HP1000 series minicomputers, hence the decision to design the new terminal extender to be compatible. The most important features of this interface which influenced the design are covered briefly below, but for full details see the HP12531 interface specification.^[2]

The serial I/O side of the HP12531 is implemented using a single 11-bit shift register for both input and output, which enforces the half duplex nature of the interface and requires some means of programming the direction of transfer. This is done by using an IN/OUT flip flop programmed from the upper (otherwise unused) bits of the data word. Thus, to program the interface, a data word is output with bit 15 set to act as a program enable bit. To stop the lower 8 bits of this word being transmitted however, the logic was designed so that the output instruction merely latches the data into the output shift register and an STC instruction is required to initiate the actual transmission.

In order to be able to detect any input characters arriving during output, the input to the shift register is left connected to the data line and the shift register contents are examined after each character transmission. Should any input character have arrived during this time, then the shift register would not contain all ones, as it would have had it just clocked in an idle line. Although this feature does not enable the input character to be determined, it is used by DVR00 to determine when a user wishes to interrupt an output stream, an otherwise difficult problem in a half duplex system such as this.

Another problem which can arise from such a simple system with no data buffering is that of an overrun error which occurs when a new character arrives at the serial input of the interface before the CPU has read the previous character. This situation is dealt with in the HP12531 plus DVR00 setup by gating the input to the shift register once a character has arrived, and leaving it closed until the character has been read. This stops any further input from corrupting the character already in the shift register. In order to detect the arrival of another character, a flip flop is set whenever the input line changes to a zero (e.g. as for a start bit) and the state of the flip flop is read back as bit 15 of the returned data, thus acting as an overrun indicator. Since there is no buffering on this card, the CPU only has one half of a stop bit time in which to respond before the next character may start, a limitation which causes frequent overrun errors.

Other features of the card were:

- (a) On card crystal controlled baud rate clock.
- (b) RS232 and 20mA current loop inputs and outputs.
- (c) Capability to turn on either printer or punch on a teletype.
- (d) Ability to run from an external clock of 8 times the data rate.
- (e) No facilities for MODEM control.

2.4 The Expander Mainframe

In developing the 16 slot terminal expander, it was decided to:

- (a) Make as much use of MSI and LSI as possible.
- (b) Centralize as many functions as possible rather than replicating them on each card.
- (c) Customize the extender mainframe specifically for terminal interfaces.
- (d) Make the unit daisy chainable so several extenders could be added if required.

The extender was designed to connect to the computer via the HP Multiplexer I/O Accessory kit^[3], which brings out all the backplane signals, buffered, onto a 100 way edge connector. A 50 pair cable connected this card to the extender, on which was mounted another 'T'-connected 100-way edge connector for the daisy chaining of extenders. The signals were also connected to the extender control card, which buffered all the signals onto the extender backplane (see Appendix A).

In an effort to reduce component count on the interface cards themselves, the interrupt request logic and priority encoding logic was moved to a master control card as shown in Fig.2.2. This achieved a saving of 2 to 3 chips on each interface card at the cost of about 5 extra chips on the master control card. More significant was the time saving that was achieved in the priority logic by using a priority encoder rather than a discrete gate on each interface. This reduced priority settling time from 16 gate delays to 3 gate delays. To save having individual baud rate generators on each interface card, a central baud rate generator was included on the master control card, common baud rate clocks being produced and wired down the backplane. Finally address selector links were added to the master card so that it could be set to answer to any block of sixteen consecutive addresses.

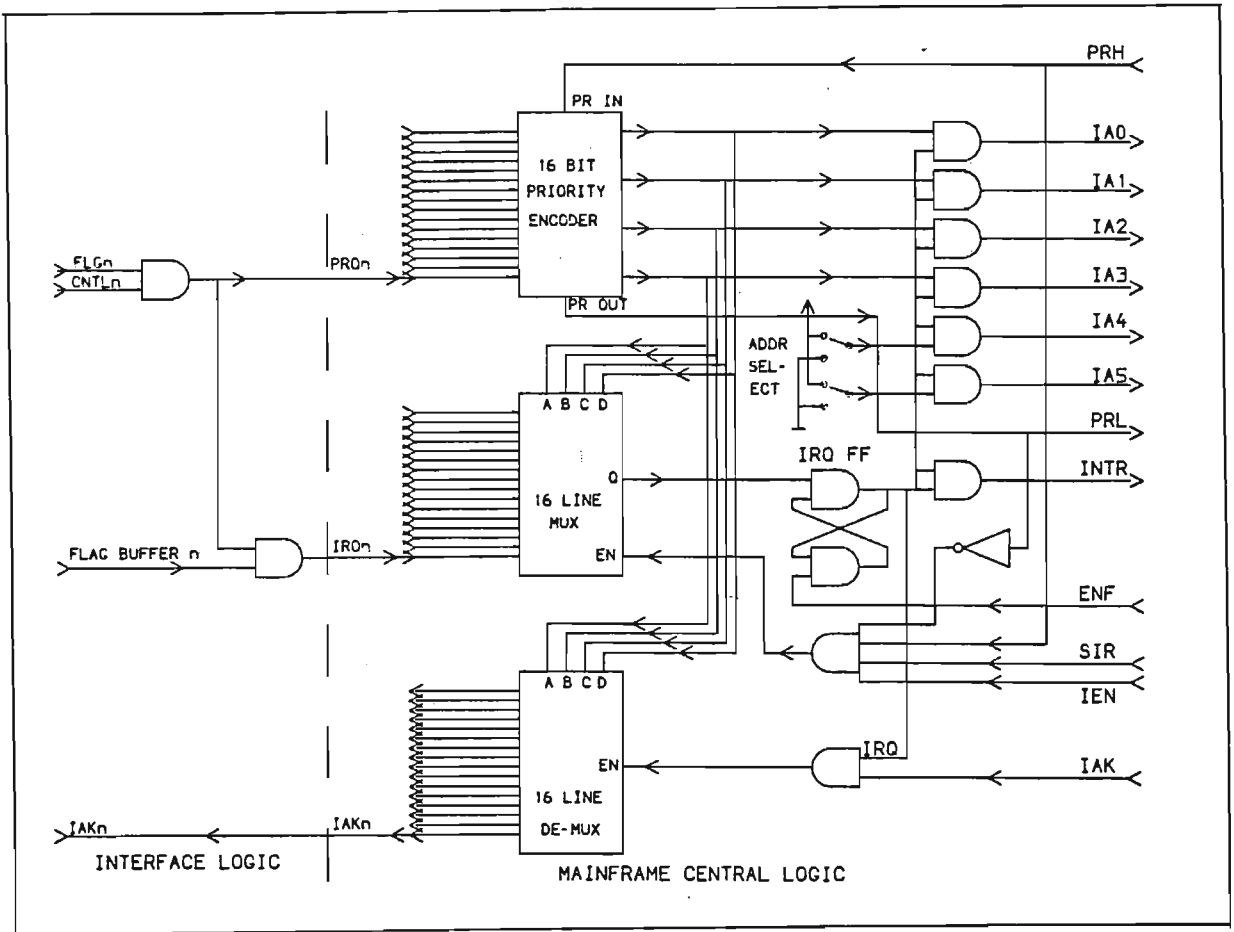


Figure 2.2 Master control card schematic.

The extender frame itself consisted of a standard 4 unit high RETMA rackmount cabinet with the power supply in the front section and the interface cards plugging into the rear section from the top. This concept of using top insert cards was only adopted to cope with the large connector hoods used on standard HP12531 interfaces, an otherwise poor idea owing the problem of plugging cards in when the unit is installed in a full rack. Either the unit needed to be slide mounted and have plenty of cable slack to allow the unit to slide out, or there had to be plenty of room above the extender, so wasting rack space.

2.5 The Terminal Interface

The interface card itself was built using a 40 pin LSI UART (Universal Asynchronous Receiver Transmitter) as the main functional unit. This UART performs every peripheral function required of a non modem compatible RS232 serial interface, and had the interface not been designed to emulate an HP12531, this UART alone would probably have been all that was necessary.

To allow for the output command to latch the data and an STC instruction to initiate transmission, an extra 8 bit latch had to be included between the data bus and the UART, while to supply an "all ones" return character whenever there was no valid character to read, required the addition of an extra eight bit tri-state buffer. This buffer was only enabled for the duration of the first read command subsequent to each received character. Thus, should the host execute a read command with no data available, the floating tri-state data bus would return the "all ones" result required by DVR00.

Due to the fact that the UART has completely separate transmit and receive sections, both double buffered, the overrun error problem which was so obvious with the HP12531 was much less so with the UART since the CPU has an entire character time to respond to an interrupt and read the character before the next input character can interfere. However to maintain exact compatibility with the HP12531, a BUSY IN flip flop was added to the interface and connected to return bit 15. This flip flop, as in the HP12531, gets cleared whenever a character is correctly received and set again whenever the input data line drops to a zero, thus allowing DVR00 to determine whether a character was arriving during a read of the previous character. Since the UART could be programmed for parity, stop bits and data word length (all features unavailable on the HP12531) switches were added to the interface to allow these options to be selected. This, while not compatible, did not interfere with DVR00 operation and could still be set to be compatible should the need arise.

The baud rate selection was implemented using a 12-way set of DIP rocker switches, allowing selection of any one of the 10 preset baud rate clocks from the backplane, or an 8 times or 16 times external clock. Since the UART requires a 16 times clock, and the HP12531 an 8 times clock, a frequency doubler had to be included on the interface. Fortunately, the UART was found to be almost totally insensitive to the mark-space ratio of the clock, allowing a simple exclusive-nor, edge sensitive pulse generator to be used for the doubling. Another non compatible feature which could be added since it did not affect DVR00 operation

was to return the three UART error signals, Framing Error, Overrun Error and Parity Error in bits 14-12. This was included to allow for possible use by some future driver, should one ever be written.

2.6 Conclusion

This project succeeded in its aims in that it supplied fourteen extra terminal interfaces which were used for four years, at about eight percent of the cost of a comparable system from Hewlett Packard, as well as providing an incentive for a thorough investigation into the HP1000's I/O system. Perhaps more than anything else, the half duplex nature of this entire system was highlighted by installing a full duplex device and then having to add extra logic to degrade the functions of this cheap and effective LSI UART in order to achieve compatibility.

In packaging the extender, several small but nevertheless important lessons were learnt:

- a) Using a manually taped PCB master resulted in excessive tolerance on edge connector fingers such that the circuit boards could be inserted so as to cause shorts or missed connections.
- b) The backplane connectors were of the "cut to length" variety in which the end stops are inserted after cutting. These connectors exhibit excessive tolerance such that when the cabinet was twisted in any way, the resultant movement of the cards invariably resulted in connector shorting.
- c) Using a layout which required a top insertion board was not a good idea at all, and so boards should be made for front or rear insertion.
- d) No edge connectors were gold plated and when the power was switched off each night, the thermal cycling and its attendant expansion and contraction cycles caused connectors to "shuffle" giving many bad connection problems. The solution was to gold plate the connector fingers and to leave the power applied continuously.

This expander did produce a useful addition to the computer's complement of terminals, and remained in service for some four years until it was made obsolete by the four terminal multiplexer described later on in this report.

Chapter 3

An Intelligent Controller for a Drum Printer

3.1 Introduction

The investigation into DVR00 and its HP12531 teletype interface, plus an isolated effort at working on the Centronics line printer driver (DVR12), illustrated clearly how intimately involved the CPU had to become with each of the peripherals it used. It was certainly a long way from the ideal situation where the host should expect the peripheral to manage all the peripheral dependent features. Now while this attitude was understandable in the past due to the expense and difficulty in producing "ideal" peripheral controllers, the arrival of the cheap and powerful microprocessor should enable "ideal" peripheral controllers to be built cheaply.

To test this view and gain some experience before attempting the final project of an intelligent four terminal multiplexer, the author designed a controller for a 300 line per minute Honeywell drum printer which had been donated, controller-less, to the Department of Electronic Engineering. This printer contained a rack of some 30 circuit boards, each 100mm square; it should also have had a controller of unknown size and, after all this, the Honeywell host CPU was still required to output the entire line of characters to be printed sixty three times, once for each character row on the print drum. This task, which meant that the CPU had to output a character every 6 microseconds, must have kept the original CPU fully occupied for the entire 200 milliseconds taken to print a line, an obvious waste of CPU time.

The final design used an 8085 microprocessor situated in the printer and communicating with the host (HP CPU) over a serial line using a very simple message protocol which was chosen to be almost completely device independent. Thus, the custom driver written for the HP was short, simple and contained no code specifically related to handling a line printer.

Since the printer controller was basically designed as a test bed for ideas to be incorporated into the final multiplexer project, and since both the printer and the printer controller were relatively complex, this Chapter covers these factors very superficially and concentrates mainly on the software layout, the use of a DMA controller and the message protocol. Appendices B, C and D cover the hardware, software and driver in more detail.

3.2 Printing on a Drum Printer

The Honeywell 112 printer is a drum printer capable of printing 300 lines per minute. To accomplish this, the printer contains a character drum rotating at 300 rpm directly in front of the paper. The drum has 63 rows of raised characters, each row containing 132 identical characters, one in each column position. Behind the paper, a row of 132 solenoid driven hammers is situated, while an inked ribbon passes between the paper and the print drum. To print a line of text, as each different row of characters comes up in front of the print hammers, the hammers are fired only in those column positions where the particular character is desired. Thus, the logic has to scan the line of text to be printed each time a new character line appears under the hammers, and decide which hammers are to be fired. Since a new line of characters appears every 3,2 ms and a print line may contain 132 characters, each character comparison would have to be performed in 24 microseconds. In practice, this time is reduced to 6–9 microseconds since much of the 3,2 ms interval must be devoted to the actual printing process of setting up and firing the hammers. Thus, any controller would have to execute the following sequence for every line of text to be printed:

- (a) Fetch the text line to be printed and store it in the scan buffer.
- (b) Await a strobe which indicates that a new line of characters is about to come up under the hammer.
- (c) Determine which character it is that is about to come under the hammer and be printed.
- (d) Scan the scan buffer, checking for the print character and where found, shift a '1' into a 132 bit print buffer, while for every false comparison shift a '0' into the print buffer.
- (e) Once the scan buffer has been completely scanned, pad the print buffer with '0's until all 132 positions are full.
- (f) Send a strobe to initiate the hammer firing process.
- (g) Check if all the characters in the scan buffer have been printed — if so, go to (a) for the next line after moving the paper, if not, return to (b) for the next print character.

This cycle occurs every 3,2 ms.

The controller would also have to look after all the paper feed control and page formatting. To do this, it was required to become part of the servo loop and control the paper feed motor speed depending upon how many lines of paper were to be skipped. This fortunately was a simple problem as the paper feed motor only possessed two speeds.

3.3 The Controller Hardware

It was obviously impossible for a microprocessor to perform the print line scanning under software, so it was decided to incorporate a hardware comparator driven by data from a DMA controller and timing signals from a small timing logic circuit. This left the microprocessor software to handle all the data input, data sorting, printing setup and paper formatting.

The Timing Logic Card

The timing and comparison logic was included on a separate card from the processor module, with signals arranged so that a simple test module could be used to test the printer in place of the microprocessor module. The logic and timing diagram of this card are shown in Appendix B (Figures B1 and B2).

The timing logic starts a sequence by first synchronising the character strobe (CHS) from the print drum with its own nine microsecond clock and then issuing the first DMA request (DRQ). This would cause the DMA controller to output the first character of the scan buffer which is then passed through two mapping eproms (described below) and two comparators. On the next clock pulse, the result of this comparison (TCP) is clocked into the print shift register (in the existing printer logic) by the printer address advance signal (PAA) while the next DMA request is issued. This sequence continues for the entire line until the shift register "sentinel" bit stops the process. On the next character strobe (CHS), the DMA controller resets to the start of the scan buffer and repeats the sequence from the start of the line again.

The dual comparators with mapping eproms were included to allow over-printing of characters since the print drum was missing several special characters which had to be made up (e.g. A "\$" was made up from an "S" and an "I"). Thus when the DMA logic outputs a "\$" code, the mapping eproms change it to an "S" and an "I" to feed the two comparators.

The Processor Module

Apart from the microprocessor itself and a DMA controller, the following major chips were included:

- (a) A serial interface UART to control communications with the host.
- (b) One kilobyte of RAM to contain a circular data buffer.
- (c) Four kilobytes of EPROM to contain the operating code.
- (d) A combination I/O – ram chip which supplied a counter for counting the characters as they are printed, a 5 bit output port, a 7 bit input port and 256 bytes of RAM used to store all operating variables.
- (e) A six bit output port to hold the character output by the DMA controller.
- (f) A baud rate generator and RS232 data buffers for the serial communications line.

Although the code occupied less than 2K of eprom, a 4Kbyte 2732 chip was chosen since it was easy to programme and allowed room for possible expansion while the difference in cost between the 2Kbyte and 4Kbyte chips was about R2.00.

The RAM although only 1K in length was given a 4K address space so that it could be treated as a circular buffer by the DMA controller. Thus when the DMA controller addressed past the end of the first 1K, the beginning of the RAM would respond again, giving the so-called circular effect.

The counter was used to count all the TCP (true compare) pulses, and when all printable characters in the line had been printed, the counter output was used to inform the processor. This was required since the printing process was under control of the DMA controller and could not otherwise be monitored by the CPU.

The I/O lines of the I/O port chip were used to monitor the front panel control signals and to drive various control signals within the printer such as the paper feed motor amplifier.

The entire processor circuitry (see Fig.B3) including the serial I/O connector was incorporated onto a single 100 mm square wire wrap board which was plugged into the main card frame.

3.4 The Printer Controller Software

The software written to control the line printer was initially conceived as two separate processes, the data producer being the receive input handler, and the data consumer being the actual printer handler with the only junction between them being the 1K circular data buffer and a few flags. To co-ordinate these two essentially independent tasks, the producer was run continuously in a background mode, while the consumer was run every 3,2 ms in a time base interrupt driven mode.

The Data Producer

The main receive data handler (RECEVE) was split into two parts:

- (a) The normal input processor was to accept print requests from the host via the serial line, store them in a circular buffer, and stop the flow of data from the host should the buffer ever become too full to handle another complete print request. Transmission would then be held in suspension until such time as only two print requests remained in the circular buffer. This approach was designed to keep the line busy in large blocks and hence reduce programme swapping in the host. The only other function of RECEVE was to convert incoming text data from the ASCII code used by the host to the Honeywell code used by the printer, and to count the number of printable characters in each request record. This printable character count was used as explained in Section 3.3 to determine the end of the printing process.
- (b) A terminal debug monitor (GETCM) which could be run in place of RECEVE allowing the contents of memory to be examined and/or altered. A single bit switch, attached to one of the input port lines was used at power up time to direct the processor to GETCM instead of the normal processor - RECEVE. By using GETCM, it was possible to place data requests into memory, alter pointers and counters and then monitor the printing process as these requests were processed. This was used during the development phase to aid in debugging the code, and was subsequently used whenever some hardware failure occurred in the printer logic.

The Data Consumer (RST6.5)

This routine, driven from the 3,2 ms character strobe interrupt constituted the most significant process in the controller as it had to supervise and co-ordinate the entire printing process.

Upon entry, each 3,2ms, it was required to first determine what state (or phase) the printer was in which was done by means of several flags (see Appendix C). If in the print phase, the character counter was checked to see whether the phase was complete, while if paper feeding was in process, then the remaining line count was examined to determine the motor speed required or to delay the necessary settling time after the motor had been stopped. If neither printing nor paper feed was in operation, then the input buffer would be examined for any new requests. All new requests were parsed to determine the desired action and then either the printing or the paper feeding process initiated.

The printer control process was used for several other minor tasks, some of the more interesting being:

- (a) To allow for the analogue paper feed servo loop to settle, the routine was entered 3 times after the paper feed motor had been stopped before starting any printing. This allowed a 10ms stabilising time.
- (b) Front panel controls were monitored so that should the printer be placed "OFF LINE" then the remaining front panel controls of "LINE FEED", "FORM FEED" and "SINGLE ORDER BUTTON" were enabled.
- (c) The page position was monitored to maintain the perforation skipover function (if enabled) and to stop multiple form feeds from wasting paper.

Interrupt Tasks

Two other interrupt driven tasks were included in the system, one driven by the paper feed motor position indicator to count lines as the paper moved and the other to fetch the input characters from the host.

Since the host could transmit at up to 19200 bits per second, characters could arrive every 500 microseconds but with the consumer process being interrupt driven, and able to run for longer than 500 microseconds, the background task (RECEVE) was incapable of polling the UART and reading all characters. Thus the UART was made to generate an interrupt with each input character and the interrupt handler then stored these characters into a sixteen character FIFO which was emptied by RECEVE. This character buffering effectively overcame the timing problem resulting from having one major task interrupt scheduled and the other running continuously.

3.5 The Message Transfer Protocol

Each request from the host was sent with a fixed format starting with an ASCII ENQ character to synchronise the message reception. Following the ENQ the host then sent the data field length, coded into two ASCII characters, in order that the printer controller could check available space left in the data buffer. Should insufficient space exist for the entire print request, the DTR line of the serial interface would be set false. This condition was to stop the host from sending any further data until such time as space became available whereupon the DTR line was set true again.

The next datum sent was a single control character derived directly from the host equipment table entry (EQT) (see Appendix H), and used to indicate the request type. Following this was the print data (if any) terminated by a Return (CR).

3.6 Conclusion

The completion of this project produced several valuable lessons to be applied to any future project, these being related to the message format, the usage of DMA in micro-processors and the software system structure.

The data message format chosen had the advantage of being an almost direct transfer from original request format of length, type and data, through the host EQT and driver, down the serial data line and into the circular buffer. Thus, the only time that any action was taken on the data content was at the printing stage in the controller itself. This made the driver and the message format essentially peripheral independent which was felt to be an important aspect in reducing host CPU load in peripheral handling.

The DMA controller as used in this project demonstrated the simplicity of adding a complex function such as DMA to a microprocessor system, and also showed how, with the addition of little extra logic, a DMA controller could make feasible a project which would be completely impossible to implement in software alone.

The major lesson in this project related to the difficulty of coping with independent concurrent tasks without an operating system to control them. Since only two tasks were involved in this project, and one of them (the Data Consumer) was very time dependent, the solution opted for was adequate to

afford each task a sufficient share of the resources. The only problem resulting from this approach was that of character input which was easily solved with the FIFO buffer. However, the number of flags used to coordinate these two activities was quite large for such a simple interaction and left a very vivid picture of how complex it could become were more tasks to be included in the structure.

Chapter 4

The RMUX Terminal Multiplexer—Design aims

4.1 Introduction

This Chapter sets out the design aims and objectives of the RMUX intelligent four terminal multiplexer, starting with a brief look at the features offered by standard interfaces existing at that time. Following this, the overall schematic of an “ideal” multiplexer is presented, containing all the features felt to be both desirable and possible. Due primarily to space limitations, not all these features could be included in a practical design, so the final design concept is then presented with reasons for all the features removed or reduced.

The conclusion of this chapter then attempts to explain how these required features influenced the hardware design and forced the selection of certain components. The final hardware design is presented in detail in Appendix E.

4.2 A Survey of Standard Interface Features

At the time of starting the design of the RMUX interface, Hewlett Packard produced two standard serial asynchronous terminal interfaces, the HP12531 teletype interface described in Chapter 2 and Appendix A, and the HP12966 Buffered Asynchronous Communications Interface (BACI).

The HP12531 offered the following features:

- (a) Both RS232 and 20mA current loop signal levels.
- (b) Only 8 bit data words without any parity option.
- (c) Five link selectable baud rates plus an external 8 times clock.
- (d) Half duplex transmission only.
- (e) Every character caused a host CPU interrupt.
- (f) Driven by DVR00 which occupied only 700 words of memory.
- (g) The driver overhead processing time for a single character interrupt of 350 microseconds resulted in a maximum character rate for DVR00 of 2880 characters per second for a fully extended host CPU.

The HP12966 BACI was a more modern interface which in an effort to reduce CPU loading contained a 128 character FIFO and a 128 character content addressable RAM for special character recognition. This interface in conjunction with its driver (DVR05) offered the following features:

- (a) RS232 signal levels only.
- (b) Modem control signals compatible with the Bell 103 standard.
- (c) Programmable number of stop bits, parity and data character length.
- (d) Wire jumper or programme selection of one of 14 baud rates or a 16 times external clock.
- (e) Half duplex transmission only.
- (f) Could be set to interrupt on every character, or when the FIFO becomes half full (64 characters), or upon the recognition of one of the programmable special characters.
- (g) The interface driver DVR05 occupied 1500 words of memory.
- (h) The average driver processing time for long records resulted in a maximum character rate of about 10 000 characters per second before the host became saturated.

Of the two interfaces, the DVR05/HP12966 set offered far more features and reduced CPU loading by almost four times compared to the DVR00/HP12531 pair. The FIFO buffering on the HP12966 resulted in lines shorter than 64 characters long needing only one interrupt per line of text, rather than one per character, hence offering a considerable potential increase in speed. Unfortunately, much of this potential was lost due to the complexity of the processing needed to service this extremely complex discrete logic card.

In order for DVR05 to be able to programme the HP12966 as well as pass it data, while only having a single 16 bit I/O data path, the top four bits of any word sent to the card are used to specify the data function. This feature adds to the CPU overhead and also requires that the CPU unpack all characters prior to output. Similarly, for input from the HP12966, the upper byte of the returned word contains status information which has to be checked, while the lower byte contains the data character which requires packing into the user buffer. This extra processing required for each character eliminated much of the advantage gained by reducing interrupts to one per block.

DVR05 uses handshake sequences with the terminal for both input and output in order to determine the state of readiness of the terminal. On output, which is typically the major traffic direction in terminal I/O, the handshake consists of an enquiry character (ENQ) sent every 33 characters to which the terminal must reply with an acknowledge (ACK) when it has space to accept the data. On input, DVR05 first issues a trigger character (DC1) before setting the interface card into a read mode. This is done so as to inform the terminal that it may send data. Both these handshake sequences add several extra interrupts to each record, further increasing the average CPU time per character.

The terminals that DVR05 was designed to drive can be quite complex units with separately addressable keyboards, alphanumeric screen, graphics screen overlay, printer, plotter and cassette tape drives. Furthermore, data from the terminal can be transferred character by character, record at a time in block mode, or a full screen at a time in page block mode. All these options and variations make DVR05 an exceptionally long and complex driver by HP standards, there being virtually no other driver in the system of this length.

4.3 Objectives for an "Ideal" Interface

From observing the way that users tend to interact with terminals, and from noting the types of terminal in common use in many installations, several observations could be made:

- (a) A very high proportion of terminal usage time is spent reading or editing alphanumeric text
- (b) Text editing benefits from frequent re-displays of the text around the most recently edited data resulting in frequent page outputs and hence, very high output data rates compared to input data rates. This input to output ratio for editing typically lies between 1:100 and 1:1000.
- (c) Since full feature terminals are expensive and their extra features appear to seldom be used in most installations (e.g. printers and cassette tapes), the majority of terminals used are straightforward text editing terminals.
- (d) During large streams of output it is often necessary for the user to interrupt the output, gain the system's attention and perhaps take some corrective action to stop the output. This should be done by pressing a single "BREAK" key of some form. Due to its half duplex nature, the HP12966 seldom notices a "BREAK" key-in and the resultant *panic key bashing* of a frustrated user is damaging to both user and terminal.

These factors, when taken in conjunction with the features described in Section 4.2, plus a strong desire to produce an efficient interface with straightforward host interaction resulted in the following design objectives:

- (a) All requests from the host should use a common format when being forwarded to the interface.
- (b) All requests should be passed to or from the interface in single burst mode block transfers.
- (c) Transfers in either direction should be 16 bits wide and capable of running very fast. Ideally, the full DMA burst mode rate of 1 million 16 bit words per second should be possible.
- (d) The interface should perform all request parsing and command interpretation, leaving the host driver merely to act as a medium for passing the requests back and forth.
- (e) All requests whether read, write, control or status should originate from the host and be acknowledged when the interface has completed processing them.
- (f) All handshake sequences should be handled by the interface quite independently from the host.
- (g) The interface need only handle text editing type terminals using either character or line mode transfers and need not attempt a full emulation of all DVR05 functions.
- (h) It should handle four terminals, all running at at least 9600 bits per second without any degradation in performance compared with running a single terminal.
- (i) User "break" interrupts should supply immediate response and should not be missed at all.
- (j) All standard line edit functions such as backspace and delete should be handled by the interface.

These objectives split naturally into two groups, those that would determine how the software handled messages, and those that would determine the hardware; while the hardware layout itself split naturally into three areas:

- (a) The standard HP flag, control and interrupt logic.
- (b) The microprocessor and its peripherals.
- (c) The interface between the HP host and the microprocessor.

Of these three hardware design areas, the first two were relatively trivial and pre-determined, but the design of the interface between host and slave offered an interesting challenge.

4.4 The Host-Slave Interface Philosophy

The desire to have the interface pack and unpack the 16 bit data from the host meant that the upper byte of each word could not be used for control or status information and since no other signals exist in the HP I/O structure, control and status information would have to be packaged serially with the data. This meant that messages would have to have some means of synchronisation built into them. Furthermore, since up to four independent requests could be active at any time, it was essential to ensure that each request would be transferred as a discrete block with its synchronisation and identification fields, to eliminate any possible mix up of requests that could occur should request transfers be interrupted in the middle and interleaved.

In order to reduce host driver processing time to a minimum, it was necessary that the slave always be ready to accept requests from the host at any speed that the host requires. Further, when the slave has a return request available for the host, it should make this available for the host to fetch at any speed, before informing the host.

To satisfy these desires, the slave had to include a Direct Memory Access (DMA) channel capable of accepting 16 bit data from the host at any time and unpacking it into slave memory at speeds of up to 2 M bytes per second. Thus the DMA controller should at all times be primed to accept data into a free request buffer and only when a request has been fully transferred, both header and data, should the slave be interrupted by the host. The slave response time to remove the full buffer and re-prime the DMA controller with a new empty buffer should be less than the minimum time that the host can take to initiate sending a new request. To cope with the return data path, a second DMA channel capable of 2 M bytes per second transfers and packing 16 bit data words, should be primed with the return buffer before interrupting the host.

Figure 4.1 illustrates these two independent data transfer processes in pseudo code form.

```

Procedure FROM_HOST;
Begin
  Fetch_free_buffer (Buffer_address);
  Prime_dma_controller (Buffer_address, Buffer_Length);
  DO Forever
    Old_buff_address := Buffer_address;
    Fetch_free_buffer (Buffer_address);
    Wait_for_interrupt (From_Host_interrupt);
    Prime_dma_controller (Buffer_address, Buffer_Length);
    Post_buffer (Old_buffer_address);
  END
END;

Procedure TO_HOST;
Begin
  DO Forever
    Fetch_return_buffer (Buffer_address);
    Prime_dma_controller (Buffer_address, Buffer_Length);
    Send_Interrupt (To_Host_interrupt);
    Wait_for_interrupt (From_Host_interrupt);
    Return_free_buffer (Buffer_address);
  END
END;

```

Figure 4.1 Host—Slave Interaction Policy

A survey of available 8 bit microprocessors with compatible two channel DMA controllers showed the Intel 8085 to offer the most satisfactory and compact solution. Furthermore, the fastest version (a 5 MHz 8085A-2) when used with an 8237-2 four channel DMA controller afforded a burst DMA data rate of 2,5 M bytes per second with a long block average of about 2,4 M bytes per second, a significant achievement for a microprocessor. This combination would allow the host to output data under its DMA system at full rate of 1 M words per second. However, a direct connection of the two DMA channels (host to slave) would not result in the desired rate, as the two different speeds would cause both machines to interleave DMA and normal CPU cycles. The resultant DMA latency on each word would considerably slow this data rate down. To overcome this problem, some form of intermediate buffering would be necessary, an IC FIFO being the ideal answer. Since single chip 4 bit wide FIFO's were available, the addition of 4 extra chips would have allowed the desired data rates to be achieved under both input and output conditions.

Figure 4.2 shows the block schematic of the slave system which would have coped with the desired host-slave interface.

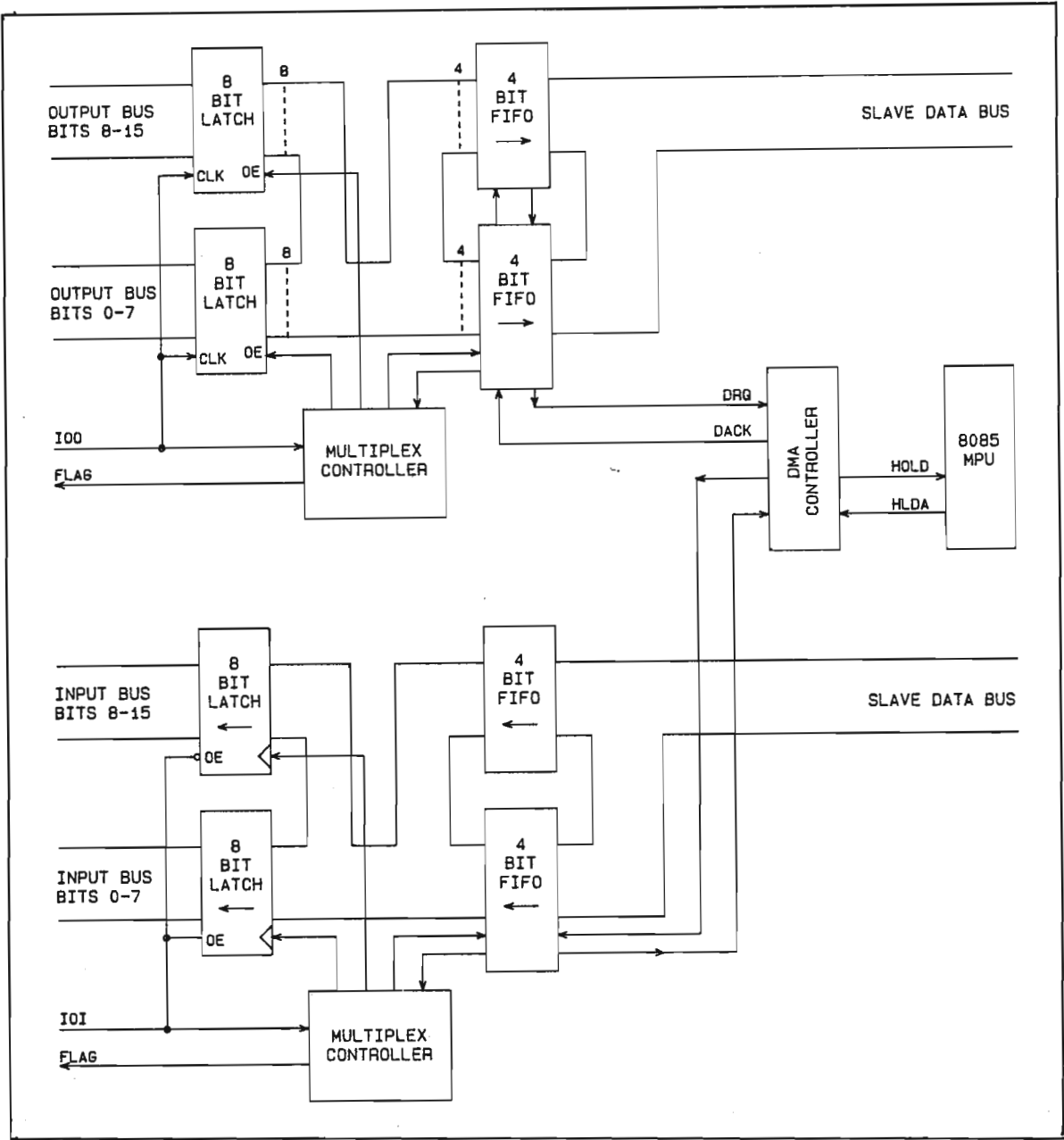


Figure 4.2 The “Ideal” Slave System Interface

One important limitation which became obvious at this preliminary stage was that the FLAG line to the host was to act as a transfer handshake line for both input and output, as well as an interrupt line to signify the readiness of a return request buffer. This restricted input, output and interrupt operations to be mutually exclusive even though from the microprocessor side they were all quite independent.

This, however, became a software sequencing problem and did not affect the hardware design.

4.5 The Slave Microprocessor Requirements

Designing a microprocessor system is principally a matter of deciding what facilities are required from the system, finding suitable IC's to implement them and then connecting all these blocks together according to manufacturers instructions.

Peripheral Choices

In choosing peripheral devices, there had to be four USART's to supply the four serial ports, a DMA controller, an eight channel interrupt controller for handling port interrupts and five programmable clock generators to supply the four baud rate clocks and a system time base interrupt. There also had to be two other interrupts into the MPU, one from the system time base generator, and one from the host CPU. These, however, were available on the 8085 MPU itself and thus did not require any extra components.

The serial port and DMA requirements could only be satisfied by using four 8251A USART's and an 8237 DMAC, but there were two choices for the clock generators. The best choice was a single 40 pin AM9513, five channel counter-timer, but a manufacturing hitch delayed production of this device for over a year, so the second choice of two 24 pin 8253, three channel counter-timers was used. Since this device only contains 14 bit counters, two channels in cascade were necessary to provide sufficient division to reduce the 1.5 MHz input clock to the 100 Hz clock used as a time base generator for the operating system.

Memory Choice

In choosing memory, it was decided that 8K bytes of programme memory would be adequate so two 2732 EPROMS were included, while for RAM, 4K bytes was about all that could be fitted on the card, and it was felt adequate to cope with four ports. In choosing the RAM, 4K by 1 chips were chosen over the cheaper more popular 1K by 4 chips as the latter devices caused the data bus capacitive loading limits to be exceeded. Since the difference in cost was about R8.00 total, and a data bus buffer was saved, the tradeoff was felt worth it.

Memory Address Allocation

With eight peripheral devices and three memory blocks, the easiest form of address decoding was used, which was to give every device its own 4K block of memory. This left five unassigned blocks for any possible last minute extras or future expansion and resulted in very simple decoding logic.

4.6 Final Hardware design

While all the preceding design choices represented a wish list, some features had to be reduced or eliminated for several reasons.

The principal problem that resulted from trying to layout a printed circuit board to standard HP interface card dimensions was that of space shortage. It seemed impossible to use a two layer PCB to contain all the flag and interrupt logic, the interface logic and the full microprocessor system, so some reductions had to be made.

The one area where some choice did exist was that of the interface logic where the hardware multiplexing and demultiplexing used several devices and was not essential. Thus the demultiplexing of 8 bit data back onto the 16 bit CPU bus was removed saving approximately 2,5 chips. The output direction multiplexing was left, as with the predominance of output, it was likely to improve the average CPU performance, whereas input from terminals is so rare, averaging 20 or 30 characters per minute, that the extra overhead of doing software packing was anticipated to be insignificant.

The HP1000 series machine only has two DMA channels which may be dynamically reconfigured to handle any interface. Typically, the disk uses one of these continuously and the magnetic tape drives use the other. Some consideration and checking showed the average terminal I/O request to be some 50 bytes long and the word transfer rate under programme control to be 7 microseconds per word. This results in an average record transfer time of 175 microseconds under programme control compared to a possible 25 microseconds under DMA control. The saving of 150 microseconds per record would have to be offset against the time taken for the CPU to free a DMA channel and allocate it to the multiplexer driver; the resultant saving being negligible. Thus, it was considered unnecessary to run the interface under DMA in the host, and so the FIFO's could be removed, saving a further five devices.

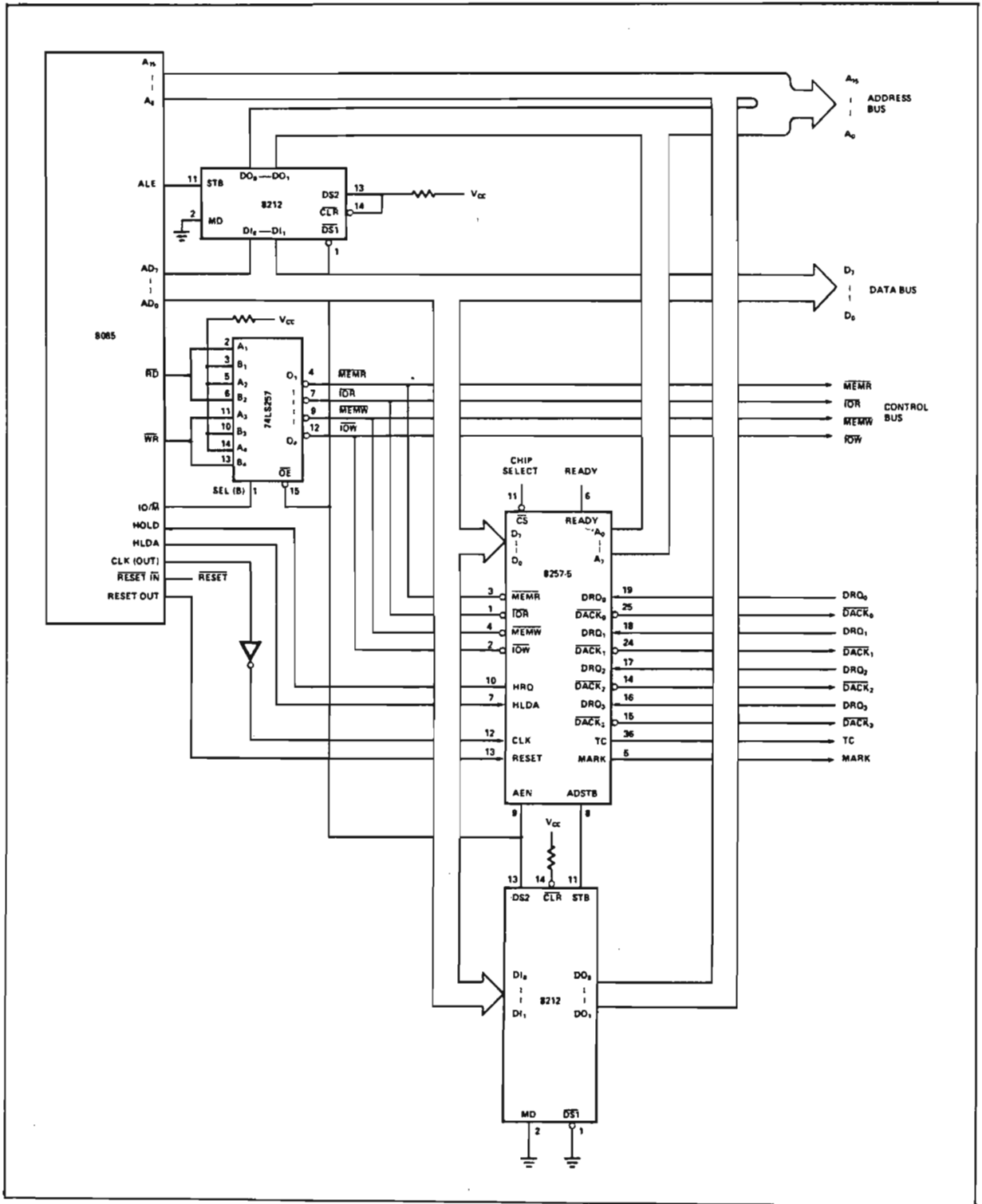
The removal of the FIFO's and the input demultiplexing circuitry saved enough space to allow the remainder of the logic to fit onto a double-sided, through-plated PCB. To save some space from power traces, two layer, perpendicular power buses were used between rows of chips, and the high capacitance and low inductance of these power buses allowed a considerable reduction in the number of discrete capacitors required to maintain a smooth supply system.

The final layout was performed by a non-graphic auto-routing package called PPLS [4] which was run on the main campus Univac 1100/10 computer. The total run time for this job was in excess of 2 CPU hours and about ten iterations were required before the final circuit of only five unconnected traces was achieved. These traces were quite easily added manually by moving some existing traces slightly.

4.7 Read and Write Signal Decoding

One of the minor design problems involved the incompatibility between the read and write signals of the 8237 DMA controller and those of the 8085 MPU. The 8237 uses 4 strobe signals ($\overline{\text{MEMR}}$, $\overline{\text{MEMW}}$, $\overline{\text{IOR}}$, $\overline{\text{IOW}}$) to indicate read or write from or to I/O or memory, whereas the 8085 uses only three signals ($\text{IO}/\overline{\text{M}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$) to do the same job. Since both devices must drive these read/write control lines, one set had to be converted to the other. The easiest way of doing this was to use a quad two input tri-state multiplexer, as recommended by Intel^[5], to convert the 8085 signals to the four line 8237 requirements. Figure 4.3 shows the connection for this conversion.

During testing of the interface, an intermittent problem existed which caused the DMA controller to revert to the power-up initialise state every so often. This problem caused endless hours of heartache until the fault disappeared when a logic analyser probe was left attached to the $\overline{\text{IOW}}$ line. Close investigation showed that a 20nsec glitch was generated on this line by the multiplexer whenever the $\text{IO}/\overline{\text{M}}$ line changed from low to high. An examination of the multiplexer construction (Figure 4.4) showed that this was due to the propagation delay of the second gate in the select (S) line which could result in both AND gates of each multiplexer having a low input with the resultant low going output glitch.



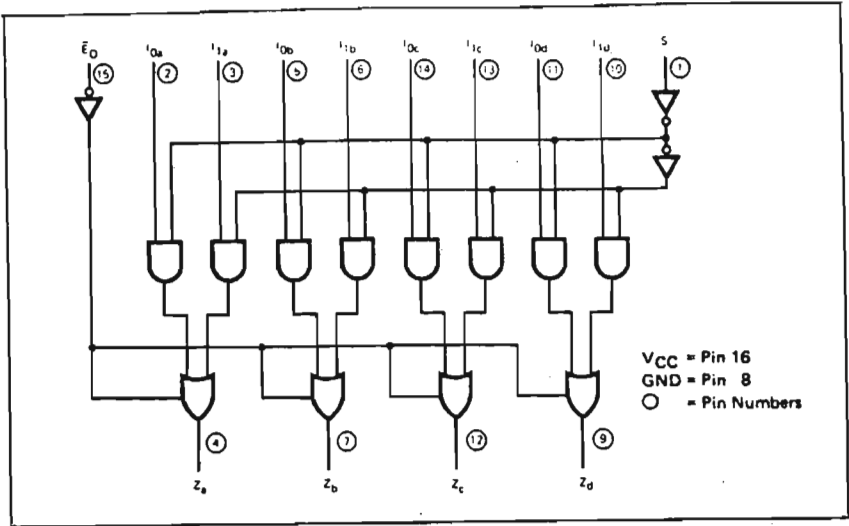


Figure 4.4. 74LS257 Quad Multiplexer Construction (Motorola)

Although the 8237 DMA controller should not legitimately respond to a 20 ns glitch on its IOW line, it appeared that this glitch could provoke some internal chaos resulting in a complete reset of the chip. No simple solution using discrete devices could be thought of to eliminate this glitch, and there was insufficient board space to use a more complex solution. Thus it was decided to remove the multiplexer completely and couple the MEMR and MEMW lines to the IOR and IOW lines for all devices except the DMA controller in which the MEMx lines were disconnected. (See Fig.4.5 for the final connection scheme). This meant that all devices would respond to both memory and I/O addresses, but due to the chip select decoding chosen previously, this caused no overlap or problems.

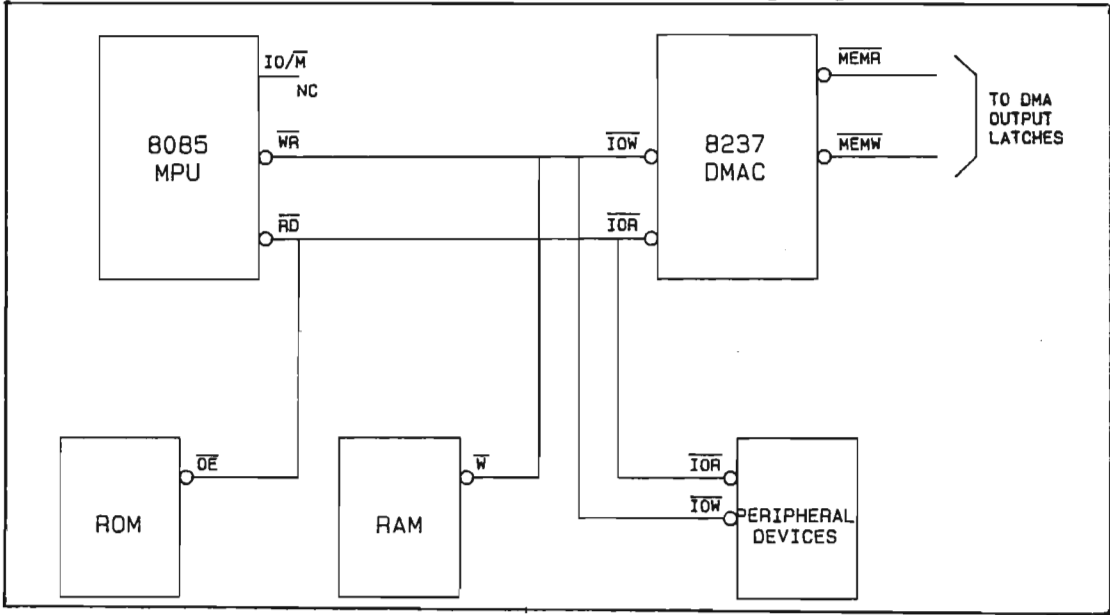


Figure 4.5 Read and Write Signal Routing

The DMA controller, when in slave mode could be addressed as either memory or I/O, but when in master mode, its I/O control lines were connected to the memory lines so that the DMA controller was effectively a memory mapped device. This is a perfectly legitimate way of connecting a DMA controller, the only change being that read and write transfers become reversed when programming the controller.

Thus the fix to this problem eventually resulted in a saving of one chip and the reversal of two commands in the software. Whether all DMA controllers would respond the same way to this glitch is unknown, but two devices were tried, both with the same results. The problem may not always occur when using the multiplexer for a decoder because the addition of a 12 pf probe was sufficient to damp the glitch and stop the reset. This indicated that with a different layout, the glitch may be rendered harmless.

Appendix E gives a full schematic drawing of the interface as generated on a Gerber IDS 80 Computer Aided Design system as well as a full description of the circuit operation.

Chapter 5

The Multiplexer Interface Operating Software

5.1 Introduction

Before starting on the design of the software for the multiplexer, the language to use had to be chosen. At that time, the only language for an 8085 for which a compiler was available in the Department of Electronic Engineering was a cross-assembler running on the HP1000 system. This meant that either assembly language be used, or a high level language compiler be obtained or written. No FORTRAN or PASCAL (the two preferred high level languages) cross-compilers could be found, so it was decided to write the code in a relaxed specification PASCAL (for structuring purposes) and then manually compile this into assembly language.

Before the software generation procedure had gone very far, it became evident that the available cross-assembler (ASMI) was frustratingly slow and also lacked some features which would be very useful. A new cross-assembler was written (see Appendix J for a user manual and full description) and several useful and unusual features were included which eased the software generation task considerably.

Since a logical well structured approach to code generation is fostered by the use of an operating system which can co-ordinate multiple, independant tasks and allow tasks to co-operate in the use of shared resources, it was decided to use a general purpose multi-tasking operating system for the kernel of the code. The Intel RMX 80 system was considered for this function, but was rejected for being too expensive to obtain and for using too much memory. Thus a small multi-tasking system (MTX) was written. It included a scheduler, a memory buffer manager and a resource management system. The final operating code was then designed as fifteen separate tasks, interaction between tasks being confined to standard operating system calls.

5.2 The A8085 Cross Assembler

The cross assembler was written in ALGOL because PASCAL, (the preferred choice) was very poorly implemented on the HP1000 at that time, being very slow in compiling and using a tremendous amount of memory. The main aim of increasing the speed of assembly was achieved by hashing both symbol and instruction tables, using an intermediate scratch file between passes one and two to eliminate line parsing in pass two, and paying careful attention to speed in those areas used frequently. These factors resulted in a basic assembler capable of assembling over 3500 lines per minute.

After the basic assembler was written and running, several other features were devised and included:

- (a) A pseudo code to allow files to be merged during assembly.
- (b) A memory segmentation scheme to allow code to be allocated to any user defined segment.
- (c) Resettable variables using the 'SET' instruction .
- (d) Ten character symbol names.
- (e) Variables allowed to be either local to a particular file or global to the entire run.
- (f) Comprehensive conditional assembly commands.

These features added significantly to the versatility of the assembler, but did slow it down to about 2000 lines a minute, the main culprit being the long symbol names.

The memory segmentation scheme was incorporated to cater for the need to separate programme code and variable data into ROM and RAM respectively. This feature — unnecessary for programmes loaded directly into RAM — becomes a necessity in stand alone microprocessor systems. The alternative was for no segmentation to exist and to specify all RAM based variables together before assigning any code to ROM. In a 4000 line assembly, this approach separates the data from its programme, making modular coding techniques impractical and reducing programme legibility.

To segment memory, the segments are first defined and given names using the NSEG (new segment) assembler directive:

```
EG      ROM:  NSEG  1000H
          RAM:  NSEG  4000H
```

A segment is enabled by the RSEG (Replace Segment) directive which saves the next available address of the current segment, and re-instates the requested segment:

```
EG      RSEG  ROM
          code goes into Rom
          RSEG  RAM
          data goes into Ram
```

The multiple source file capability was included to allow separate logical blocks of code to reside in separate files, allowing for easier editing and handling while also providing a logical structure. Making all symbols (or labels) local to the file unless specifically declared as global (prefix of ^) significantly eased the generation of

meaningful labels, (since the same label could be used in several files) and made programmes easier to follow as the scope of all labels is immediately obvious. These features plus all the others mentioned previously are described fully in Appendix J.

5.3 The Requirements of a Multi-tasking Executive

The function of a multi-tasking executive is to allow many independent tasks to utilize a single processor without affecting one another or altering one another's data. The executive is essentially a suite of programmes supplying facilities to the user tasks in such a manner that each user task believes it has the full processor to itself. To create the situation where a task's run time can be controlled, the executive has to have some means of determining the natural delay points that occur in most tasks (particularly real time ones), and use these delay times to give other tasks a share of the processor.

Since the most common delay in any task is when the task is waiting for some peripheral device to respond, it follows that one useful function for a multi-tasking executive to supply would be some means of synchronising tasks to peripheral hardware devices.

In a system of tasks where each task accesses only its own I/O devices and where there is no inter-relationship between any tasks, an adequate executive would only need to have some means of stopping and starting the tasks and of controlling the peripheral I/O wait times. However, this situation would be very rare indeed, the reality being that tasks will be inter-related, will be likely to share peripherals and memory and often alter one another's data. In this situation, the executive needs to supply additional facilities to allow user tasks to co-operate, share resources without conflict or corruption, and signal one another for synchronisation.

To supply the facilities required above, a fairly simple system need only consist of the following subsystems:

- (a) A scheduler and dispatcher to initiate suspended programmes that are ready to run.
- (b) A resource manager to handle resource lock requests and to suspend requests for resources already locked (a resource here may be data, a peripheral or memory).
- (c) An inter-task communication manager to pass signals and messages between tasks, suspending tasks that await a signal until the signal has been issued.

In operation, any user task wishing to access a peripheral or any other resource must first call the resource manager. Should the resource not be free, the resource manager will suspend the task and pass control to the scheduler to determine

whether any other tasks are ready to run. The choice of the task to run next will then be made and the task will then be passed to the dispatcher to initiate execution. Similarly, any task wishing to synchronise itself with some event (such as the completion of some peripheral action) will call the communication manager to wait for a signal from the peripheral interrupt handler. The task will then be suspended until such time as the signal is sent, and the scheduler releases the task again. In this fashion, tasks can be controlled, allowing several independent tasks to share the same processor without each task ever realizing that it is being suspended, or that it is sharing the processor.

5.4 Semaphores for Task Control

The semaphore, as originally proposed by Dijkstra ^[6] and discussed extensively by Brinch Hansen ^[7] in his almost classical primer on operating systems, is an ideal mechanism for coordinating concurrent tasks and in some form or other has become the basis or core of most real time operating systems.

The general valued semaphore is defined as a non-negative valued integer S which is operated on by two primitive operations $P(S)$ and $V(S)$ both of which must be indivisible. The operations are defined by:

$V(S)$ increment the value of S .

$P(S)$ for $S=0$ suspend the calling task until $S>0$

for $S>0$ decrement the value of S

A more common implementation is the binary valued semaphore in which S is constrained to only take on the values zero and one. Then $V(S)$ will set S to a one, and $P(S)$ sets S to a zero if it is non zero, otherwise the calling task is suspended.

To use semaphores for task control, two forms of the semaphore can be defined:

- (a) The *resource* semaphore
- (b) The *signal* semaphore

The *resource* semaphore is used for resource management by associating a particular semaphore S with each shared resource in the system. Then any task wishing to use a resource must first of all execute a $P(S)$ operation on the associated resource. If all tasks wishing to use the resource always execute a P -operation before using the resource and then execute a V -operation when finished only one task at a time will ever be able to access the resource.

The *signal* semaphore requires the normal semaphore action to be reversed, so that a task wishing to wait for some signal must execute a $P(S)$ operation on the semaphore associated with the signal but since the task must be suspended, the

semaphore must start life in the zero state. This will cause the required suspension until some other task executes a $V(S)$ operation on the semaphore thus releasing the waiting task. Thus, the semaphore can be used for both resource control and event signaling, the only difference being that resource semaphores initialise in the free state while signal semaphores initialise in the busy state.

5.5 The “MTX” Multi Tasking Executive Kernal

While the semaphore provides the means to control concurrent tasks, some further data constructs are necessary to allow task definition and to maintain the task suspend lists.

The MTX system was designed around two basic structures:

- (a) The *Task Control Block* (TCB) for task definition.
- (b) The *Exchange* to enable suspended task's TCB's to be maintained.

The Task Control Block

The function of a task control block is to define the status of a task, maintaining all static task descriptors as well as all temporary and dynamic data that is valid when the task is suspended. The static information is usually the task identification and initial start up address, while dynamic data that must be saved during suspension consists of CPU register contents and the programme stack with all the subroutine return addresses etc.

In MTX, the TCB was chosen to be 72 bytes long, 14 bytes for predefined data and the 58 bytes for a stack. The 14 byte header of the TCB contains:

- (a) A two byte pointer for linking TCB's into lists.
- (b) Two bytes to hold the value of the stack pointer during suspension.
- (c) A status byte.
- (d) A TCB (or task) identity number.
- (e) A 2 byte pointer for linking TCB's into the time list. This feature allows suspension while waiting for some resource to be time limited.
- (f) The timeout clock itself. A sixteen bit integer count of the 10ms timebase intervals left to run before the timeout expires. When a TCB is linked into the time list, this value is decremented every 10ms by the timeout processor.
- (g) Four spare bytes for possible future expansion.

The TCB was not made to hold the task start address as this value was placed in a separate ROM based jump table to ease the task startup process.

Before choosing the stack size at 58 bytes, all the code was written and the stack usage of each subroutine determined. The largest amount of stack used by any task was less than 32 bytes, so 32 bytes were allowed for normal usage, while the remaining 26 bytes were allowed for interrupt usage. In the worst possible case, interrupts can be nested three levels deep and since each interrupt handler must save all the registers it wishes to use, interrupt stack usage was high. It is in situations such as these that the single stack machine becomes a nuisance, as here every TCB has to have 26 bytes of stack space available just to allow for the interrupts when that task's stack is in use. A machine with separate user and system stacks for use by interrupt procedures, would have made coding easier and saved nearly 400 bytes of RAM.

The Exchange

The exchange entry is the core of the MTX data structure and consists of an eight byte entry for each semaphore in the system. Since any task executing a semaphore lock request is liable to be suspended until the semaphore becomes free, the system requires some means of linking the waiting tasks to the semaphore itself. Thus the exchange contains not only the semaphore itself, but also a linked list header for linking waiting TCB's (via the TCB link field) to the semaphore.

In MTX, all linked list headers have the same format:

- (a) A two byte head link pointer.
- (b) A two byte task link pointer.
- (c) A one byte list length indicator.

The semaphore itself was implemented as a single byte binary state semaphore with zero being defined as free and non zero as busy. By defining busy as non zero rather than one, the ID number of a locking task could be used to fill the semaphore and lock it, while at the same time, providing a means for identifying the task currently locking the semaphore. This was necessary for the situation where a task is aborted while it owns some resources, as the system abort processor can then scan all exchanges to find any owned by the aborted task and free them for subsequent users.

Since an exchange entry could be used as either a resource or as a signal event type semaphore, an indicator byte was included in the exchange to mark its type, a zero value indicating a resource and non zero indicating an event. At startup, event semaphores must assume a locked state, as described earlier, and so the indicator is initially copied into the semaphore to lock events and free resources, thus eliminating any special processing for the two different exchange types.

The Use of the Exchanges

At system startup, task zero is executed removing control of the processor from the operating system. In order for the operating system to regain control, the task must suspend itself against an exchange by awaiting an event (or signal). Thus the startup task should initialize itself, perform some startup processing and then execute a WAIT FOR EVENT call on some exchange. The system will then link the TCB onto the specified exchange and then run the next task.

In this manner, all tasks will eventually become suspended waiting for events or resources, and so the operating system remains in control. The scheduler will then check each exchange in turn to see if any free exchange has a task waiting against it. Should such an exchange with a zero semaphore and a non zero list length exist, the TCB at the head of the list is unlinked, the semaphore is set to the task ID number and the task given control.

To simplify the use of exchanges, several user callable routines were written, a brief description being given below. Detailed calling sequences are given in appendix F.

Event Calls

WTEVNT (Exchange_#)	Wait for a signal or event
STEVNT (Exchange_#)	Set an event or send a signal
CLEVNT (Exchange_#)	Initialise an event exchange to ensure that it is in the <i>locked</i> state as it should be after startup.
TWTEVN (Exchange_#,delay)	Wait for an event only as long as the time delay specifies. If a timeout occurs then return with error flag.

Resource Calls

GTRESC (Exchange_#)	Get or lock a resource exchange.
TGTRSC (Exchange_#,delay)	Attempt to lock a resource but limit the suspension time to the value specified.
RLRESC (Exchange_#)	Release a locked resource exchange.

The RLRESC call was implemented in such a manner that only the task that locked a resource could release it again. This prevents other tasks from accidentally unlocking a resource that they do not own which could result in two tasks both using a resource that both thought they had locked exclusively. This feature could not be applied to event exchanges as the normal case is for different tasks to lock and free the exchanges.

The timed exchange requests allow calling tasks to specify a maximum time that they are prepared to wait at the exchange, the time being the number of 10ms timebase 'ticks' to use. Should the task be released from the exchange before the timeout has expired, the return timeout flag will be cleared. However should the exchange still be locked when the timeout expires, the task will be re-scheduled as normal, but with the return timeout flag set to inform the calling task that the exchange access was not successful.

To allow tasks to use the timeout feature for fixed time delays and for time sequencing, a *dormant* exchange was included which is always busy and never ever made free. Thus any task wishing to suspend itself for a fixed time merely need execute a TWTEVN call on this exchange to ensure that the full timeout will occur. All dormant tasks in the system are also queued against this exchange.

Another special exchange was incorporated to hold tasks 'ready to run' after a timeout or a re-schedule call. This *active* exchange is maintained in the free state by the scheduler itself with the task at the head of the list being the next task to execute. Whenever any task pending against some exchange times out, its TCB is unlinked from whatever exchange it was on and relinked onto this active exchange. Similarly any dormant task being re-scheduled will be unlinked from the dormant exchange and relinked to the active exchange.

5.6 Memory Management Facilities

The basic function of any memory management system should be to supply user tasks with a set of calls allowing them to dynamically request or return blocks of memory. The memory manager should supply these demands from a pool of spare memory and since, on average, all tasks will not use their maximum requirements simultaneously, several tasks may end up sharing the same memory. In comparison with permanently allocating each task all the memory it may require, a memory management scheme can reduce memory requirements considerably.

A memory management system may be implemented to supply either fixed size blocks of memory, or to supply the exact amount specified in the user call.

The fixed block length scheme has the disadvantage that the block size must be as large as the largest possible request, resulting in memory wastage when only small amounts of memory are needed. This management philosophy, is however very simple to implement since memory can be pre-divided into the requisite blocks which then merely need be maintained in a linked list.

The variable length memory management scheme has the advantage that memory utilisation is more efficient, but it requires considerably more sophisticated management techniques. Several different strategies exist for allocating blocks of

memory, but irrespective of the system used, the pool of memory becomes fragmented into many small and random sized blocks, increasing the search time for suitable sized blocks. To compensate for this, the memory return system normally implements some form of compaction to try and coalesce adjacent empty blocks together. While the processing overhead of this scheme dictated against its inclusion, the deciding factor was that the task fetching data from the host would be unable to determine the required block size until the complete block had been transferred. Since the DMA controller must always be capable of accepting maximum length records, the fixed block maximum length scheme for memory management was necessary and hence was adopted.

After allocation of all system data structures and variable storage, the remaining memory was divided into nine blocks of 272 bytes. The first five bytes of each block were used to hold two linkage pointers and a memory block status byte, with the remaining 267 bytes being left for the application task to define. One of the links was used to link the blocks onto the task message queues (described next), while the other was used to permanently link all the blocks together. This chain is used by the *abort processor* to find memory allocated to an aborted task in order that it may be returned to the pool queue. To assist in establishing ownership of a block, the status byte is set to the queue number for a block in a queue, or the task ID number for a block owned by a task. The top bit of the status word is used to differentiate between the two conditions.

In the multiplexer system, the memory blocks are only used to contain request messages which are passed from host to destination task and subsequently back to the host again for acknowledgement. Each task was thus allocated an input queue on which all messages for the task could be placed.

The calls supplied for moving buffers (blocks of memory) around are:

PUTBUF (Queue_#,buffer)	Places the given buffer onto the specified message queue
GETBUF (Queue_#,buffer)	Fetches a buffer from a queue, and suspends the caller if no buffer is available.
TRYBUF (Queue_#,buffer,error)	This is a no suspend version of GETBUF which will return immediately with an error if no buffers are available.

Since a caller to GETBUF must be suspended if the queue is empty, each of the message queues has a resource exchange associated with it. The GETBUF, PUTBUF and TRYBUF routines take care of the resource lock calls so that this does not involve the user.

5.7 Task Control Facilities

In order to allow tasks to schedule one another and also to terminate other tasks, three calls were provided:

STACTV (TCB_#,Time_delay)	Sets a dormant task active if Time Delay is zero or places the TCB into the time list if Time Delay is non-zero. [The time delay option allows tasks to time schedule other tasks to run at some later time. The delay can be a maximum of 654 seconds using a 16 bit word with ten millisecond resolution.]
STDORM (TCB_#)	Sets the given task dormant by placing it into the dormant queue. All memory and exchanges owned by the task are cleared.
STOP	Places calling task into dormant queue.

The abort processor (STDORM) attempts to clean up the system when a task is aborted by releasing all resource exchanges owned by the aborted task, and returning all its memory buffers to the pool queue. However, since the MTX executive was not written to supply any standard I/O handlers, it cannot clean up I/O and interrupts left by the aborted task. To get around this, each and every task must contain a user written abort subroutine which will properly close down all interrupt and I/O activity should the task be aborted. The abort processor (STDORM) will then call this subroutine when the task is aborted.

Jump Table

Instead of a task's start address being stored in its TCB, a jump table was set up with six bytes being allocated for each task; the first three contain a jump to the task start, while the second three contain a jump to the abort subroutine. This table being ROM based is set up by the assembler at compile time. The reason for this approach was the simple code needed to use the jump table approach compared to that which would have been required should the start address have been stored in the TCB itself.

5.8 MTX – A Summary

Interrupt Handling

Since this operating system was designed expressly for the RMUX multiplexer in which the anticipated tasks were all essentially only I/O handlers, it was not necessary to add much in the way of centralized I/O handling facilities to the

system. Each task could handle its own I/O with less overhead than if the system were to supply the I/O handlers. The only function supplied by the system is a pair of interrupt handling routines which use an interrupt vector table to determine the interrupt handler location. The first routine intercepts all interrupts, saves the registers, determines the handler address and branches to the handler. When the handler has completed, a simple RETURN instruction will return control to the second routine which restores the registers and returns to the interrupted task. This philosophy eliminates duplication of the register save and restore code and eases the return from interrupt procedure without imposing any overhead on the I/O process.

Coding the System

Once all data structures had been decided on, the routines were coded into PASCAL, a regimen that tends to force well structured code. Originally the idea was to actually compile and run the PASCAL code on the HP1000 first in order to do some high level debugging before moving to the assembly code. PASCAL performs a very rigorous data type checking and some quite legitimate operations in assembly code required far more complex data structures in PASCAL than were warranted. An example of this was in the linked list handling procedures. Both memory buffers and TCB's are linked onto various lists, and both have the same link format. However, from PASCAL's point of view, they are two different record types and so cannot be treated by the same link procedures without special case handling. Since the major benefit of using PASCAL was to provide the correct structuring and not for high level debugging, the code was left in a relaxed specification pseudo-pascal and then manually compiled into 8085 code.

This exercise in coding definitely proved effective, as the code was all written, compiled and tested in a week, the debugging phase taking only one afternoon.

The system code including the pseudo-pascal code, which was left in as comments, took 1170 lines of code and occupied just over 1K bytes of ROM. A more complete description of this system code can be found in Appendix F, while for full details, the source code printouts (available from the Department of Electronic Engineering of the University of Natal) can be used.

5.9 Time List Handling

The time base interrupt interval was chosen as 10ms using the TRAP non maskable interrupt input of the 8085. TRAP was chosen because it could be connected directly to the output of a 100Hz timer/counter without any intervening logic necessary to clear the interrupt source. This is due to the TRAP input only responding to a rising edge signal.

However, since TRAP is the highest priority interrupt in the system, and cannot be disabled it was essential to make it's handler as short as possible. The time list feature is really rather unimportant in this system and any missed time base ticks would have very little effect on the system, whereas missing an input character due to an overrun error would be far more serious. When one considers that all four ports could be receiving in block mode at 9600 baud, resulting in there only being 250 microseconds available to process each interrupt, the time required to chase a time list and count down all time base clocks cannot be afforded.

Thus, instead of making the timebase interrupt handler process the time list, it merely sets an event to signal a task to run. This task (TOUT) uses one of the last exchanges in the exchange list and so has a very low priority; the scheduler will always release this task last. The time list chasing and decrementing of all the time base clocks are all performed by TOUT, which since it is a background task cannot interfere with the interrupt processes. Should the system become very busy such that TOUT is never scheduled during a 10ms time base interval, then the interrupt handler will merely set the signal event again which will result in a missed interval, an insignificant occurrence in this system.

A debug feature was added in which TOUT sequentially reads through the entire RAM at the rate of 4 bytes per 10ms. This causes the entire RAM to be read every 10 seconds, enabling a logic analyser to see what RAM contains. This was especially useful during early task debugging as the contents of tables, schedule lists and memory buffers could be examined. Since this debug read used only 0,34 percent of the available CPU time, it was left in the final version of the code, so that memory can be examined at any time.

5.10 Application Task Structure

The MTX system was expressly designed to control I/O intensive tasks that would supervise their own I/O. To fit into the system, all tasks were required to have the same structure resulting in every task consisting of three essential blocks of code: a main task, an interrupt handler and a termination routine.

The main task is the central co-ordinating body of code which performs all the interaction with the system. In every case, it's function is to fetch messages (blank or meaningful) from some queue, process the header to determine the requisite action, initiate the I/O transaction and, then wait against an event exchange for a signal from the interrupt handler. Once released from the exchange, the results of the interrupt transaction are processed, the message is placed on some output queue and the main task then loops back to fetch another input message. The main task's duty is therefore overall coordination of its I/O, message header processing and system interaction.

The interrupt handler is always entered due to an interrupt from the peripheral device it controls, the first interrupt of any message usually being initiated by the main task's action. The interrupt handler should then perform any processing that the data may require and start the next peripheral transaction. It should be a completely self-contained block of code and may not interact with the system at all except at the end of a complete transaction, when it may set the signal event to release the main task for final processing.

The termination routine is a small subroutine that the system abort processor will run whenever the main task is aborted. Its task is to clean up for the main task. Although the system will recover any memory owned by the task, and will release all resources, it cannot control I/O, so the termination routines principal job is to close down any associated peripherals correctly and disable their interrupts. In many cases the termination routine will be a null routine consisting only of a return.

5.11 Host Interaction Tasks

Host interaction is controlled by two independent application tasks, FROM_HOST to receive requests from the host and dispatch them to the various port handlers, and TO_HOST to fetch messages from the port handlers and pass them back to the host. Each task has its own DMA channel for data transfer, but they both have to share a single interrupt line to the host (FLG), and a single interrupt signal (STC) from the host. Thus they share a common interrupt handler which has to determine the cause of an interrupt before it can signal the correct task.

TO HOST

The TO_HOST task was an exceptionally simple task to produce, as it had only to fetch a message from its input queue, prime the DMA controller to send the message to the host, interrupt the host to inform it of the return message's presence, and then wait against an event until the interrupt handler signals that the host has accepted the message. It then loops to fetch another message as is illustrated in the overview given in figure 5.1.

Using the FLG signal to interrupt the host required a little care, since this same signal is used by both DMA channels to pace input and output. Thus, as explained in Chapter 4, the TO_HOST task was constrained from sending interrupts whenever the host is busy performing data transfer, the CNTL flip flop being cleared during this process. TO_HOST was written to always first check the CNTL flip flop state (connected to the RST5.5 input) and then only interrupt the host if the flip flop is clear. Should TO_HOST not be able to interrupt, it suspends itself for 20ms to allow the host to finish the data transfer and set the CNTL flip flop again. TO_HOST then loops and tries again to set the interrupt, this loop repeating indefinitely until the CNTL flip flop is found clear.


```

PROCEDURE To Host
BEGIN
  DO forever BEGIN
    GetBuffer (ToHostQueue, BufAddress);
    SetDmaChannel (BufAddress);
    DO BEGIN
      WHILE CntlFF = Clear DO Wait-20ms;
      SetFlagInterruptToHost;
      WaitEvent (ToHostSync#, OneSecond, Error);
    END UNTIL Error # Timeout;
    PutBuffer (BufAddress, PoolQueue);
  END;
END.

```

Figure 5.1 The TO HOST Task layout.

The average data transfer process takes about 300–400 microseconds with at most one transfer every 10ms. resulting in very little chance of TO_HOST having to loop more than once while waiting for a free period during which the host is not transferring data.

Once the FLG interrupt has been set, TO_HOST waits up to one second for the host to acknowledge the interrupt and read the message from the DMA controller. This is plenty of time for the host to respond, the maximum host response time being in the region of 50 to 80 ms even during periods of high I/O activity. There is, however, a small chance that the interrupt to the host could have been sent during the one very brief period upon entry to the host driver when it ignores interrupts, but has not had time to clear the CNTL flip flop. In this case, the interrupt will get lost, an inevitable problem when using the FLG flip flop for so many functions. Hence if after one second, TO_HOST has not been acknowledged, it will time out, discover that the interrupt has been lost and try again.

Once the host has eventually accepted the message and signalled this by sending an interrupt to the STC interrupt handler, TO_HOST will deposit the now empty message buffer onto the memory pool queue and loop to fetch another message from its input queue.

FROM HOST

This task, whose flow is outlined in figure 5.2, is far more complex than TO_HOST as it must ensure that there is always an empty buffer for the input DMA controller to use whenever the host starts sending. Furthermore, once it has a message, it must determine the message destination which can be any one of eight different queues for the read and write requests on the four ports.

Once a complete message has been sent by the host and an interrupt issued, the message must be removed from the DMA controller and a fresh empty buffer put in its place. All this must occur before the host has time to start outputting another message and since this interval may be as little as 200–300 microseconds, when many requests are queued in the host, there is insufficient time for FROM_HOST to be re-scheduled and perform the buffer replacement. The buffer replacement was therefore left to the interrupt handler which can perform the replacement within 80 microseconds of the interrupt. However, since the interrupt handler may not interact with the system it cannot request a fresh buffer from the pool. Hence, to ensure that there is always a spare buffer available for the interrupt handler to use, FROM_HOST always tries to keep four spare buffers in reserve.

```

PROCEDURE From Host
  BEGIN
    InitializedDma;
    StartUpAllTasks;
    FOR I:= 1 TO 5 DO MarkPot (I,"Busy");
    GetBuffer (PoolQueue, BufAddress);
    FillPot (1, BufAddress);
    SetDmaChannel (BufAddress);
    DO forever BEGIN
      FOR Pot := CurrentPot TO CurrentPot+4 DO BEGIN
        IF Pot^Status = Full THEN SendOff (Pot);
        IF Pot^Status = Empty THEN TryFillPot (Pot);
      END;
      WaitEvent (FromHostSync#);
    END;
  END.

```

Figure 5.2 The FROM HOST Task layout.

FROM_HOST manages this using a circular list of five flagged POTS which each hold a buffer address and may be flagged BUSY, FULL, EMPTY or FREE depending upon the state of the buffer pointed to by the pot. A BUSY pot contains a buffer currently set up in the DMA controller, a FREE pot contains an empty or available buffer, a FULL pot contains a buffer awaiting dispatch to a port task, and an EMPTY pot contains nothing, waiting to be filled. On start up, all pots are labelled EMPTY before the normal run loop is entered. This main run loop checks each pot for FULL or EMPTY status, full pots being sent off to the *message dispatch* routine, while EMPTY pots are given to a routine which tries to fill them. This TRY_FILL routine uses the non-suspending form of buffer access (TRYBUF) (Section 5.6) so that the task will not suspend should the buffer pool be empty. When this does happen, although the pot will remain empty, this will have little effect on the operation as there will always be some FREE pots in the list. This can be determined since due to the host limitation of only one request pending per port,

there can at most be only four valid request buffers in the system. Allowing one more at most for a system break request, (defined later), there must still be four buffers left in the pots to receive abort requests which are the only requests that may override an already pending request. Since at most there can only be four abort requests forthcoming, the nine buffers in the system will always be able to accommodate all possible host output.

FROM_HOST checks for FULL pots and dispatches these, thus creating EMPTY pots which are then filled when possible to create FREE pots. The FREE to BUSY to FULL cycle is not performed by FROM_HOST, but is left to the interrupt handler (STC_INT) described later.

The *message dispatcher* which sends full buffers off to their respective port handlers, has first to determine which port the message is for. This is done by using the lower two bits of the EQT entry in the host (see Chapter 6 and Appendix H) to index into a *port look up table* which contains the port address associated with each EQT entry. The lower two bits of the EQT entry address are sufficient for EQT identification since as EQT entries are 15 words long, and the four EQT entries for each interface are required to be adjacent in the EQT table, the lower 2 bits of each entry address will be unique within an interface.

The *port address table* is set up by the initial port configuration call (CN,30B call) (see Appendix K) which specifies the port number in the lower two bits of the configuration word. FROM_HOST intercepts these configuration requests, and should the *port look up table* entry for that EQT entry not be set up, the lower two bits of the configuration word are stored in the *port look up table* entry with a flag bit to indicate that the table entry has been initialised. The configuration call is then passed on to the port handlers for normal configuration processing.

One other host request that is processed by the *message dispatcher* routine is the abort call (CN,1u,0) (Appendix K) which must cause any pending requests on the relevant port to be aborted. Since this call cannot be processed by the port handler itself, FROM_HOST simply issues three STDORM calls to abort the three port handling tasks associated with the port and then reschedules them again. This method very simply removes the request and re-initialises all the port tasks. The abort request message is then returned to the host, where it acts as an acknowledgement for the original aborted request.

Two types of errors are detected in this section, one when the double SYN synchronization pair of characters cannot be found, and the other when a non-configuration request arrives for an unconfigured port. In the former case, the message cannot be acknowledged or returned since all addressing has been lost, so the buffer is merely returned to the pool queue which leaves the calling user suspended indefinitely, or until the EQT entry times out should a timeout have been set. In the latter case, requests for an unconfigured port are returned directly to the

host with a zero transmission log which is equivalent to an empty record.

STC_INT

STC_INT, the common interrupt handler for both FROM_HOST and TO_HOST is activated by an STC instruction from the host, causing an interrupt on the RST 7.5 interrupt of the 8085. This interrupt is only sent by the host when it has completed a data transfer in one direction or the other, so STC_INT has to determine which direction the transfer occurred in so as to invoke the correct task to process the request. The two DMA channel count registers are examined to see which channel count has decreased by at least 12 bytes, this being the length of a message header. This allows the channel upon which a transfer has occurred to be determined.

For return requests to the host, the DMA count register is reset to a correct buffer length so that it will not cause an incorrect decision on the next interrupt, the DMA channel is disabled, and TO_HOST is signalled to complete processing.

Requests from the host require somewhat more processing, since the pot associated with the recently filled buffer must be marked FULL, and the buffer from the next free pot must be loaded into the DMA controller ready for the next request from the host. Once this is done, the DMA count will of course have been reset during the reload and so will not cause errors on subsequent interrupts. Thus all that is left is to signal FROM_HOST to process the request and re-load any empty pots.

```

PROCEDURE STC_INT
BEGIN
    SaveRegisters;
    IF ToHostDmsCount <= BuffLen - 12 THEN BEGIN
        ToHostDmaCount := BuffLen;
        DisableToHostDma;
        SetEvent (ToHostSync#) END
    ELSEIF FromHostDmaCount <= BuffLen - 12 THEN BEGIN
        MarkPot (CurrentPot, "Full");
        CurrentPot := NextPot;
        SetDmaChannel (CurrentPot);
        MarkPot (CurrentPot, "Busy");
        SetEvent (FromHostSync#);
    END;
    RestoreRegisters
END.
```

Figure 5.3 STC_INT – The Host Interrupt Handler.

This interrupt handler was required to be very fast and finally occupied just over 100 bytes with a maximum processing time of about 80 microseconds. Figure 5.3 illustrates the overall approach of STC_INT the host interrupt handler.

5.12 Port Handler Tasks – An Overall View

The host processor has three types of request that it can send to the interface, namely read, write and control calls, while the interface can be in three distinct, active states, viz.

- (a) Outputting data due to a write request
- (b) Inputting data due to a read request
- (c) Inputting data with no read request pending.

The initial problem was how to break up the port handler into separate tasks to cope with these three request types and three active states, the final choice being:

- (a) TX_TASK to handle write and control requests
- (b) RX_TASK to handle read requests
- (c) BREAK to handle unsolicited input for system 'break' requests

Each of these tasks was written as a conventional three module task as described in Section 5.10, but due to the need for each task to access both the transmitter and the receiver of a port, the interrupt handlers are slightly interrelated. While this was inevitable, careful structuring of the interrupt handlers has reduced this relationship to a fairly simple clear cut one.

The four ports required four sets of similar tasks, the only differences being the port addresses and the data associated with each port. This was necessary because the 8085 instruction set does not allow for register indirect addressing of I/O devices. Since ample ROM had been included, and processing time was considered important, it was decided that no parts of any handler would be made common to all port handlers and that no special code would be included to save memory at the expense of speed. The resultant code for each port used just over 1K byte of ROM and for simplicity of tracing, each port was given 500 Hex bytes (1280) to ensure that all port handlers started on even boundaries and that a listing of one port's software was usable for all ports during debugging. The effort in maintaining four sets of port handlers with identical code except for RAM and port addresses, was eased by the multifile, local label and resettable variable features of the assembler (see Appendix K). This allowed two files to be kept, one (&RXDRV) a driver control file to set up port dependent variables and then merge in the other, (&RXTX1) the port handling software, once for each port. A copy of &RXDRV appears as Fig.5.4 to illustrate the approach used.

```

;-----&RXDRV. The port driver load file
;
;   this file holds the setup values needed to make
;   4 copies of the port control drivers.
;
^PORT#:          SET 1                ; Define the port handler #
^PORT_DATA:      SET ^PORT1DATA       ; Set the Data register address
^COUNTER:        SET ^COUNTER_1       ; and the Baud Rate generator
^COUNT_MODE:    SET ^CNTR1_MODE      ; Global BRG counter setup
^C_MODE_WRD:     SET ^C1_MODEWRD      ; This channel BRG setup
^TX_BUF_Q#:      SET ^MEM_Q#_TX1      ; Buffer Queue numbers
^RX_BUF_Q#:      SET ^MEM_Q#_RX1      ;
^PORT_RESRC:     SET ^PORT1_RESC      ; Resource # to lock port in/out
^FLAG_1#:        SET ^TX1_EVENT       ; Both TX & RX use same sync event
^FLAG_2#:        SET ^PORT1_BRK#      ; Port 1 break task sync exchange #
^RX_INT_VEC:     SET ^RX1VEC          ; Interrupt vector address's
^TX_INT_VEC:     SET ^TX1VEC          ;
^TX_MASK:        SET 10H              ; This bit on 8259 stops tx intr
;
;           MERG &RXTX1                ; Finally merge in the driver
;
^PORT#:          SET 2                ;
^PORT_DATA:      SET ^PORT2DATA       ;
^COUNTER:        SET ^COUNTER_2       ;
^COUNT_MODE:    SET ^CNTR2_MODE      ;
^C_MODE_WRD:     SET ^C2_MODEWRD      ;
^TX_BUF_Q#:      SET ^MEM_Q#_TX2      ;
^RX_BUF_Q#:      SET ^MEM_Q#_RX2      ;
^PORT_RESRC:     SET ^PORT2_RESC      ;
^FLAG_1#:        SET ^TX2_EVENT       ;
^FLAG_2#:        SET ^PORT2_BRK#      ;
^RX_INT_VEC:     SET ^RX2VEC          ;
^TX_INT_VEC:     SET ^TX2VEC          ;
^TX_MASK:        SET 20H              ;
;
;           MERG &RXTX1
;
< similar code is used for the remaining two ports>

```

Figure 5.4 Partial Listing of &RXDRV.

One feature of the Hewlett Packard terminal interaction policy is that any input from the terminal that occurs when there is no input request pending implies that the user is requesting a System Request. In order to cater for this, the driver software in the host must schedule the programme PRMPT whenever it detects a System Request.

To catch these unsolicited inputs, the input interrupt is directed to the break task interrupt handler BREAK_CNTL whenever no input request is active. This handler plus it's associated main task (BREAK) will create the schedule request.

Control requests from the host very seldom require any peripheral interaction, the only one on this interface that does being the *space line* request. Since this is basically an output request, and since all requests from the host are mutually exclusive, it was logical to treat control requests in common with write requests rather than generating a separate task for control requests.

The remainder of this chapter describes the three tasks required for a single port, mainly covering the design philosophy and principal problems of each task. All the code paths and options are not covered here since most of these are tedious sets of special character checks with very little processing of any interest. Appendix G covers these tasks in greater detail, while for full detail, refer to the code itself which is extensively commented^[8].

5.13 TX TASK, the Port Transmit Handler

TX_TASK is set up as the father of all port tasks, so once scheduled, it schedules the other two port tasks before entering its normal execution loop in which it remains indefinitely. Refer to Figure 5.5 for an outline of this task.

The task spends most of its life at the start of the main execution loop waiting for a message buffer to arrive on its input queue. When a buffer arrives, and the memory manager releases the task from suspension, TX_TASK checks if the message is a control or a data buffer. If a data buffer, the port resource is locked and the handshake flag, as set by the last configuration call, is examined to determine handshake type. The Qume or HP style handshakes require that an ETX or ENQ character respectively be sent to determine terminal readiness, this character being sent directly by the main task. No data may be sent until the terminal responds with an ACK, so TX_TASK sets the receive interrupt vector to point to a small interrupt handler TX_ACK and waits for an event signal from this handler. When the ACK arrives, TX_ACK resets the interrupt vector to BREAK_CNTL (the 'break' processor) and signals TX_TASK to continue.

To perform the actual output, the transmit interrupt vector is set to the output interrupt handler (TX_CONT) and the transmitter interrupt enabled. TX_TASK then has only to wait for a signal from the interrupt handler notifying that all the output has been completed. It then updates the header status byte (see Appendix K) and posts the completed empty message buffer to the host as an acknowledgement.

```

PROCEDURE TX_TASK
BEGIN
  Schedule (RxTask);
  Schedule (Break);
  DO Forever BEGIN
    GetBuffer (TxQueue, BufAddress);
    IF Buffer^RequestType(BufAddress)=Write THEN BEGIN
      GetResource (Port);
      MarkPortStatus (Active);
      UnPackHeader (BufAddress);
      DO BEGIN
        IF HandShake = EnqAck THEN BEGIN
          SendEnq (Port);
          WaitEvent (PortEvent, FourMinutes, Error);
        END;
        EnableTxInterrupt (Port);
        WaitEvent (PortEvent);
        UNTIL Buffer^DataLeft (BufAddress) = 0;
        PackCommonIntoHeader (BufAddress);
        MarkPortStatus (Inactive) ~END
      ELSEIF Buffer^RequestType(BufAddress)=Control THEN
        SetControlVariables;
      END
    END;
  ;
  ;
  INTERRUPT PROCEDURE TX_CONT;
  BEGIN
    IF PortStatus = Active THEN BEGIN
      IF Buffer^DataLeft (BufAddress) = 0 THEN BEGIN
        IF Buffer^Mode (BufAddress) # Honest THEN
          Send (Cr + Lf);
          SetEvent (PortEvent);
          DisableTxInterrupt (Port) END
        ELSE
          Send (NextChar);
        IF Buffer^DataLeft (BufAddress) mod 80 = 0 OR
          Buffer^DataLeft (BufAddress) = 0 THEN BEGIN
          SetEvent (PortEvent);
          DisableTxInterrupt (Port)
        END
      END
    END
  END
END

```

Figure 5.5 An Overview of TX TASK, The Output Handler.

One slight deviation from this simple process occurs during an HP style handshake transfer which requires a new ENQ/ACK handshake sequence every 80 characters. This is simply implemented by having the interrupt handler perform a normal end of line return every 80 characters during an output record (which may be 256 characters long). TX_TASK always checks the remaining character count

(maintained in the message header) and until this count reaches zero, TX_TASK will loop and perform another ENQ/ACK handshake sequence, returning control to the interrupt handler once again after the handshake has been completed.

Control request processing consists of examining the control request type and transferring control to one of six routines. There are only six requests that are processed, as the abort request (CN,LU,0) is handled by FROM_HOST and the buffer flush request is only ever used by the host driver and is not transferred to the interface at all. With the exception of the space line request (CN,lu,11B), all control requests simply set some variables in the data common attached to the port and then return the buffer to the host as an acknowledgement.

There are two versions of the space line control call, a positive line space count of $+n$ meaning that n lines must be spaced, while any negative line space count implies a form feed. The former case is treated by changing the control call to a write call, setting the data count to the number of lines needed to space and, then filling the data buffer with the requisite number of LF (line feed) characters. Control is then returned to the normal transmit processor for it to treat it as a normal line of output. Similarly, in the form feed case, a single form feed character (*FF*) is placed into the data buffer and the data count set to 1.

The transmit termination subroutine TX_KILL is one of the few cases where a termination routine requires some code, in this case the transmitter interrupts are disabled and the transmitter active flag is cleared to indicate that all activity should be dead.

5.14 RX TASK, the Port Receive Handler

As with the transmitter, RX_TASK has to first get a message buffer from its input queue, a point at which it remains suspended for much of its time. Once a buffer has arrived, the header is examined and the control bits checked for ECHO, HONESTY and BINARY modes, the ECHO flag in common being set accordingly. The other flag in common of significance is the BLOCK_MODE flag which is set to indicate that a program enabled block read is in progress, indicated by bits 5 and 6 of the control word both being set. In this case, a block read must be triggered with a DC1 character, no echo will be given, irrespective of the echo flag, and the input will be terminated with a CR,LF pair.

The handshake mode is then examined, and if HP style, then a DC1 input trigger character is sent (see Appendix K). Finally, the receive interrupt vector is set to point to the receive interrupt handler, (RCV_CONT) before RX_TASK suspends itself against an exchange to await the signal from RCV_CONT at the end of the input record. The wait for event call used is a timed call, with the time delay value being set to that issued in the last set time delay configuration call.

Once a complete record has been input, the termination being dependent upon mode, (see interrupt handler description), the main task resets the receive interrupt pointer back to the `BREAK_CNTL` interrupt handler, updates the status byte in the message header and common, returns the message to the host and then loops to fetch the next request.

The port resource is locked by `RX_TASK` prior to any I/O being performed on the port, and released after the last transaction is complete. This resource is used to ensure mutual exclusion of the two port handlers `RX_TASK` and `TX_TASK`, a feature not normally required in an Hewlett Packard RTE4B environment as the operating system ensures this exclusion by only allowing one request on a port at a time. However, should two EQT entries be configured to use the same port via the `configure (cn, LU, 30b)` call, then the resource locking will eliminate any interaction between possible overlapping requests. This same feature is also of value on another version of this interface, where it is used with a different host system which allows both read and write requests to be present simultaneously.

RECEIVE INTERRUPT Handler

The receive interrupt handler `RCV_CONT` has the basic function of responding to an input interrupt, fetching the input character, checking it for special character processing and, then placing the character into the message buffer. The major part of this essentially very simple process is the special character processing, a full list of the processing that occurs being found in Appendix G.4. However, a few of the more interesting special characters and the different modes of operation are described below.

The *BINARY* bit when set specifies that no special character checking should occur, all characters should be added to the buffer, and the read should only be terminated once the requested number of characters have been input. Finally, should an odd number of bytes have been requested, then a zero padding byte must be appended to the buffer.

In contrast to this, *ASCII* mode (non-binary) requires that several characters assume special significance and need special processing. Also records will only be terminated upon receipt of a carriage return (CR) and will be padded with an ASCII 'SPACE' character. However, in the *ASCII* mode, an *HONESTY* bit can be set which then avoids all special character checking. These two bits, *HONESTY* and *BINARY*, are set in the calling EXEC control word (`CONWORD`) (see Appendix K).

Of the special characters that are processed, *DC2*, *CAN*, *BS*, *DEL* and *CR* are the only ones that deserve special mention here.

CAN, entered using a CNTL X key is used as a means of attracting PRMPT, the system break mode processor's attention at the end of an input line. Whenever RCV_CONT detects a CAN character, it passes control to BREAK_CNTL the break task interrupt handler. This feature, which is not a standard HP feature, is a very useful feature for getting PRMPT's attention when some errant program insists on accepting all input from a terminal and will not go away. PRMPT can now be scheduled and the errant program can then be terminated easily.

DC2 in HP style handshakes is specified as the response to a DC1 trigger indicating a LONG MODE BLOCK transfer [9]. This DC2 will be followed by a CR return character which must then be responded to with a second DC1 trigger. Figure 5.6 illustrates this handshake sequence which should only occur if the DC2 character is the first character in a record.

TERMINAL		RMUX MULTIPLEXER
Indicate a long mode	DC1	Send Normal HP Handshake
Block Transfer		trigger character
Pending	DC2,CR(LF)	Wait for CR & ignore it
		Turn off echo
	DC1	Indicate readiness for
		Block mode with a second
		trigger
Send data terminated		
with a return	DATA + CR(LF)	

Figure 5.6 Long Mode Block Transfer Sequence.

The return after the DC2 is troublesome, since it normally indicates a record end, hence a flag (BLOCK_FLAG) is set in common whenever the first character is a DC2. Then when a return (CR) is detected, BLOCK_FLAG is checked and if set, it is cleared and the DC1 trigger is sent. There is otherwise no difference in processing except that echo is turned off for block mode reads.

BS and DEL are used to edit a record before it is terminated. BS removes the last character from the buffer, unless the buffer is empty, in which case the backspace responds as for the delete. The response to a DEL depends upon the type of handshake set. For HP style handshake (ENQ/ACK and DC1 triggers), an HP terminal is assumed and a terminal specific clear line command is issued, the complete output sequence being CR to return to start of line, ESC K to clear the current line and finally a backslash (\) to act as a delete indicator and replacement for the prompt that usually exists at the start of most input lines. For non HP handshake mode, no terminal specific characteristics can be assumed, so the string \,CR,LF,\ is issued which places the cursor below the deleted line and again starts the line with the backslash prompt replacement.

To simplify the process of sending special character strings, a small subroutine (OUT_CHNG) was written which when called, outputs the character in the A register, pops the return address off the stack and stores this address into the transmit interrupt vector. This causes the next interrupt to pass control to the statement following the call. The concept of moving the interrupt vector continuously led to very readable and simple code and eliminated the need to maintain any state flags.

Upon receipt of a return (CR) the BLOCK_FLAG is checked first and if set, then the DC1 sent, otherwise a return and line feed are sent if the echo flag is set, or if the transfer was a block mode transfer. The special check for block transfers is due to the fact that block transfers must not echo but do require the CR, LF termination.

Echoing of characters on this interface is a software performed function the received character being transmitted after reception, whereas the standard HP interface cards include extra logic on the card to route the incoming signal out to the transmit data line. This method uses extra logic and causes any receive errors to pass unnoticed.

However, the echoed character is coincident with the received one whereas in the RMUX interface, there is a single character time shift between received and echoed characters. The receive interrupt handler does not use transmitter interrupts to signal echo completion, nor does it check for this, as transmit and receive baud rates are always the same meaning that the transmitter can always transmit a character in the time taken for the receiver to receive one.

5.15 BREAK TASK, the Unsolicited Interrupt Handler.

BREAK_TASK waits for a signal from it's interrupt handler, and when it gets one, a buffer is fetched from the pool, the header is filled with the correct EQT address (stored in data common), a zero conword to indicate a 'break' request, and the terminal status with the break bit set. The buffer is then dispatched to the host before looping to await the next signal.

BREAK_CNTL the break mode interrupt handler is entered whenever any unexpected input arrives, and its first function is to clear the interrupt (by reading the USART) and check the character. All characters are ignored except 'SPACE' and 'CAN' (CNTL X) which are the only two recognised break characters. If one of these characters is recognised then the terminal 'break enabled' status is checked. This status is set by the terminal enable calls (CN,lu,20B and CN,lu,21B) (see Appendix K), and if disabled, stops all system break requests. Should the break status be enabled, however, BREAK_CNTL checks to see if either the transmitter or the receiver have a valid buffer as indicated by the port resource being locked. If a valid buffer exists, then BREAK_CNTL merely sets the 'break' bit in the header

status word of this buffer, and terminates, otherwise it signals the main programme BREAK_TASK for it to compile a blank message and send it to the host.

Chapter 6

The Host Software Driver Routine – DVX05

6.1 Introduction to DVX05

Every individual type of interface connected to an HP 1000 computer running the Hewlett Packard RTE operating system requires a special module of operating system code called a driver. This module remains completely invisible to the normal user of the system, but must translate all user requests from the standard format defined for I/O requests within HP's RTE operating system, into the commands necessary to drive the actual peripheral. Drivers in RTE have fairly rigidly defined rules as to how they interact with the rest of the operating system, and how they must be written.

The standard operating system uses an equipment table to define all the peripheral devices in the system, with one entry (known as an EQT entry) for each interface card in the system. This one to one mapping between peripherals (the hardware) and the EQT entries (the software pseudo peripherals) results in only one user being able to use an EQT entry and, hence a peripheral at one instant. This of course is a requirement when an interface card controls a single peripheral as it eliminates the possibility of two users corrupting one another's I/O.

However, with four peripherals on a single card, the RMUX multiplexer required that the operating system be altered to accommodate multiple EQT entries being associated with a single interface. Since the actual operating system itself could not be altered, the driver *DVX05* was written to assume some of the system functions for itself in order to map in the correct user to answer each interrupt. This required a small routine to be placed outside the usual driver area in a position where it would never be mapped out, Table Area 1 being chosen for this.

This chapter describes the major features of *DVX05* and the mapping routine *\$DVM5*, with particular emphasis on their interaction with RTE. Appendix I gives further details on the more minor internal features and of the driver interaction with the card; appendix H gives a description of RTE's I/O system and all its I/O tables and appendix K describes the I/O calls and functions applicable to the interface.

6.2 The Normal Interrupt Response Procedure

Under the RTE system, any interrupt causes the system map to be enabled and the trap cell instruction to be executed. For normal interrupts, this instruction is a jump to the system routine *CIC, the Central Interrupt Controller*. CIC saves the machine state, determines the source of the interrupt, and uses this information to address an entry in the *Interrupt Table* which specifies how to handle the interrupt. The usual Interrupt Table entry contains a pointer to an EQT entry which in turn contains the address of the interrupt handler. CIC then saves this EQT address information into a special area in the machine base page reserved for the current EQT entry address, and then via the *Driver Mapping Table*, (indexed by EQT number) determines the correct driver map to load and enable.

Several possible situations exist, as the data for the request may be in the system map as in the case of buffered requests, it may be in the special user map reserved for memory resident programmes, or it may be in the user map of a disk resident programme. The two word driver mapping table entry specifies the case in question, and for disc resident programmes, the address of the user programme's base page. From this information, CIC reloads the user mapping registers to point to the correct user and the correct driver before finally passing control to the driver.

The driver is then totally unaware of which map it is in, as it merely thinks it is in a simple 32K machine with the data, driver and peripherals all available within its address space. Upon completion, the driver returns to CIC in the system map via a link in Table Area 1, the only area included in both the system and all the user maps.

When a user programme calls the system to perform some I/O, the system places the ID address of the calling programme into word one of the EQT of the desired peripheral, which locks the EQT to the caller for the duration of the request and stops other users from using the same peripheral.

The peripheral EQT lock implies that every peripheral that may be accessed without excluding some other one, must have its own EQT entry, an easily supplied feature within the RTE system structure. However, as each peripheral slot in the machine has only one interrupt table entry which can only contain one EQT entry address, it is impossible to have four independent, simultaneously usable peripherals with four EQT entries, all attached to a single slot in the I/O backplane, and still remain within the standard operating system. What was required therefore was the ability to define four EQT entries for a single card slot and then have some means of altering the interrupt table entry to point to the correct EQT entry whenever the interface causes an interrupt. The requirement

that this be accomplished without any alterations to standard system code was obvious, as the driver *DVX05* had to survive upgrades of the operating system without having some user patch to add into each upgrade.

6.3 Operating System Feature Changes – The Philosophy.

In order to identify the EQT that each return message was destined for, the EQT address was included in the message header as the first two bytes after the synchronising characters. This enables *DVX05* to determine the correct destination EQT for each interrupt, but however such information arrives too late for CIC to use, as once CIC has passed control to the driver, all EQT and user mapping has been completed.

With the EQT address determined, it is a simple matter for *DVX05* to alter all the EQT addresses in the base page and access the correct EQT since all EQT's are stored in Table Area 1 and hence are accessible from all maps. CIC does not maintain any record of which EQT has been stored in base page, other than the addresses in base page themselves, so by changing these addresses from within the driver, *DVX05* effectively fools CIC into thinking that it loaded the correct EQT into the base page. CIC uses these addresses after a completion return from the driver in order to work out which user programme to release from I/O suspension and reschedule.

For write requests and control requests, the return messages contain no return data to go into a calling programme buffer, the only return information being stored in the EQT. There is therefore no need to map in the correct user for these requests. The system always restores the user map to it's status prior to the interrupt and takes no notice of changes it made prior to entering the driver so it does not matter that on entry to the driver the EQT in base page and the user map correspond, while on exit from the driver, they do not. Thus, once the correct EQT has been loaded into the base page address area, write and control requests can be processed quite normally, without any concern over which map is enabled, or whether the correct user map is installed.

Read requests however contain data in addition to the message header information, and this data has to be transferred to the caller's data buffer in the calling routine. For *system requests* and *buffered read requests*, both of which emanate from the system map, the driver needs to access data buffers in System Available Memory (SAM) which means that the system map must be mapped into the driver mapping space. For unbuffered user map read requests, only the correct user map must be loaded into the mapping registers. Since a user programme may only

have one I/O request active at a time, having the correct user map enabled guarantees that the corresponding driver partition will be mapped in as well. This situation does not however exist in the case of the system map as many different I/O requests may be concurrently active from the system map.

RTE provides a standard routine (*\$XDMP*)^[10] which system map resident drivers can call to have a new user loaded into the user map. This routine requires the calling driver to save the contents of the 32 word mapping registers first and then call *\$XDMP*, passing it the ID address of the programme that must be mapped in. This call is expected to be used by drivers placed in the System Driver Area (SDA), a special driver area included in the system map^[11] which must do all its own mapping. Placing drivers into the SDA has the disadvantage that it uses up system address space, since all SDA drivers are stored consecutively and not overlaid as are normal drivers. For this reason it was decided that the multiplexer driver should not be forced into the SDA but should appear as a normal driver. This meant however that there would be times when the driver would need to change the user map from the one in which it had been invoked, to the correct one for the return message while having no copy of the driver present in the system map from which to call *\$XDMP*.

This led to the requirement for a stable base from which *\$XDMP* could be called. This area had to always be available from the system map, and preferably also be accessible to all user maps although this latter requirement could have been eliminated by performing cross map data transfers from the system map. Table Area 1 in RTE is a two page (2K words) region that fits these requirements perfectly since it is included in all maps of the system. However, Table Area 1 is a small area which contains all the system tables and should not be extended beyond 2 pages if it is not seriously to reduce the available programme space. Hence it was necessary to ensure that only the minimum amount of code would be included in Table Area 1 to perform the necessary map changing.

Since it is only for read requests that this special mapping code need be accessed, the code could be reduced to a simple and straightforward procedure. Its first act upon being called is to determine if the required data space is in the system map or the user map. If in the system map, the transfer can proceed immediately followed by a normal return to the standard driver in its initial map. If the data is in the user map, then the user map registers must be saved, and *\$XDMP* called to load in the correct programme whose ID address is obtained from the EQT. The user map must then be re-activated before the actual data transfer can occur. After this the system map must be re-enabled and the original user map restored before returning to the driver in this original user map.

These requirements were incorporated into the main driver routine *DVX05* and its associated Table Area 1 mapping routine *\$DVM5*.

6.4 Mapping System changes – *\$DVM5*

This section describes the actual implementation, in *\$DVM5* and *DVX05*, of the mapping alteration code.

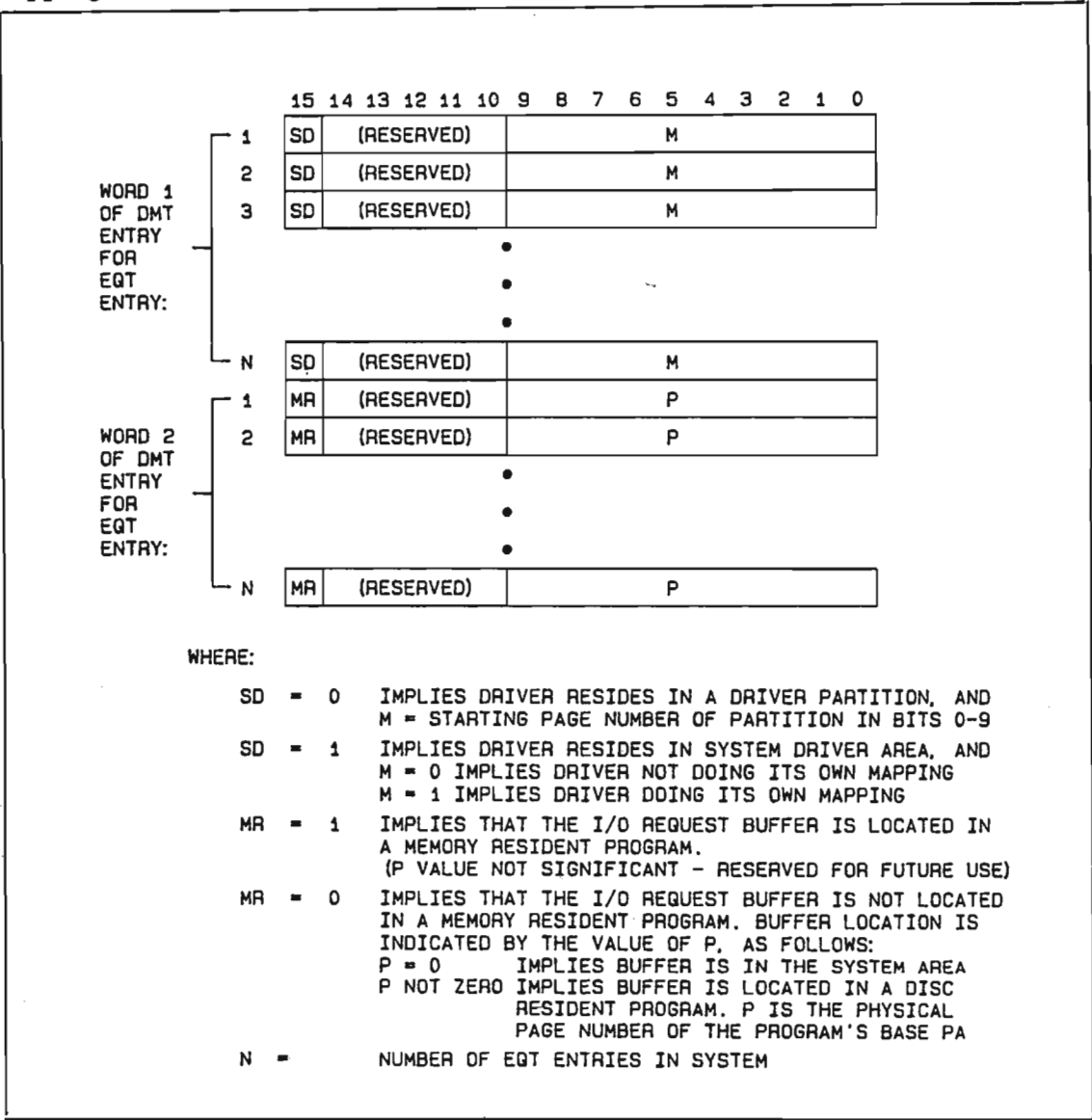


Figure 6.1. The Driver Mapping Table (DMT) format.

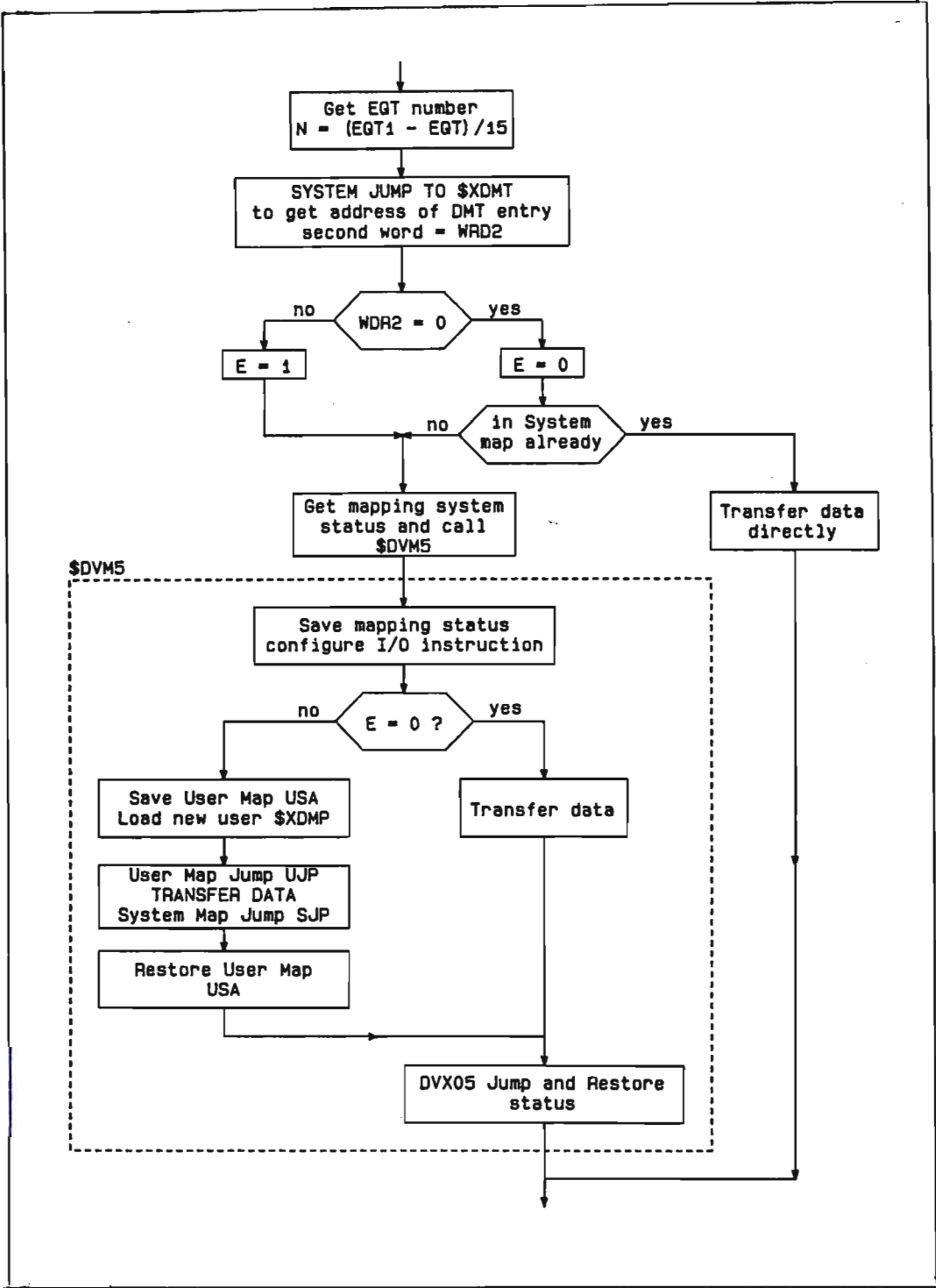


Figure 6.2 Mapping System Changes For READ Requests.

The first item needed after the correct EQT addresses have been loaded into the base page is the EQT entry number, which is found by subtracting the start address of the EQT table (found in variable *EQT1* in base page common) from the current EQT address and dividing by 15, the length of an EQT entry. This number

is needed to index into the driver mapping table (DMT) to find the user map status of the user currently locking the EQT. The address of this mapping table is held in an external system variable *\$DVMP* which is only accessible from the system map. Thus, as a part of *\$DVM5*, a small routine (*\$XDMT*) was written to swap maps and determine the address of the driver mapping table entry. Since the DMT table itself is located in Table Area 1 it can be accessed from within the normal driver *DVX05*. The second word of the DMT entry (the DMT format is shown in Figure 6.1) is tested for zero to determine if the system map must be enabled. In this case, the current map status is tested to see if the system map is already enabled. If not, then a map change is required, as is also the case should the DMT entry be non zero. The mapping system status is then saved, and the system map is enabled in a jump to *\$DVM5*.

At this stage, *\$DVM5* saves the mapping system status, configures it's I/O instructions to the correct I/O slot and forks to use the system map or to reload the user map. To reload the user map, the current user map is saved using the single 'USA' instruction, the ID address is obtained from the EQT entry and *\$XDMP* is called to perform the actual reload. The user map is then enabled and the data is read, byte by byte from the interface into the user buffer. Finally, the system map is re-enabled and a 'jump and restore status' instruction performed to return to *DVX05*. This process is illustrated in the flow chart in Figure 6.2, the emphasis on this mapping section being necessary due to its non standard nature.

6.5 The Layout Of The Main Driver – DVX05

The driver, as with all HP drivers, consists of two major modules: the initiation section to pass user requests to the interface, and the completion section (interrupt handler) to receive the returned user requests. The driver was coded to use subroutines in an effort to keep the code modular and easy to follow. The comparatively simple process of sending messages to the interface and receiving the replies was complicated by the following auxilliary functions which the driver had to perform to suit the RTE system.

- * Accept a buffer flush request causing all buffered requests to be ignored until the buffer queue has been emptied.
- * Process invalid or illegal interrupts.
- * Process power fail requests and re-configure the interface.
- * Process time-out commands from the system and issue abort requests to the interface.
- * Manage the special situation when a terminal is the system console.

The Initiation Section:

Figure 6.3 shows a block layout of the initiation section of DVX05.

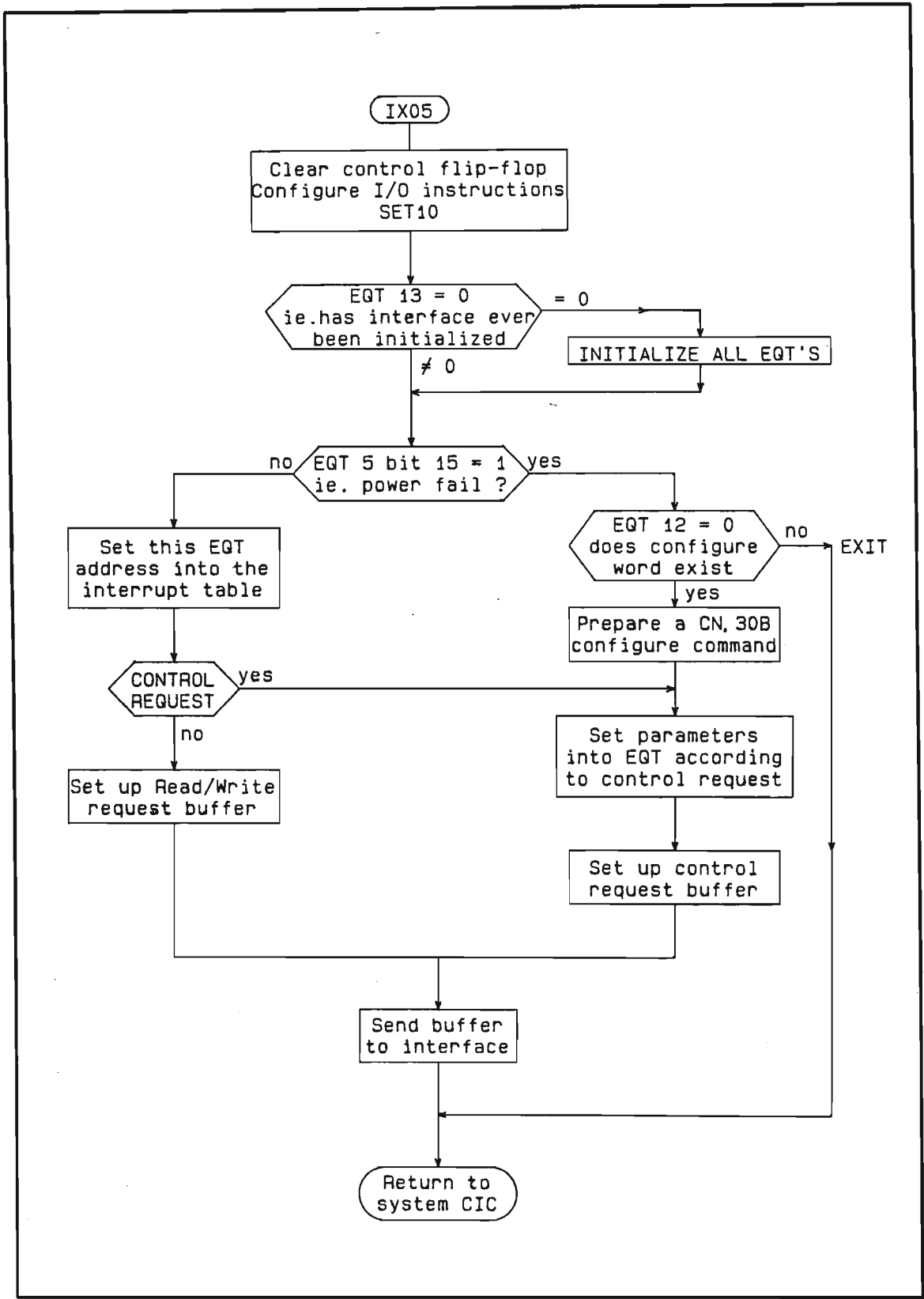


Figure 6.3 The Initiaton Section Of DVXO5.

Configuration of I/O Instructions.

The configuration of I/O instructions is a standard requirement for all RTE drivers since only one copy of each driver is used to drive all interfaces of the same type. Thus the driver must rebuild all its I/O instructions by including the correct interface slot address into them. The first I/O instruction executed clears the control flip flop, and the remaining configuration process then supplies the necessary twelve microsecond delay for the interface to notice the clear control flip flop (Section 5.11).

Cold Start Initialization.

On the initial entry after system start up, all four EQT's for the same I/O slot are initialized to hold the ID address of the programme PRMPT and the address of the interrupt table entry for the slot. All four EQT's are initialized at once, *DVX05* finding them by searching all EQT's for those with the same slot number (held in word 4) and same initiation section address (held in word 2). PRMPT's ID address (or -1 if not defined) is stored in EQT word 13 and acts as an indicator to show that the initialization has been performed.

Power Fail Handling.

When the power fails in an HP1000 system, a battery backup maintains the memory contents, but all peripheral power disappears. Thus on power up, the operating system resumes from where it left off, while the interface starts up from scratch and needs to be fully configured.

To accommodate this, the power fail recovery system re-issues all outstanding I/O requests setting bit 15 in each EQT word 5 to indicate that the power failed. Since the main memory contents were not lost during the power outage, the EQT will still be valid. Now whenever a configuration call is made, the control section of *DVX05* was made to store a copy of the configuration request parameter into EQT word 12. Thus when the system issues the power fail request, the driver changes the request into a configuration call using the configuration parameter previously saved in EQT word 12 and sends this to the interface, thus re-establishing the configuration that existed prior to the power failure. If EQT word 12 is found to be zero, then the power fail call is ignored, as will be all subsequent calls until a valid configuration request is issued.

Minimizing Map Changes.

Whenever an interrupt from an interface occurs, the driver is entered with one of the four EQT's that belong to the interface, but as described previously, this EQT may not be the correct one resulting in the re-mapping process described in Section 6.4. In an attempt to reduce the number of times that the entry EQT is wrong, the

initiator always places the EQT address of the current request into the interrupt table, with the result that every interrupt will enter with the EQT address of the channel that last received an initiation request. When all four channels are busy, this feature has little effect, and the number of erroneous EQT entries made is likely to be the same as if a single EQT was always used. However, as less channels are used, the chance of the entry EQT being correct increases, with the result that the mapping activity is reduced. When traffic patterns are examined, this feature can be seen to be quite effective as most terminal transactions occur in bursts, with relatively long idle periods between these bursts while the user reads the screen, decides what to do, etc.

Request Processing.

The user request is examined with read and write requests being treated together, while control requests are processed separately. For read and write requests, the EQT parameters are merely packed into the header (see Appendix I.4) and the complete message of header plus data is output to the interface, one word at a time. Control requests however are checked first with those requests that affect the EQT status being processed before being dispatched to the interface.

The interface offers programmable handshake features as described in Chapter 4 and can therefore appear to emulate either an HP12966 interface with DVR05 or an HP12531 interface with DVR00 depending upon the handshake programmed. The last two digits (in octal) of the driver name are stored in the EQT 5^[12] (see also Appendix H) as the driver type code and many programmes check this type code to determine the type of terminal they are dealing with. Thus, whenever a configuration call is made on any EQT, the handshake bits are checked, and if an HP style handshake is selected, then the driver type code is set to '05', otherwise it is set to '00'.

When outputting the message header to the interface, the *FLG* flip-flop is used in conventional HP manner to pace the data exchange process and act as the handshake line. For each word of the header sent, this flag is checked and if it has not set within 20 μ s of the output instruction, then an error condition is assumed, as the interface should always be able to respond within this time. The error is treated by setting an error code of 3 into the EQT word 5 status field, and taking the *device not ready* exit back to the system. The system then marks the EQT as *not ready* (or down) and reports the condition to the system console. Should the header of the message be accepted correctly by the interface, any remaining data is sent without *FLG* checking as the input DMA channel on the interface has been proved to be working and the data transfer can then proceed as fast as the code will allow at one word every seven microseconds.

The Completion Section.

Figure 6.4 shows the block layout of the completion section of *DVX05*, the entry point name being *CX05*. After the normal configuration of I/O instructions which has to be repeated every time the driver is entered, the timeout entry bit is tested, this being the only condition other than an interrupt (although it is due to an interrupt from the time base) which can invoke the completion section.

Timeout Processing.

A timeout means that the interface request should be terminated, and so an abort message (CN,l,u,00) is sent to the interface just as any normal control request. However, should the interface have timed out due to an interface fault rather than a slow terminal user response, then the system must be informed, and so to check for a fault, a short timeout of 1,2 seconds is set into the timeout clock (EQT word 15) and bit 6 is set in the EQT status field to indicate that the interface is already processing a timeout request. The abort request should always be completed in much less than 1,2 seconds and hence, if no acknowledge is forthcoming in this time, then the interface must be faulty. Thus, for every timeout entry, the *second timeout* bit (bit 6 in EQT5) is checked and if set, an interface fault is reported by taking a timeout exit to the system. This will cause the device to be marked *down* and a console report to be generated. If, however, the abort request is processed correctly, then the original aborted request will be completed with a zero transmission log and the *second timeout* bit will be cleared.

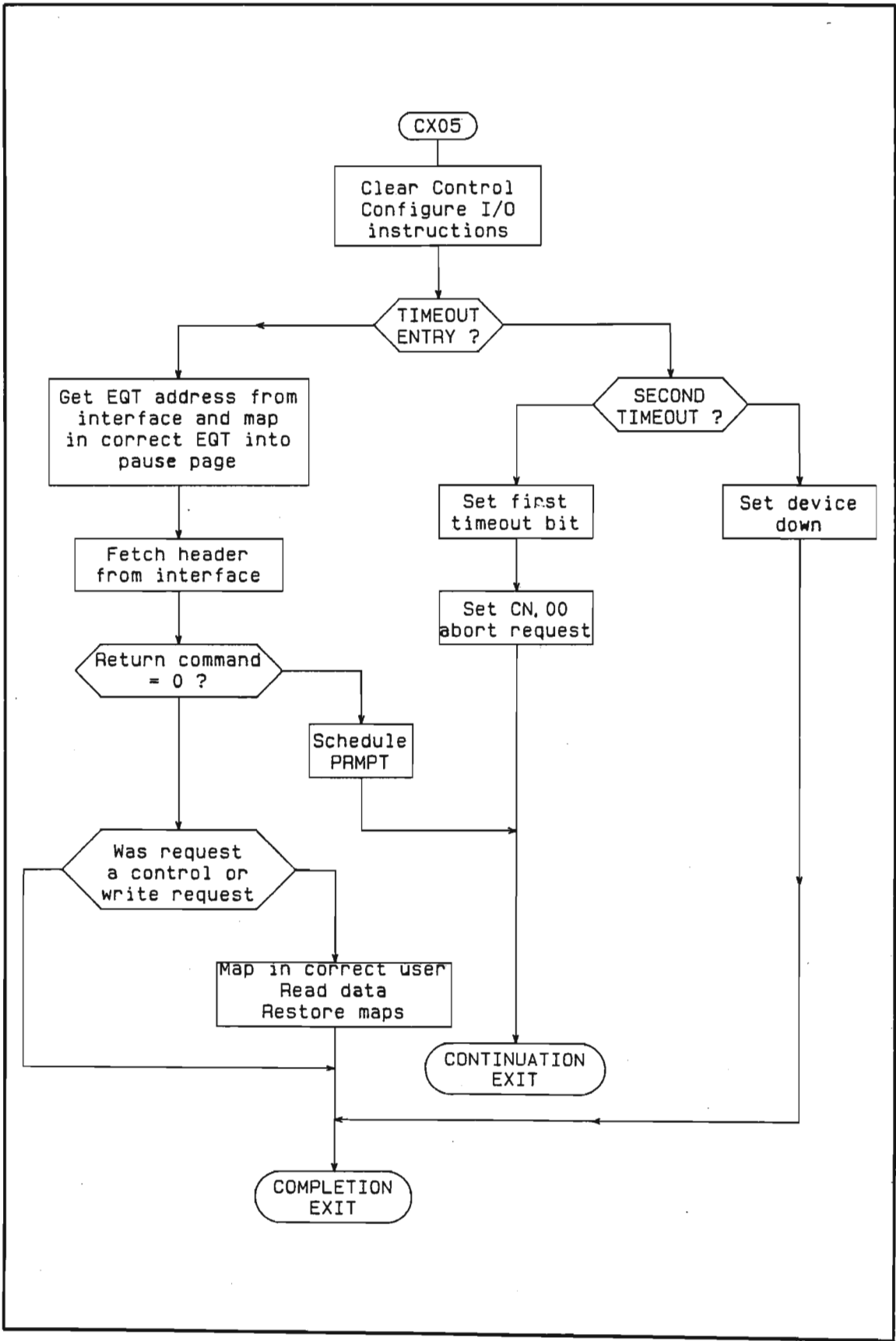


Figure 6.4 The Completion Section Of DVXO5.

Normal Interrupt Processing.

If the entry was due to a normal interrupt then the first action is to read the EQT address from the interface and load the correct EQT into base page. This done, the remainder of the header is read from the interface and unpacked into the EQT. The returned *CONWORD* ^[13] is then checked to determine subsequent action. For write and control requests (*CONWORD* = 2 or 3) no further action is needed, so a completion return to the system is taken.

In the case of PRMPT schedule requests, (*CONWORD* = 0), the program PRMPT is scheduled by a call to the external system routine *\$LIST* ^[14] and a continuation return taken.

For read request returns, the correct user memory must be mapped in before the data can be read as was explained in detail in Section 6.4. After the map has been changed, the data is read from the interface a byte at a time, and then finally the original map is restored and the completion exit is taken to return to the system.

As with the output process, the *FLG* flip flop is monitored during the header input process, and any delay in excess of 20 μ s is treated as a fault and reported accordingly by taking the error exit and setting an error code of 3 into the status word. Other errors are checked for, these being:- (see Appendix I.7)

- * Error 2 when the user map specified cannot be mapped in.
- * Error 4 when the returned EQT from the interface is not a correct EQT for the slot.
- * Error 5 when the synchronization characters cannot be found within the first 5 characters from the card.

Each of these errors will cause the interface to be marked *down* and an I/O NOT READY message to be displayed on the system console.

6.6 Timing Considerations

There were several points at which timing had to be carefully checked, as the shortage of handshake and interrupt lines in the HP1000 meant that not all transactions could be properly paced.

Cold Start Delay.

During the boot up procedure the system performs a general preset instruction (*CLC 0*) which resets the interface completely, forcing it through a power-up restart procedure. This procedure, which involves the checking of all RAM, the

setting up of all tables and the scheduling of all 15 tasks on the interface, takes approximately 20 ms . During this time, RTE will be busy going through its initialization phase, which may, depending upon the user defined start up procedure, attempt to access the interface, with the result that the interface input task may not be ready, causing an error 3. This problem is more pronounced after a power failure recovery as the interface requests will be re-established very quickly after the power restoration, the process not being user definable. To overcome this possible problem, a comparatively long delay of 120 ms was added into the configuration control call handler, since this is always the first call that occurs after a power fail or during the boot up procedure. This delay does not cause a problem in normal use since this call is seldom used.

Driver Exit Delay.

During the output process, the desired operation after outputting the last character to the interface is to issue a (*STC,C*) set control and clear flag instruction and exit the driver. However, the delay between the last output instruction and the clear flag command would be less than the six microseconds worst case response time of the DMA input channel on the interface, with the result that the set flag action of the *DMA* input channel could occur after the clear flag action of the *STC,C* instruction. Since no valid instructions were needed which could be included between the last output (*OTA*) and the *STC*, a dummy pair of rotate instructions were included to add the $6\mu\text{s}$ delay.

After the final *STC,C* instruction at the end of an output sequence, it can take up to $80\mu\text{s}$ for the interface micro-processor to respond to it's interrupt and replace the memory buffer in the DMA controller. During this interval then, the host must not attempt to output a new record to the interface, otherwise an error 3 would result and the interface would be marked not ready. Checking the time taken between entry into the initiation section and the start of any output to the card, shows a delay of greater than $104\mu\text{s}$ which ensures that the interface will always be able to replace the DMA buffer without the need for any host padding delays before output can begin.

CLC to Driver-Output Delay.

The last delay that had to be ensured was the delay between issuing the clear control *CLC* instruction upon entry to the driver and the beginning of any output. This delay, as described in section 5.11, is necessary to stop any interference between host data output and interrupt signals from the interface. Placing the *CLC* instruction at the start of the I/O configuration routine, which is much longer than the required twelve microseconds, was all that was necessary to ensure this delay.

With the exception of the 6 μ s delay at the end of each output, no regular delays were needed in the code, resulting in good speed efficiency from the driver. The long delay in the configuration section of the driver is so seldom used that its effect upon system throughput is insignificant.

Chapter 7

Performance Measurements and Results

7.1 Introduction

The multiplexer was designed with two aims in mind:

- * To increase the number of terminals that could be connected into the computer main frame.
- * To reduce the loading on the host during I/O.

The connectivity aim was achieved resulting in a four times increase in the number of terminals that could be plugged into the available slots in the machine mainframe. All that was required to further prove this objective was to check that the multiplexer could handle all four ports at once, a factor which was easily verified.

In order to justify that host loading was reduced, some measurements had to be made to compare the performance of the RMUX multiplexer with that of the two standard HP interface cards which it was designed to emulate. This chapter describes two sets of measurements made to evaluate the multiplexer performance:

- * Microprocessor timing measurements made using a logic analyser.
- * Host CPU programme measurements.

The results of these measurements show that the interface is capable of supporting a total character throughput of 6000–7000 characters per second, uses 20% of the host's time compared to the HP12966 and only 5.2% compared to the character orientated HP12531.

7.2 Interface Timing Measurements

The principal instrument used for debugging software and hardware was the Tektronix 7D02 logic analyser with an 8085 personality module. This instrument plugs into the microprocessor socket in place of the 8085, and allows one to monitor all bus and signal line activity. The instrument contains an extremely powerful and sophisticated triggering facility, can store and display data which matches quite complex qualifying conditions and can perform timing measurements between any two trigger conditions.

The timers (there are two available in the analyser) can be set to measure time or events, and in time mode can be stopped, started and reset at will. They thus presented an excellent tool for making measurements of interface performance.

Although the 8085 microprocessor instruction times are known, and therefore subroutine execution times should be calculable by counting machine cycles, this cannot be done in this case due to there being three possible levels of nested interrupts and a DMA channel all interfering with normal execution. While it is still feasible to calculate worst case times for these effects, the level of uncertainty makes actual timing measurements mandatory.

Times that were checked were:

- * Time to process a single normal interrupt on input and output.
- * Time from arrival of a request at the interface until ready for the first character.
- * Time to dispose of a completed message.
- * Time to process the time list.
- * Time to replace the DMA input buffer.

The results of these measurements are shown in Figure 7.1, the maximum or worst case times occuring during periods of peak activity with all four channels running at 9600 baud.

Measurement	* Best Time	* Worst Time
Interrupt handler time for a single input character.	235	265
Interrupt handler time for a single output character.	145	175
Message initialisation time for a read request.	193	223
Message initialisation time for a write request.	180	180
Message disposal time from last char to host interrupt	337	—
Time to process the time list. (max 4 entries)	251	490
DMA Input buffer replacement time.	136	176
Timer interrupt routine.	30	30

* all times in micro-seconds.

Table 7.1. Interface Timing Results.

From the message initialization, character handling and message disposal times, the following conclusions can be made:

- * The message handling overhead in total amounts to about $530\mu\text{s}$ giving a capability of 1800 null messages per second on both input and output requests.
- * The character handling times indicate that the interface could handle a total burst rate of 4250 characters per second on input and 6900 characters per second on output.
- * Assuming message lengths of 10, 50 and 250 characters, the processing time per output message amounts to $2300\mu\text{s}$, $9300\mu\text{s}$ and $44300\mu\text{s}$ respectively, giving average effective character rates of 4300, 5400 and 5650 characters per second.
- * The time-list processing time which occurs every 10 ms indicates that the time list processor occupies less than 5,2% of the system time.
- * The DMA buffer replacement time was measured from the time that the host issued its *STC* interrupt command until the DMA controller was finally reloaded and re-enabled with a new empty buffer. The worst case measured time of $176\mu\text{s}$ ties in well with the calculated worst case figure of $167\mu\text{s}$.

These figures indicated that the interface performance would be more than adequate to support four terminals running at 9600 baud although not enough to run all at 19,2Kbaud should all attempt to transfer at maximum speed simultaneously.

7.3 Host Throughput Measurements

To measure the effect on host performance, some means of exercising the terminals was required while the CPU utilization could be monitored. The terminal exercise routine would have to sustain high data rates for long periods of time to allow easy measurement, and since this is difficult to achieve in input mode, all terminal activity was measured in output mode only.

Terminal testing was done by a programme *PRINx* of which multiple copies were made, one for each terminal being used in the test (e.g. *PRIN1*, *PRIN2*, etc.). *PRINx* was written to output 160 records of 252 characters each to its terminal and time the total transaction to find the average character rate. This transfer takes in the region of 45 seconds when using 9600 baud communications, a long enough time to average out any timing irregularities.

In order to measure CPU activity, a simple programme was written (*LOADT*) which merely performed a simple CPU intensive calculation several thousand times. *LOADT* was also made to time itself and upon termination, print out the total time taken. The number of times the programme looped was adjusted to ensure that *LOADT* would complete in less than half the time taken for *PRINx*, the final programme taking 17,70 seconds to complete on an otherwise inactive machine. Using this as a base figure, any reduction in available CPU time would display itself as a lengthening of the time taken for *LOADT* to complete, allowing the percentage time available to *LOADT* to be calculated.

To test a combination of terminals, each terminal was allocated its own copy of *PRINx* with *PRIN1* having a priority of 61, *PRIN2* a priority of 62, etc. This staggering of priorities was done to check whether the priority was affecting the programmes; if the priority was to affect a programme's performance, then *PRIN1* should show a higher throughput than *PRIN2* etc. for identical terminals and interfaces. In practice, the data rate of four programmes driving four HP2621A terminals proved to be repeatable to 1 character per second, and quite insensitive to programme priority. Furthermore, the priority made no difference whether the four terminals were all on a single multiplexer, or whether they were on two multiplexers, each feeding two terminals.

The desired copies of *PRINx* were then all scheduled simultaneously with one copy of *LOADT*, and at the end all the times and data rates recorded. Figure 7.2 shows the values recorded for various combinations of terminals and interfaces. Two derived results from these figures are also included, these being the percentage CPU utilization for a standard character rate of 1000 characters per second, and the percentage efficiency of data transfer when running at 9600 baud; a nominal data rate of 960 characters per second.

The figures for character transfer rate show that the RMUX interface is significantly better than standard interfaces. In the non-handshake mode, when compared to DVR00, the RMUX manages 99 percent of the nominal data rate compared to 73 percent for the HP12531 with DVR00. This is due to the RMUX using a double buffered USART which ensures that each character starts directly after the previous character's stop bit; whereas the HP12531 causes an interrupt after each character which DVR00 has to process. In fact, the character transfer rate of 701 characters per second implies an average character time of 1,43 ms which, with a nominal character time of 1,04 ms, implies that the average interrupt overhead in DVR00 is in the region of 380 microseconds; quite a tribute to the rather weak I/O system of the HP machines.

No of Terms	Terminal Type	Interface Type	Driver Type	LOADT Time Note(1)	Char Rate Note(2)	% Char. Effic'y Note(3)	% CPU Time Note(4)	% CPU /1000cps Note(5)
1	HP2621	HP12531	DVR00	23,57	701	73,0	24,9	35,5
2	HP2621	HP12531	DVR00	34,27	700	73,0	48,4	34,5
1	HP2621	HP12966	DVR05	19,17	819*		7,7	9,4
2	HP2621	HP12966	DVR05	20,85	819*		15,1	9,2
1	HP2621	RMUX	DVX05	18,03	950	98,9	1,84	1,93
2	HP2621	RMUX	DVX05	18,37	945	98,4	3,63	1,92
3	HP2621	RMUX	DVX05	18,71	932	97,1	5,39	1,91
4	HP2621	RMUX	DVX05	19,04	917	95,5	7,00	1,91
4	HP2621	RMUX	DVX05		860*			
8	HP2621	RMUX	DVX05	20,54	917	95,5	13,83	1,89
0	LOADT Measurement only			17,70				

* Indicates ENQ/ACK HP style Handshake enabled.

NOTES:

- 1 LOADT runs for 17,70 seconds on an otherwise empty machine. All data transfers were long enough to complete after LOADT had completed.
- 2 The transfer rate relates to output only transactions without any handshake enabled.
- 3 An indication of transfer efficiency for a nominal rate of 960 characters per second.
Calculated from : [(CHAR RATE) * 100]/960
- 4 The percentage utilisation for 'n' ports running at the given data rate.
Calculated from : 100*[(LOADT TIME) - 17,70]/[LOADT TIME].
- 5 CPU utilisation normalised to a base of 1000 characters per second for easy comparison.
Calculated from : [(PERCENT CPU TIME)*1000]/[(NO OF TERMS)*(DATA RATE)].

Table 7.2. Host Efficiency Figures for Different Interfaces.

When comparing the character transfer rates with ENQ/ACK handshake enabled, the RMUX interface performs marginally better than the standard HP12966, the ratio being 860 characters per second to 820. Since the HP12966 also employs a double buffered UART, the character transfer rate should be the same for both interfaces, so the difference derives from the increased interrupt handling time of DVR05 which will service 10 interrupts from the interface during a 252 character record compared to the one interrupt from the RMUX.

The character transfer rates are, however, of small significance compared to the load placed on the CPU by the different interfaces. To make the comparison fair, the CPU usage figures were all normalised to indicate the load that the CPU would suffer if each terminal were to receive output at a rate of 1000 characters per second. In this comparison, the differences are dramatic, with the RMUX being some eighteen times more frugal of host usage than the simple HP12531/DVR00 combination. Even when compared to the more sophisticated buffered HP12966/DVR05 combination, the ratio is five to one, the RMUX using less than 2 percent of the CPU's time for 1000 characters per second. This is where the fast

and simple transfer procedure of the host—RMUX interface comes into its own, because as the actual character rates show, the hardware limitations are very similar.

One of the test runs was performed with four HP12531 interfaces driven by DVR00 all transmitting data at 9600 bits/sec. When this was run, the actual data rates varied from 580 characters/second to 650 characters/second, and LOADT never even started execution until the first terminal transfer had completed. Thus, four DVR00 terminals could completely clog the entire HP1000 CPU leaving it incapable of doing anything else.

Perhaps in summary, the relative performance improvement that was achieved by the RMUX can best be highlighted by considering the case of the 24 terminal HP1000 RTE system that was in use in the Department of Electronic Engineering at the time of writing. Had these terminals been connected to the host using standard HP12531 terminal interfaces all running at 9600 bits/sec., then if all terminals were in operation only 16% of the time, the host CPU would be totally occupied with terminal I/O.

Using the RMUX interfaces, however, the 24 terminals doing I/O 16 % of the time, would use only 8% of the CPU's capability, leaving it relatively free to do its main job of processing.

Chapter 8

Conclusion

This thesis has covered the design and implementation of three separate projects, all aimed at improving the Input/Output capabilities of an HP1000 series minicomputer. Two of the projects, although independent and complete in themselves, each contributed toward the final objective of expanding the number of terminals that could be connected to the HP 1000 while at the same time reducing the CPU loading on the HP 1000 host.

The initial project of producing a dumb terminal expander system using modern LSI and MSI logic was successful in increasing the number of terminal ports of the host while reducing circuit complexity from the standard HP terminal interface. However, since the expander was designed to emulate the standard product exactly, there was no reduction in CPU loading.

This highlighted the necessity of adding peripheral intelligence to the terminal expander system, a concept long used in the mainframe computer industry. This would then off-load much of the tedious I/O work from the host CPU. The second project acted as a test bed for some concepts on adding peripheral intelligence, while creating a useful line printer controller at the same time. This small project nevertheless highlighted certain weak design philosophies and gave valuable experience in the design of microprocessor slave controllers.

The third and major project was the design of a single printed circuit board comprising an intelligent microprocessor controlled four terminal multiplexer. This unit constructed to professional standards, has fulfilled all the expectations held for an intelligent interface. It has proved itself able to significantly reduce the CPU loading during input/output transactions while also increasing the number of terminals that can be physically accommodated in the host machine frame.

A further benefit from the users point of view has been the increase in terminal options and facilities made possible by the software controlled approach of a microprocessor driven interface. Features such as on line configuration of all communication parameters and message protocol have considerably enhanced the ease of use of the system and aided cabling standardisation. Software configuration has completely removed the need for the special cables, special switch settings and multiple jumpers so often used to change features on dumb discrete wired logic controlled interfaces.

The incorporation of a microprocessor eased both the design and implementation (debug) phases of this project; the design simplification being due to the very simple and systematic way in which standard members of a microprocessor family

connect together, while the debugging reduction was due to the short lead times taken to change software compared to making alterations to complex hardwired logic. The hardware design was produced and committed directly to a through plated, double-sided, printed circuit board without any breadboarding or testing or any sort. The layout, routed on an auto-routing design package, used over 30 metres of track and contained 1900 holes, yet was debugged totally in five days, requiring only nine patch wires (mainly routing mistakes) and one chip removal due to a 'glitch' sensitivity in the DMA controller.

This of course was not the end of debugging since the software development and testing was by far the largest effort. However, by adopting a systematic approach to software layout, and by using a multitasking operating system for overall control, the code was produced as many relatively independent modules, all about one page long. This modularity eased debugging dramatically, as well as reducing the number of errors generated since large rambling blocks of code are far more error prone than multiple short blocks with well defined functions.

As with most designs, a retrospective look at the design will produce new and simpler ways to implement things. When the design was completed, the density increase that occurs continuously in integrated circuits, had altered to such an extent that it would have been possible to produce an 8 channel multiplexer, controlled by a 16-bit microprocessor, with four times as much memory and implement it in the same physical space. The extra memory and ports would have been most useful with the 16-bit processor being needed to maintain performance. These changes were due solely to improvements in technology rather than alterations in philosophy.

One philosophy change that would however have improved matters involved the message passing format.

The message format was designed to pass all the useful variables in the EQT to the interface in as compact a form as possible, which required the host to unpack the EQT and repack it into the message header for output and vice versa for input. A more simple approach would have been for the host to merely post the entire EQT entry to the interface with the interface micro-processor doing all further processing. This would have made the host driver simpler, shorter and more universal.

In summary this four channel multiplexer has achieved all the aims originally envisaged, and with some eight years of use from over thirty boards, has proved itself to be a valuable addition to the HP 1000 system.

Appendix A

A Terminal I/O Extender for the HP1000

A.1 Introduction

This appendix describes the first approach taken to try and increase the terminal handling capacity of the HP1000 minicomputer without incurring the expense of buying a standard Hewlett Packard I/O extender and several standard HP terminal interface cards. The approach taken was to design a custom I/O extender and terminal interface to be totally compatible with the standard Hewlett Packard buffered teletype interface [15]. Where possible, modern MSI and LSI circuits were used to implement the design, subject to the constraint of maintaining compatibility with the standard interface and its driver software (DVR00) [16].

The first section of this Appendix describes the standard HP I/O backplane signals, their use and their timing relationships. It also illustrates the standard logic used by HP to implement this timing. The next two sections cover the design of the extender mainframe and the terminal interface card.

A.2 HP I/O Backplane Signals. Normal Implementation.

The HP1000 series machine contains a single 16 bit bi-directional I/O bus and a multitude of timing and control signals to perform the input and output and to synchronise interrupts and interrupt priority. Most I/O control circuitry is distributed on the interface cards themselves. Figure A.2 shows the logic found on every HP1000 interface card, and figure A.1 shows the timing relationships. [17,18]

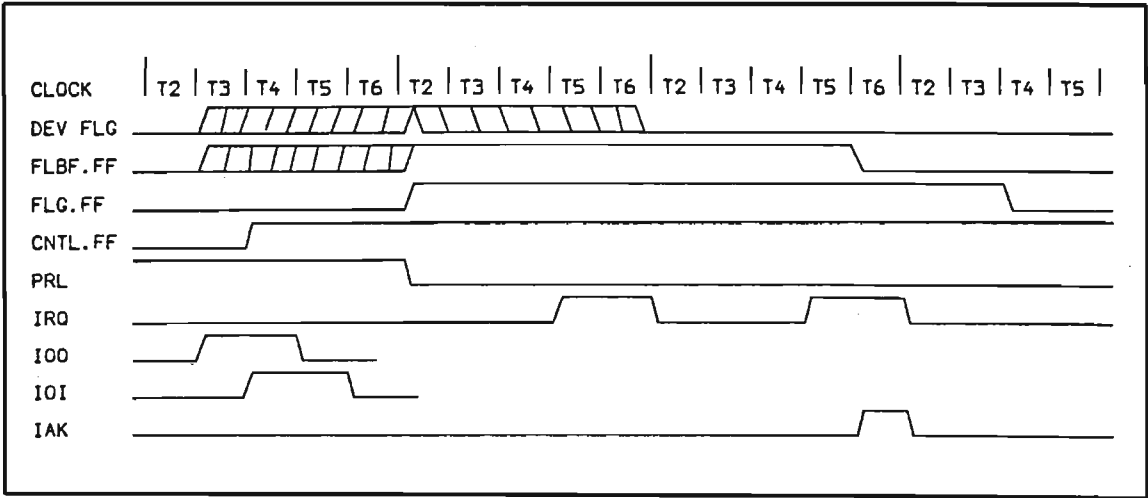


Figure A.1 Timing of Flag, Control and Interrupt Logic.

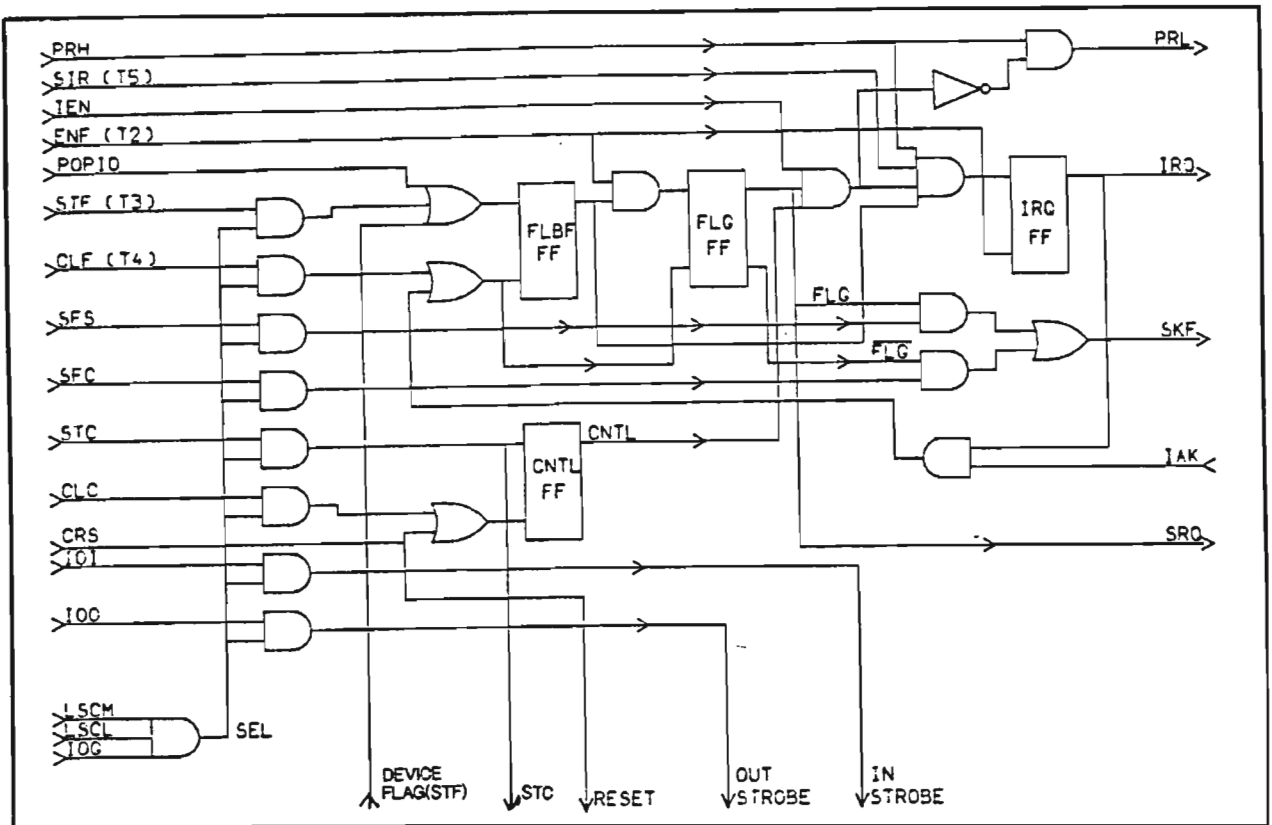


Figure A.2 Standard HP I/O Interface Logic.

The control and interrupt logic shown in figure A.2 revolves around two basic blocks, the CNTL flip flop and the FLG flip flop both of which may be set or cleared under programme control. When the CNTL flip flop is enabled the interface may interrupt the CPU provided several other conditions (described below) are met. Since this flip flop is usually left set, its set command line (STC) doubles as a 'ready' signal to the peripheral to strobe data in and out.

When a peripheral device wishes to interrupt the host it issues a 'device flag' signal which sets the flag buffer flip flop (FLBF). This signal is synchronized with timing signal T2 (see figure A.1) and used to set the FLG flip flop. Provided CNTL is set, and the interrupt system is enabled (IEN true), the priority chain will be disrupted and PRL will go false, disabling all lower priority devices. At time T5 (see figure A.1) the SIR (Service Interrupt Request) signal occurs which causes the IRQ flip flop to set providing PRH (priority in) is high and FLBF is still set. Since all FLG flip flops are set each T2 time and the IRQ flip flops at T5, the periods T2, T3 and T4 are allowed for the priority chain to settle. Should the CPU acknowledge the interrupt it issues the IAK signal during T6 which clears the FLBF flip flop, hence disabling further interrupts. It does not, however, clear FLG and so does not release the priority chain. This is only done under programme control via the CLF instruction. Should the host not acknowledge the IRQ signal during the first T6 interval, the IRQ flip flop will be cleared at the next T2 (when FLG gets set) so that any new flags may alter the priority chain. This ensures that only one interface ever has its IRQ request set at any one time.

The IOO and IOI signals are gated with the SEL (slot select) signal, (as are all the other commands) to produce the output and input data strobes respectively. The only other signal of use to the peripheral section on the remainder of the interface is the CRS (reset) signal which issues a reset pulse both when the front panel preset is pressed and when the CLC 0 instruction is used.

A.3 Terminal I/O Extender Mainframe.

In order to add an I/O extender to the host machine, the HP 'Multiplexed Input/Output Accessory Kit' [19] was used to buffer all the backplane signals and bring them out on a 50 pair cable. These signals then had to be buffered (or received) again on arrival at the extender to minimize signal transmission errors. Thus the extender mainframe consisted of a 16 slot cabinet, a power supply and a cable (or master) interface circuit board. In order to reduce the FLG and CNTL logic on individual interfaces, it was decided to try and centralise as much of it as possible. However the only part that could be centralised while still maintaining all standard signal timing was the interrupt and priority request logic plus the encoding of the interrupting device's address required for the multiplexed I/O kit. Figure A.3 shows a simplified diagram of this interrupt and priority logic.

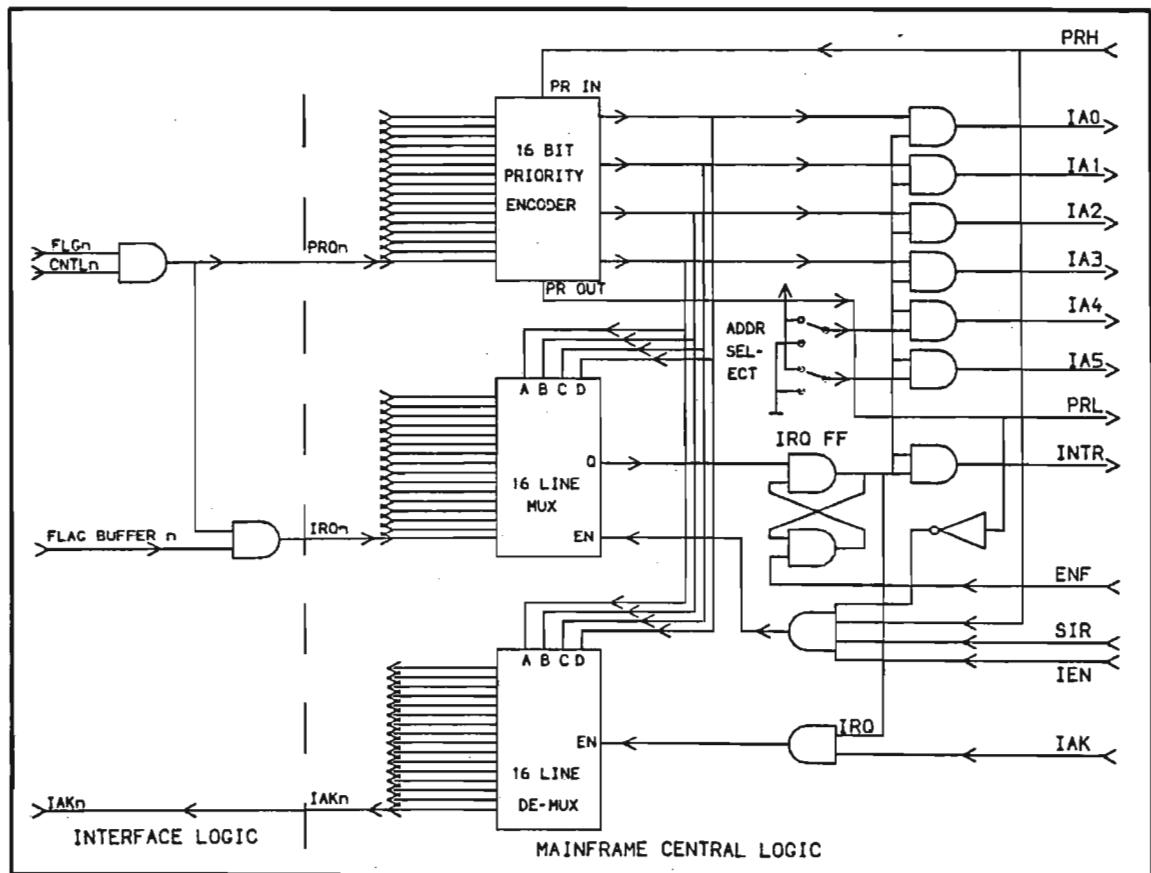


Figure A.3 Priority and Interrupt Logic in the I/O Extender.

In this system each interface card generates the signals $PRQ_n (=FLG_n \cdot CNTL_n)$ and $IRQ_n (=PRQ_n \cdot FLBF_n)$ which are sent to the master control card. The PRQ_n logic produces a 4 bit address corresponding to the highest priority device requesting service and breaks the priority chain. The address is used to select the IRQ_n output of the requesting channel, which is then used to set a single central IRQ flip flop. The signals required in order for the interrupt request flip flop (IRQ) to be set, namely $PRH \cdot \overline{PRL} \cdot SIR \cdot IEN$, are all gated together to enable the IRQ multiplexer. This single IRQ flip flop will be set every T_5 and cleared every T_2 until the CPU issues an IAK (interrupt acknowledge) signal.

The IAK is gated with IRQ and routed to the correct interface card via the 16 line IAK demultiplexer which is also addressed by the priority encoder output. This system saves about 3 integrated circuit packages per interface card and also speeds up the priority determination logic. Figure A.4 shows the full logic diagram of the mainframe master control card which apart from the centralized priority control logic contains receivers and buffers for all the signals from the multiplexer I/O accessory kit.

Since the extender was designed specifically for serial terminal I/O, a master baud rate generator was included on the master card to generate ten of the most common baud rate clocks used for serial I/O. These clocks were derived by suitable division of a 10.7 MHz oscillator and supplied at 16 times the nominal baud rate to the extender backplane. The clocks were supplied at 16 times the baud rate since this convention is becoming a 'de-facto' standard for most integrated UARTS (universal asynchronous receiver-transmitters), one of which ^[20] was used on each serial interface card.

The extender mainframe was wired so that all signals from the HP CPU cable passed through it and could be accessed on a rear mounted circuit board edge connector wired to be identical to the normal HP 'Multiplexed I/O accessory kit' connector. This allowed for multiple extenders to be daisy chained together, each one offering sixteen terminal slots assignable to any block of sixteen adjacent channel select codes. The only signal that was not wired straight through was the PRH/PRL signal pair, which passed through the priority logic before being passed on to the output connector.

The mainframe was constructed using a 7 inch high rack mount case with an 8 amp 5 volt power supply mounted in the front section. The interface cards were plugged into 116 way connectors in the rear section, with the terminal cables being led from the top of the cards and out of the rear of the cabinet. The 50 pair multiplexed I/O cable was brought into the rear panel via two 50 way D-type connectors, and fed out again on the rear panel via the 100 way printed circuit edge connector mentioned above.

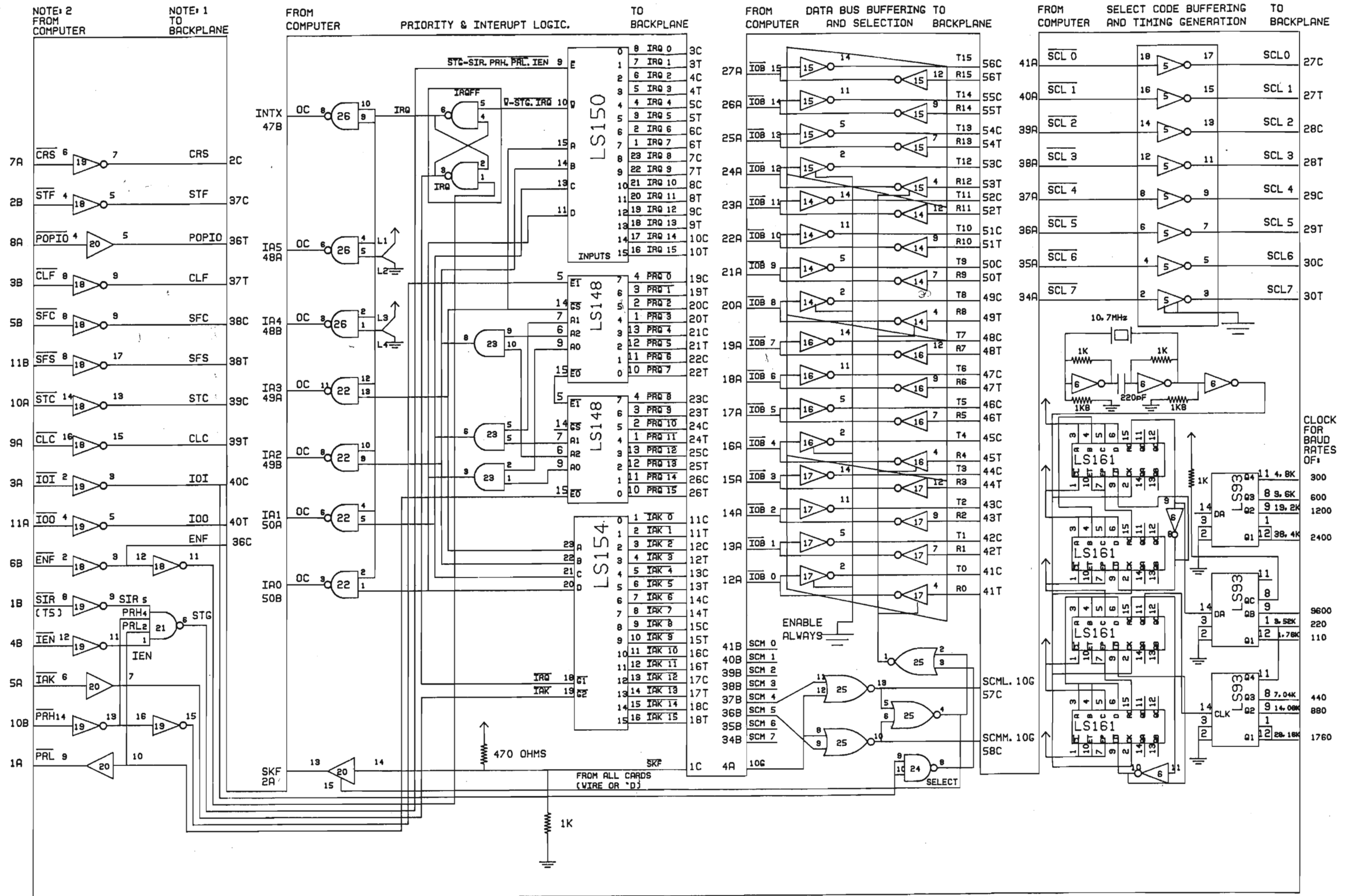


Figure A.4 Master Control Board Logic.

A.4 The Terminal Interface Card

The terminal interface card (Figure A.5) was designed to appear exactly the same as a standard HP buffered teletype interface card to the HP driver DVR00. However a standard LSI UART was used instead of the 10 bit shift register of the standard interface, to create a double buffered full duplex interface as compared to the singly buffered half duplex interface of the standard system. To maintain compatibility however, the interface was only used in half duplex mode, the direction of signal transfer being determined by the state of the IN flip flop in the control register. The control register is a three bit register, loaded by any output instruction which has data bit 15 set. The three bits (bits 14, 13 and 12) are the IN, PRINT and PUNCH bits used to set up the interface direction and enable the printer or punch on an HP modified model ASR33 teletype [21]. The discrete components used to implement the PRINT and PUNCH COMMAND signals are used to route the 20mA output signal through either or both of the PRINT or PUNCH solenoids in the TTY. Reference [21] describes these signals and their function within the teletype in greater detail.

The READ COMMAND, a ground true signal to enable the teletype keyboard, is only activated when the READ flip flop is set. This flip flop is set by a 'set control' (STC) instruction being issued when the interface is in input mode and is cleared whenever a data character is received on either the current loop or the RS232 inputs. Input characters may be echoed back to the terminal when in input mode if either PRINT or PUNCH is set.

Some extra logic which was required to ensure compatibility with the standard interface, degraded the interface's capability and would not have been necessary on a newly specified design. The first such area was to only allow the host to read an input data character once. The act of reading a character had to clear the interface so that a subsequent read would input 'all ones'. Since it was not practical to clear the UART used, a BUSY IN flip flop and some gating was used to ensure that the backplane drivers could only be enabled (by IOI) when either BUSY IN was set, or when the UART Data Ready (DR) signal was true. Since IOI clears the DR signal (via Data Ready Reset (DRR)), DR had to be delayed by about 1 microsecond to allow IOI to complete its input. The BUSY IN flip flop is set by the incoming character start bit, and is cleared by the Data Ready Delayed (DRD) signal. A more sensible design using the UART would merely have been to include the DR bit in the return to the driver for it to examine rather than have it check for an all 1's word for duplicate reads. This would however have required driver changes and lost the compatibility goal.

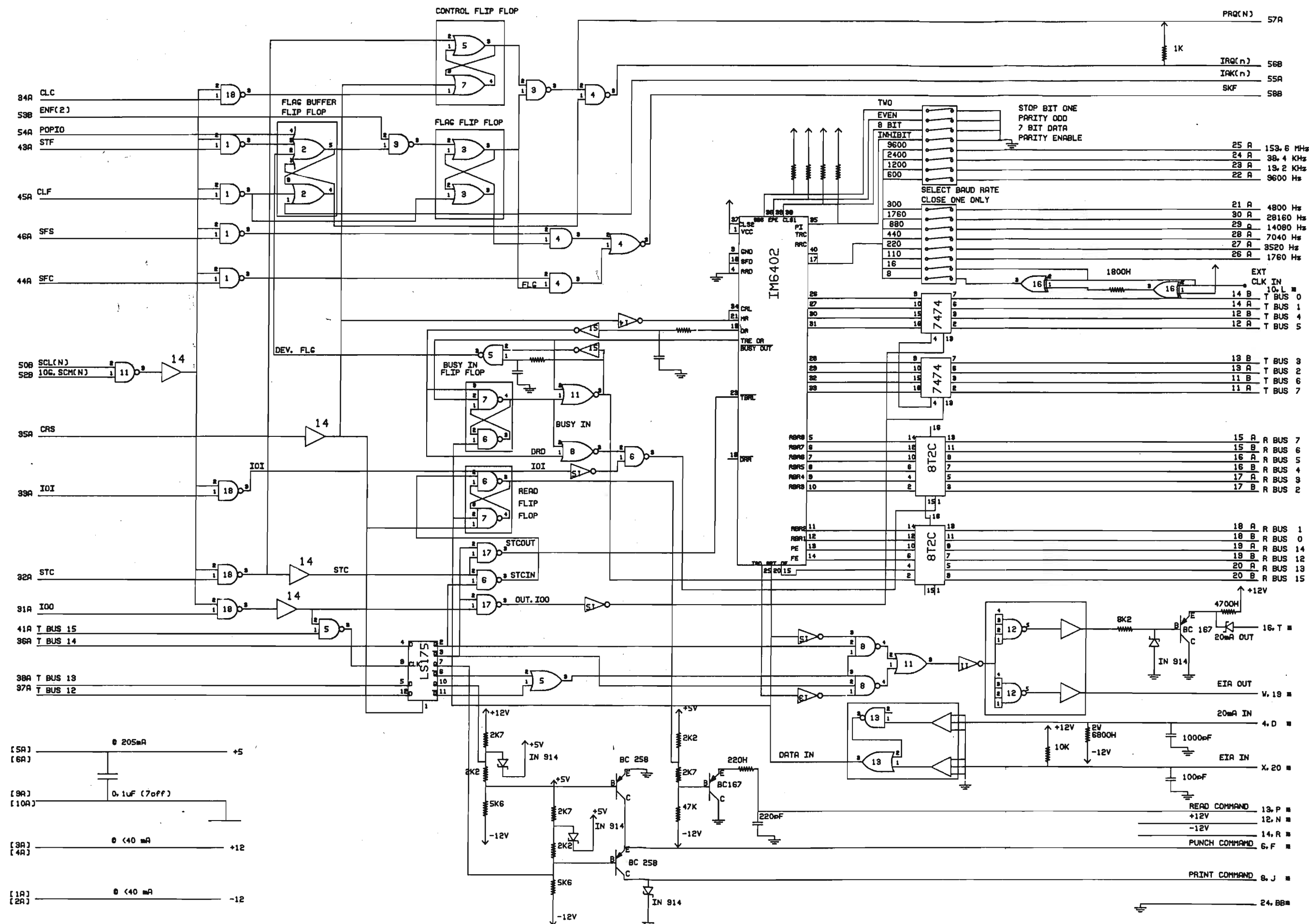


Figure A.5 Terminal Interface Logic.

The second set of undesirable logic was to make the card accept an external data clock running at only eight times the data rate. This was required since several of the older HP terminals supply such a clock which is used to enable the terminal to control the interface data rate. The newer terminals all supply the 16 times clock but at the time of design, several of the older devices were in use. Since the UART required a 16 times clock, a frequency doubler was added. This consisted of an edge catching monostable made up from two exclusive or gates and a delay circuit set to about two microseconds. This time was chosen to achieve a roughly even duty cycle at the maximum expected clock rate of 307,2 KHz corresponding to a 16 times 19,2K baud clock. Measurements showed that the UART could accept clock pulses as narrow as 500 ns, and that the duty cycle had no effect on the UART operation, hence this simple frequency doubler proved adequate.

Switches on the card allowed any of the 10 internal baud rates to be selected, or the 8 or 16 times external clock. A further four switches allowed for the setting of one or two stop bits, seven or eight data bits and odd, even or no parity. While these features were not available on the standard interface, they did not affect the compatibility of this interface, but merely enhanced its ease of use. Similarly, the parity, framing and overrun error indicators available from the UART were fed to the host backplane as bits 14, 13 and 12 of the returned data. This was done so that should a different driver routine ever be written the information would be available, while DVR00 ignores these bits and so was not affected.

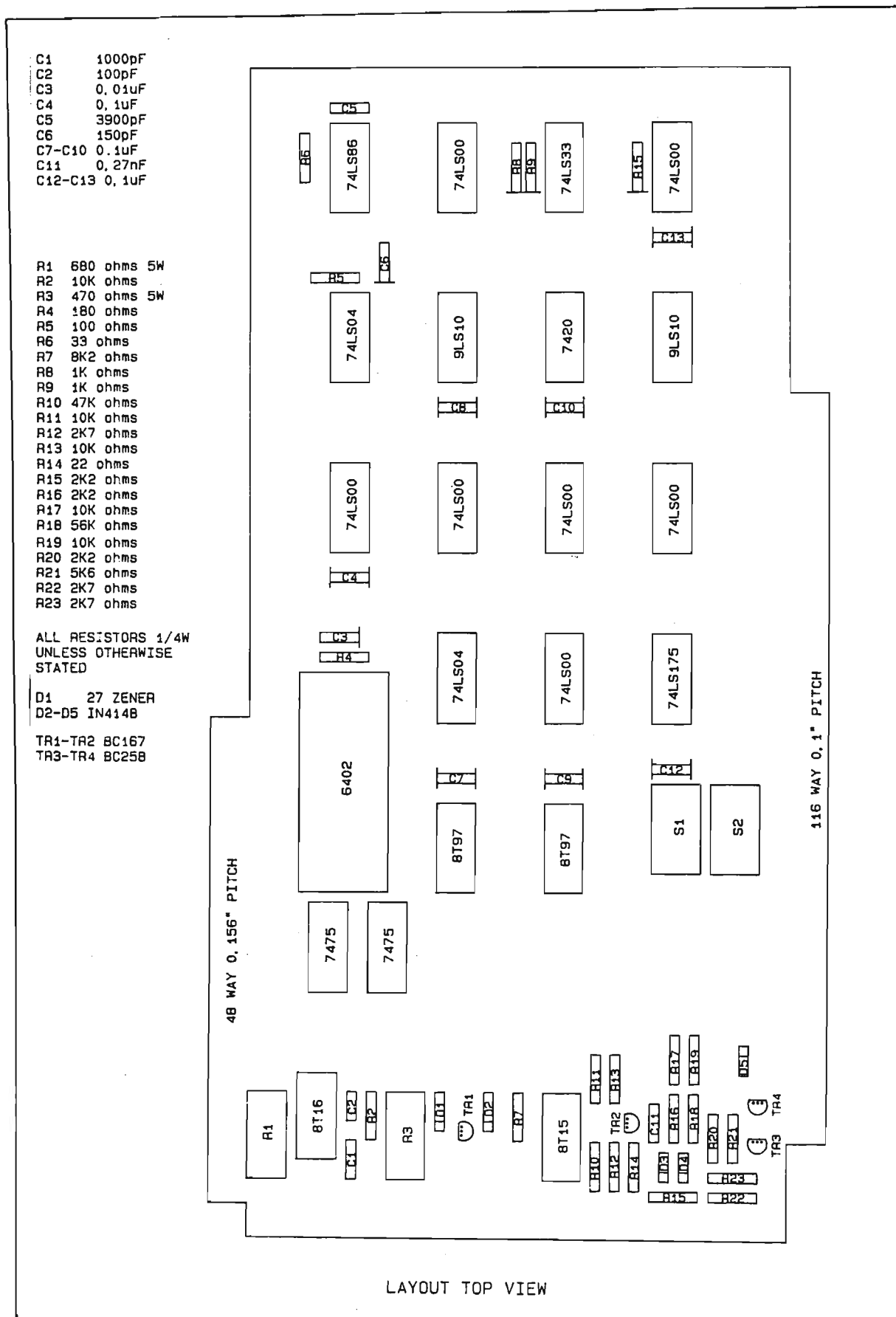
Figure A.6 shows the terminal interface component layout and tables A.1 and A.2 give connector assignments for the interface card, the 48 pin connector using exactly the same pins as the standard HP interface it emulated. The wiring from the computer to the extender was wired in accordance with the directions in the HP 'Multiplexed I/O accessory kit' manual ^[19] with the same pin numbers being used on the master control card when the signals entered the extender.

Table A.1 48 pin Terminal connector assignments

PIN NO.	SIGNAL DESCRIPTION	SIGNAL LEVEL
4 & D	Current loop input	20mA
6 & F	Punch command	20mA
8 & J	Print command	20mA
10 & L	External 8/16X clock input	TTL
12 & N	+ 12 volt supply output	
13 & P	Read command	20mA
14 & R	- 12 volt supply output	
16 & T	Current loop output	20mA
19 & W	RS232 output	RS232
20 & X	RS232 input	RS232
24 & BB	GROUND	

Table A.2 116 pin Interface backplane signals

TRACK SIDE CONNECTIONS		COMPONENT SIDE CONNECTIONS	
Pin #	Signal	Pin #	Signal
1 & 2	- 12v	1 & 2	
3 & 4	+ 12v	3 & 4	
5 & 6	+ 5v	5 & 6	
9 & 10	GND	9 & 10	
11	TBUS 7	11	TBUS 6
12	TBUS 5	12	TBUS 4
13	TBUS 3	12	TBUS 2
14	TBUS 1	14	TBUS 0
15	RBUS 7	15	RBUS 6
16	RBUS 5	16	RBUS 4
17	RBUS 3	17	RBUS 2
18	RBUS 1	18	RBUS 0
19	RBUS 14	19	RBUS12
20	RBUS 13	20	RBUS15
21	CLK. 300 Baud	21	
22	CLK. 600 Baud	22	
23	CLK 1200 Baud	23	
24	CLK 2400 Baud	24	
25	CLK 9600 Baud	25	
26	CLK. 100 Baud	26	
27	CLK. 220 Baud	27	
28	CLK. 440 Baud	28	
29	CLK. 880 Baud	29	
30	CLK 1760 Baud	30	
31	IOO	31	
32	STC	32	
33	IOI	33	
34	CLC	34	
35	CRS	35	
36	TBUS 14	36	
37	TBUS 12	37	RBUS 8
38	TBUS 13	38	RBUS 9
39		39	RBUS 10
40		40	RBUS 11
41	TBUS 15	41	TBUS 8
42		42	RBUS 9
43	STF	43	RBUS 10
44	SFC	44	RBUS 11
45	CLF	45	
46	SFS	46	
47		50	SCL n
48		52	IOG.SCMn
49		53	ENF (T2)
54	POPIO	54	
55	IAKn	55	
56		56	IRQ n
57	PRQn	57	



Appendix B

The Honeywell Line Printer Controller. – Hardware Description

B.1 Introduction

This chapter describes the two hardware modules which were designed to control the operation of the Honeywell Model 112 line printer. The line printer logic itself was only slightly altered and these alterations are also described. Further detail of the printer operation and interface specifications can be found in the modified 'Printer Operation and Maintenance Manual', [22] which fully describes the printer operating functions as well as all the modifications made to incorporate the printer controller.

Both the modules described in this appendix were contained on single circuit boards compatible with other Honeywell cards, the timing generator module being solder wired while the processor module was wire wrapped. Both modules were installed in the printer card frame, the timing card in an unused slot (WVA1H) and the processor module in place of one of the line driver modules (WVA3C). Three modules of line drivers were removed (WVA3A to WVA3C) since the printer was no longer remote from its controller, and some of the wire wrap connections on the printer cardframe backplane were altered to accommodate the new controller. The references WVAxx refer to card locations in the Honeywell card frame [22]. Power and most of the signals required for the cards were obtained from the 36 pin connectors which connect to the backplane. The timing and processor cards were connected together by a 14 way connector and cable and the processor card contained a standard RS232 type D25 connector to bring in the serial data line.

B.2 The Timing Generator Module.

As each row of characters on the print drum is about to come under the print hammers, a magnetic pickup causes a character strobe signal (CHS) to occur, its occurrence rate being approximately every 3,2ms. On receipt of the CHS strobe, the printer controller must cycle through the print buffer, producing a clock pulse (PAA) for each character position and a True Compare Pulse (TCP) for every character to be printed (See figure B.1 and [23]). The printer signals a line end by a pulse on the Sentinel Bit Detect (SBD) line one clock pulse from the end. The timing generator module controls the generation and timing of the clock signals and also performs the comparison between each print buffer character and the current print drum character. The timing logic is controlled by a sequential circuit driven by an oscillator with a period of nine micro-seconds.

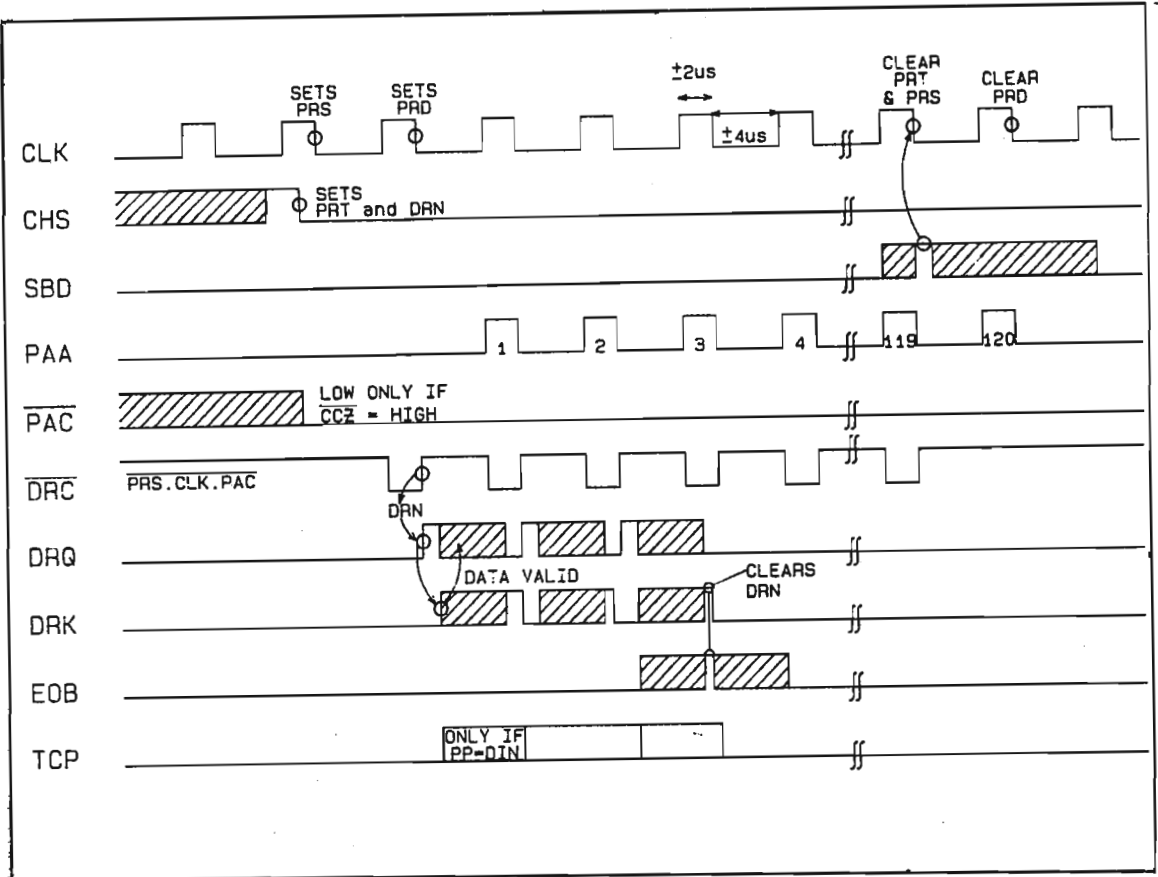
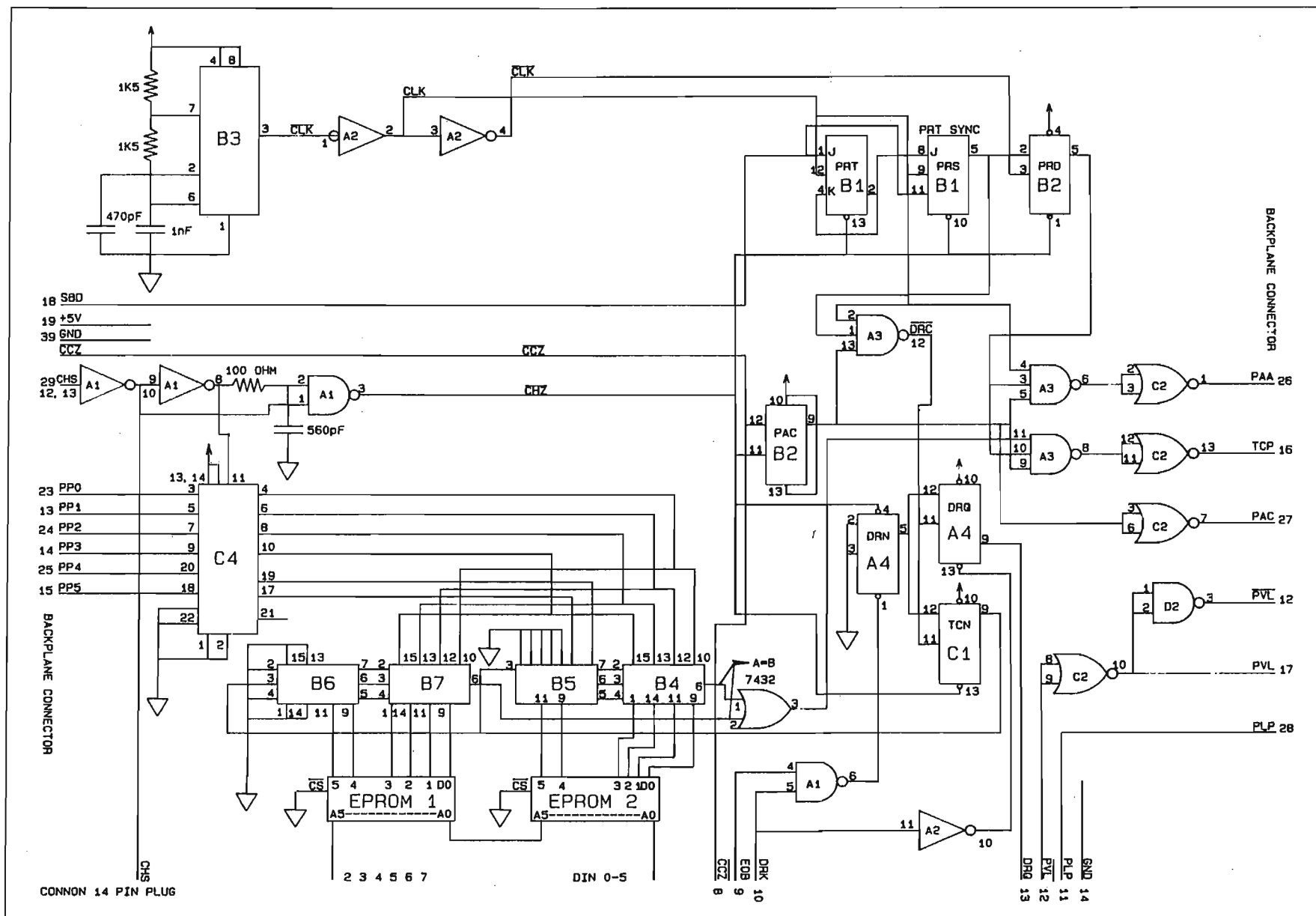


Figure B.1 Timing Diagram for the Timing Generator Module.

The negative edge of the CHS strobe is used to set the flip flops PRT and DRN. PRT is a holding flip flop to allow PRS to synchronize the print cycle with the clock, while DRN is the DMA request enable flip flop. The same edge of CHS is also used to set the Print And Compare (PAC) flip flop which signals the printer that a print cycle is about to start. This can only be set if CCZ is false, i.e., the character count is not zero and there are still characters to print for as soon as CCZ is true and there are no further characters to print, CHS will clear the PAC flip flop and stop further printing. The first clock after CHS will cause PRS (the printer sync.) to set as well as DRQ, which issues a DMA request. The processor will issue the data on the DIN lines and send a DMA Acknowledge (DRK) signal. The character is passed through two converter eeproms, which enable overprinting to occur, is then compared to the current drum character held by the PP latch, and the result is then fed out on the (A=B) line.

On the next clock cycle, PRD (Printer Sync Delayed) sets, which providing PAC is true causes a PAA clock signal to be issued. If A=B was true, a TCP (True Compare) pulse would also be issued. At the same time, the next DRQ pulse is issued to fetch the next character from the controller print buffer. This sequence with DRQ one cycle ahead of PAA continues until the print line scan is complete at which time the DMA controller (DMAC) in the processor issues an end of block (EOB) signal. This resets the DMA enable signal (DRN), which on the next clock

Figure B.2 Timing Generator Module Schematic



pulse will clear the TCN (True Compare Enable) and DRQ signals, hence ceasing all further DMA requests and disabling the comparators on the (A=B) line. At the end of the line of print (120 or 132 characters), the sentinel bit (SBD) pulse causes PRT and PRS to reset, which then resets the PRD signal after the one cycle delay. This stops the cycle until the next CHS causes it to repeat provided there are still characters to print (CCZ not true). Each line cycle takes 1,2ms, leaving 1,9ms for the printer to transfer the bits just created out of the buffer register and into the print register for printing. The printer manual^[22] describes all the timing details of the standard printer in greater detail.

Overprinting was implemented using two eprom lookup tables. For normal single print characters one eprom feeds through directly while the second eprom puts out a code which does not exist on the print drum. For overprint characters, the two eproms store the two overprint character codes causing both characters to print in the same position; e.g. the '!' was made up from an apostrophe and a full stop. Overprint characters were assigned to the codes for unused drum characters such as the symbols for quarter etc., for which no ASCII codes exist.

B.3 The Printer Controller Processor Module

The processor module shown in figure B.3 was based on a 4MHz Intel 8085 processor with ram, rom, serial and parallel I/O and a direct memory access controller (DMAC). The DMAC and processor were wired according to standard procedure, with octal latches B2 and B1 being used to de-multiplex the top and bottom halves of the address bus from the data bus. B2 was used for DMAC de-multiplexing when AEN (DMAC address enable) is high, and B1 for processor de-multiplexing when AEN is low. Rom consisted of a single 2732 4Kbyte eprom at address 0000H-0FFFH while ram was split into two: 1K bytes at address 8000H-83FFH (folded 4 times to address 8FFFH), and 256 bytes from 2000H-20FFH. The chip select decoder used A15, A13 and A12 as address inputs, so selecting the first four and third four of the 16 possible 4K blocks. As the decoder did not incorporate read or write strobes, the ram select signal was further gated to include the RD+WR timing strobe. All other devices fed from this decoder incorporated the strobes so needed no further logic.

The 8085 and the DMAC produce different sets of read/write timing strobes so the quad 2 input multiplexer was used to convert the three 8085 signals (\overline{RD} , \overline{WR} and $\overline{IO/\overline{M}}$) to those used by the DMAC and everything else (i.e. \overline{MEMR} , \overline{MEMW} , \overline{IOR} & \overline{IOW}). Only one DMAC channel (#2) was used, with latch G3 to latch the output data. The latch clocks on a negative pulse from C2-3 which 'ands' together IOW and DMA acknowledge.

The Intel 8155 combination ram, I/O and timer chip at F1/2 has two of its I/O ports, its ram and its counter/timer in use. The counter/timer was used as a down counter to count the true compare (TCP) pulses from the timing generator, port A to input 6 bits of status information from the printer, three of which were latched, and port C to produce 5 bits of output. Two lines (PVL and PVH) were used to switch the paper feed motor to low and high velocity modes while the remaining three were used to reset the input latches, reset the CHS signal latch and supply the CCZ signal to stop printing. The 256 bytes of ram were used for operating system variables and stack, while the 1K ram at 8000H was used exclusively as a 1K circular data buffer. In order to enable the DMAC to cope with a data record in this buffer overlapping the ram boundary, the ram was made to repeat at addresses 8400H, 8800H and 8C00H thus, for single records, enabling the DMAC to treat ram as a linear list rather than a circular one.

The three RST interrupts on the processor were all used: RST7.5 being fed by PLP, the paper line pulse which indicates when the paper has moved one line; RST6.5 by a latched version of CHS to act as a 3ms time base interrupt; and RST5.5 by the RXRDY signal from the serial port indicating when a new character has arrived.

The Intel 8251 serial port used RS232 drivers and receivers to interface the most common RS232 modem signals. A CMOS baud rate generator (MC14536) was used to generate different data clocks up to 19200 b/s. This clock, which runs at 16 times the data rate, was divided by two prior to being fed out to the RS232 port, since the interface used in the host HP computer required an 8 times clock. The clock used a simple RC time constant circuit to produce the reference frequency of 614,4 kHz, due partly to the difficulty in obtaining a suitable crystal, and partly since asynchronous I/O can withstand ± 1 percent error in the clocks without data error.

Table B.1 lists most of the signal mnemonics used in the schematic (figure B.3) along with a brief description of their meaning. Many of the signals originate from the printer itself and use the standard Honeywell names and definitions. For further detail on the signals and the printer operating procedures refer to the manual of reference [22].

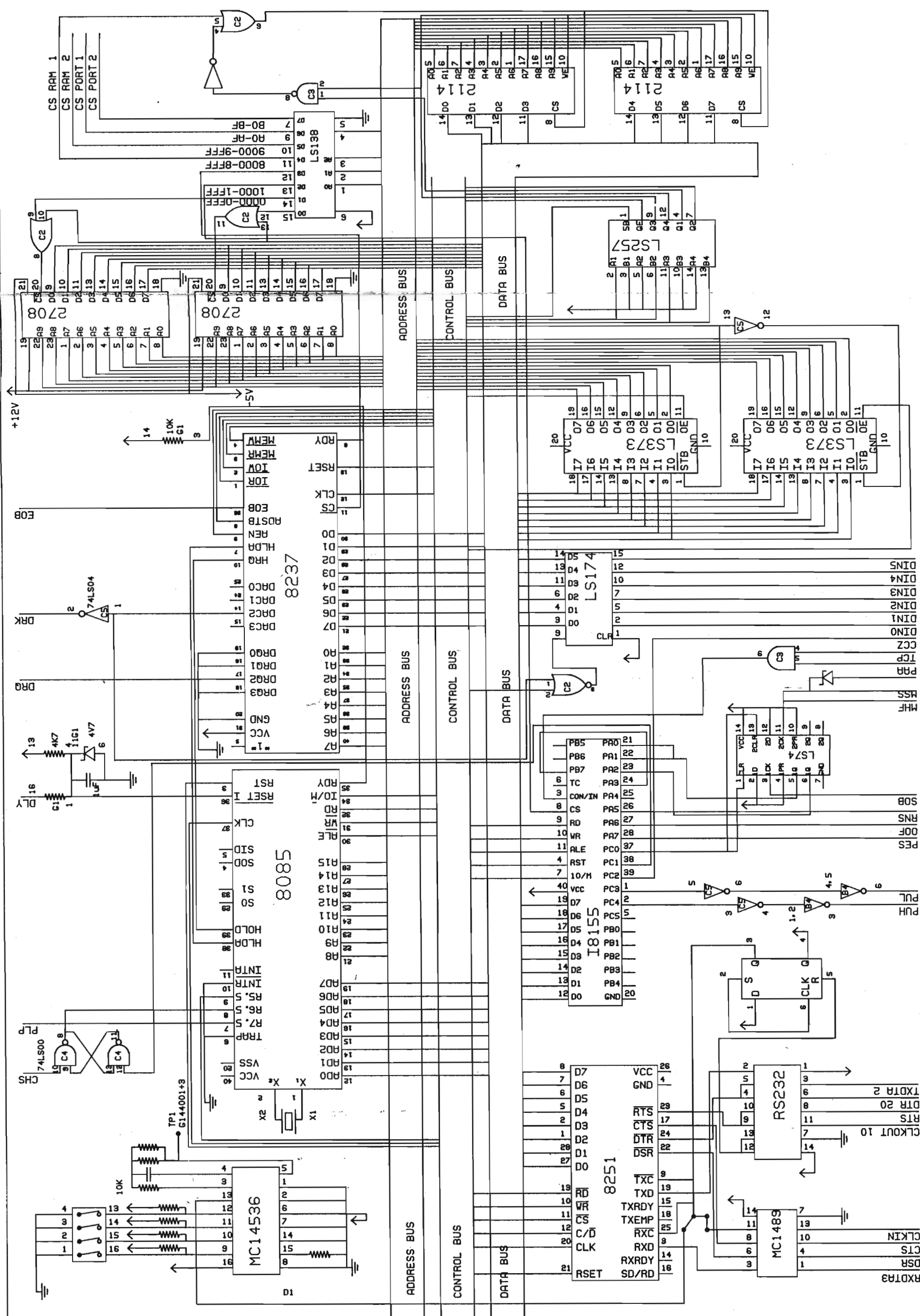


Figure B.3 Printer Controller Module Schematic

Table B.1 Printer Signal Definitions

PVL	Paper velocity low	- engage paper feed motor at low speed
PVH	Paper velocity high	- engage paper feed motor at high speed
PES	Printer emergency stop	- any error condition sets this bit
OOF	Out of forms	- special case error bit
RNS	Run state	- printer to controller to indicate ON/OFF state
SOB	Single order button	- performs 1 print cycle in offline mode
MHF	Manual head of form	- from front panel button
MSS	Manual single space	- " " " "
PAA	Print address advance	- clock to shift data to printer buffer
TCP	True compare pulse	- set with PAA for each printable character
CCZ	Character count zero	- signal from μ P when print cycle complete
DIN0-5	Data in	- 6 bit data from μ P to printer under DMA
PLP	Paper line pulse	- pulse from printer to μ P on each line move
DRQ	DMA request	- from printer logic to μ P for 1 DMA cycle
DRK	DMA acknowledge	- μ P to printer to acknowledge DMA cycle
EOB	End of block	- signal from μ P DMA at end of record
DLY	Delay	- printer to μ P signal after 10 sec. warm up delay
CHS	Character strobe	- printer pulse on each drum character move
PP0-5	Printer pattern	- 6 bit code for character next under the hammers
SBD	Sentinel bit detect	- bit from printer 1PAA pulse before end
PAC	Print and compare	- to printer to indicate print cycle active
AEN	Address enable	- DMAC to μ P to indicate DMAC has bus
ALE	Address latch enable	- latches μ P address A0-7 from data bus
ADSTB	Address strobe	- latches DMAC address A8-15 from data bus
RST	Reset	- from μ P after DLY
CLK	Clock	- 2MHz from μ P
IOW	I/O write	- negative true timing strobe
IOR	I/O read	- " " " "
MEMW	Memory write	- " " " "
MEMR	Memory read	- " " " "
IO/M	I/O or memory	- 8085 generated signal
WR	Write	- 8085 generated signal
RD	Read	- 8085 generated signal
HOLDR	Hold request	- DMAC wants bus from μ P
HOLDA	Hold acknowledge	- μ P grants bus to DMAC

Appendix C

Line Printer Controller Software – HEZLP

C.1 Software Overview

This appendix describes the software system used to control the Honeywell 112 line printer and connect it via a serial line to an HP1000 series minicomputer. The hardware is described in the preceding appendix, and the driver software for the host machine in appendix D. The system code (called HEZLP) was written in 8085 assembly code and was assembled using the A8085 assembler described in appendix I. The system consisted of two alternate main background routines and three interrupt driven control routines as shown in figure C.1.

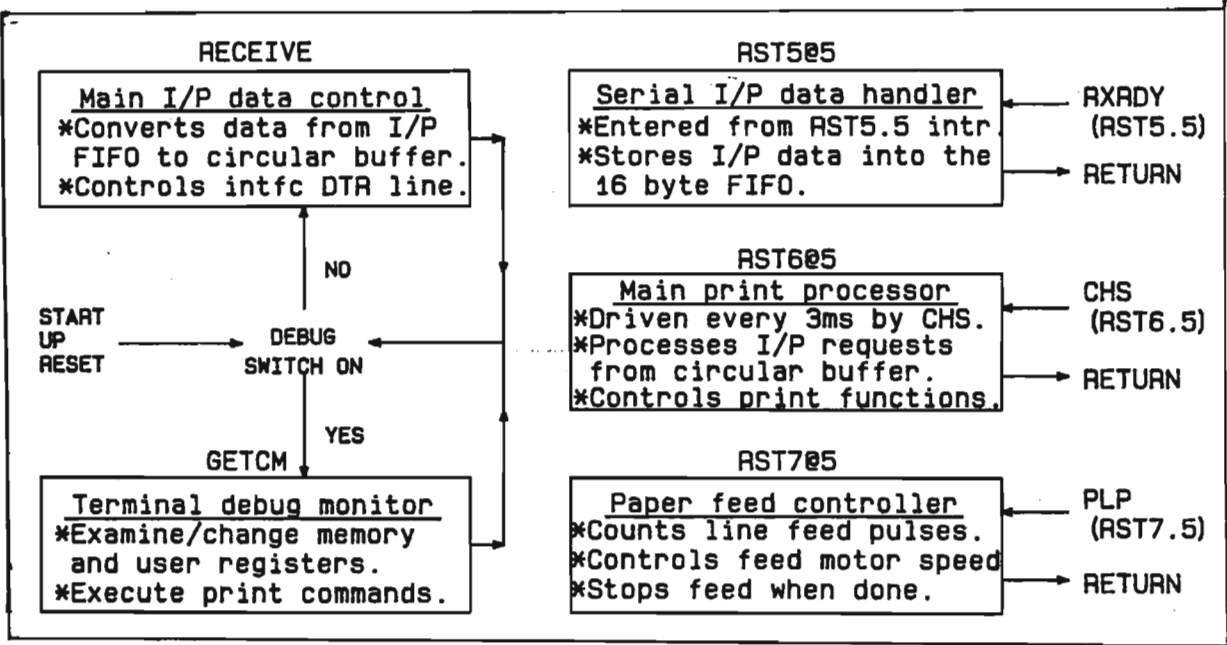


Figure C.1 Printer control programme. Module layout.

The actual background routine executed depends on the state of a switch mounted on the processor card. In normal operation, routine RECEIVE is used exclusively, but for testing, the GETCM terminal debug monitor was used to set up print records and test and alter memory. The major operating sections are RECEIVE which fetches data into the 1K circular buffer, and RST6@5 which executes the command records stored in the circular buffer. These two processes are described separately below as they are completely independent apart from the data buffer. The other background routine GETCM will be described briefly with RECEIVE, while the other interrupt routines will be described with RST6@5 since they are highly dependent upon RST6@5, and are also very small and straightforward.

C.2 Background Data Input Processor (RECEIVE)

After reset, a small section of code (RST0) runs to initialize ram and the serial and parallel ports and to set the default data values needed for the printing process. It then sets up a dummy null record in the data buffer, enables interrupts and checks bit 4 of input port A. This bit determines whether the RECEIVE routine or GETCM should be executed.

The GETCM monitor uses a subset of the commands available on most 8085 SDK demonstration kits namely:

the *X' command* for examine and change registers

the *S' command* for examine and change memory

the *G' command* for GO to user programme with registers as set by the *X' command*.

An extra command, (*R'*) was included to cause a direct jump to the RECEIVE routine. GETCM comprised routines XCMD, SCMD and GCMD to execute the three normal command options and several data formatting utilities to input and output hex characters.

The *S' command* is used by entering 'S' after the monitor prompt (a period) and following this by an address in hexadecimal. A 'space' will then cause the contents of the address to be printed, whereupon it may be altered. Another space will access the next location, and a 'return' will revert control to the command input phase.

An *X'* followed by the name of a register (e.g. XA) will print the contents of the user register and allow it to be changed. A space will sequence to the next register.

The *G'* is followed by an address and then a 'return' to cause control to pass to the given address.

An *R'* will pass control directly to the routine RECEIVE.

The RECEIVE routine has the task of synchronizing itself with the incoming data, breaking this into discrete requests, converting these to a form suitable for use by RST6@5, and storing them in the circular buffer. It also has the task of stopping the incoming data stream when the buffer becomes full, and starting it again when space becomes available.

RECEIVE gets input data from a 16 character first in—first out buffer (FIFO) which is filled by the interrupt driver routine RST5@5 as the data arrives from the host. This FIFO allows the printer to use line data rates up to 19200 bits/sec which would otherwise be impossible if RECEIVE were to accept the data directly owing

to the comparatively long periods during which RST6@5 takes control of the processor from RECEIVE.

Once RECEIVE detects an ENQ character (the start of record signal) it fetches the next two characters, combining the lower 4 bits of each to form an 8 bit record length value. It then checks that the printer is on line, that no faults exist, and that there is space in the circular buffer for the record. Should any of these conditions be false it sets the DTR line false to stop the host from sending any further data. RECEIVE then loops continually until all these conditions are met before setting the DTR line true again. To make the wait times as long and infrequent as possible, RECEIVE fills the data buffer and then holds off transmission until only two records remain to be printed.

Once all conditions are met, RECEIVE sets the DTR line true, fetches the next character, which is a buffer control character as outlined in table C.1, sets up a request record in the data buffer in the format shown in figure C.2 and links it into the linked list of data records. RECEIVE then fetches characters from the input FIFO, converts them from ASCII code to Honeywell code [24] and stores them in the data request record. This phase is terminated either by an ASCII 'return' (CR) arriving or by the full number of characters as specified in the record length.

Table C.1 Print Record Control Characters

ASCII CHAR	HEX EQUIV	EXEC CNWRD	CONTROL FUNCTION SPECIFIED
0	30H	000B+lu	Print line follows. Column 1 to be printed.
1	31H	200B+lu	Print line follows. Column 1 forms control.
I	49H	CN,l,u,9	Control function to skip lines or a page.
J	4AH	CN,l,u,10	Enable auto skip of page perforations.
K	4BH	CN,LU,11	Disable auto skip mode.
L	4CH	CN,l,u,12	Set physical page length in lines.(Dflt=68)
M	4DH	CN,l,u,13	Control request to force a page skip.
N	4EH	CN,l,u,14	Set printable lines/page for auto skip.(62)

WORD	CONTENTS
** 0-1	Two byte linkage pointer to next print record.
2	Control character. (See table C.1)
3	Printable character count of current record.
4	Record length. (N bytes.)
5	—— not used ——
6+	N data bytes sent in ASCII but stored in Honeywell code.

** Not sent from host.

Figure C.2 Print Request Record Format

As each character is converted, it is checked against one of four conditions, these being:

- * ASCII control characters which are replaced by a small square symbol
- * Printable characters cause the printable character counter to be incremented by one
- * Overprint printable characters cause the printable character counter to be incremented by two
- * An ASCII carriage return causes the record fill phase to be terminated.

The characters are converted from 7 bit ASCII to the equivalent 6 bit Honeywell codes via an ASCII character indexed look-up table which holds the Honeywell code in the lower 6 bits. Bit 7 is used to indicate the printable characters, and bit 6 to indicate the overprint characters.

Once all input data has arrived, been converted and stored, the printable character count and record length fields in the record header are set and the record counter is incremented. Finally, the RECEIVE/GETCM switch is checked to see whether to return to the start of RECEIVE, or to pass control to the GETCM monitor.

C.3 The Interrupt Driven Print Processor (RST6 5)

This print processing routine, entered every 3 ms in response to an interrupt from CHS (the character strobe), is used to initiate whatever is necessary for the next print action. Since some print cycle actions can take a comparatively long time to initiate, the interrupt system is left enabled while RST6@5 executes, with a single 'in process' flag being used to stop any subsequent CHS interrupts from attempting re-entrant processing of the same action.

A typical print cycle starts with a print cycle to print the line of text, followed by a line feed cycle to move the paper the required amount.

The print process.

To print a line, RST6@5 sets the DMA controller (DMAC) to output the print record data repeatedly by setting the DMAC into auto-initialize mode [25]. The 8155 counter is then loaded with the printable character count, and the CCZ signal (character count zero) is set high (false) to start the print process. The timing generator module (see appendix B) then repeatedly causes the print record to be scanned, once for each row of characters on the drum, until the counter counts down to zero. While in the printing phase, a print flag (PRFLAG) is set to cause RST6@5 to check the counter each interrupt. Once the counter reaches zero, RST6@5 stops the print cycle by setting CCZ low and clears PRFLAG. It then waits for a two character delay (6ms) to allow actual printing to complete since the processor runs ahead of the mechanical printing process.

The paper feed process

The next check is to see whether the paper is to be moved, and if so, by how many lines. For single line moves, the paper feed signal PVL is set causing a low velocity move, while for more than one line, the PVH signal is set for a high velocity move. RST6@5 thus determines the number of lines to move, (taking page boundaries into account), starts the feed motor at the relevant speed and sets a feed flag (FDFLAG) to indicate that the paper moving process is in operation.

For each line moved, a pickup on the paper feed motor causes a pulse on the RST7.5 interrupt line (PLP pulse). The interrupt activates the interrupt handler RST7@5 which decrements the remaining line count, changes PVH to PVL when only one line is left and removes both PVH and PVL and resets FDFLAG when zero lines are left.

RST6@5 checks FDFLAG on each CHS interrupt and if it is set no further action is taken and no other flags are checked. Once FDFLAG has been cleared, RST6@5 waits three CHS interrupts ($\approx 10\text{ms}$) to allow the paper feed servo system to settle and stop all paper motion. The print record just processed is then unlinked from the list, discarded and processing of the next record can begin.

Prior to processing each print record, RST6@5 checks that the printer is on line, and if not it checks the two manual paper feed controls, LINE FEED and FORM FEED. If either is pressed, the requisite action is taken in the same manner as for programme controlled feeds. Since records are not processed when off line, the 'Single Order Button' (SOB) is used in the off line mode to enable a single print record to be processed for each press of the button. The three manual buttons (SOB, MHF and MSS) each set a latch when pressed and cannot set the latch again

until completely released. RST6@5 resets these latches when it detects a button pressed by pulsing bit zero of port C.

RST6@5 also processes control requests most of which merely result in one of the default variables being changed. A subsection of RST6@5 called DOCNTL is used to process these control records. There are several other subroutines to RST6@5, each one being called to perform some major function such as start paper feeds (STRTFD) and start printing (STRTPR). Very little of the code is executed with the interrupt system off, but where it was necessary to ensure no interrupts could occur, the interrupt system was disabled for as short a period as possible.

The total code space required by the system was just over 2Kbytes which fitted easily into the 4K EPROM used while ram usage amounted to about 80 bytes, excluding the 1K data buffer. A copy of the source code is available from the Digital Processes Laboratory of the University of Natal should further detail be required.

Appendix D

The Host Software for the Honeywell Printer — (DVP12)

D.1 General Driver Layout

This appendix describes a driver routine written to run on the Hewlett Packard HP1000 series minicomputer under control of the RTE IVB operating system. The driver accepts standard commands from the host operating system and reformats them to suit the Honeywell line printer controller described in appendices B and C. The driver also controls the sending of the message to the printer, but does not maintain any record of the printer status.

In keeping with standard practice for RTE IV drivers, DVP12 (12 being the RTE code for printer drivers) comprises two major sections, the initiator which accepts requests from RTE and initiates the transaction by outputting the first character of the request, and the continuator which is entered in response to an interrupt to process the remainder of the transaction. DVP12 is a very small and simple driver by RTE standards since the printer controller is intelligent enough to perform most of the request processing itself.

The following sections describe the initiator and continuator sections separately with only brief mention of the standard RTE I/O system. For further detail on the requirements for RTE drivers and on the HP I/O system itself see appendix H and the relevant HP Reference Manuals [26],[27].

D.2 The DVP12 Initiator Section

The initiator routine is entered at entry point IP12 directly from the RTE IVB executive, with the A register containing the slot number of the interface to be operated upon. Since the same driver may handle several interfaces, it must configure all its I/O instructions to the relevant slot number before it can start processing the I/O request. RTE passes all the details of a request to the driver by means of a 15 word data block called an EQT entry and before entering the driver, the RTE executive places the addresses of these words into 15 reserved base page locations thus allowing the driver to access any EQT entry via indirect addressing through fixed locations. DVP12 thus fetches the request CONWORD [28] from EQT word 6 and from the lower two bits determines whether the request was a write or a control request.

For a write request the routine PRINT is called which sets up the character address and negative character count into the EQT entry words 9 and 10. It then converts the character count into two length digits, each of which holds 4 bits of the record length (see figure D.1) according to a simple hexadecimal coding scheme, and stores

these digits into EQT word 12. Finally the control character (see table C.1 in appendix C) is determined and stored in EQT 13.

Hex Digit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ASCII Char	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?

Figure D.1 The Hexadecimal to ASCII coding scheme.

For control requests the routine CNTRL is called which first checks that the request is a valid one, and if so, whether it uses an optional parameter in EQT word 7. For those that do use the parameter, it is broken into 4 four bit nybbles, each being represented by an ASCII character (see figure D.1) stored in EQT words 7 and 8. Then as for the PRINT routine, EQT 9 is set up to the character address of EQT 7 since this now contains the transmit data, EQT 10 is set to the negative transmit data length, (either 0 or -4), EQT 12 is set to ASCII '00' and EQT 13 is set to the control request character according to table C.1.

At this point, both print and control requests now appear identical and so all further processing is independent of request type. Thus the initiator sets the interface to output mode and sends the ENQ character to signify the start of a record. The initiator then returns control to RTE IV to await the completion interrupt.

The continuator is entered after each interrupt from the interface which indicates that the last character transmission is complete. After performing the standard I/O instruction re-configuration, the continuator fetches a mode pointer from the three least significant bits of EQT 5 (set to zero by the initiator) and uses this pointer to index into an address table. This table contains the addresses of six code sections that deal with the six phases involved in transmitting a record to the printer.

These are:

- * send most significant length character
- * send least significant length character
- * send the control character
- * send the data characters
- * send the completion carriage return
- * clear the interface and take the completion exit back to RTE IV.

As each phase is completed the mode pointer is incremented to cause the next phase routine to execute on the next interrupt. This method of having separate routines for each phase resulted in an extremely fast and compact interrupt continuator, helped also by the fact that no significant processing was required as most processing is done by the printer controller. The longest path through the continuator used 31 instructions while the shortest used only 18, which with average instruction time of $1,2\mu\text{s}$ resulted in an extremely rapid interrupt response.

Appendix E

RMUX Hardware Description

E.1 Introduction

This appendix serves to detail the hardware design of the RMUX four channel terminal multiplexer for HP 1000 series computers. The multiplexer consists of a single printed circuit board which occupies one standard I/O slot in a Hewlett Packard computer. The board contains an 8085 microprocessor plus associated peripheral devices which enables it to interface four asynchronous serial terminals to the host computer using standard RS232-C signals. The microprocessor enables the multiplexer to take care of all the terminal protocol and terminal editing capabilities so relieving the host of much of its I/O interrupt load.

Data transfer between the multiplexer and the host is performed on a 'message at a time' basis using direct memory access (DMA) into the multiplexer memory. This enables the host to exchange messages with the multiplexer as fast as it can, and reduces each message to one burst transfer and a single host interrupt.

Apart from logic required to interface the microprocessor to the host machine (the HP 1000), the multiplexer board contains 4 kilobytes of RAM, 8 kilobytes of EPROM, 4 full duplex serial I/O channels, each with its own programmable baud rate generator, 10 interrupt sources, a 10ms time base generator, and a 4 channel DMA controller.

The host interface logic is the most complex and so has been divided into two sections for explanation purposes: viz., the FLAG and CONTROL logic, and the Data Path logic. The remaining logic on the multiplexer consists of the processor, the memory and the peripheral interface. These 5 sections, shown in block diagram form in figure E.1 are all described separately in the following sections which all refer to the main circuit diagram in figure E.4.

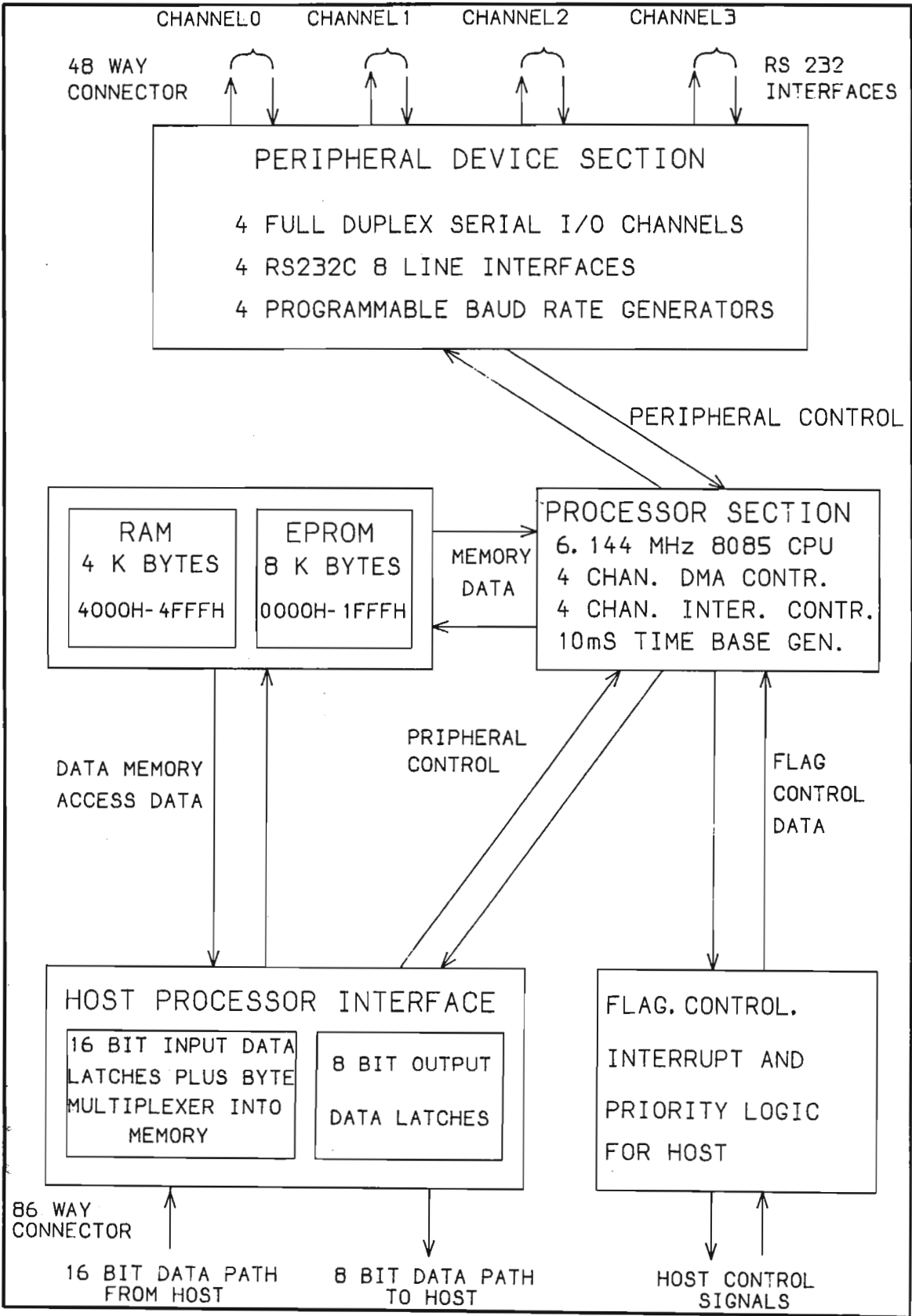


Figure E.1 RMUX Hardware Block Diagram

E.2 The Processor Section

The processor section consists of an Intel 8085 micro-processor, an Intel 8237 DMA controller, two 74LS373 octal address latches, a 74LS155 and a 74LS138 address decoder and an Intel 8259 eight channel programmable interrupt controller. Channels 0 and 1 of one Intel 8253 three channel programmable timer are cascaded to provide the 10ms time base generator.

Address De-multiplexing

The 8085 processor is run off a 6,144 Mhz crystal which was chosen to enable all common communication baud rates to be simply derived. The host signal *CRS* which occurs for both a host preset and a host *CLC 0* instruction was used to reset the 8085 and hence initialise the software. Since the 8085 multiplexes the *lower half* of the address bus onto the data bus, an octal latch was used to de-multiplex the address when the signal *ALE* is high. The 8237 DMA controller (DMAC) also uses a multiplexed address and data bus but multiplexes the *upper half* of the address bus onto the data bus. Hence it cannot use the same de-multiplexing latch as the processor which necessitated the use of a second octal latch. The correct latch is enabled in response to the DMA controller signal *AEN* which the DMAC asserts high when it has control of the bus.

Read and Write Generation

The connection of the read and write (\overline{RD} and \overline{WR}) lines requires some explanation as it deviates from the standard system recommended by the supplier. The circuitry adopted was chosen to overcome some problems which were experienced when using the standard approach. The initial problem arose due to the 8085 CPU and the 8237 DMAC using different methods for generating these read and write signals. The DMAC uses a 4 line system, providing separate lines for I/O read, I/O write, memory read, and memory write (\overline{IOR} , \overline{IOW} , \overline{MEMR} , and \overline{MEMW}), whereas the CPU uses a 3 line system of Read, Write and I/O or memory (\overline{RD} , \overline{WR} , and $\overline{IO/\overline{M}}$). Although Intel recommends a single chip solution to convert the 3 line approach to the 4 line standard, ^[29] the solution generates very narrow (20–30ns) glitches on the output lines when changing from I/O to MEM. These glitches do affect some of the peripherals and in particular they caused the DMAC to reset itself occasionally. To produce a glitch free 3 line to 4 line tri-state decoder would have required at least two chips, so an alternate scheme was used which effectively overlapped the memory and I/O address spaces.

The \overline{RD} and \overline{WR} signals from the CPU were used directly as the system \overline{RD} and \overline{WR} signals and the $\overline{IO/\overline{M}}$ line was unused. This meant that the I/O and memory address spaces overlapped, but this was accommodated by placing all I/O devices above I/O address 80H (memory address 8000H) and all memory

addresses below 8000H (I/O address 80H). This way all the I/O peripherals could be addressed as either I/O devices or memory devices. The only problem was then the DMAC which uses all 4 lines. Since it uses the $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$ lines as inputs when being addressed as a peripheral, these had to be connected to the system $\overline{\text{RD}}$ and $\overline{\text{WR}}$ lines. This then meant that the DMAC's *I/O control lines* were connected to the *system's memory control lines*, so that when the DMAC is to act as a bus master, it has to perform an I/O read in order to read memory. This problem was simply solved by reversing the definition of the DMAC read and write cycles in exactly the manner suggested by the Intel manual when memory mapping the DMAC [30]. The $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ signals from the DMAC which are only ever used as outputs when the DMAC is in master mode, are then simply gated with the DMA acknowledge signals to act as strobes for the peripheral I/O channels from and to the host respectively.

Interrupts

The system needed 10 interrupts, 4 for the 4 transmitters, 4 for the receivers, one for the time base and one for the host. Although two would have been better for host interaction, this proved impossible as the HP 1000 computer has only a single interrupt input per slot; and a single control bit per slot, hence limiting interaction to one interrupt on the 8085 CPU.

The 8 terminal related interrupts were handled by a single 8259 programmable interrupt controller (PIC) which connects to the 8085 via the *INTR* and *INTA* lines. This device accepts the highest priority input and issues an interrupt to the 8085. Upon receipt of an acknowledge (*INTA*) from the CPU, the PIC releases a 3 byte call instruction, hence producing the effect of a vectored interrupt. The inputs to the PIC are positive true and must remain set until acknowledged which was ideal for direct connection to the *TXRDY* and *RXRDY* interrupt outputs from the 8251 serial interface chips.

The time base interrupt was derived from the output of two cascaded channels of the 8253 programmable timer which was programmed to produce a square wave with a 10ms period. By feeding this square wave directly into the non maskable *TRAP* input of the 8085, the need for any interrupt latches and acknowledge circuitry was eliminated. This resulted from the fact that *TRAP* is only activated by a rising edge signal, and so only ever gets accepted once per cycle of the square wave.

The interrupt from the host occurs when the host programme executes an *STC* (set control) instruction. This signal is processed by the flag and control logic section to produce a positive true 500ns pulse (*STCG*) which is fed directly into the *RST7.5* input of the 8085, which being a positive edge triggered and latched input eliminates the need for any further interrupt holding latch.

Single Bit I/O

The 8085 needed to be able to examine the state of the *FLAG* and *CNTL* flip flops of the host interface, and also needed to be able to set the *FLAG* flip flop. The Serial Out Data (*SOD*) line of the 8085 supplies the single bit output while the Serial In Data (*SID*) line supplies a single bit of input. Rather than add an extra input port solely for the *CNTL* flip flop, the *RST5.5* line was used as a simple input port since the RIM instruction supplies the state of this line even when the *RST5.5* interrupt is disabled.

Address Decoding

The 74LS138 octal decoder supplied the chip selects for the 8 peripheral chips with each device being allocated 4Kbytes of memory address space above address 8000H (see figure E.2). Since this corresponded to the overlapping I/O address space where each device was given 16 I/O addresses above I/O address 80H, either I/O or memory instructions could be used to access the peripherals.

Since each of the peripheral devices was connected to the \overline{RD} and \overline{WR} timing strobes, the only signals necessary for the production of the I/O strobes were *A15* to cover the address's from 8000H to FFFFH and *AEN* from the DMAC to ensure that no peripherals would respond when the DMAC was in control of the bus.

The 74LS155 dual two bit to 4 line decoder was used to provide the chip selects for the RAM and EPROM. To generate the RAM chip select at address 4XXXH, address *A14* was 'and'ed with the composite signal (*RD* and *WR*) and used to enable decoder 1. This inclusion of *RD* and *WR* was necessary since the 2141 RAM chips used could only be selected once their addresses were stable, hence the inclusion of the read-write timing strobe. The EPROM however has an output enable (\overline{OE}), which was connected to the \overline{RD} line, and required the full address set up time for the chip select as well as the address lines. Thus the chip select decoder for the two EPROMS at 0XXXH and 1XXXH did not include any *RD* or *WR* signals.

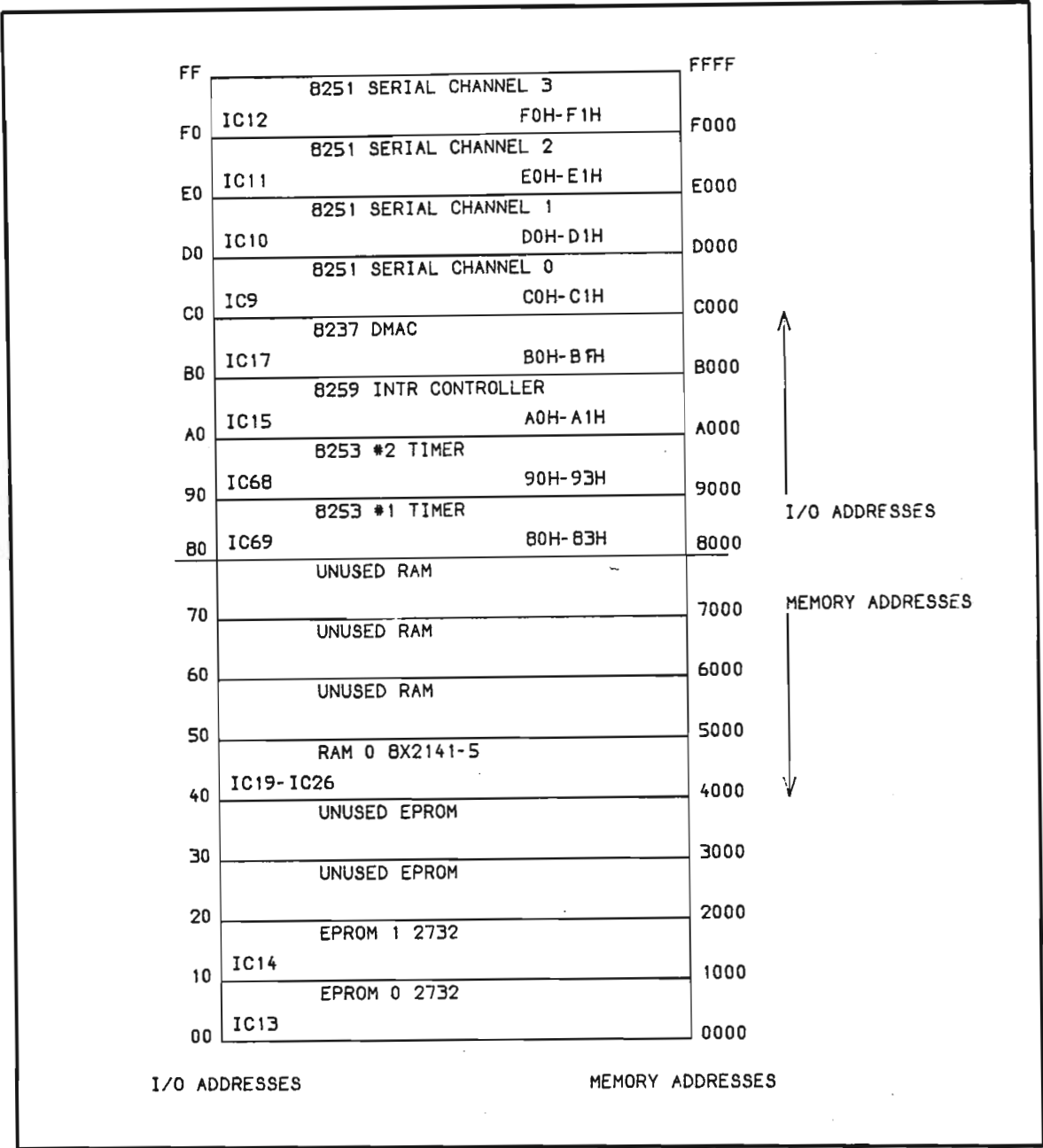


Figure E.2 RMUX Address Map

Time Base Generator

Two three-channel timers were included on the card giving six 14 bit timers. Four of these channels were used to supply the four baud rate clocks, and the remaining two were cascaded to produce a 10ms period wave which was fed into the 8085 TRAP interrupt. Since the timers have a maximum input frequency of 2MHz, the 3,072MHz processor clock was halved using a divide by two toggle before being fed into the timers.

E.3 The Memory Section.

The memory consisted of two socketed 4Kbyte EPROMs (Intel compatible 2732's) and eight 4Kbit RAMs (2141 type).

Eprom's

The 2732 EPROM was chosen as the sole EPROM device as it was the cheapest device available at the time the design was started, as well as being the largest readily available device. The address to data access time of these devices of 450 ns was more than adequate for the 570 ns requirement of the processor and since no buffers were used on either address or data buses, timing was extremely straightforward and easy to achieve.

The 2141 4K by 1 bit RAM was chosen as it only represented a single chip load on the data bus and required a single 4K block select signal whereas the cheaper and more popular 2114 type device with its' 1K by 4 configuration would have required further address decoding to produce the four 1K blocks and would also have presented 4 loads to the data bus. This level of data bus loading would have exceeded the 150pF drive capability that most of the devices exhibit and hence altered timing margins. Although in most cases the extra time lag could have been met, some of the chips gave no specifications for time delay versus bus loading, leading to some uncertainty in system timing.

E.4 The Peripheral Interface System

This section consists of the four Intel 8251 Universal Synchronous/Asynchronous Receiver/Transmitters (USART's) each of which has four RS232 level line drivers and four RS232 level line receivers associated with it.

Interface Signals

For each channel, the Transmit Data (TXD), the Data Terminal Ready (DTR), the Request To Send (RTS) and the clock out (CLKO) signals were all buffered by the MC1488 type RS232 line drivers and fed onto the 48 pin connector. The four inputs, Clear To Send (CTS), Data Set Ready (DSR), Receive Data (RXD) and clock in (CLKI) were received by MC1489 line receivers and fed into the requisite 8251 USART's. For normal modem-less or direct connect operation, the RTS, DTR, and CLKO lines are looped back respectively to the CTS, DSR, and CLKI lines and only the TXD and RXD lines are used. However the modem control lines are honoured and may be used.

Interrupt Signals

The TXRDY and RXRDY interrupt outputs from each USART were connected to the 8 inputs of the Intel 8259 PIC to provide each function with a separate interrupt path to the 8085. The RXRDY lines were connected as the highest priority since data reception is more time critical than transmission.

Address Assignments

Table E.1 gives the address assignments for the data register and the control—status register for each of the 8251 USART's, the corresponding baud rate generator's address and also the edge connector pin numbers for the eight interface signals.

Table E.1 Serial Port Address and Pin Assignments

	PORT 0	PORT 1	PORT 2	PORT 3
Data register address	C0H	D0H	E0H	F0H
Control status address	C1H	D1H	E1H	F1H
Baud rate counter address	90H	91H	92H	82H
Baud rate control address	93H	93H	93H	83H
TXD	Y	W	10	J
RXD	X	16	F	3
DTR	22	17	H	4
DSR	AA	14	E	B
RTS	21	U	8	6
CTS	23	P	D	2
CLKO	Z	19	K	5
CLKI	20	T	7	C

E.5 Host Processor Data Path Logic

Data transfers both to and from the host processor (the HP 1000 series minicomputer) are performed by DMA transfers out of and into the multiplexer RAMs. Since in typical terminal transactions, the majority of data is transferred from the host to the terminal, this data path was made 16 bits wide with hardware unpacking of bytes to save host processor time. Board space limitations excluded this option in the reverse direction so only an 8 bit data path to the host was implemented with byte to word packing being done in the host by software.

Host to multiplexer data path

The 16 I/O data lines from the host backplane were terminated by 2K2 pull down resistors to -2 volts in order to maintain compatibility with the CTL logic levels used on HP 1000 backplanes. The data was then latched into two octal latches by the host output strobe signal IOOG which simultaneously presets two J-K flip flops. The first flip flop supplies a DMA request to the DMAC which will then acknowledge, when ready with a write strobe (*WSTB1*) as discussed in section E.2. This write strobe is gated with the output of the second flip flop to produce the high byte enable signal *HBEN* which enables the most significant 8 bits of input data from the octal latch onto the data bus, allowing the DMAC to store the first byte into memory. The trailing edge of the write strobe (*WSTB1*) clocks the *HBEN* flip flop so causing it to change state. Since the DMA request flip flop is still set the next write strobe (*WSTB1*) to arrive will be routed by the *HBEN* flip flop to produce the signal *LBEN* which enables the low byte from its' latch onto the data bus. *LBEN* is also used to clear the DMA request and issue a 'set flag' signal (*STF2*) to the host to indicate that the complete 16 bit word has been stored. The timing diagram in figure E.3 indicates this sequence and also gives time limits for the 6 MHz processor.

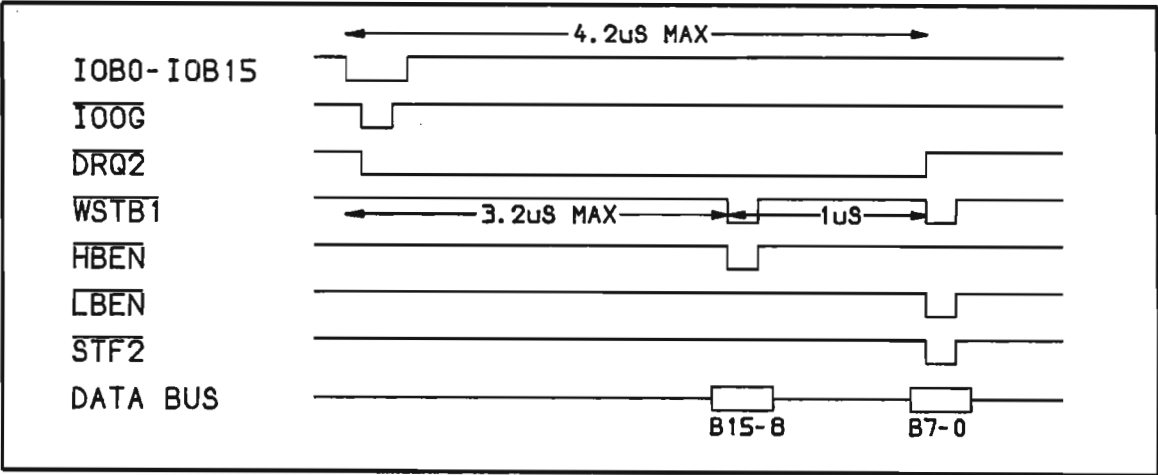


Figure E.3 Backplane Data Input Timing

Since the two bytes are input via the DMAC in burst mode, the two cycles will always be consecutive, taking a maximum of 3 clock cycles each. This time added to the maximum DMA latency of an 8085 running at 6,144 Mhz indicates that the maximum delay from the IOOG signal to the return STF2 signal will always be less than 4,2 μs. This indicates that the maximum sustainable input data rate is 475 Kbytes per second.

Multiplexer to host data path

The *DRQ3* flip flop starts off clear after the initial reset, which places a DMA request signal on the *DRQ3* line. Once the DMAC is enabled, it will acknowledge the request by placing the data on the data bus and issuing the acknowledge strobe *STF1*. This latches the data into the 74LS374 latch, sets the *DRQ3* flip flop to remove the DMA request and sets the *FLG* flip flop to inform the host. The host performs a read via the *IOIG* signal which enables the 8T13 backplane drivers, placing the 8 bits of data from the latch onto the host I/O backplane. The trailing edge of *IOIG* clocks the *DRQ3* flip flop, clearing it and causing it to issue another DMA request. The maximum delay from *IOIG* to the *STF1* strobe is that of a single DMA cycle — ie 3,2 μ s giving a maximum output rate of 300 Kbytes per second.

E.6 Flag and Control Logic

The flag and control logic section consists of four R-S flip flops and associated logic gates which were all required to implement the standard HP backplane control protocol. The logic takes 18 signals from the backplane, comprising 9 instruction related signals, 3 device select signals, 2 timing signals, 3 interrupt control signals, and one power-on preset signal. These signals are combined to produce control signals for the rest of the multiplexer logic and to synchronise the multiplexer interrupt to match the host processor's interrupt timing sequence. The following description will cover the signal implementation on the multiplexer only; for any further detail on the HP 1000 signal relationships, consult the HP 1000 Interface Manual. [31]

Backplane signal selection

The three logic signals *SCL*, *SCH* and *IOG* (subchannel low and high and I/O group) are 'anded' together to produce the signal *SEL* which will only be true when an I/O instruction for the correct subchannel is executed. This signal is used to gate in all the other I/O signals, these being:

- CRS Controlled Reset activated by a power-on preset, the front panel preset button and the instruction *CLC O*
- IOO These are respectively the backplane data output and input strobes which
- IOI occur for the *OTA*, *LIA* and *MIA* instructions.
- STC These signals set and clear the control flip flop and are activated by the
- CLC *STC* and *CLC* instructions.

STF These signals set and clear the flag and flag buffer flip flops. *STF* is
 CLF activated by the *STF* instruction only while *CLF* occurs for the *CLF*
 instruction as well as all I/O instructions which accept the (,C) option.

SFS These are the flag test instructions skip if flag set and skip if flag clear.
 SFC

The *IOO* and *IOI* signals become *IOOG* and *IOIG* after gating with *SEL* and these are used to strobe the backplane data into and out of the multiplexer as described in section E.5. The *CRSG* signal is used to reset the 8085 microprocessor and also clears the control flip flop (*CNTL*). The remaining signals operate on the control and flag flip flops and will be described separately in the following sections.

Control flip flop logic

The control flip flop (*CNTL*) of any HP 1000 interface is the major activating switch for the interface interrupt system. When set it enables the card to interrupt and break the priority chain whenever the flag buffer becomes set. When *CNTL* is clear all interrupt activity from the interface to the host becomes disabled and the host priority chain cannot be disturbed. The *STC* (set control) instruction is used to set the flip flop, but also performs the secondary function of acting as a single bit I/O strobe signal. Hence this signal is routed to the 8085's RST 7.5 interrupt input so that the host can inform the 8085 when it has completed a transaction. The state of the *CNTL* flip flop is also routed to the 8085's RST 5.5 input which is treated as a single bit input so that the state of the *CNTL* flip flop can be determined.

Flag flip flop logic

The flag buffer flip flop (*FBFF*) is set by the multiplexer whenever it wants to interrupt the host. Since this may occur after output or input, or simply when the 8085 requires attention there are three *STFn* signals coming from the rest of the multiplexer (see section E.2 and E.5). These are combined with the host generated power-on preset signal (*POPIO*) and the standard set flag signal *STFG* to set the flag buffer flip flop. Since this flip flop can be set at any time, a second flip flop, (*FLG*), is set by the combination of *FBFF* and timing signal *T2* which ensures *FLG* is always synchronised with *T2*.

The signal *FLG* is gated with backplane signals Interrupt Enable (*IEN*) and *CNTL* to create the signal *IENA* which breaks the host interrupt priority chain *PRH* to *PRL*. It is further gated with *FBFF*, the host timing signal *T5* and incoming priority signal *PRH* to set the interrupt request flip flop *IRQF*. This flip flop drives two backplane lines via the open emitter 8T13 drivers (*IRQB* and *FLGX*) which are used to interrupt the host and supply the interrupt source address.

The *IRQF* flip flop is cleared at the following *T2* and set again at *T5* time provided the interrupt has not been acknowledged or lost priority. This removal of *IRQF* each *T2* to *T5* time allows time for the priority chain to alter and settle, since the priority chain can only ever change state after a *FLG* gets set which is at *T2* time. When the host acknowledges the interrupt, it issues an interrupt acknowledge signal *IAK* which clears only *FBFF* and not *FLG*. This stops further interrupts, but does not release the priority chain. To release this, a clear flag (*CLF*) instruction must be issued.

The state of the flag flip flop can be tested by the host by using the *SFC* and *SFS* (skip if flag clear/set) instructions. These cause the signals *SFC* and *SFS* which are gated with \overline{FLG} and *FLG* respectively and then or'ed to produce the backplane signal 'skip on flag' (*SKF*) which will cause a host programme instruction skip if the flag was as required by the instruction.

The major communication between the host processor and the 8085 is via the *CNTL* and *FLG* flip flops as the 8085 can examine both flip flops, can set the *FLG* to interrupt the host and is itself interrupted by a host *STC* instruction.

E.7 Conclusion

This appendix has described the logic of the multiplexer interface by breaking it up into five major sections. The descriptions have concentrated most on those areas where the design has not followed standard laid down design rules, and has not attempted to cover those areas where little choice exists in the standard interconnection of components. For example, there is little point in detailing how to connect an 8085 CPU to the 8237 DMAC, since there is only one fixed way which is clearly spelt out in the specification sheet for the 8237 [32]. Timing diagrams have not been included except where necessary as most timing in the multiplexer is generated by software.

Tables E.2 and E.3 supply full details on the two edge connector pin-outs, while figure E.4 gives the full schematic diagram and Table E.4 a full parts list.

The schematic is all on one sheet as it was drawn on a CAD system which required full interconnections to be shown before it could do net list checking and automatic printed circuit board generation. The printed circuit board layout shown in figures E.5 and E.6 was produced by an auto-routing package, contains 31 metres of track, 1963 holes and took 45 minutes of CPU time to create with only 5 unconnected tracks out of 786. These 5 had to be manually routed which proved relatively straightforward.

Table E.2 48 pin Peripheral Connector Assignments

Signal (Track)	Pin no.		Signal (Comp)
COMMON	A	1	COMMON
DSR 3	B	2	CTS 3
CLKI 3	C	3	/RXD 3
CTS 2	D	4	DTR 3
DSR 2	E	5	CLKO 3
/RXD 2	F	6	RTS 3
DTR 2	H	7	CLKI 2
/TXD 3	J	8	RTS 2
CLKO 2	K	9	
	L	10	/TXD 2
	M	11	
	N	12	
CTS 1	P	13	
	R	14	DSR 1
	S	15	
CLKI 1	T	16	/RXD 1
RTS 1	U	17	DTR 1
	V	18	
/TXD 1	W	19	CLKO 1
/RXD 0	X	20	CLKI 0
/TXD 0	Y	21	RTS 0
CLKO 0	Z	22	DTR 0
DSR 0	AA	23	CTS 0
COMMON	BB	24	COMMON

Table E.3 86 way Backplane Connector Assignments

Signal (Comp)	Pin no		Signal (Track)
COMMON	1	2	COMMON
PRL	3	4	FLG
SFC	5	6	IRQ
CLF	7	8	IEN
STF	9	10	IAK
	11	12	SKF
CRS	13	14	SCM
IOG	15	16	SCL
POPIO	17	18	
SRQ	19	20	IOO
CLC	21	22	STC
PRH	23	24	IOI
SFS	25	26	IOB0
	27	28	
IOB1	29	30	IOB2
	31	32	T5
	33	34	
	35	36	
	37	38	
+5V	39	40	+5V
	41	42	IOB4
+12V	43	44	+12V
IOB3	45	46	ENF
-2V	47	48	-2V
	49	50	
IOB5	51	52	IOB7
IOB6	53	54	IOB8
IOB11	55	56	IOB9
IOB12	57	58	IOB10
	59	60	
IOB13	61	62	
	63	64	
IOB14	65	66	
	67	68	
-12V	69	70	-12V
	71	72	
	73	74	IOB15
	75	76	
	77	78	
	79	80	
	81	82	
	83	84	
COMMON	85	86	COMMON

Table E.4 Parts List for RMUX Multiplexer

QTY		COMPONENT
4	MC1488	Quad EIA RS232 line driver
4	MC1489	Quad EIA RS232 line receiver
4	P8251A	Intel USART serial interface
2	P8253	Intel programmable 3 channel timer
8	P2141-5	4Kx1 Static rams. (250ns)
2	74LS373	Octal latch
3	74LS374	Octal flip flop
1	P8259A	Programmable interrupt controller
1	P8085A	Intel Micro-processor
1	P8237	Intel DMA controller
1	74LS138	3 to 8 line decoder
1	74LS155	Dual 2 to 4 line decoder
6	8T13	Dual CTL line drivers
2	74LS122	Dual J-K flip flop
1	74LS04	Hex inverter
1	74LS279	Quad R-S flip flop
1	74LS32	Quad OR gate
1	7408	Quad TTL AND gate
3	74LS00	Quad NAND gate
1	74LS10	Triple NAND gate
1	74LS11	Triple AND gate
2	1000pf	Capacitors
3	0.1 μ F	
3	4.7 μ F	
3	10K Ω	Resistors 0.25W, 10%
1	150 Ω	
1	180 Ω	
6	2K2 Ω	8 pin SIP RESNET
1		6,144Mhz Crystal
2	24pin	Low profile sockets

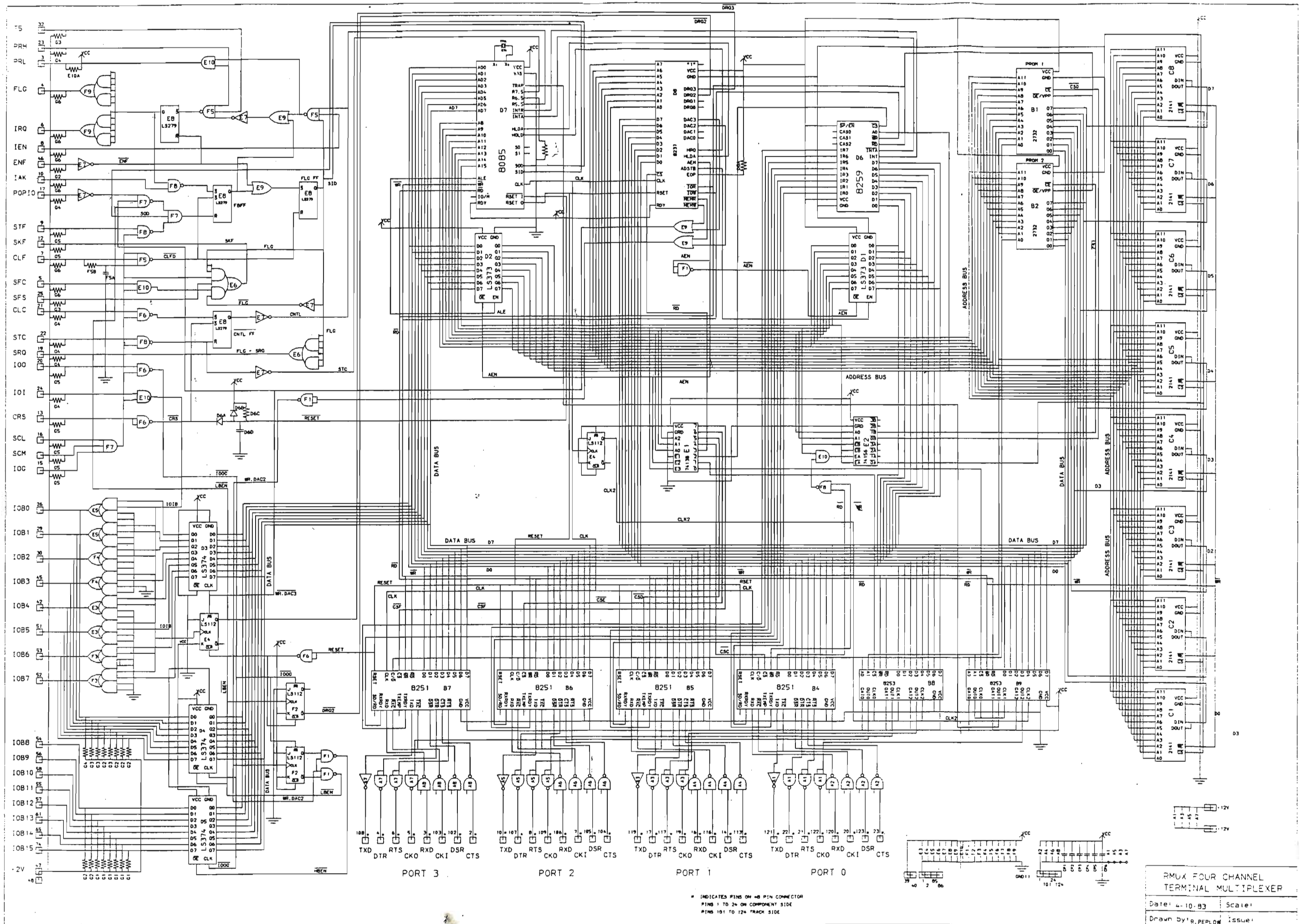


Figure E.4 RMUX Multiplexer Schematic

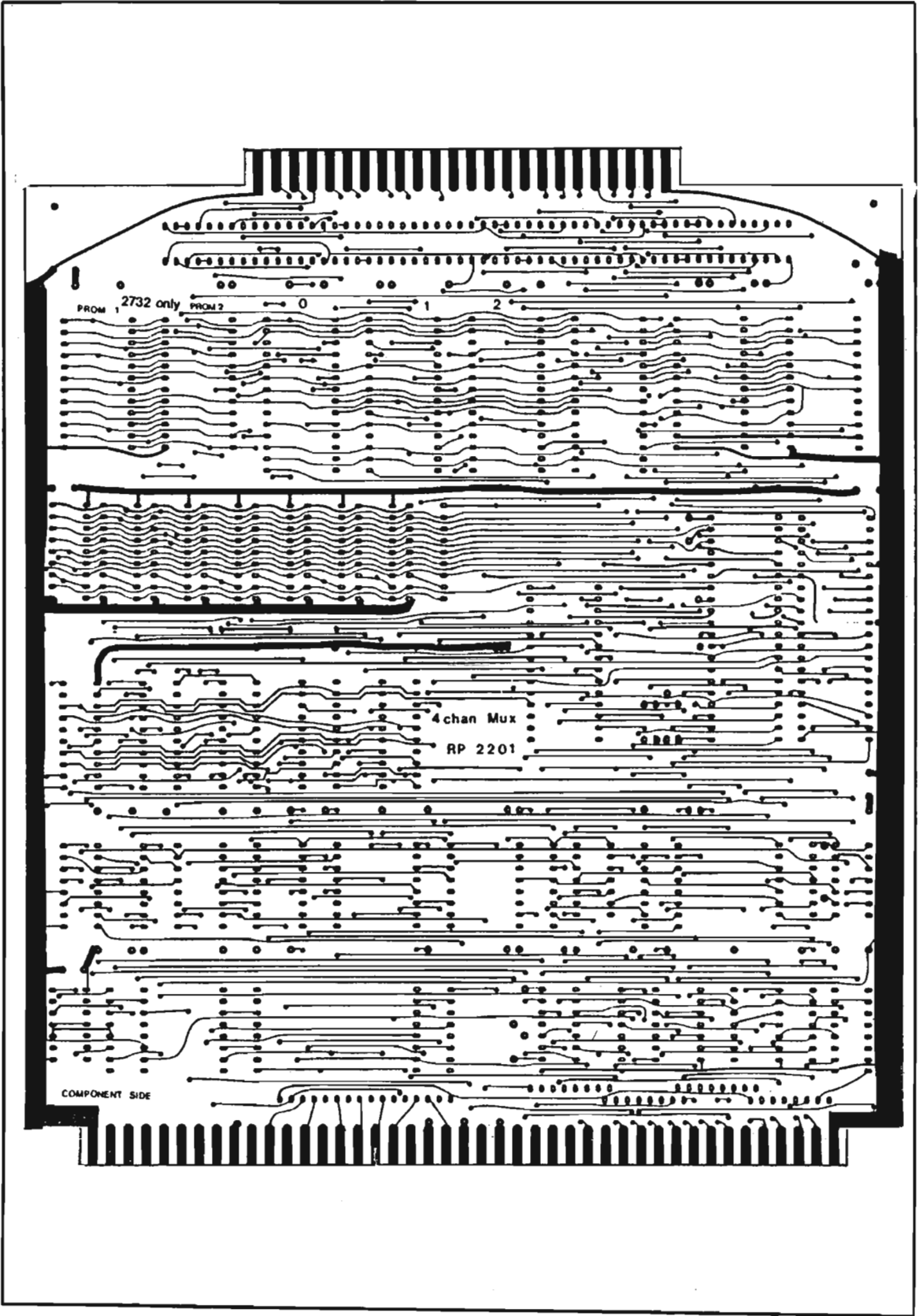


Figure E.5 Component Side Track Layout



Figure E.6 Solder Side Track Layout

Appendix F

MTX Software Description and User Manual

F.1 Introduction

MTX is a small multi-tasking executive written for an 8085 microprocessor with the specific objective of simplifying the task of developing software for the RMUX interface for HP 1000 series computers.

Although written for a custom application, *MTX* is general in nature and hence could be used for many different applications on many different sets of hardware.

The system is quite simple and offers only a very rudimentary I/O control system. However it does have a scheduler and dispatcher which allow multiple tasks to run concurrently. Facilities are supplied for time scheduling, resource locking, event synchronisation, and message handling, all of which are implemented as simple executive calls.

This appendix describes only the *MTX* executive itself and does not attempt to cover any application tasks at all. The description will first cover the operation of the system, detail the application of all the data structures and tables, and then cover the usage of the system in the form of a user manual. This user manual section will detail how to set up the system for any application and how to use all the executive calls.

F.2 System Layout – an Overview

The major objective of the system is to allow multiple independent tasks to run concurrently. To implement this, each task, which is defined here as one logically independent module of code, has it's own **Task Control Block (TCB)** to define it. The TCB is used to store all static and dynamic information pertaining to a task, and it is the TCB which is maintained by the system to allow the concurrency.

A task will execute continuously until it suspends itself waiting for an event, a resource or a message. All these three options of suspension are implemented using a data structure termed an **Exchange**. Each resource, event or message queue in the system uses its own exchange, and it is at these exchanges that *MTX* queue's waiting tasks.

The dispatcher scans all the exchanges searching for any exchange that contains a task that is ready to run. When one is found, the task is released and run until it again suspends itself against some exchange whereupon control returns to the dispatcher. Interrupts may interrupt any task, but no task may be suspended due

to an interrupt. This means that each task may only suspend itself on an exchange and can never be suspended by any other source. While this is a simplification of normal executive techniques, it does simplify both *MTX* and the application tasks. In the application system for which *MTX* was specifically written, all tasks are of equal importance, thus obviating the need for one task to usurp another. Figure F.1 illustrates how tasks are queued against the different forms of exchanges.

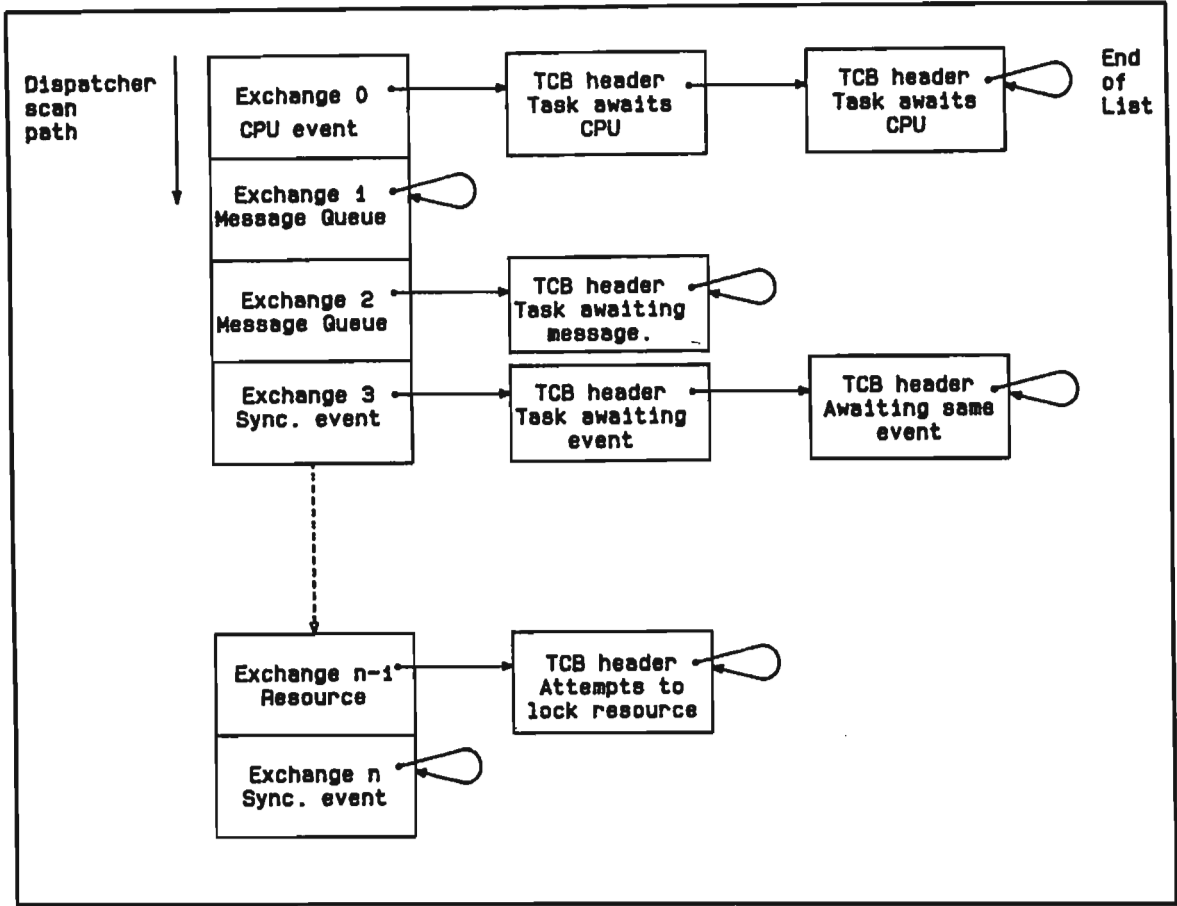


Figure F.1 Exchange Format

Messages in *MTX* consist of fixed length buffers which are initially held in a pool queue of empty buffers. Using standard *MTX* calls, tasks can fetch messages from designated message queues and can place messages onto any queue. There is no limit to the number of message buffers or message queues in the system other than the amount of RAM available.

All interrupts that arrive via the programmable interrupt controller (*PIC*) pass through a small *Central Interrupt Control module (CIC)* which saves all the registers and disables the *PIC* from giving any further interrupts. Control is then transferred via an interrupt vector to the correct handler for each vector. At completion the interrupt handler executes a *RET* (return) instruction, whereupon the *Central Interrupt Control* routine restores all the registers, enables the *PIC* and resumes processing. Note that *CIC* does not allow for a change in task as the result

of an interrupt.

A 10mS non-maskable interrupt on the *TRAP* input provides a time base which *MTX* uses to provide time limited waits and time delay schedule calls to other tasks. A time of day clock is not kept so all time calls may only specify a delay in 10mS steps. Any task pending against an exchange has the option of placing a time limit on it's wait and can then be notified upon return of the success or failure of its request.

F.3 MTX Data Structures and Table Formats

Interrupt Vector Table

A small table consisting of 8 two byte vectors is kept in RAM for use by the Central Interrupt Control (*CIC*). These vectors contain the addresses of the 8 interrupt handlers that correspond to the 8 interrupts from the PIC. These vectors can be modified by application tasks and by interrupt handlers to direct interrupt jumps anywhere. A RET (return) instruction will always restore control to *CIC* which will resume normal task processing from where it was interrupted. The dynamic re-allocation of interrupts is extremely useful where, for example, the significance of an interrupt relies upon the history of previous interrupts.

Exchange Table

The exchange table is the major data structure in *MTX* and consists of several 8 byte entries allocated contiguously in RAM. There is a maximum of 64 exchange entries allowed as there is only a six bit field allocated to take the exchange number in the TCB status field (see TCB format below). The exchange layout (see figure F.2) starts with a standard 5 byte list header which consists of a 2 byte list head pointer, a 2 byte list tail pointer to simplify list additions, plus a single byte list length counter. All lists and queues in *MTX* use the same format to simplify the queue handling routines. The last link pointer in a list is indicated by a zero pointer, and an empty list is indicated by a zero head pointer as well as a zero list length.

Byte 5 of an exchange entry is the exchange flag byte which when zero indicates that the exchange is free. When the exchange is busy, this byte is set to the TCB number of the task that currently owns the exchange. In the case of a resource type of exchange, only the task that locked the exchange can free it again, so the flag byte is checked on Release—Resource calls to ensure that the current task is in fact the exchange owner.

Byte 6 of the exchange entry is an indicator byte to indicate whether the exchange is a resource type or an event type of exchange. The two major differences are that a resource type can only be released by the calling task that locked it, and it initially starts with its flag in the free or zero state. An event type of exchange can be set or cleared by any task, since it is used mainly for inter-task synchronisation, and it must initialise to the busy state so that the first call to 'Wait for the event' will suspend the caller until the first 'set event' call occurs. The indicator byte is non-zero for events and zero for resources which simplifies initialisation since the indicator is merely copied into the flag byte.

Byte 7 holds the exchange number which is used in the status field of the TCB whenever a TCB is queued against the exchange (see TCB exchange format below). Although this number could be calculated dynamically instead of being stored, this method is slightly faster and an 8 byte exchange entry is easier and quicker to derive from its entry number than a 7 byte entry would be.

0	2 byte Head link
1	pointer
2	2 byte Tail link
3	pointer
4	Queue Length
5	Flag byte (0=free)
6	Indicator (0=resource)
7	Exchange Number (from 0)

Figure F.2 Format of an Exchange Table Entry

Task control blocks

The task control block or TCB is a 72 byte data structure used to store all static and dynamic information about a task. Each task has its own TCB and all TCB's are placed contiguously in RAM at the time of assembly, hence TCBs cannot be dynamically created or removed. The TCB consists of a 14 byte information header (see figure F.3) and a 58 byte stack area for the task to use while executing.

The first two bytes (0 and 1) comprise the link address field, which stores the link pointer when a TCB is queued against an exchange. This pointer is set to zero when the TCB is the last item on any queue, otherwise it points to the next TCB on the queue.

Bytes 2 and 3 hold the *stack pointer* that was current when the task suspended itself. At initialisation it is set to the top of stack in the TCB — ie., the end of the TCB. By giving each task it's own stack in it's TCB and saving the stack pointer when the task suspends, a task may suspend with any data it likes on the stack, or from any level of subroutine nesting.

Byte 4 is the *Task Status Byte* which has bit 7 set when the task is executing. If the task is not the currently executing task, then it must be suspended on some exchange, and so bits 5 through 0 of the task status byte are used to store the exchange number. Since the TCB could also be suspended in the time list, bit 6 of the status field is set when the TCB is in the time list.

Byte 5 contains the *Task Identity* number with each TCB being numbered sequentially from one. This value is static and may not be altered.

Bytes 6 to 9 are currently unused.

Bytes 10 and 11 store the *Time List Link Pointer* which links together all the TCB's in the time list. Note that the TCB's are all linked on bytes 10 and 11 in the time list queue and that this link pointer does not point to the top of the TCB.

Bytes 12 and 13 store the 16 bit *Time Delay* value which is the number of 10ms intervals to go before the task times out and is re-scheduled. This value stored least significant byte first is decremented every 10ms by the time base processing task.

0-1	2 byte exchange list linkage pointer
2-3	2 byte stack pointer save area
4	TCB Status B7=executing, B6=time list, B5-0=exchange number
5	TCB Identity number
6-9	Not used
10-11	Time list linkage pointer
12-13	Time list delay value (10ms)
14-71	58 byte stack area 32 bytes for user task 26 bytes for Interrupts

Figure F.3 Task Control Block Format (TCB)

The remaining 58 bytes (bytes 14 to 71) are used as stack area for the task. However since a separate stack for interrupts does not exist, each user stack must leave 26 bytes for the interrupt routines, which leaves 32 bytes of stack for the user task itself. The 26 bytes of interrupt space are allocated as 4 bytes to the non-maskable time base interrupt, 8 bytes for the RST 7.5 interrupt from the host CPU (in the case of the RMUX application), and 14 bytes allowed for a single level of user interrupt.

As a TCB is a fairly long entry, its size could be altered for a specific application should it consume too much RAM, providing the stack area is not required to be so large. However the library routine ^GTCBAD (see next section) which gets the address of a TCB given its number, relies on all TCB's being contiguous and 72 bytes long. So if the TCB length is altered then the routine ^GTCBAD should be altered accordingly.

Message Queue Header Table

This table consists of 6 word queue header entries with the queue zero header being reserved for the pool queue which is where all blank memory buffers are initially stored.

Bytes 0 through 4 form the standard list header format (see figure F.4) with a 2 byte head pointer, a 2 byte tail pointer and a single byte length counter.

Byte 5 holds the number of an exchange which is associated with the message queue header. Thus if a Task requests a message from a queue and there is no message available, the task will be suspended against the corresponding event exchange. When a message is added to the queue, the event will be set, hence releasing the caller.

0	2 byte list head
1	pointer
2	2 byte list tail
3	pointer
4	list length counter
5	queue exchange number

Figure F.4 Message Queue Header Table

Memory Buffer format

Memory buffers in the current RMUX version of *MTX* are 272 bytes long, but this length is variable depending on application. The buffers require a 5 byte header used by *MTX*, and then the remaining space can be application determined. Memory buffers are defined as starting at byte 3 which leaves the normal list linkage pointer at address zero.

Bytes -3 and -2 comprise the list linkage pointer which ties all the memory buffers together in a permanent list. This list is used by *MTX* to recover buffers from killed tasks and generally to keep track of all the buffers. This system also enables buffers to be allocated in a non-contiguous manner providing they are all linked together at startup.

Byte -1 holds the buffer status which reflects buffer ownership. If bit 7 is a 1, then bits 5—0 hold the message queue number while if bit 7 is a 0 then bits 5—0 hold the TCB number of the task that last requested the buffer.

Bytes 0 and 1 hold the normal queue linkage for linking messages onto the various message queues as defined above.

The remaining bytes in this buffer are user defined as *MTX* does not alter them at all. Furthermore, buffers may be of different lengths, for apart from the startup routine, *MTX* does not maintain any information about buffer lengths.

-3	2 byte linkage holding all buffers
-2	together in a list
-1	Buffer owner status (B7+Q# or B7+Task#)
0	2 byte linkage to hold buffers on
1	individual message queues
2	User available data space unaffected by the system. May be any length but is not dynamically alterable.

Figure F.5 Memory Buffer Format

F.4 MTX User Callable Functions

The principle function of *MTX* is to supply facilities for the handling of resources, events, messages and tasks, and hence all the user callable functions apply to these four facilities. This section will first of all cover how each of these facilities is used and then will cover the user calls.

Resource numbers

A resource number is merely an exchange number which two or more co-operating tasks can use to protect some shared facility, be it data or a peripheral. Any task wishing to access a shared facility should execute a GET RESOURCE call on the resource number associated with that facility. *MTX* will allow the task to continue if the resource number is free, and will lock the number to the calling task. Should the resource already be locked then the calling task will be suspended until the resource becomes free. Once a task has completed operating on the shared facility, it should execute a RELEASE RESOURCE call on the same resource number which will free the resource number and allow any other task awaiting the resource number to continue. Thus a resource number is used to ensure mutual exclusion of co-operating tasks when dealing with a shared facility.

A resource number is initially unlocked at power-up time, and can only be released by the task that locked it. Should an invalid release call be made, no action is taken and the resource stays locked.

Events

An event is a means by which co-operating tasks can pass signals to one another to synchronise their activities. A call to WAIT EVENT by a task will cause the task to be suspended against that event until some other task performs a call to SET EVENT on the same event number. This would then release the waiting task and clear the event in readiness for the next set of calls. Should the SET EVENT call be executed before the WAIT EVENT call then the WAIT EVENT caller would not be suspended. Events are all initialised to the clear state, but a CLEAR EVENT call may be used to ensure a clear event if it's history can be ignored.

Several WAIT EVENT calls may be performed by different tasks which will cause all the tasks to be queued against the event. When a SET EVENT occurs, only the first task in the queue will be released; all the others remaining queued for subsequent SET EVENT calls.

Messages

Messages in *MTX* are all stored in memory buffers which are of fixed length, determined at the time the system is assembled. A task wishing a message will execute a call to GET BUFFER, giving a message queue number. Should a message be available on that queue, it's address will be returned immediately, otherwise the task will be suspended until a message becomes available. The first two bytes of the message buffer are reserved for the list linkage, but the remaining space can be used as the task sees fit.

To pass a message, a task will execute a call to PUT BUFFER and supply the buffer address and a queue number. This will place the message buffer on the message queue specified, ready for some other task to fetch. Since *MTX* does not keep track of message buffer lengths, the user task must know the length and not exceed it otherwise it could write over other buffers.

Tasks

A task is essentially a self contained block of code which performs some discrete function, independently of other tasks. Where two or more tasks share some function or resource, they should use resource number or event calls to secure the resource and synchronise themselves. Thus a task should be able to run completely independently of any other task, and where this cannot occur, system calls should be used to ensure safe and error free co-operation. Tasks may be scheduled, time scheduled, aborted or halted via various system calls described below. Since *MTX* does not rigidly supervise the I/O system, each task must contain a KILL section of code which *MTX* will execute whenever it aborts or halts a task. This code should tidy up after a task, particularly in the area of I/O and interrupts. *MTX* will release resources locked by the aborted task and return any memory buffers that it owns to the pool.

Function calls

^GTRESC:	This call gets or locks a resource number to the caller, or if the resource number is locked to another task then the calling task is suspended until the resource number becomes available.
INPUT DATA:	A register holds resource number
OUTPUT DATA:	Nil
REGISTERS USED:	All
STACK USED:	6 bytes
^TGTRSC:	Timed get resource call which, should the resource not be available, will only suspend the caller for a specified time for the resource.
INPUT DATA:	A register holds resource number. BC = 16 bit time delay in 10mS increments
OUTPUT DATA:	Z bit set for a timeout return. Z bit clear for resources locked
REGISTERS USED:	All
STACK USED:	16 bytes
^RLRESC:	Release a resource number. The resource number must belong to the calling task or else no action results.
INPUT DATA:	A register contains resource number
OUTPUT DATA:	Nil
REGISTERS USED:	HL, DE and PSW (BC preserved)
STACK USED:	4 bytes

- ^WTEVNT:** Causes the calling task to suspend against an event until the event gets set by another task, unless the event is already set in which case, no suspension occurs. The event is cleared once the task is released.
- INPUT DATA:** A register contains event number
- OUTPUT DATA:** Nil
- REGISTERS USED:** HL, DE, and PSW (BC preserved)
- STACK USED:** 6 bytes
-
- ^TWTEVN:** Timed wait for event call which will only allow the task to be suspended for at most the specified time while waiting for an event.
- INPUT DATA:** A register contains event number. BC = 16 bit time delay in 10mS increments
- OUTPUT DATA:** Z bit set for timeout return. Z bit clear if event occurred OK
- REGISTERS USED:** All
- STACK USED:** 16 bytes
-
- ^STEVNT:** Set an event. May be executed by any task.
- INPUT DATA:** A register contains event number
- OUTPUT DATA:** Z bit set indicates invalid event number
- REGISTERS USED:** All
- STACK USED:** 4 bytes
-
- ^CLEVNT:** Clear an event. Normally only used to ensure an initial startup condition.
- USAGE:** (Identical to ^STEVNT above)
-
- ^GETBUF:** Gets a message buffer from a specified message queue. If no buffer is available, then the calling task is suspended until one does become available on the queue.
- INPUT DATA:** A register contains message queue number
- OUTPUT DATA:** DE = the start address of the buffer
- REGISTERS USED:** All
- STACK USED:** 12 bytes
-
- ^TRYBUF:** Gets a message buffer from a specified message queue. If no buffer is available then an immediate return is made to the caller with the Z-bit set, else a buffer address is returned with the Z-bit clear. This call does not suspend the caller at all.
- INPUT DATA:** A register holds the message queue number.
- OUTPUT DATA:** DE holds address of buffer start else Z bit set indicates no buffer available
- REGISTERS USED:** All
- STACK USED:** 12 bytes
-
- ^PUTBUF:** Puts a message buffer onto a specified message queue.
- INPUT DATA:** A holds destination message queue number. DE holds address of the start of the buffer
- OUTPUT DATA:** Nil
- REGISTERS USED:** All
- STACK USED:** 10 bytes

^STACTV: Sets a specified task active, I.E. schedules a task to run, optionally after a given time delay. The task must be dormant and not in the time list, otherwise the STACTV call has no effect.

INPUT DATA: A register contains task TCB #. DE holds time delay before task will start.

OUTPUT DATA: Nil

REGISTERS USED: All

STACK USED: 10 bytes

^STDORM: Sets a specified task dormant immediately. This is an Abort request which kills a given task and releases all its system resources.

INPUT DATA: A register contains the task TCB number.

OUTPUT DATA: Nil

REGISTERS USED: All

STACK USED: 20 bytes

^STOP: System call to allow a task to terminate itself. This call does not release any system resources owned by the task, or perform any clear up, hence all cleaning up must be done by the task prior to the stop call.

REGISTER AND STACK USAGE: Immaterial

F.5 System Library functions not directly user callable

MTX was written on a modular basis and hence contains several library routines which may be of use to some user routines. Although these functions are not system calls, they may be used as general subroutines for user programs on a library basis. All are re-usable so may be called by any task at any time.

^SETIMT: Links the current task into the timelist and sets its time delay value to that contained in the DE register. Uses all registers and 10 bytes of stack.

^CLTIMT: Removes the current task from the timelist. Saves BC register only and uses 8 bytes of stack.

^KLTIMT: Removes the TCB whose address is specified in the HL register pair from the timelist. Saves BC register only and uses 8 bytes of stack.

^RLMEMR: Releases all message buffers owned by the task whose TCB number is specified in the A register, and places them back on the pool queue. Uses all registers and 16 bytes of stack.

^LINKIN: Links the object whose address is in DE onto the end of a list whose header address is in HL. The list header must be in the standard 5 byte form specified in section F.3. Alters only the PSW register and uses 8 bytes of stack.

^UNLINK: Unlinks the object specified by the address in DE from the list whose address is in HL. The list header must be in the standard 5 byte form specified in section F.3. The Z bit is returned set if the object is not found in the list. Alters only the PSW register and uses 6 bytes of stack.

^GETOFQ: Unlinks the first object from the top of the list whose header address is in HL and returns the object address in DE. If an empty list is encountered then DE is returned with zero in it. Alters only the PSW and uses 6 bytes of stack.

^GTMQAD: Returns in HL the message queue header address that corresponds to the queue number in the A register. Alters HL and PSW and uses 2 bytes of stack.

^NXTLNK: Loads HL with the contents of the address pointed to by HL. Alters HL and PSW and uses 2 bytes of stack.

^GTCBAD: Returns in HL the TCB address corresponding to the task number in the A register. Alters all registers except BC and uses 2 bytes of stack.

^BYPASS: List removal routine that stores the contents of the address pointed to by DE into the address pointed to by HL, ie. $(HL) := (DE)$. Alters A register, increments HL by one and uses 2 bytes of stack.

^CHKHDL: Checks if DE equals the memory pair pointed to by HL and sets the Z bit if they are equal, ie. checks if $DE = (HL)$. Alters A register and uses 2 bytes of stack.

F.6 Dispatcher and Scheduler

All calls to *MTX* that result in a task being suspended will call one of the routines **^QUEME** or **^TQUEME** to perform the actual suspension and link the task against the specified exchange. **^TQUEME** is a subsidiary routine that adds the caller into the time list before calling **^QUEME**, and removes it from the time list once the task has been released again.

^QUEME is the major suspending task which saves the user stack pointer into the task's TCB, updates the task's TCB status byte and links the TCB onto the correct exchange. The return address of the routine that called **^QUEME** will be the last value on the stack, so that it is a simple matter to return to the correct point at a later stage. **^QUEME** then changes the stack to the dispatcher stack and passes control to the dispatcher (**^DSPTCH**).

^DSPTCH starts at the top of the exchange table and examines each entry in turn, searching for an exchange which has a task queued against it (head link non-zero) and which has a zero flag byte. Should no ready exchange be found, **^DSPTCH** starts again at the top of the table and tries again, looping continuously until some exchange becomes ready. When an exchange becomes ready, the TCB on the top of the exchange queue is released, the exchange is marked as being busy to that task, the task stack is re-instated and then control is passed to the task. This will cause the task to continue executing directly after the point at which it called **^QUEME**.

`^QUEME` and `^DSPTCH` maintain two variables in RAM called `^CTCBAD` and `^CTCBID` which contain the address and ID number of the TCB of the currently executing task. These variables are set to zero and -1 respectively, when no task is executing and `^DSPTCH` is looping.

Calling specifications are:

`^QUEME:`

INPUT DATA:	DE points to the exchange to be queued on.
OUTPUT DATA:	see <code>^DSPTCH</code> which is the output section
REGISTERS USED:	All
STACK USED:	4 bytes

`^DSPTCH:`

INPUT DATA:	Nil, as this route is not directly callable
OUTPUT DATA:	HL points to the released exchange flag byte
REGISTERS USED:	All
STACK USED:	4 bytes of USER stack (see <code>^QUEME</code>)

`^TQUEME:`

INPUT DATA:	DE points to the exchange to be queued on. BC holds 16bit time delay in 10mS steps
OUTPUT DATA:	Z bit set if timeout exit occurred
REGISTERS USED:	All
STACK USED:	14 bytes

F.7 System Start-up Routine

An initialisation routine `^STRTUP` is the first code to execute after a reset occurs, and it is responsible for setting up all the tables in RAM and generally performing all the required initialisation functions listed below.

- * Clear entire 4K RAM to zero .
- * Create the message queue header table and initialise each entry to reflect its corresponding exchange number. These exchange numbers are obtained from the ROM table `^MEXTBL` which consists of one single byte exchange number for each message queue header. The variables `^MEM_Q_0` and `^MAX_MEMQ#` are used to hold the start address of the table and the last entry number respectively.
- * Create the exchange table using starting address `^EXCHG_0` and last entry number `^MAX_EXCHG#`. Each entry is initialised with its exchange number, and has its flag and indicator bytes set to indicate whether it is an event or resource type entry. This information is recovered from the ROM table `^RSCTBL` which holds the indicator byte for each exchange.

- * Initialize all message buffers by linking them onto the master list (header: `^MEMLST`) via their master linkage (bytes -3 and 2) and also onto the pool queue via their normal linkage. The variables `^MEM_BUF_0`, `^MEM_BUFLen`, and `^MAX_MBUF#` are used to define the start address of the message buffers, their length, and the number of the last buffer respectively.
- * Create the TCB's and initialise them by setting their status to the number of the DORMANT exchange to indicate their dormant status. The variables `^TCB_01`, `^TCB_LEN`, and `^MAX_TCB#` are used to define the start address, length and number of TCB's respectively. The dormant exchange is merely the last exchange which since it is never checked by the dispatcher is used to hold all inactive TCB's.
- * Schedule task 1, the timeout processor, and task 2, the first user application task, both with zero delay.
- * Initialize the time base generator to run at 100Hz.
- * Set up the programmable Interrupt controller to reflect the correct interrupt vector address and mode of operation.
- * Exit to the dispatcher.

This start-up routine could be added to to cause it to perform other start-up tasks, but all user initialisation should be done in the first user task (task 2) which should also be the one to start up any other tasks required for any application.

F.8 Time Base Generator and Time List Handling

A small non-maskable interrupt driven routine (`^CLOCK`) is included with the system which executes every 10mS in response to the 10mS time base generator. This routine uses 4 bytes of user stack, and merely sets an event to release the timeout processor TOUT (always TASK 1). The event is `^TIMER` and can be placed anywhere in the exchange table depending upon the priority it requires. The priority is due to the exchange table being scanned from the top, giving low numbered entries a higher priority.

TOUT is an application task which is used to process the time list. It suspends itself against the event `^TIMER`, and each time the event gets set (by `^CLOCK`), TOUT runs down the time list decrementing each TCB's time delay value. When a time delay that decrements to zero is found, the TCB is unlinked from the time list, and from its exchange, and is queued against the CPU exchange (`^CPU_EXCHG1`) which is always the first exchange and is never locked. This will cause the timed out task to execute when the dispatcher next runs. When all the TCB's in the time list have been checked, TOUT re-suspends itself against the `^TIMER` event.

F.9 Adding a Task to the System

To add a new task into the system, several tables may need alteration to reflect the requirements of the new task depending on what facilities the task requires. The following section describes all the changes on a step by step basis.

- * Write the task itself using calls to any of the system routines described in section F.4. Add a KILL subroutine which will clean up any resources that would be left open by the main task if it were to be aborted at any point. Once these are written the source code file can be included in the assembly by use of the assembler MERG instruction (see appendix I).
- * Allocate a TCB for the task, giving the task a TCB number. If this requires an extra TCB then indicate this by altering the variable ^MAX_TCB# to reflect the highest TCB number in the system. This is all that is necessary to add a TCB.
- * Inform the system of the task's start address and kill routine address by adding the following code to the start of the task.

```

ORG  TASKNO-1* 6+^JUMP_TABLE
JMP  <task start address>
JMP  <task kill routine>
RSEG ^ROM1

```

..

This fills the task start address jump table which is used by the ^STACTV routine to start up a task. TASKNO should contain the actual number of the task being added and is normally assigned by using the assembler SET instruction.

- * If any *new exchanges* are required, add them to the exchange list and update the variable ^MAX_EXCHG# to reflect the highest exchange number in the system. The table ^RSCTBL in the same file must also be updated to indicate whether the exchange is an event or a resource. Should more than 32 exchanges be used then ^RSCTBL should be moved from address 4 to elsewhere in ROM by replacing the ORG 4H statement by an RSEG ^ROM1 statement.
- * If any *new message queue* headers are required then add them into the memory queue table and update the variable ^MAX_MEMQ# to match the number of the last message queue header. An extra exchange will also have to be added for this queue and so proceed as given above. Finally update the message queue exchange table ^MEXTBL by adding the new queues exchange number into the table.

- * If any more *message buffers* are required, or if the buffer length is to change, merely alter the variables `^MAX_MBUF#` or `^MEM_BUFLLEN` respectively.

Assemble the master file `&MASTR` (see figure F.6) which should merge in all the system files and routines, and then check the variables `^ROM1ND` and `^RAM1ND` which will give the usage of ROM and RAM, these being the next available addresses in each segment respectively. Ensure that these figures do not overflow the ROM or RAM available.

Figure F.6 gives a sample control file `&MASTR` which was used in the RMUX interface to install the system and four copies of the port driver module which contains three separate tasks.


```

8085,LE      "Multitasking monitor  R.Peplow  <850104.1531>"
^RAM1:      NSEG 4000H
^RAM_BEG:   EQU .
^RAM_END:   EQU .+4095
^ROM1:      NSEG 0040H      ;Load system into first rom
;              ;directly after interrupt vectors
                MERG &RXTBL  ;First the table definition module
                MERG &RXSYS  ;Then all the system routines
                MERG &RXTIM  ;The time list handler task
                MERG &RXHST  ;The host interaction tasks
;
; Now to load the 4 port driver tasks setting port parameters for each
;
^PORT#:      SET 1
^PORT_DATA:  SET ^PORT1DATA
^COUNTER:    SET ^COUNTER_1
^COUNT_MODE: SET ^CNTR1_MODE
^C_MODE_WRD: SET ^C1_MODEWRD
^TX_BUF_Q#:  SET ^MEM_Q#_TX1
^RX_BUF_Q#:  SET ^MEM_Q#_RX1
^PORT_RESRC: SET ^PORT1RESC
^FLAG_1#:    SET ^TX1_EVENT      ;both TX & RX use same sync event
^FLAG_2#:    SET ^PORT1_BRK#     ;port 1 break task sync exchange #
^RX_INT_VEC: SET ^RX1VEC
^TX_INT_VEC: SET ^TX1VEC
^TX_MASK:    SET 10H             ;this bit on the 8259 stops tx intr
                MERG &RX801
;
^PORT#:      SET 2
^PORT_DATA:  SET ^PORT2DATA
^COUNTER:    SET ^COUNTER_2
^COUNT_MODE: SET ^CNTR2_MODE
^C_MODE_WRD: SET ^C2_MODEWRD
^TX_BUF_Q#:  SET ^MEM_Q#_TX2
^RX_BUF_Q#:  SET ^MEM_Q#_RX2
^PORT_RESRC: SET ^PORT2RESC
^FLAG_1#:    SET ^TX2_EVENT      ;both TX & RX use same sync event
^FLAG_2#:    SET ^PORT2_BRK#     ;port 2 break task sync exchange #
^RX_INT_VEC: SET ^RX2VEC
^TX_INT_VEC: SET ^TX2VEC
^TX_MASK:    SET 20H             ;this bit on the 8259 stops tx intr
                MERG &RX801
;
;.....repeat the same procedure for ports 3 and 4
;
SCRATCH:     NSEG OFFFHH          ;this will close all segments
ROMIND:      EQU ^ROM1            ;now report all the end address's
RAMIND:      EQU ^RAM1
END

```

Figure F.6 Sample Master Control File

Appendix G

RMUX System Software

G.1 Introduction

This appendix describes the application software written for the RMUX multiplexer. The initial section covers the system in very general terms, shows how messages move around, and how all the tasks interact.

The next section describes how the host interacts with the multiplexer and details how the two host interaction tasks in the multiplexer operate. There is very little description of the HP RTE IV software since this is described in appendix H.

The last section describes how the multiplexer interacts with the terminal and goes into some detail as to how the terminal handling tasks function.

G.2 Overall System Operation

The function of the multiplexer is to take messages from the HP1000 host, distribute them to the relevant message handlers to operate on and return completed messages to the host. Messages in this context may be transmit requests, where a line of text is to be output from the host to some terminal; receive requests, where a line of input from a terminal is to be placed in the message and returned to the host; or they may be control requests informing the multiplexer to perform some function on a terminal. (E.G. space lines). The common factor in all is that the message when completed is returned to the host in acknowledgement.

The task block diagram in figure G.1 shows the various modules that handle these messages and how the messages move throughout the system. The full complement of tasks is only shown for one port, as the other three ports have identical layouts.

All messages originate in the host as some request, and are passed to the multiplexer via a multiplexer DMA channel, 16 bits at a time. When a complete message has been placed in the multiplexer memory the host informs the `^FROM_HOST` task (figure G.1) in the multiplexer by issuing an *STC* instruction. `^FROM_HOST` checks the message and determines its destination to be one of eight message queues depending upon which port the message was for and whether it was a receive request or not. The message then passes via these queues to one of the eight port handling tasks for processing.

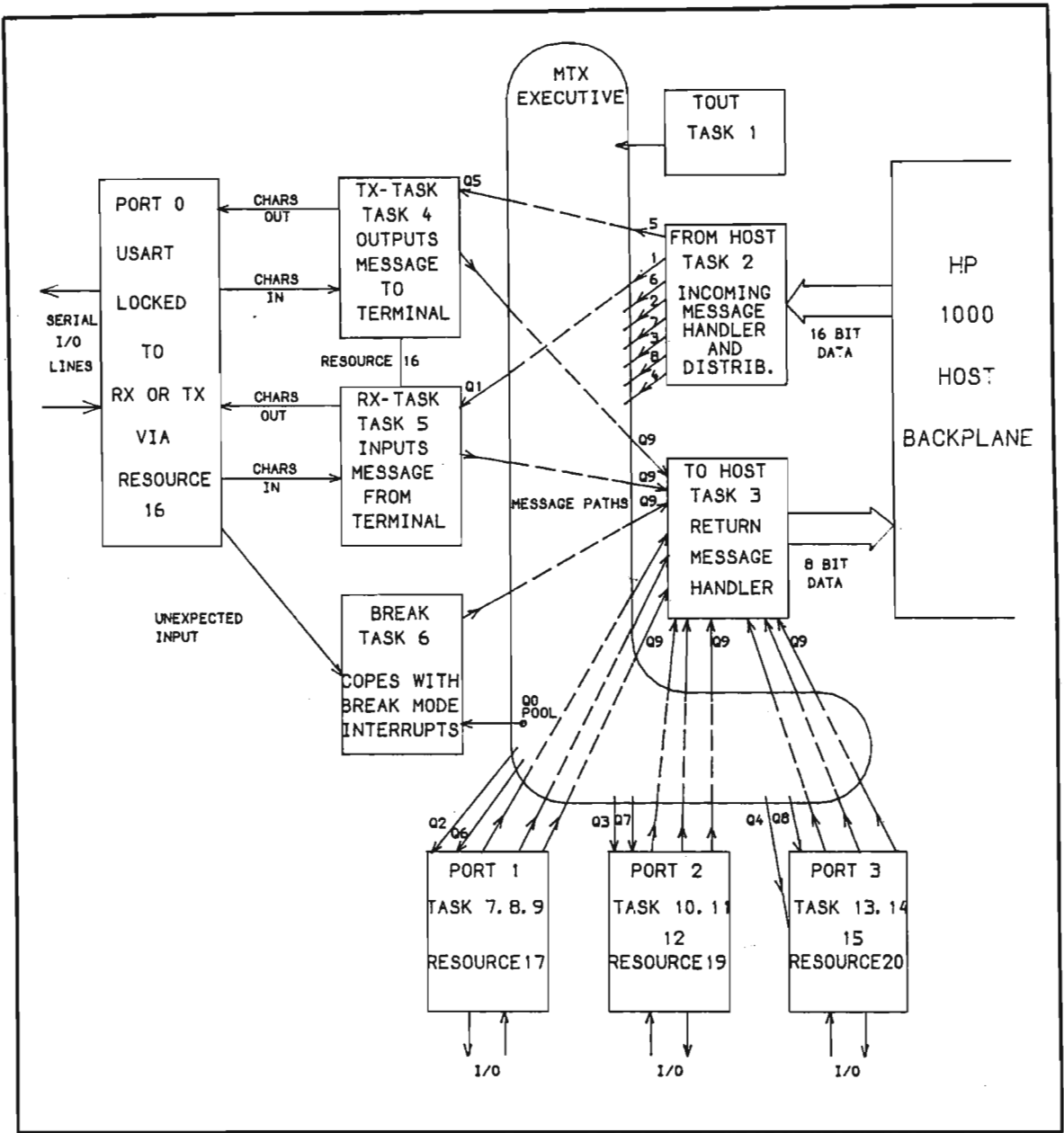


Figure G.1 RMUX Multiplexer Message Flow Diagram

When a port task has a message it locks the resource number associated with its port in order to eliminate any possibility of simultaneous transmission and reception on one terminal. Once locked, it outputs (or inputs) data as specified by the request until the request is complete. The task then releases the port resource and passes the message to the ^TO_HOST task via queue 9. ^TO_HOST sets up the DMA channel to the host to output the message and interrupts the host which then reads the message from the multiplexer. When complete the host interrupts the multiplexer to inform it that the message has been accepted. The host can then determine the destination for the message and release the originating program.

Should a character be received from a terminal when no receive request is pending it will be ignored unless it is a '*space*' or a '*CTRL X*'. These two characters are used to attract the host's attention which is done by '*BREAK*', the task to which all unexpected input characters are diverted. To attract the host's attention *BREAK* fetches a blank buffer from the pool queue, sets up the header to indicate which port requires attention and sends it to the host.

One requirement of modular programming is that each module should be independent and self-contained with all interaction outside the module being confined to a well defined set of calls. The structure shown in figure G.1 attempts to ensure this by confining all task interaction to message transfers or resource calls.

G.3 Host-Multiplexer Interaction

All messages from the host to the multiplexer are handled by the task *^FROM_HOST* and those from the multiplexer to the host by task *^TO_HOST*. Transfers in both directions are performed under DMA in the multiplexer which enables rapid transfers to occur without requiring the multiplexer software to synchronise with the host software. However this results in the data transfer being invisible to the micro-processor and so the host sends an interrupt (*RST7.5*) to the multiplexer micro-processor whenever it completes a transaction, either input or output. This single interrupt for two different transactions was forced by the lack of suitable signals from the host. The interrupt routine in the multiplexer is *^STC_INT*, and it determines which transaction the host is signalling completion of by examining the two DMA channel word count registers. The channel that has reduced by at least one message header (12 bytes) is the channel that the host last used and hence *^STC_INT* can inform either *^FROM_HOST* or *^TO_HOST* by either setting the event *^FROM_SYNC#* or the event *^TO_SYNC#* respectively.

^FROM_HOST

Since this task must always be ready to receive a request from the host, *^FROM_HOST* attempts to always maintain a reserve of 5 empty buffers. To speed up the use of these buffers, they are not kept in a queue or linked list but are stored in 5 individual data structures termed *POTS*. A pot is a 3 byte record which holds a single status byte and a two byte buffer address. The pot status may be *FREE* if it has an blank buffer in it, *FULL* if it has a full buffer in it, *EMPTY* if it contains no buffer and *BUSY* if it contains a buffer which is currently being filled. *^FROM_HOST* initially tries to create 5 *free* pots setting the buffer from the first *free* pot to be filled by the DMA controller and marking the pot *busy*. This done it waits against the event *^FROM_SYNC#* for the buffer to be filled via DMA.

When the host has filled the buffer it interrupts the multiplexer using an STC instruction. This causes the 8085 control to transfer to the interrupt routine ^STC_INT (figure G.2) which checks the state of the two DMAC word count registers. Should it find the ^FROM_HOST channel altered (DMAC channel 2) it will check that the current pot is *busy* and if so set it to *full*. It then checks each pot in turn until it finds a *free* one, assigns it to the dma controller to be filled, marks the pot *busy* and finally sets the event ^FROM_SYNC# to release FROM_HOST.

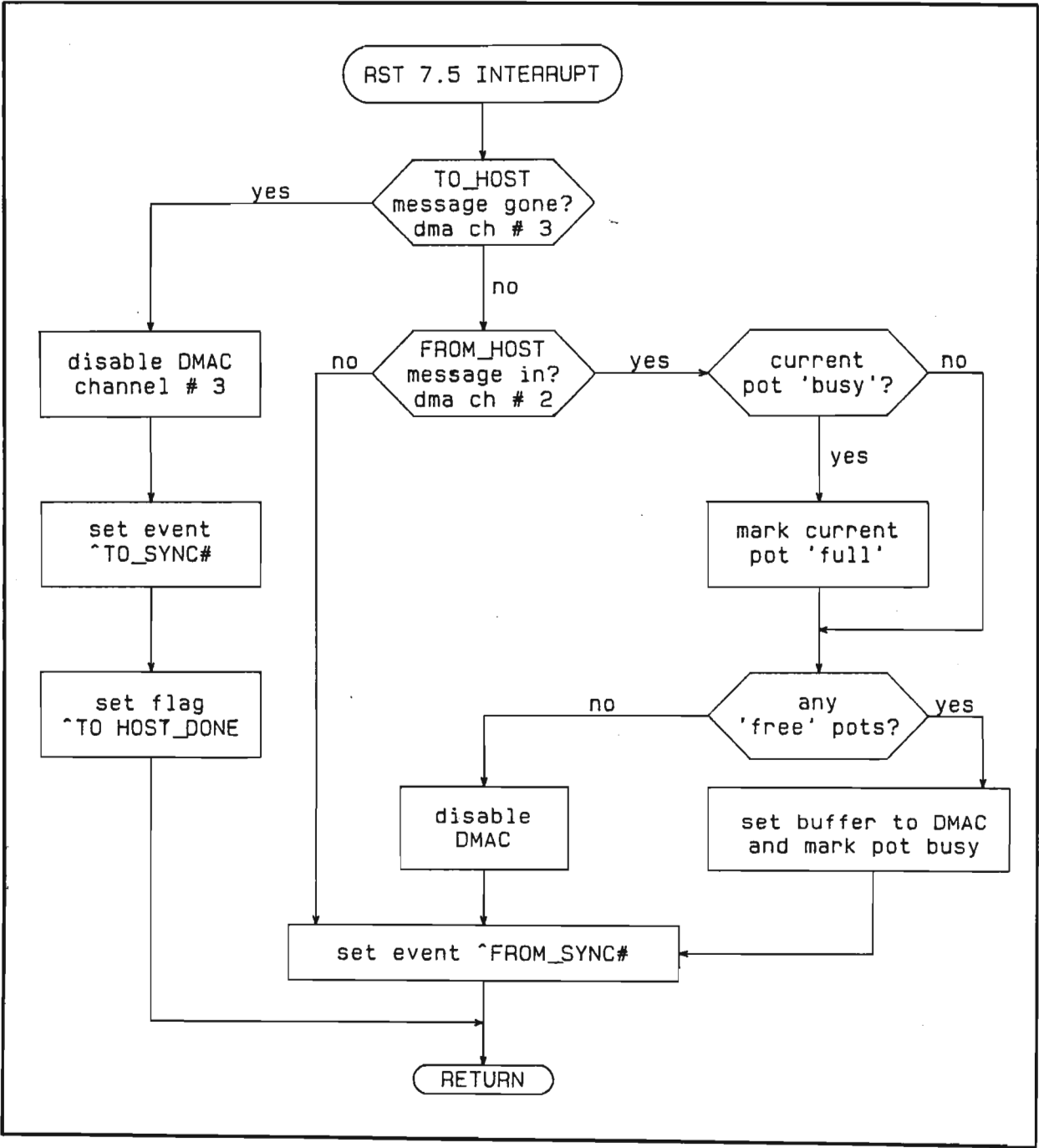


Figure G.2 Flow Chart for ^STC_INT Interrupt Routine

When released from the event wait, ^FROM_HOST searches for *full* pots, dispatching their buffers to the relevant port handlers. Providing the port has been configured (see below) ^FROM_HOST routes the four port read requests to message queues 1 to 4, and the write requests to message queues 5 to 8. Control requests other than type zero and type twenty four (RTE:CN,LU,0 and :CN,LU,30B,xx requests) are treated as write requests and sent to the respective TX_TASK's via queues 5 to 8.

Port task abort

The control request type zero (:CN,LU,0) when sent to a port requires that the current operation should be aborted immediately. This abort request is processed by ^FROM_HOST simply by aborting all three tasks associated with the port, (using calls to ^STDORM) and then re-scheduling them all using a call to set the TX_TASK active as TX_TASK re-schedules the other two tasks. The type zero control request message is then returned to the host (via queue 9) as confirmation.

Port-configuration

The RMUX software uses the least two significant bits of the equipment table entry address in the host (see appendix H for further detail) as an index to the port number. Since equipment table entries (EQT's) are 15 words long, all combinations of the least two bits are obtained if the 4 EQT's are contiguous in the host, a requirement for the multiplexer. However channel zero need not have its EQT index as zero, so the first call to each multiplexer channel after power-up must be a configuration call (RTE:CN,LU,30B,nnnnnnB) where the least two significant bits of the configure call specify which port should be associated with the EQT address in the message header.

^FROM_HOST detects configuration calls to unconfigured ports and sets up a four byte cross reference table (^PORT_ADDRS) using the EQT address least two bits as an index. Bit zero of the ^PORT_ADDRS table entry is set if the port has been configured, and bits 2-1 hold the port number to be used for the given EQT address. Once the ^PORT_ADDRS table has been set up, the configuration request is passed on to the relevant TX_TASK since it contains interface set up data.

Once ^FROM_HOST has dispatched all *full* pots, marking them *empty* as it does, it then attempts to fill all *empty* pots from the pool queue. When all pots are *full* (or no buffers remain) it scans all pots to ensure there is a *busy* pot. If not then a *free* pot is changed to *busy* and the DMAC set up to fill the buffer held in the pot. This covers the situation where ^STC_INT could not find a 'free' pot to set 'busy'. Finally ^FROM_HOST once again suspends itself against the event ^FROM_SYNC#.

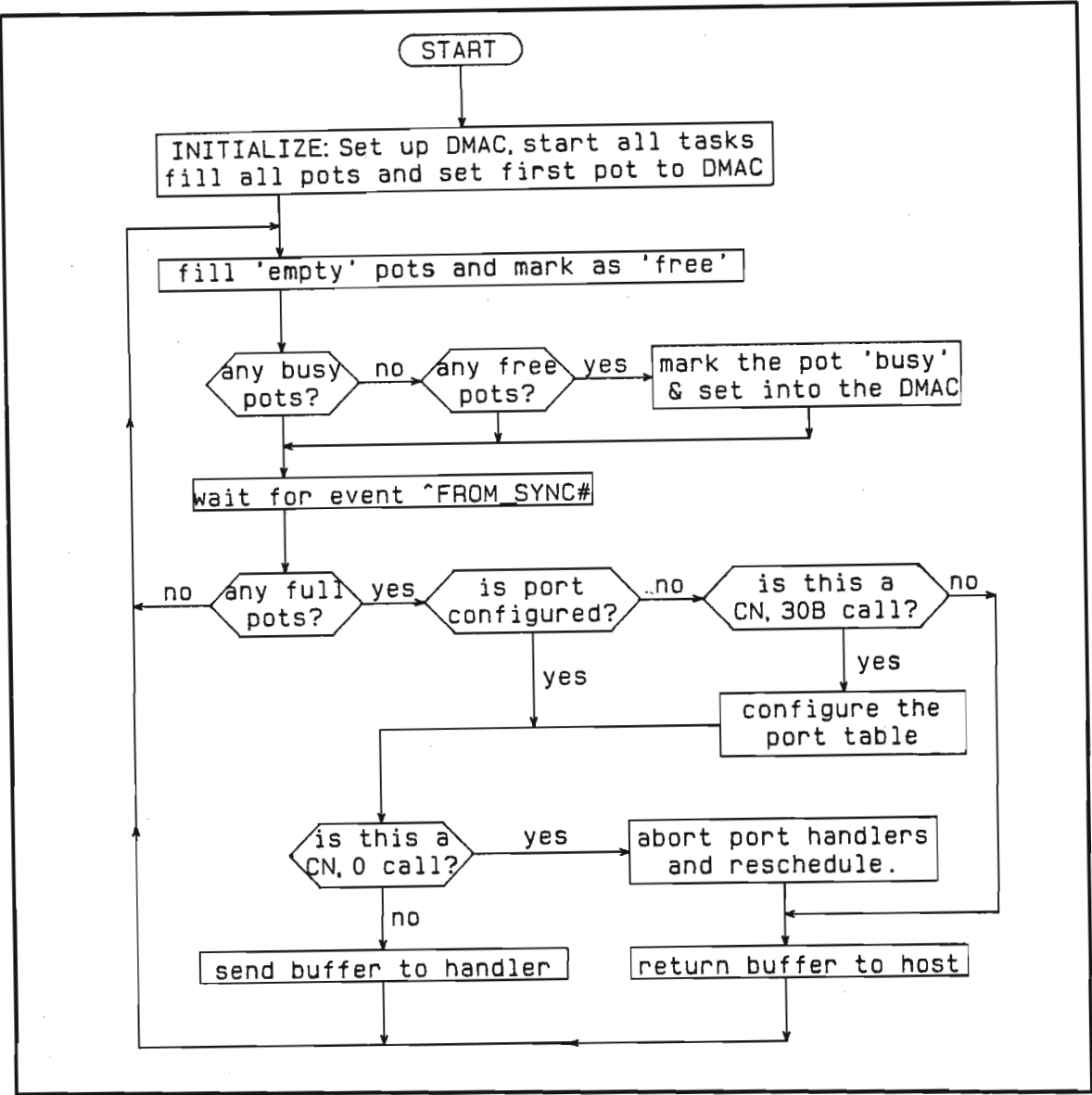


Figure G.3 Flow Chart for the ^FROM_HOST Routine

^TO_HOST

Initially, ^TO_HOST suspends itself against message queue 9 waiting for a message to arrive. When one does it sets up channel 3 of the DMAC to output the buffer and checks to see if the host's CNTL flip flop is set or not. Should it be clear, it implies that the host is busy outputting data to the interface and so should not be interrupted. This is a problem with only having a half duplex backplane in the host. If the CNTL flip flop was set then the host could be interrupted and so ^TO_HOST sets the host FLG flip flop by pulsing the SOD line (appendix E). In either case, ^TO_HOST then suspends itself for 20mS against the event ^TO_SYNC#. When released from this event, either by a timeout or by ^STC_INT setting the event (as described previously), TO_HOST checks to see if the DMAC

has sent the message. If not, then it loops back and again checks the host's CNTL flip flop before setting the host's flag. Thus, should the host be busy with the interface, or if it ignores the flag interrupt, ^TO_HOST will re-try every 20mS until the host eventually responds.

Once the host has accepted the message, ^TO_HOST returns the used message buffer to the pool queue and loops back to the beginning to fetch another message. The flow chart in figure G.4 gives a brief picture of the sequence which ^TO_HOST executes.

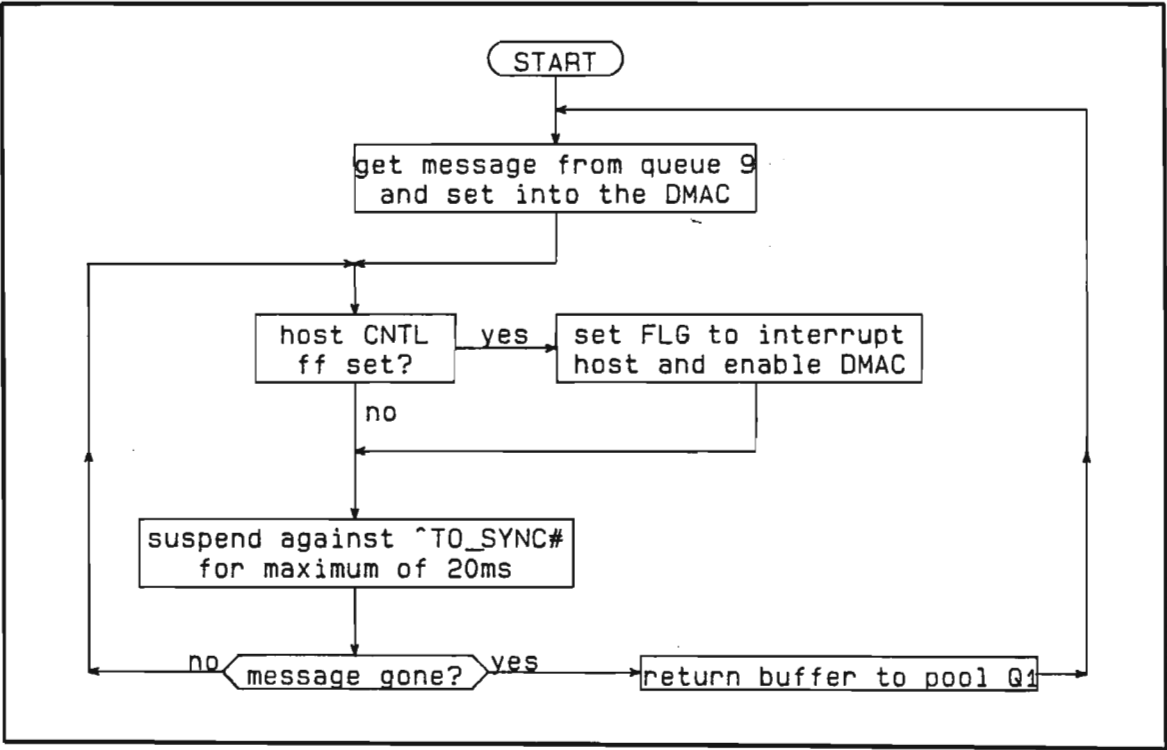


Figure G.4 Flow Chart for the ^TO HOST Routine

G.4 Terminal Handlers

Each port has three separate tasks to control the terminal, TX_TASK to handle transmit (output) and control requests, RX_TASK to handle receive (input) requests and BREAK to handle unsolicited interrupts and attract a host prompt if necessary. The transmit and receive tasks each have their own input message queue and send the completed requests to queue 9 for return to the host. BREAK fetches blank messages from the pool queue and sends the completed message to the host via queue 9.

Each task consists of two distinct and relatively independent sections; the main task body which runs under control of the MTX dispatcher and an interrupt handler which runs under control of the interrupt system independently of the state of it's main task body. The standard manner in which the tasks are written is that the main task gets a request, sets up variables to direct the operation, enables the interrupt handler and the device interrupts and then waits against an event for the transaction to complete. The interrupt handler is entered for each character interrupt, processes the interrupt, and initiates the next character transaction. When the entire transaction is complete, the interrupt handler typically disables further interrupts and sets the event to awaken the main task. This then tidies up, completes the request, and returns the entire message to the host as an acknowledgement before fetching a new request.

The following sections describe each of the handlers in turn, but for full details of all the handler options and functions the listings should be referred to as the handlers carry a very wide range of options to cope with specific terminals operating in various modes. Where relevant, these options will be mentioned but not covered in detail.

TX TASK

Flow charts of the operation of the transmit control task (TX_TASK) and the transmit interrupt handler routine (TX_CONT) are covered in figures G.5 and G.6 respectively. TX_TASK is responsible for the message request management while TX_CONT is responsible for the control of the serial port and the maintenance of the data output process. TX_TASK first schedules the other two port tasks RX_TASK and BREAK, and then waits for a message from its input queue. Once it has one it examines the CONWORD (see figure G.7) which specifies the type of request that the message contains. The full format of the 267 byte message buffer is described in figure G.7.

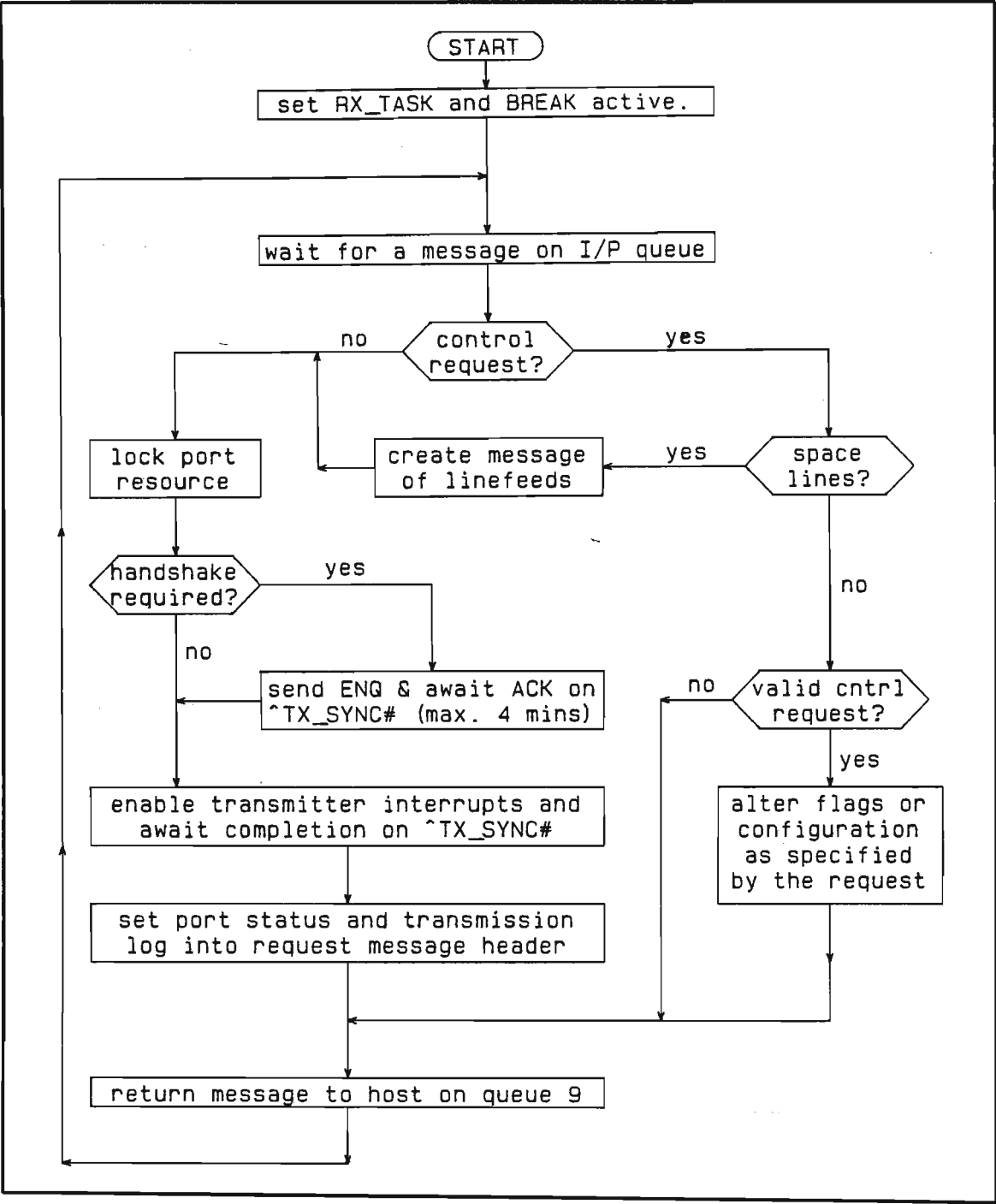


Figure G.5 Flow Chart for Terminal Task TX-TASK

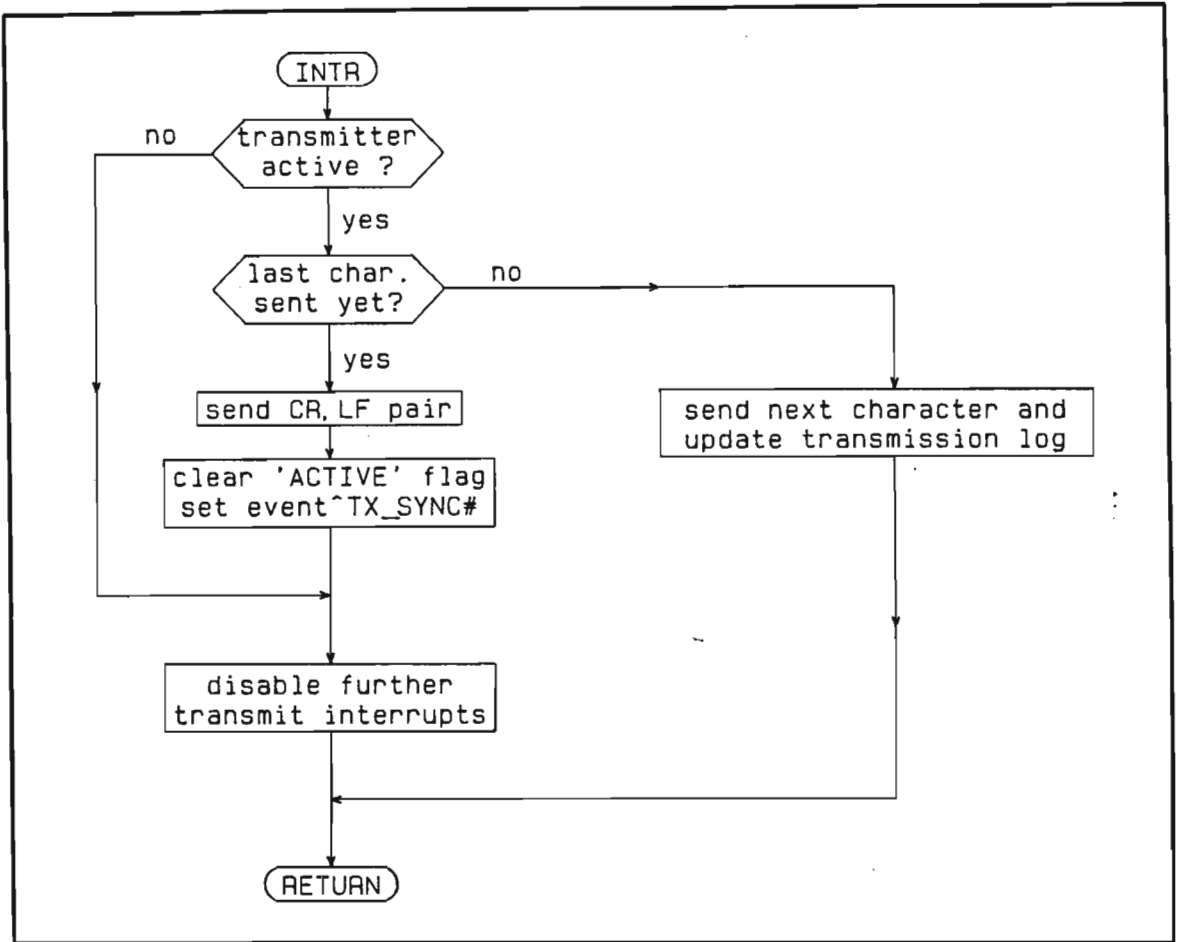


Figure G.6 Flow Chart for TX_CONT Interrupt Handler

For control requests (CONWORD = 3) the task branches to the section CONTRL which determines the type of control request. Most such requests merely involve changing flags or variables in the port's common data area. Typical examples are to set the timeout value or enable or disable the terminal from attracting host attention through unsolicited interrupts. The configuration control request causes CONTRL to re-programme the baud rate counter and the USART in order to allow on-line alteration of the baud rate, number of stop bits, number of data bits, and parity. The only control request which actually causes terminal I/O is the space line request which either performs a form feed or a specified number of line feeds on the terminal. CONTRL decodes this request and fills the message buffer with the form feed character or the specified number of line feed characters. It then passes the buffer back to the normal transmit section of TX_TASK for output as a normal message.

0-1	Buffer linkage address in RMUX but to or from host = SYN-SYN for message synchronisation.	(LSB-MSB)
2-3	EQT entry address for this port.	(MSB-LSB)
4*	EXEC call CNWRD. Host bits 11-6 shifted to bits 7-2 here.	
5	Input buffer length (max 255 bytes)	
6	On input:- holds a subchannel number on return:- holds the transmission log in bytes	
7**	Interface status byte echoed in EQT 5 status byte in host.	
8	Optional EXEC call parameter # 1	(MSB-LSB)
9	Optional EXEC call parameter # 2	(MSB-LSB)
12-267	max 255 bytes of data + 1 spare byte	

* CNWRD bits	B1-B0: call type. 01=READ, 10=WRITE, 11=Control B2: ASCII/Binary bit. (0/1) B4: Echo bit. (1 = echo input data) B6-B5: 11 = Program block read. IE. echo CR & LF only. 10 = Honest mode. All I/P accepted and no CR,LF sent
** Status byte bits	B0: Set on return to invoke 'break' mode response. B1: Set when terminal enabled for 'break' mode. B3: Set when a parity error was detected in last message. B4: Indicates status of interface DSR line at message end. B5: EOT bit set to indicate an EOT (CNTRL D) was input.

Figure G.7 Message Buffer Format

For transmit requests TX_TASK first locks a resource number which both TX_TASK and RX_TASK associate with the serial port itself. This ensures mutual exclusion between transmit and receive message requests. Once the port is locked the message header is unpacked and stored into port common. Specifically, the buffer address, conword, and subchannel are all saved so that TX_CONT (the interrupt handler) can access them. The transmit active flag (TX_ACTIV) is set to indicate to TX_CONT that a request is being processed and then the handshake flag is checked. This flag is set up by the configuration call, and determines which type of handshake is to be used with the terminal.

The three handshake types currently implemented are

- * no handshake,
- * an enquiry using the 'ENQ' character every 80 characters
- * an enquiry using the 'ETX' character before each line of output.

Both the last two types will hold transmission of the message until the terminal sends an 'ACK' acknowledge in response to the enquiry. Handshake 2 corresponds to that used by HP terminals while Handshake 3 is used by the QUME daisy wheel printer and several other printers.

Where a handshake sequence is required, TX_TASK sends the requisite enquiry character, sets up the Receive interrupt vector to TX_CONT and then waits against the event ^TX_SYNC# for the acknowledge to arrive. When the acknowledge character ('ACK') arrives, TX_CONT receives it and sets the event to release the main task which then moves the receive interrupt vector back to the

BREAK task interrupt handler (BREAK_CNTL) which processes unsolicited input. TX_TASK then moves the transmit interrupt vector to point to the transmit interrupt handler TX_CONT, enables the transmitter interrupt and waits for transmit completion against event ^TX_SYNC#.

When TX_CONT has completed the transmission (described separately below) it sets the event ^TX_SYNC# which releases TX_TASK. TX_TASK then clears the transmit active flag, stores terminal status into the message status word (see figure G.7), releases the port resource and returns the message buffer to the host by sending it to queue 9, ^TO_HOST's input queue. TX_TASK then loops back to the beginning to fetch another message buffer.

TX CONT

TX_CONT is entered initially due to an interrupt since the transmit section of the USART will interrupt whenever it is ready to send a new character and thus will interrupt as soon as the transmitter interrupt is enabled. TX_CONT first checks that the transmit active flag (TX_ACTIV) is set, and if not, assumes an invalid interrupt, taking the spurious interrupt exit disabling the transmitter interrupt as it exits. If the active flag is set, then the remaining character count is checked to see if the last character has been sent and if not then the next character is output, the transmission log is incremented and the buffer length counter is decremented. The routine then returns from the interrupt via the MTX operating system to the programme that was interrupted.

If the last character had been sent then a carriage return is sent, and the interrupt pointer moved to cause the next entry to be at a different point. On the next interrupt entry a line feed is sent and again the pointer is moved. The next interrupt indicates that the line feed has been transmitted and that the record is complete so the active flag (TX_ACTIV) is cleared and the event ^TX_SYNC# is set. Finally the transmitter interrupts are turned off and a return from interrupt executed.

TX_CONT has two extra features not shown in the flow chart; The first of these supports the HP convention that the terminating carriage return and line feed will be omitted if the last character in a line is the underscore '_' character. The second is the so-called 'HONEST' mode which is invoked by setting bit 6 in the input conword and also causes the carriage return and line feed to be suppressed.

RX TASK

This task, which is the main body for handling receive requests (those requests requiring input from the terminal to the host) is very similar to TX_TASK with the exception that it does not have any processing of control messages to do, and that the terminal handshake is somewhat simpler.

`RX_TASK` starts by waiting for a message from its input queue, and when it gets one it locks the port resource and unpacks all the relevant header variables into port common. It next determines whether a handshake is required and if so sends out a DC1 character. This is the HP type of input handshake where a DC1 (11H) character must be sent to the terminal to trigger input. The next step is to set the receive interrupt vector to point to the receive interrupt handler `RX_CONT` and then set a timed wait on event `^RX_SYNC#`. The time limit is alterable by the set timeout control request and is re-instated each time a character arrives at the interrupt handler `RX_CONT`. The reason that the timeout value is reset each character is so that the request will only time out after a period of inactivity rather than when a user is in the middle of typing a long and complex line of text.

Once the `RX_TASK` is released, either as the result of an event being set by `RX_CONT`, or as the result of a timeout, it resets the receive interrupt vector back to the `BREAK` interrupt handler (`BREAK_CNTL`), sets the status byte in the message header, returns the message to the host via queue 9 and then loops to the start to fetch a new message.

`RX_CONT`

The receive interrupt handler is in essence a very straightforward handler complicated by the large variety of input options it is required to cope with. The flow chart in figure G.8 shows only the standard terminal options, having left out the options of binary mode, honest mode, block mode, and all the various handshakes for the sake of clarity. For further detail on the handshake protocols see appendix K and for greater implementation detail consult the source code listings.

`RX_CONT` fetches the input character immediately after the interrupt to reduce the possibility of overrun and checks to see if it is a `CNTL X` (`CAN`) which is used as a break mode character. If it is a `CNTL X` then the break bit is set in the header status byte. If not then the timeout is restored to its full value and the character checked against the special control characters such as delete, backspace, (`CNTL Z`), and (`CNTL D`).

`CNTL Z` allows the echo mode to be toggled within a line of input, while `CNTL D` deletes the line returning a zero length line to the host. Delete has two results, depending upon the terminal type. For HP terminals (determined by the handshake type), the delete causes the string `'CR,ESC,K,\'` to be output which clears the current line and prints the back slash character `'\'` as a prompt. On non-HP terminals the output string for delete is `'\,CR,LF,\'` which issues a backslash, moves to a new line and gives a backslash prompt.

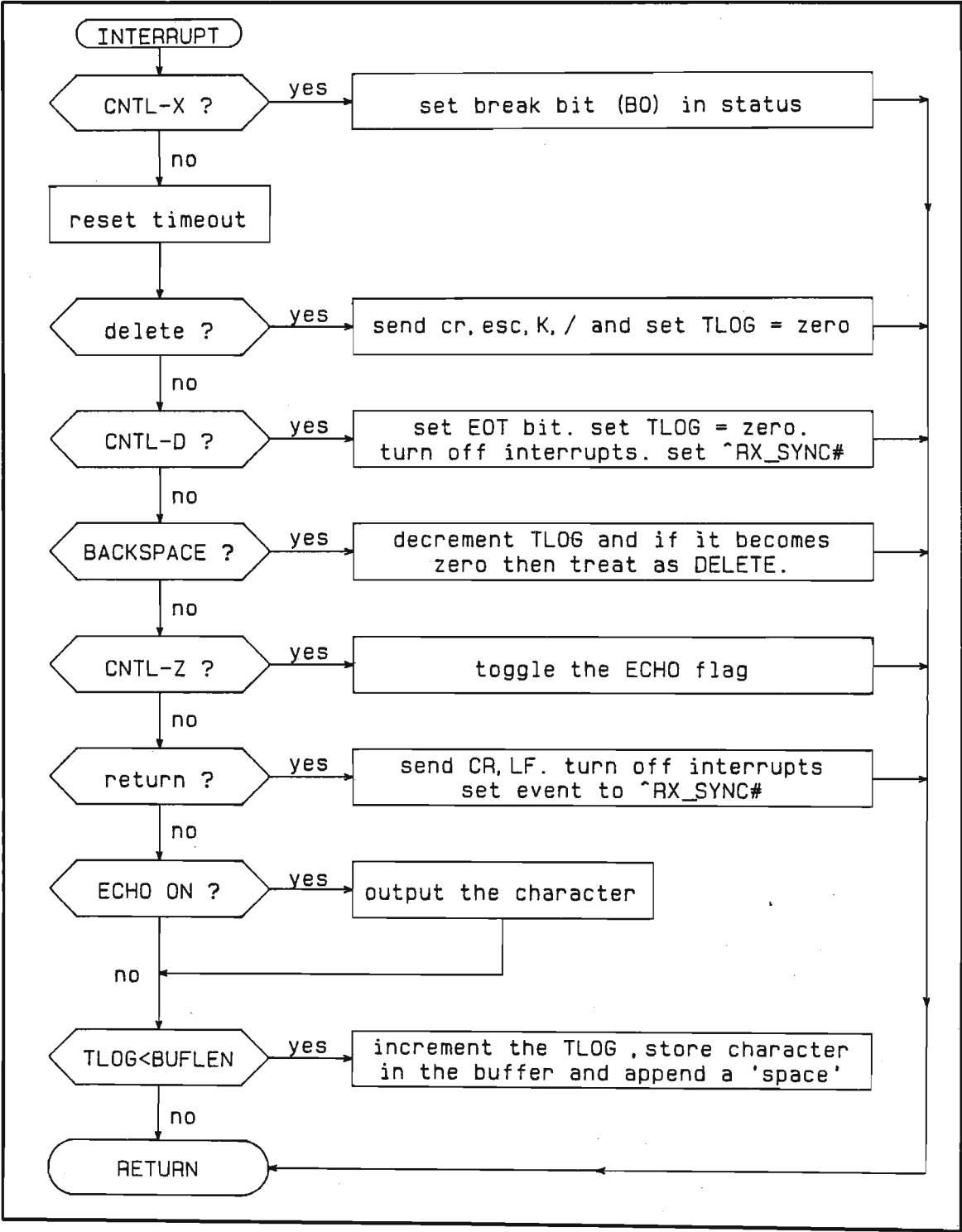


Figure G.8 Receive Interrupt Handler Flow Chart

If the input character is not an option character, then the next check is to see whether an echo is required and if so the input character is transmitted back to the terminal via the transmit port of the USART. The final check is to see whether the transmission log, which is a count of the number of characters input so far is still less than the requested buffer length. If so, then the character is stored in the buffer, the transmission log is updated and a padding 'space' is stored in the next

byte of the buffer. This is needed since the host only inputs complete words, and requires the last word to be padded with an ASCII 'space' character if an odd number of characters is input. RX_CONT then returns from the interrupt.

The special options not shown in the flow chart are:

i) Honest mode

In honest mode all input characters are stored in the buffer with no special character checking being performed.

ii) Binary mode

In this mode no special character checks are made, all characters are stored in the input buffer irrespective of what they are until the buffer length equals the transmission log. There is no early termination character in binary mode. Finally in this mode the last character in the buffer is followed by a null rather than an ascii space.

iii) Block mode

Here no normal characters are echoed, but the final carriage return line feed is still sent to the terminal.

iv) Handshake mode

Although not strictly a mode, with handshake on if the first character from the terminal is a DC2 (12H) then the following carriage return and line feed is ignored, and a DC1 (11H) trigger character is sent to the terminal to initiate a block mode transfer. This rather odd handshake is termed 'Long Handshake Trigger Mode' by HP.

BREAK

During the times when no input has been requested by the host, and when TX_TASK is not waiting for an 'ACK' handshake acknowledge, the receive interrupt vector is left pointing at the break task's interrupt handler BREAK_CNTL. Thus all unsolicited input is processed by BREAK_CNTL which ignores all characters except 'space' and 'CNTL X' (20H & 18H), these two characters being used to attract the attention of RTE IVB. In order to do this, bit zero of the returned status word is set which causes the host driver routine DVX05 (see appendix H) to attract RTE IVB by scheduling a program called PRMPT.

On receipt of one of the 'break' characters, BREAK_CNTL examines the port resource flag to determine if the port is locked to either RX_TASK or TX_TASK. If the resource is locked then the port common area in RAM will contain pointers to the current message buffer and so BREAK_CNTL merely needs to access the message status byte and set the least significant bit. This will then attract the host

as soon as that message buffer gets returned to the host. If however the resource is not locked, then no current buffer is available to have it's break bit set. BREAK_CNTL thus sets the event ^BREAK# which releases the main task BREAK. BREAK fetches a buffer from the pool queue, fills the header with the correct equipment table entry address, sets the conword, buffer length, and transmission log to zero, sets bit zero (the 'break' bit) in the status byte and sends this buffer to the host. BREAK then loops back to the beginning and suspends itself on the event ^BREAK#. Figure G.9 shows both these routines in flow chart form.

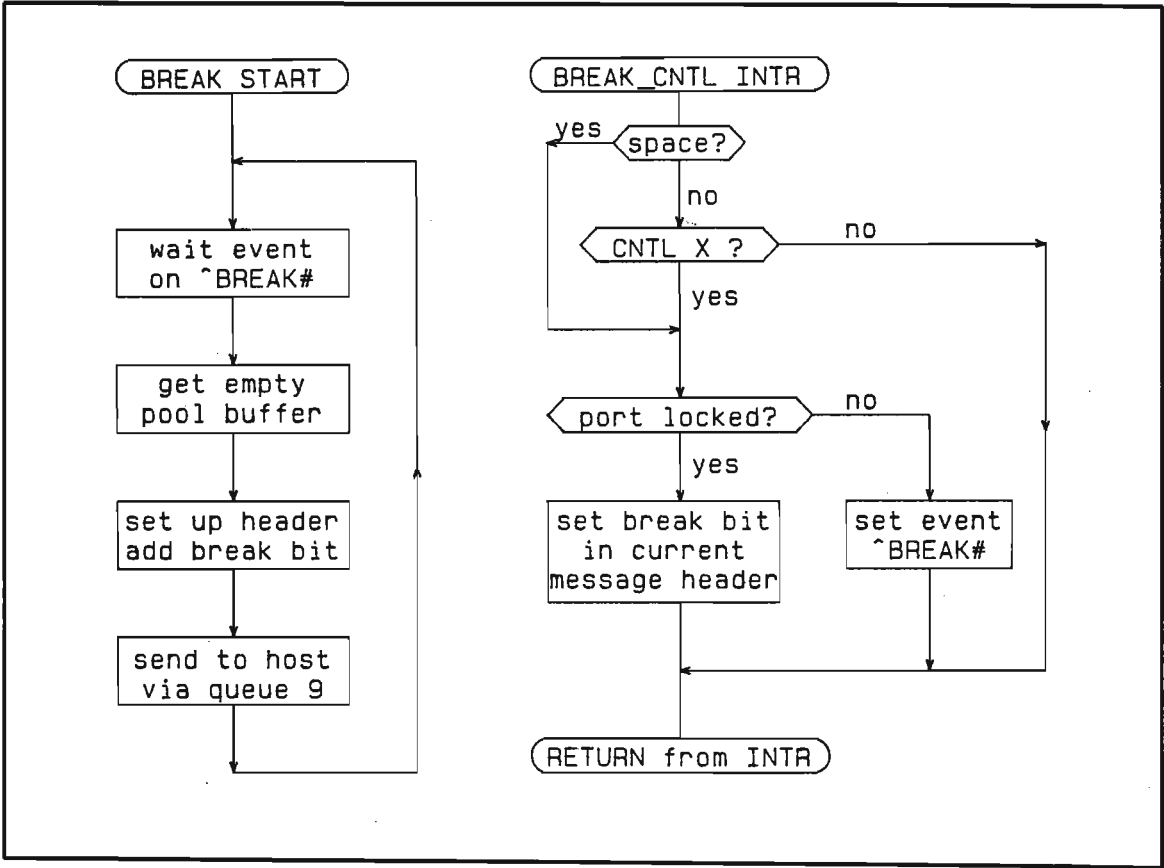


Figure G.9 Flow Chart for BREAK Task and Interrupt Handler

G.5 Summary of Terminal Characteristics and Options

The RMUX terminal system handles three types of requests from the host: read, write, and control, identified by the CONWORD which consists of Bits 11–6 and 1–0 of the standard HP EXEC call CONWORD (see figure G.7). Bits 1–0 determine the request type while bits 7–2 (HP bits 11–6) enable various options. Table G.1 describes the function of these bits for read and write calls. For control calls, these six bits have no meaning as the fifth byte in the header contains the control code. This code allows several different functions to be specified, as described in table G.2. Table G.3 expands on the specific options available through the configure terminal (30B) call.

Normal read and write requests may contain some special characters which affect the operation of the multiplexer. These are described in table G.4 for read requests and table G.5 for write requests.

Table G.1 Conword option bits for Read and Write calls

RMUX CONWORD	7	6	5	4	3	2					1	0
corresponds to	:	:	:	:	:	:					:	:
HP CONWORD	11	10	9	8	7	6	5	4	3	2	1	0

Bit	Function
0–1	00 – invalid, 01 – READ, 10 – WRITE, 11 – CONTROL
2	BINARY BIT. On READ's this bit stops special character processing and causes request termination to occur on buffer full. Odd byte count causes a NULL byte pad character. On WRITE's it has no effect.
4	ECHO BIT. On READ's this causes all input to be echoed back to the terminal except for some special characters. ECHO can be toggled during input with CNTL Z. On WRITE's, ECHO has no effect.
6	HONEST MODE BIT. On READ's this stops special character processing. CR still terminates input however. On WRITE it stops the CR,LF pair from being sent at the record end.

Table G.2 Control Request Options

Request code	Function	Parameter
11B	Space lines request.	'N' = +ve no of lines 'N' = -1 for new page
20B	Enable terminal break mode	nil
21B	Disable terminal break mode	nil
22B	Set read request timeout delay	'N' = +ve no of 10ms
30B	Configure terminal options	'N' (see Table G.3)

Table G.3 Configure Request Format (Control 30B call)

Parameter word	C	C	X	T	S	S	P	P	H	B	B	B	B	H	N	N
bit definitions	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit field	Function	
15-14	(CC)	Set USART character length (00 = 5 bits) (01 = 6 bits) (data field only) (10 = 7 bits) (11 = 8 bits)
11-10	(SS)	Set number of transmitted stop bits (01 = 1 bit) (10 = 1.5 bits) (11 = 2 bits)
9-8	(PP)	Set parity for transmission and checking (00 & 10 = no parity) (01 = odd) (11 = even)
7-2	(HH)	Set terminal handshake mode (00 = no handshake) (01 = HP style handshake, ENQ/ACK on transmit and DC1 to trigger receive) (10 = QUME style handshake, ETX/ACK before each transmitted line)
1-0	(NN)	Set port number to be used (only valid on first call)
6-3	(BBBB)	Set terminal baud rate: 50 75 110 135 150 300 1 2 3 4 5 6 1200 1800 2400 4800 9600 19200 7 8 9 10 11 12
12	(T)	Set to enable terminal to use break mode interrupt
13		Not used

Table G.4 Special Character Processing for Read Requests

Character	Normal Mode effect	Binary mode effect	Honest mode effect
EOT (^D)	Immediate return with zero record length	Add to the input buffer	Add to the input buffer
BS	Echo and reduce transmission log by 1. If TLOG = 0 then issue delete sequence.	Add to the input buffer	Add to the input buffer
LF	Echoed but ignored	Add to the input buffer	Echoed but ignored
CR or RS	Terminate input phase Send CR LF sequence. Return message to Host	Add to the input buffer	Terminate input request
DC2 (^R)	If it is the first character then send a DC1 back to the terminal and ignore the first (CR, LF), else add to buffer as normal.		
CAN (^X)	Set break mode attention bit (B0) in status byte.		
SUB (^Z)	Toggle echo flag on/off	Add to the input buffer	Add to the input buffer
US	Echoed but ignored	Add to the input buffer	Add to the input buffer
DEL	Send CR,ESC,K,\ or \,CR,LF,\ to terminal and set TLOG to zero	Add to the input buffer	Add to the input buffer

Table G.5 Special Character Processing for Write Requests

Character	Normal Mode effect	Honest mode effect
Underscore ' _ '	If last character in buffer then terminate output and complete without sending CR, LF. If not last then output as normal.	Output as normal

Appendix H

The RTE IVB Input/Output System

H.1 General Overview

The RTE IVB real time executive system provides central control of all input/output operations through the system module RTIOC. The system can interface to a maximum of 56 I/O devices, each consisting of a single printed circuit board controller driven or controlled by a software module termed a system driver. RTE allows several drivers, one for each different type of I/O controller connected to the machine.

RTIOC uses three tables to define the system I/O structure, these being:

- * The equipment table
- * The interrupt table
- * The Device reference table. (DRT)

These tables are defined during the system generation phase and only the device reference table may be modified on line.

The equipment table contains of a 15 word entry (EQT) for each I/O device connected to the system. The entry is used to specify the driver address, the device type, status, and all temporary information needed to process a request.

The interrupt table consists of a single word entry for each I/O controller (each occupied slot), to specify how an interrupt from the device is to be handled. The handler may be either a system driver, or a user supplied programme.

The Device reference table contains a two word entry for every logically distinct device in the system, and is used to link each 'logical unit' to a particular subchannel of a piece of equipment.

All user I/O requests to a logical unit are routed via the device reference table to an EQT, from which the driver address can be found. The driver initiator section would start the request and then suspend the calling programme pending a completion interrupt. Each interrupt causes RTIOC to determine the correct interrupt handler from the Interrupt table and pass control to that handler. Normally this handler will be a system driver whose continuator address is also stored in the EQT. After the final interrupt the continuator returns control to RTIOC to release the calling programme to the schedule state.

H.2 The Equipment Table Entry (EQT)

Each physical I/O controller plugged into the computer must have an EQT to be recognised by RTE. These entries, each 15 words long, are established when the system is generated and are not alterable on line. The EQT (figure H.1) is used to hold status information such as the two driver entry point addresses, the channel or slot number, and the equipment type, as well as dynamic information and temporary data.

word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	R															
2	R															
3	R															
4	D	B	P	S	T											
5	avail															
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																

R - Reserved bit for system use only
D - Set if driver requires a DCPC (DMA) channel.
B - Set if automatic output buffering is used.
P - Set if driver is to process power failure recovery.
S - Set if driver is to process a device timeout.
T - Set on driver entry if the device has just timed out.
avail - I/O availability indicator.
00 - available for use
01 - device disabled (down)
10 - device busy (currently in use)
11 - driver is waiting for a DCPC channel.

Figure H.1 Equipment Table Entry format.

Word 1 of the EQT is the link word used to link together the ID segments of all programmes wishing to use the EQT. The first one in the chain is the ID segment of the programme currently busy with the EQT, while all the rest wait in the queue for the current request to complete.

Words 2 and 3 hold the entry point addresses of the initiator and continuator sections of the driver. The initiator is called from RTIOC to start the request while the continuator is entered to process an interrupt.

Word 4 holds driver status information, bits of which may be altered by RTIOC, while word 5 holds the static device type field plus the dynamic device status which may be altered by the driver.

Words 6 to 10 are set prior to entry into the initiator section to hold the current request information. Thereafter all words from EQT 6 to EQT 13 may be used to store temporary data as they are not altered by the system until the next initiation request.

Words 14 and 15 are used for timeout processing, word 15 being the actual timer incremented from a negative value every 10mS while word 14 is the reset value to be stored in word 15 at the start of each timeout period.

H.3 The Interrupt Table

The interrupt table consists of a one word entry for each I/O slot in the machine. The table is established at generation time, each entry being one of three different types:

- * A zero entry is used for those I/O channels where no I/O controller exists and hence no interrupt is expected.
- * A positive interrupt table entry is used to hold the address of the equipment table entry that will be used to process the interrupt.
- * A negative value indicates that a user supplied programme will handle the interrupt, the complement of the entry being the ID segment address of the programme that should be scheduled.

H.4 The Device Reference Table

The RTE IV system provides for a maximum of 255 logical units and only 63 EQT's, the device reference table (DRT) being the cross map table to link logical units to the physical devices (EQT's). Each DRT entry consists of two words:

- * The first word contains the six bit EQT number, the subchannel number and a lock flag.
- * The second word is a link pointer to link pending requests to a logical unit.

The concept of a subchannel is used to allow a single physical device to contain several logical devices. A typical example of this is the Hewlett Packard 2648 graphics terminal which can contain two cassette tape drives, a graphics memory, and an external printer in addition to the normal alpha screen and keyboard. This terminal uses five subchannels and hence five logical units, all associated with a single item of equipment and a single EQT. With the subchannel system, only one subchannel of any EQT may be busy at a time as an EQT is only capable of handling one transaction at a time.

H.5 The Standard RTE System Driver Layout

Under the RTE system, all I/O transfers to and from a peripheral device are performed under the control of a two part module called a 'driver', the two parts being termed the 'initiator' and the 'continuator'.

The 'initiator' is called by RTE which passes the user's call parameters via the EQT. Since the driver may handle several EQT's, the initiator must configure all I/O instructions to reflect the correct slot address. It should then initialise all software flags and variables, check the request for validity, prepare the request for further processing, initiate the first I/O transaction of the call and then return to the operating system leaving the caller suspended.

The 'continuator' is called by RTIOC whenever an interrupt arrives from the associated I/O controller. The continuator should first check the link word in EQT 1 to check that the EQT is in fact busy. Should this be blank, then no request is outstanding and the interrupt should be ignored unless special processing is to be performed for unsolicited interrupts. Otherwise the continuator should then configure all I/O instructions again since some other EQT could have used the driver in the interim period. The next I/O transaction is then started and a return made to RTIOC. Should no further I/O exist, a special return is made to RTIOC to indicate request completion at which the caller will be re-scheduled.

H.6 Mapping system usage

When a user programme calls a driver to perform an I/O request, RTIOC must first include the specified driver in the user's memory map so that the driver may access the data contained within the user's programme. To enable this, all drivers are relocated into two

page overlays, any one of which may be incorporated into the two pages (pages 2 & 3) set aside in the user map specifically for drivers. (A page in the HP 1000 amounts to 1K of 16 bit words). Thus when the request is first processed, the requisite driver overlay is mapped into the user's driver partition, and the normal initiation process taken.

When an interrupt occurs, the user programme in the current user map is unlikely to be the programme awaiting the interrupt since that programme is suspended. Hence before RTIOC can pass control to the driver's continuator section, it must install the correct user into the user map and save the state of the interrupted user's map. Once this is done, RTIOC passes control to the driver which remains mapped into the user's driver pages. When the driver continuator returns to RTIOC, RTIOC removes the interrupting user from the user map and re-instates the user that was executing prior to the interrupt. RTIOC is a system routine in the system map which is always the map enabled to process an interrupt.

H.7 Summary of the I/O process in RTE

- * The user makes a call to RTE EXEC specifying the I/O transaction type, data buffer length and address, and device logical unit.
- * EXEC passes the request to RTIOC which uses the Device Reference Table to determine EQT number and subchannel.
- * From the EQT, RTIOC determines the driver to be used and via a driver mapping table (\$XDMT) [33], maps the correct driver into the driver partition.
- * The call parameters are transferred to the EQT.
- * The driver initiator is called which configures I/O instructions, sets up variables and initiates the transaction, returning to RTIOC.
- * RTIOC suspends the user awaiting I/O completion and transfers control to the scheduler which maps in a new user and transfers control to it.
- * On interrupt, the hardware enables the system map and executes the trap cell instruction which normally contains a JSB to RTIOC. RTIOC then uses the interrupt table entry to

determine the correct EQT address.

- * From the EQT word 1, the address of the calling programme's ID segment is obtained, and after saving the current contents of the user map, RTIOC calls a system routine \$XDMP passing it the ID segment address.
- * \$XDMP determines the correct map contents to enable the required programme and loads the user map with these values. This installs the interrupting programme and its driver.
- * RTIOC gets the continuator address from the EQT and transfers to it.
- * The continuator processes the interrupt, accessing data from the user programme as necessary. That done, it returns to RTIOC.
- * RTIOC reinstates the interrupted user map and returns to normal processing via the scheduler. If the continuator had indicated completion of the transaction, the calling programme would be moved into the schedule queue to compete with other programmes for processor time.

Appendix I

The Host Driver to Control the RMUX Interface

I.1 Introduction

This appendix describes the driver programme used by the RTE IVB operating system to control the RMUX interface. The programme consists of two distinct modules: DVX05, the major driver routine which resides in a normal RTE IVB driver segment (see appendix H) and a special purpose mapping routine \$DVM5 which resides in Table Area 1.

The protocol involved in passing messages to and from the interface is described, as is the message format itself, and the usage that the driver makes of all the system tables.

I.2 Incorporating Multiple Interrupt Sources per Slot

As discussed in appendix H, the HP RTE IVB system only caters for one interrupt source per interface slot due to the one to one correspondence between EQT entries and interface slots. However the RMUX interface with four independent channels requires four EQT entries so that four independent programmes may simultaneously use the channels. This required a section of code to extend the operating system function and perform user programme mapping. This code had to be available from all possible maps, both user and system, so that it could change user maps or access the system map. A small module (\$DVM5) was therefore written to be included in Table Area 1 (TA1), the only area accessible to all maps. \$DVM5's function when called from the main driver DVX05 is to save the contents of the current user map, enable the system map and then call an RTE IVB module (\$XDMP) to switch in the correct user map for which the data is destined. \$DVM5 then transfers a block of data (one record) between the interface and the user programme before finally restoring the original user map and returning to the main driver DVX05.

The main driver module (DVX05) determines the mapping system status before jumping to \$DVM5 with the system map enabled. The current user map can then be saved before \$XDMP is called to install the correct map for the interrupt. Since this may be either a user or the system, \$DVM5 checks to see whether to move back to the user map (the new user) or to stay in the system map for the data transfer. If it needs the system map then data transfer can commence immediately and a direct return to the caller can be made, restoring the original mapping system status at the same time. If, however, the data is for a new user map, this user map must be enabled before data transfer. After this \$DVM5

returns to the system map, restores the original user and returns to the caller. This process occupies only 88 words of Table Area 1 which has little impact on the normal operating system memory requirements.

The inclusion of \$DVM5 allowed a single driver, while servicing a single interface card to map in any user and transfer data. However the driver had to have some means of determining the correct map corresponding to an interrupt. This was implemented by passing the calling EQT entry address to the interface at the initiation of each request and then retrieving this address from the interface following each interrupt, thus enabling the driver to identify the correct EQT to handle each interrupt. From the EQT entry address, the operating system (via \$XDMP) can determine the correct user map, while all information pertinent to the entire transaction is available to the driver from the EQT entry.

RTE IVB always places the addresses of all the elements of the current EQT entry into a fixed location in base page immediately prior to entering the driver as discussed in appendix H and in reference [34]. Thus once DVX05 has determined the correct EQT entry it places the list of element addresses into the base page where they are then treated normally. Upon completion of the driver, no attempt is made to restore the addresses found upon entry since the operating system makes no use of these addresses after completion of a driver call other than to determine which user to release from suspension.

I.3 Driver Structure

In keeping with accepted driver writing practice for HP computers [35] the driver DVX05 is written in two major parts, an initiation section with entry point IX05 and a completion section with entry point CX05.

I.4 The Initiation Section

The initiation section is called from the RTIOC section of RTE IVB at the start of each request, and its function is to perform all the tasks necessary to start the request. Upon entry for the first time after a re-boot of the system, all EQT entries associated with a particular slot must be initialised. This involves getting the ID segment address of the programme PRMPT (the LOGON and Break mode terminal supervisor) from the interrupt table and placing it EQT 13. Then the power-fail and time-out flags must be set in each EQT to inform the system that time-out and power-fail recovery are both to be handled by DVX05 itself.

Normal requests are then analysed to decide whether they are read, write or control requests and are processed accordingly. For read and write requests a common header is constructed (see figure I.1) and sent to the interface one word at a time followed, for write requests only, by the output data. For control requests, no data is sent but the header sent has a different format to that of the read and write requests (see figure I.2).

Byte #	Function
0	First synchronization character. ASCII "SYN" (Hex 16)
1	Second synchronization character
2	Most significant byte of EQT entry address
3	L.S. Byte of EQT entry address
4**	Conword from EXEC call (see below) EQT 6 bits 11-6 & 1-0 packed into bits 7-0
5	Host request buffer length (max 255 bytes)
6	Subchannel # sent.TLOG returned.
7	Interface status byte (lower 8 bits of EQT 5)
8	MSB of optional parameter #1 (from EXEC call)
9	LSB of optional parameter #1
10	MSB of optional parameter #2 (from EXEC call)
11	LSB of optional parameter #2
12	Data bytes from request buffer.
-	Sent only for write requests one word at a time
271	Fetched only for read requests one byte at a time. (max 255 bytes plus blank pad)

** Conword B1-0 define request type 01=read, 10=write
 B2 indicates binary mode - read and write
 B4 indicates read with echo - read only
 B6-5 = 11 for block mode trigger - read only
 B6-5 = 10 honest mode. All special characters are
 ignored - read and write

Figure I.1 Read/Write request message format

Output to the interface is performed one word at a time since word to byte unpacking is done by hardware on the interface. The header is sent by subroutine SENDT in DVX05 which constructs each word in turn and then passes the word to subroutine OUT for actual output. OUT sends the word to the interface waiting for a maximum of about 20µS for the interface to flag its acceptance of the data. The only time that this delay could expire on a functioning interface card would be if there were no blank buffers available for the data to be stored into (see appendix G) but with the card programmed as at present, this should never occur. Thus if there is no acceptance flag within the delay time, the card is assumed faulty, is set down (unavailable) and an error message is sent to the system console.

Byte #	Function
0	First synchronization character (ASCII "SYN")
1	Second synchronization character
2	M.S. Byte of EQT entry address
3	L.S. Byte of EQT entry address
4	Conword from exec call B1-0 =11 only
5	Control request function from B11-6 of EQT 6
6	Subchannel number of requesting port
7	Interface status byte - (lower 8 bits of EQT 5)
8	M.S.B. of control request parameter #1
9	L.S.B. of control request parameter #1
10	M.S.B. of control request parameter #2
11	L.S.B. of control request parameter #2

Figure I.2 Control Request Message Format

Once the header has been sent using subroutine OUT, the data (write requests only) is sent without checking for acceptance flags at all. The reason for this is that it saves some time, allowing a data transfer rate of 140 Kilo-words per second. It is unnecessary to check for flags since, as the header has just gone, the interface must be working, it must have an available storage buffer, and once a buffer is set up, the DMA controller on the interface is guaranteed to accept data at any rate up to 237 Kilo-words per second.

After the last word has been sent, a set control, clear flag instruction (*STC sc,C*) is sent to the interface to inform it that the DMA input buffer has been filled and is ready for processing. The initiation section is then complete and so DVX05 returns to the operating system to await an interrupt.

Power fail processing.

Power fail processing is performed by the initiator whenever B15 of EQT 5 is set upon entry. This indicates to the driver that it is to try and recover from the power failure and re-initialise the interface. The last configuration word (see appendix G.4) sent to the card is retrieved from EQT 12, where a copy is always stored, and sent to the interface as a normal configuration control request. This is the only power fail processing required as the system will always try to repeat all I/O requests that were pending prior to the power fail, and it always sets bit 15 of EQT 5 to signify power fail recovery processing.

I.5 Completion Interrupt Processing

Upon completion of any I/O request, the interface sets a flag to interrupt the host which then transfers control to the completion section of DVX05. There are several cases that may cause the completion section to be entered and so the following conditions must be checked for:

- * Completion of a normal request
- * Arrival of an unsolicited request
- * Time out due to no response

Completion of a normal request

The first response to a non time-out entry is to read the header of the return message from the interface and from this determine the EQT entry that corresponds to the message. This entry is then moved into the base page area and word 1 is checked. A non zero value indicates that a request is pending on this port and so the interrupt must be an acknowledgement. Thus the transmission log, status, and optional return parameters are transferred from the header to the EQT.

Next, status bit zero, (see figure I.6) is checked to see if the 'break' mode processor PRMPT is to be scheduled. (This bit would have been set if either the space bar was struck during output, or a CNTL X was entered at any time. Should either of these conditions have occurred, the interface would have tried to find a message that belonged to the port; if found, it would have set the 'break' bit, otherwise it would have assigned a new buffer, filled it with the current status and EQT entry address and sent this to the host.) When the 'break' mode bit is found set in a normal return request, the subroutine SCHED is called to schedule the programme PRMPT whose ID segment address is stored in EQT 13. SCHED calls the system routine \$LIST^[36] to perform the actual scheduling.

Finally the CONWORD from EQT 6 is checked to see if the returned message was as a result of a read request, for if so then the returned data must be read from the interface and stored into the user buffer in the user's address space. If this is the case, then the correct user map must be fetched (as described in section I.2) before any data transfer can occur.

Handling of unsolicited requests

When a message header has been read in and the correct EQT entry has been moved into the base page, if the EQT word 1 is found to be zero then the message occurred as the result of an unsolicited interrupt which occurred while there were no requests pending for the port. The response to this condition is to use the subroutine SCHED to schedule the programme PRMPT and then take an exit route to RTE informing the system that the interrupt was a spurious one.

Time-out Processing

If a time-out has been set on the EQT entry associated with any port, then the operating system will reset the time-out clock immediately prior to entering either the initiation or completion section of DVX05. The time-out clock is stored in EQT 15 and contains the number of ten milli-second time base 'ticks' required to fill the time-out interval. This clock is incremented by the operating system every ten milli-seconds providing it is non zero and should it reach zero, then the operating system will pass control to the completion section of DVX05 with the time-out bit set (EQT 4 bit 11), to inform the driver that the interface has not responded within the required time and that corrective action should be taken.

The action DVX05 takes in response to a time-out is to try and force a recovery of the interface by sending an abort request (CN,lu,0), setting a short 1,2 second time-out. Status word bit 6 is also set to indicate that an abort recovery is in process. This abort request should cause the RMUX operating system to restart all the processes (TX-TASK, RX-TASK and BREAK – see appendix G.4) associated with the relevant port and hence recover. The response to the abort request should come through in far less than 1,2 seconds, in which case the status word bit 6 will be cleared and a normal completion request taken with a zero length transmission log. This action looks to the calling programme as though the request had terminated correctly but that no data had been transferred.

Should the interface not respond within the 1,2 second time-out interval, then it implies that the entire interface or its operating system is at fault. Thus if DVX05 detects a time-out entry and discovers status word bit 6 already set, indicating an abort request already in process, it takes a time-out exit to the system which causes RTE IVB to mark the device as down and unavailable and prints a message to this effect on the system console. In this situation there is no corrective action that can be taken programmatically, hence the message is all that can be done, leaving the corrective action up to the system administrator.

I.6 Message Passing Protocol

The Hewlett Packard HP 1000 range of computers impose a rather rigid and limited set of restrictions on I/O transactions in that the entire I/O process is fundamentally half duplex. This is due to there being only one signal for indicating data ready (the CNTL strobe) and one signal for interrupt and data acceptance (the FLAG signal). Thus in trying to implement four channels of full duplex communications between host and interface, great care had to be taken over timing problems. A very rigid discipline had to be exercised over the information flow and to achieve this the following limitations were applied:

- a All messages between the host and the interface consist of complete records, each with its own header; no partial record transfers can occur.
- b Each request for the interface will be passed to the interface as a discrete package which when fully processed will be returned as confirmation.
- c All message packages, irrespective of content have a common format consisting of a 12 byte header (Fig. I.1) and a maximum of 255 bytes of data.
- d The host will clear the control flip-flop (CLC instruction) at least 20 μ S before starting any output to the interface.
- e The interface may not set the flag flip-flop, to inform the host of the presence of a return message, if the control flip-flop is clear.
- f Should the host not respond to a 'set flag' interrupt from the interface within 1 second then the interface will repeat the interrupt.
- g The host waits at least six microseconds after the last word of output data before executing a set control, clear flag instruction (STC sc,C).
- h Message headers start with the double ASCII 'SYN' characters to enable message synchronization.
- i The host always transfers at least a full 12 byte header to or from the interface for each transaction.

The first restriction ensures that an interrupt to either the host or the interface processor indicates the completion of a full message only, with all data transfer performed on a non interrupt basis. (DMA on the interface and a tight programme loop in the host). This also allows the host to use DMA to output a message to the interface, although this would not save any host time due to the limitation of only having two DMA channels in an HP 1000.

The requirement that all requests from the host be returned creates a simple and standard acknowledgement procedure for all types of requests. Acknowledging all requests after processing is complete ensures that the interface will only ever contain one message for each port, and so memory overflow cannot occur. If write requests were acknowledged immediately after transfer, but before transmission from the interface, then extra signals would be required to limit the number of messages that could be queued for each port without causing memory overflow. All this special processing for different request types and the extra signals would achieve very little improvement in throughput, especially in a multi-user system where one user's I/O can usually be overlapped with another user's processing, so keeping the host CPU fully occupied.

The requirements limiting message size and specifying standard headers allow for simple memory handling on the interface, since a pool of fixed length buffers can be maintained rather than having a more complicated dynamic memory management system required for variable length buffers. A further advantage of fixed length buffers is that they can be pre-assigned to the DMA channel in anticipation of input, ensuring that as soon as the host has data ready, it can send it without having to wait for the interface to find and assign memory.

Due to the shortage of suitable control signals in the HP 1000 I/O system, the flag flip-flop has to be used by the host to synchronise the transfer of each item of data, as well as being the only means for the interface to interrupt the host. Thus if the interface were to set the flag FF indicating the presence of a return message, while the host were using the same flag FF to transfer data to the interface, the resulting conflict could result in the loss of a data word. In order to eliminate this potential problem, the interface is not allowed to interrupt the host whenever the control flip-flop (CNTL) is clear, and the host is constrained to always clear the control FF while performing output to the interface.

A problem could still occur should the interface sample the CNTL FF immediately prior to it being cleared, decide an interrupt was allowed and hence set the flag FF just as the host started using the flag FF for output. To overcome this, the host may not perform any output for at least 20 μ S after clearing the CNTL FF thus giving the interface adequate time to set the flag while it is not being used and hence not causing any damage. Should this condition arise, the interrupt intended by setting the flag would not occur and would be lost since the interrupts are disabled while the driver is in control. To cover this possibility, the interface places a one second time-out on any returned message and should the message not be accepted by the host within this time then another interrupt would be sent to the host. Thus restrictions (d) through (g) effectively overcome all the problems encountered in using the single flag FF for two functions.

The restriction of waiting a minimum of $6\mu\text{s}$ after the last output before executing the set control/clear flag is to allow sufficient time for the interface DMA controller to accept the last data word and set the flag. If the set control/clear flag preceded the DMA controller acknowledge which sets the flag flip-flop then an invalid or spurious interrupt would be generated since the flag would now be interpreted as an interrupt.

During input from the interface the host will read only the desired number of bytes in the message, whereas the DMA controller on the interface will be set to output the full message of 12 header bytes plus 256 data bytes. Thus when the host takes its last data byte and stops input, the interface DMA controller will always fetch the next byte and deposit it in the interface output latch. The next time the host reads from the interface, before it gets the new message as expected, it will first get this left over byte from the previous message. Thus the host must read data from the interface and discard it until such time as it reads two successive 'SYN' characters and achieves synchronism.

The final problem to be overcome in the communication between the two processors was due to there only being one signal line by which the host could interrupt the interface processor, this being the STC strobe line activated by a set control (STC) instruction. The host has to interrupt the interface when it has accepted a message, and when it has sent out a new message. The interface processor thus examines the DMA count registers to determine whether the host has just completed an output or an input. Whichever count register has reduced by at least the header length is then taken as indicating the completed transfer that the host wishes to signify. This is the reason why condition (i) was imposed. Figures I.3 and I.4 show these sequences in flow chart form.

In summary, the problems created by having too few signal lines to maintain a proper full duplex communication between the host and the interface were overcome by placing both data and timing restrictions on the message transfer process. These restrictions do not however severely impair the operation of the interface as is shown in the main body of this report.

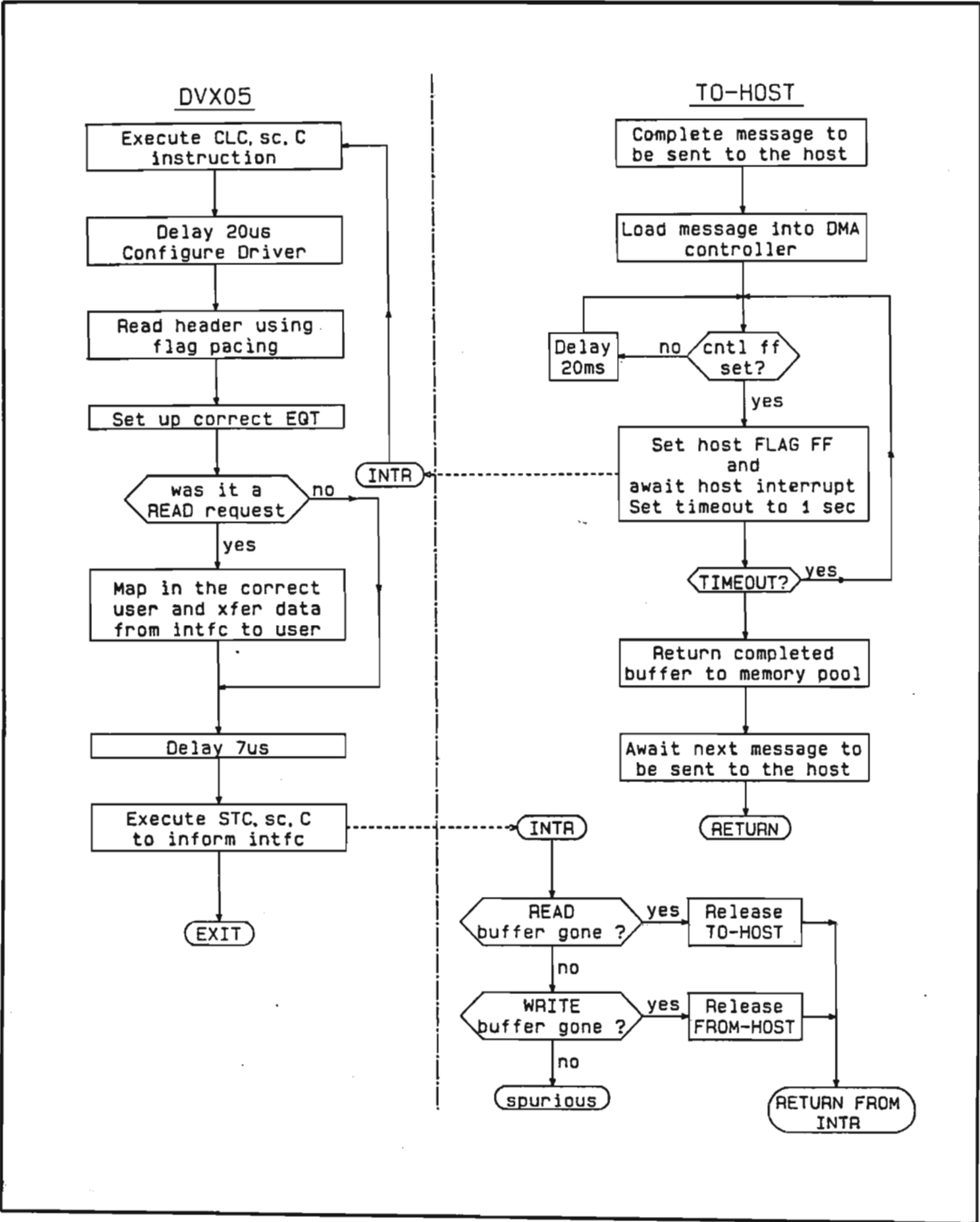


Figure I.3 Message packet input flow.

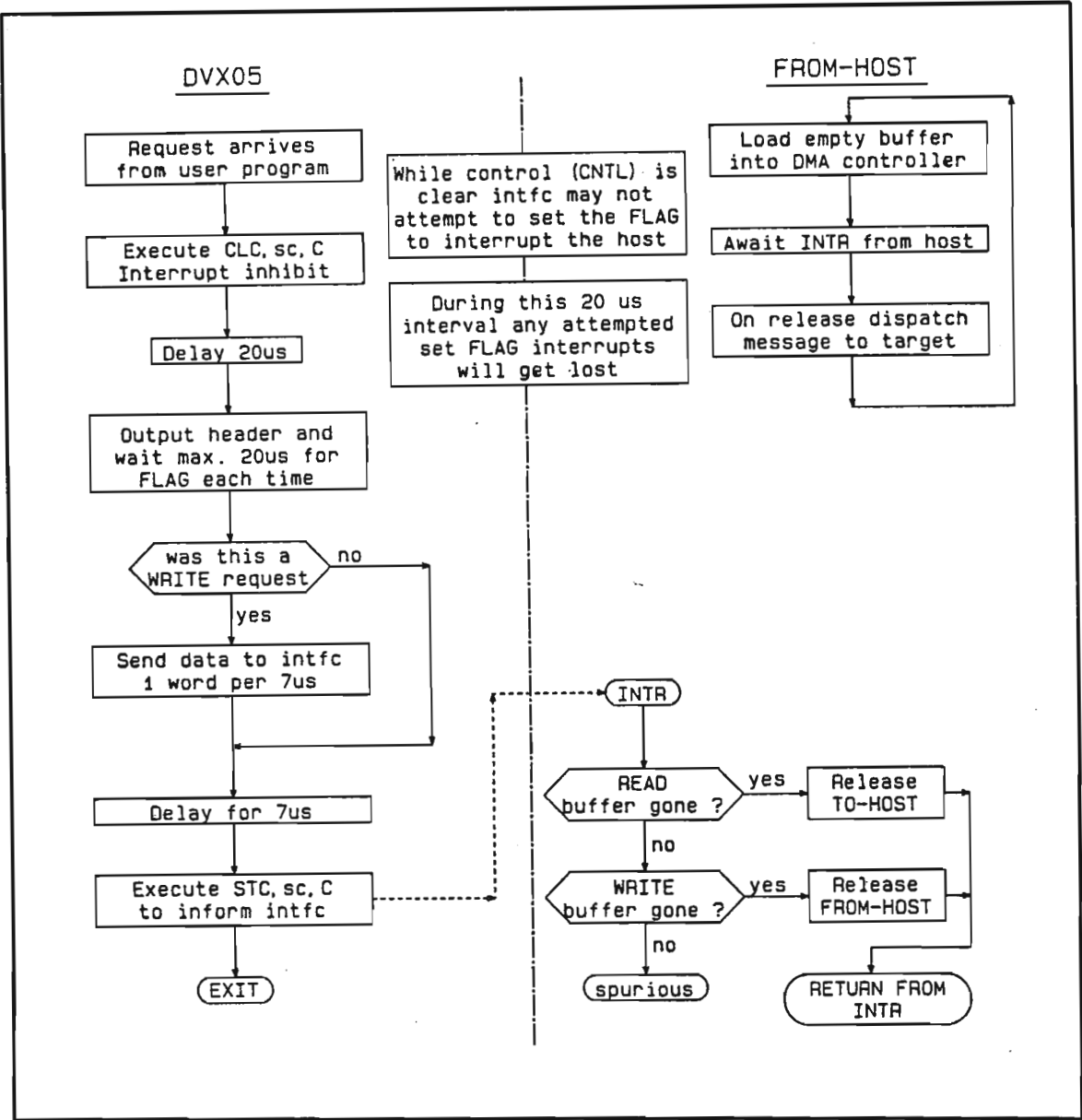


Figure I.4 Message packet output flow.

I.7 System Table Usage

Equipment table

The driver uses only the equipment table entry (see figures I.5 to I.9) for temporary storage and since most data is stored on the interface itself, no EQT extension was required. EQT words 1 to 4 are system defined and are not altered by the driver, EQT 5 is used to hold interface status information while EQT 6–10 hold all the call information sent by the system. EQT 11–13 are used exclusively by the driver for variable storage and EQT 14–15 are used for the time-out clock.

Word #	Function
1	System link word. Hold caller's ID address
2	Address of initiator entry point (IX05)
3	Address of continuator entry point (CX05)
4**	D. P S T . Subchannel . Slot number
5	Driver type & Status
6	Exec request conword (figures I.8 and I.9)
7	Exec request buffer address
8	Exec request buffer length
9	Exec request optional parameter #1
10	Exec request optional parameter #2
11	Interrupt table entry address.
12	Copy of last configure word sent
13	ID segment address of programme PRMPT (or -1)
14	Time out clock reset value
15	Time out clock

** D - Driver requires a DMA channel
 P - Driver is to process power fail
 S - Driver is to process time out
 T - Interface has just timed out

Figure I.5 EQT entry usage by DVX05

Bit #	Function
0	Break bit – set on return if a ‘break’ occurred
1	Terminal enabled
2**	Driver error encountered
3	Parity error in last request
4	Shows status of modem DSR line
5	EOT flag – set if CNTL D entered
6	Used by driver to show time-out abort in process
7	Used by driver to show buffer flush in process

** for driver errors Bits 5–3 hold error code (see figure I.7)

Figure I.6 Status Bits in EQT 5

2	Error occurred in trying to set up user map
3	No flag from interface in time – no response
4	Invalid return EQT address – message corrupted
5	No 'SYN-SYN' found from card in 11 characters
7	This port or subchannel not yet initialised

Figure I.7 Driver Return Error Codes (B5–3 of EQT 5)

0	Read request bit	These bits are mutually exclusive
1	Write request bit	
6	Ascii/Binary request (0/1)	
7	Not used	
8	Echo bit for echo of input	
9	When set with bit 10 implies programme block read	
10	Honesty bit or see bit 9	

Figure I.8 Read/Write Conword Bit Definition

Function	Code
00B	Interface clear (abort request)
11B	Space number of lines given by EQT 7
20B	Terminal enable
21B	Terminal disable
22B	Set time-out value
23B	Enable flush of all Queued I/O
30B	Configure terminal (configure word in EQT 7)

Figure I.9 Control Request Conword Bit Definition

Interrupt table

The first time the driver is entered after boot up, it has to initialise the equipment table and the interface. The one address that is necessary if the port is to handle an interactive terminal is the address of the break mode processor — PRMPT. If the system has been generated to make this port an interactive terminal the Interrupt table entry corresponding to the select code that the card is plugged into will contain the negated address of the ID segment for PRMPT. The negation is done to differentiate an ID segment address from an EQT entry address.

DVX05 calculates the address of the Interrupt table entry corresponding to the interface it is about to initialise, fetches the contents, and if negative then this address is reset positive and saved in EQT 13 as the address of the ID segment of PRMPT. The interrupt table entry is then altered to make it hold the address of the EQT entry. This is necessary to enable the driver to process interrupts. If however the interrupt table entry were positive on initialisation then the port could not be an interactive terminal and so EQT 13 is set negative to indicate this. EQT 13 is also used as the means by which the driver can determine whether to initialise the interface or not, for once initialised EQT 13 will never again be zero as it was upon first entry.

DVX05 alters no other system tables but does use two system routines and accesses the driver mapping table.

\$LIST

The system routine \$LIST, which has an indirect linkage through table area 1 (TA1), is used to schedule the programme PRMPT whenever a break mode request arrives from the interface. The address of EQT 4 is sent to \$LIST as well as the ID address of PRMPT. PRMPT uses the address of EQT 4 to determine which equipment table entry, and hence LU, requested the break mode processing. For further detail on \$LIST see the HP Driver writing Manual [36].

\$XDMP.

The system routine \$XDMP is used to map in a new user map. This routine is called from the special TA1 driver module \$DVM5 (see I.2), and the only data required by \$XDMP is the address of the first word of the EQT entry, which is passed in the A register. Before \$XDMP is called to change the user map, \$DVM5 saves the original user map contents so as to be able to restore the original map once the data transfer has been completed.

Driver Mapping table.

Whenever a read request completes, DVX05 must determine where the request came from so that the return data may be copied into the caller's request buffer. If the caller were from the system map, user mapping would be unnecessary, but should the caller be in a user map then the correct user map must be mapped in as discussed previously. To determine which map to use, word 2 of the driver mapping table entry for the EQT in use has to be examined. Should this word be zero then the system map is required otherwise the user map must be changed.

To access the driver mapping table, which is in table area 1 and easily accessible, is however a little awkward, as the address of the table is stored in a system variable named \$DVMP. Since \$DVMP is stored in table area 2 (TA2), which is inaccessible

from most user maps, an indirect link to \$DVMP was incorporated into \$DVM5 at entry point \$XDMT. DVX05 executes an enable system map and jumps to \$XDMT which accesses \$DVMP and returns the mapping table address to DVX05.

Appendix J

The 8085 Assembler — A8085

J.1 The Assembler Description

A8085 is an absolute addressing only, two pass assembler for the 8085 microprocessor. It accepts its input in standard mnemonic form as defined by INTEL corporation [37] and assembles the code into a non INTEL format binary file and listing. The source code is extensively checked for errors in both pass one and pass two of the assembler, with all error messages being descriptive rather than cryptic, and appearing immediately after the erroneous code line in the listing. All standard INTEL defined pseudo codes (or assembler directives) are recognised by the assembler, as well as several new extended directives designed to make the process of absolute code generation more effective.

This appendix serves as a user manual describing all the assembler directives and those assembler features not covered in the standard assembler reference manual. It does not cover the assembler language itself as this is adequately covered in the reference manual. However a summary of all assembler mnemonics is given for completeness.

The assembler was written in HP ALGOL to run on the Hewlett Packard 1000 series computers under RTE IVB. It consists of approximately 1400 lines of code and can assemble source files at the rate of about 2000 lines of code per minute on an otherwise unused HP 1000 machine. The assembler accepts source from either files or logical units and directs its binary output to either a file or a logical device such as a punch. Listings can only be directed to logical devices such as a printer or a terminal. The assembler is invoked in the same manner as all HP assemblers and compilers, and offers nearly all the same optional features.

J.2 Conventions

The following symbols and conventions are those used throughout this manual.

- * < >: Anything enclosed in angled brackets is optional.
- * Namr: The HP convention for a disc file name including optional security code, and cartridge reference separated by colons.

EG. &FILE1:sc:cr

J.3 Running the 8085 Assembler

In order to use the assembler a source file must be prepared using standard INTEL mnemonic operation codes and instructions. Labels (or symbols) may contain any number of characters, but only the first 10 characters are used by the assembler, hence all labels must be unique in the first ten characters.

The first line of the source file must contain a control statement holding the options and an optional heading.

Control Statement format – line 1 starting in column 1

8085,<opts> "optional header (max. 20 characters)"

where OPTS can be any set of the following characters in any order

B = Brief listing with all page formatting & comments removed

L = Listing required on list device

S = Sorted symbol table at end of pass 2

C = List clock cycles of each instruction

T = Print sequential symbol table at end of pass 1

E = Print an error summary of last 100 errors at end of assembly

(Options may be separated by commas or merely packed together. A space terminates the option list)

EG. 8085,LESC "TEXT a text processor"

The optional 20 character header may be enclosed in either single or double quotes (' or ") and is printed on the top of each page of the listing.

The last line of the source file should contain an "END" instruction in the op-code field, where the op-code may start anywhere other than column 1.

To invoke the assembler the following run string is used:

A8085, src file namr,<list device lu>,<object file>,<lines>,<OPTS>

All fields in the line are optional except for the source file name. If this is not given, the assembler will output a message on the terminal as to how the assembler may be invoked.

Source file namr: This field must be given, and can contain either a disc file namr or a device logical unit. If the input logical unit is LU1 (the terminal) then the assembler will prompt each line with an "!. In this case input terminates with an "END" opcode, a "/E" or a zero length line.

List device LU: Must be a logical device and not a disc file. If not given, the list device defaults to the user terminal.

- Object file:** This may be either a disc file name, a logical device LU or a minus sign "-". If a "-" is given and the source file name begins with an ampersand "&" then a default binary file name will be used, having the same name as the source file only with the "&" replaced by an exclamation mark "!". If the binary file cannot be found, it will be created as a type 8085 file on the same cartridge as the source file and with the same security code. If no object file name is given, no binary will be produced.
- Lines:** Allows the user to set the number of lines per page on the list device. The default is 56 lines.
- Options:** Any of the 8 options "*BLTECS*" described previously in any order. The options must not be separated by commas and the first space or comma terminates the list. If any options are given here, then all options in the control statement are reset and overridden.

J.4 Source Code Format

A standard instruction line has the following format:

<Label> OPCODE <operand 1>,<operand 2> <;comment>

- Label** if given, must start in column one and should end with a colon. It can be any length but only the first ten characters are used.
- Operands** if required, may be expressions or symbols, and where multiple operands are required they must be separated by commas. A space terminates the list.
- Comment** This field is optional but should begin with a semi-colon and must be separated from the last operand by at least one space.

If no label exists, the op code may not start in column one.

Variants on this standard format are:

LABEL:

A line may consist of a label only, in which case the label will take on the value of the programme counter at that point. This allows for multiple labels of the same value and can aid in programme legibility.

;

This is a comment only line if the ";" begins in column one.

J.5 Operand Format

An operand may consist of a constant, a symbol or an expression.

A constant is a fixed value given in one of the following formats.

- * An ASCII constant consisting of a single ASCII character enclosed in either single quotes or double quotes (' or ").
- * A decimal constant consisting only of the digits 0 through 9.
- * An octal constant consisting only of the digits 0 through 7 and either beginning with an "@" or ending with an "O" or a "Q".
- * A hexadecimal constant consisting only of the digits 0-9 and A-F and either beginning with a "\$" or ending with an "H".
- * A binary constant consisting only of the digits 0 or 1 and either beginning with a "%" or ending with a "B".

A symbol is any character string starting with an alphabetic letter (A-Z) or the special character "^" (see section J.6 iv). Any symbol used must be defined somewhere in the programme as a label of some instruction.

Either of the special symbols "." or "*" may be used in place of a normal symbol to denote the current value of the programme counter.

An expression is a string of symbols and/or constants separated by one of the following operators:

+	addition	
-	subtraction	
*	multiplication	
/	division	
!	Boolean OR function	[always performed on 16 bit data]
.	Boolean AND function	

Expressions are evaluated on a *strictly left to right basis* with a space, comma or semi-colon terminating the expression. They are calculated as single precision reals and converted to a 16 bit 2's complement integer at the end of evaluation.

Where an operand is to be used as a 16 bit value it may be in the range -32768 to +32767 and where it is required as an 8 bit value it may only be between -128 and +255. If these limits are not met then an error is reported.

J.6 Assembler Directives or Pseudo-Ops

Assembler directives are instructions to the assembler rather than for the destination machine, and are used to ease the programming task and improve programme legibility. They can be divided into several groups as given below.

Constant definition:

LABEL:	EQU operand	;Symbol definition
LABEL:	SET operand	;Allow multiple symbol redefinition
<LABEL>	DS operand	;Define storage space by increasing the PC by the value of the operand.
<LABEL>	DW operand<,operand>	;Define 16 bit word value constants stored L.S. byte first
<LABEL>	DB operand<,operand>	;Defines 8 bit constants one byte per operand. This also has the special case operand of an ASCII string of any length enclosed in either single or double quotes. The characters are stored in ASCII one per byte.

Programme counter definition:

LABEL:	ORG operand	;Define programme counter (PC)
	NSEG operand	;Save current PC in previous segment variable (if defined) and set PC and the new segment variable (LABEL) to the value of operand.
	RSEG segment variable	;Save current PC in previous segment variable (if defined) and restore PC to value of new segment variable.
	SVPC segment variable	;Save current PC in segment variable.

Optional code inclusion:

IFZ operand	;If operand evaluates to zero then include the code between this statement and the next ELSE or ENDF statement.
IFNZ operand	;If operand evaluates to non zero then include all code between this statement and the next ELSE or ENDF statement.
ELSE	;Reverse sense of last conditional assembly statement.IE If previous code was excluded then include following code until ENDF and vice versa.
ENDF	;Terminates an IFZ, IFNZ, or IF(Z/NZ)—ELSE construct.

Source file handling:

MERG file	;Opens the given file and includes it in the ;assembly. If file begins with an assembler ;control statement, the control statement is ;ignored.
SCAN file	;Opens the given file and includes it in ;pass one only. No listing or object code is ;generated but symbols defined in this file ;are included in the symbol table.

Listing control:

SUP	;Suppress listing of second and all subsequent ;operands in each DB and DW statement.
HED <string>	;Start a new page on the list device and add ;the string to the header line.
SKP	;Skip to a new page and print heading
PAGE	;Skips same as SKP
SPC n	;Causes n lines to be spaced on the list device
UNL	;Terminates listing on the list device
LIST	;Resumes listing only if the control statement ;option specified a listing.

Several of the directives given above are non-standard and require further explanation.

SET. This command allows a symbol to be redefined during the flow of the assembly. It is the only statement which allows a single label to be used more than once and is useful for changing some constant in a piece of code which gets used more than once (see also the file merge section for this).

Code segmentation. The 'NSEG' and 'RSEG' directives allow the source code to be segmented into several segments of memory, such as ROM, RAM, I/O etc. Segments are defined using the 'NSEG' statement which sets the 'LABEL' as a segment name to the value of the operand. The programme counter is also set to the same value after it's previous value has been saved in the previous segment variable (if one was defined). To restore code generation to a previously defined segment requires the use of the 'RSEG' directive giving the segment name. The current PC is saved in the current segment variable and then the new segment variable value is loaded into the PC. This allows code generation to continue directly from where it last left off in the new segment.

For example:

Assume a ROM starts at address 0000H, a RAM starts at 2000H, and the programmer wishes all fixed code to go into ROM while all variable data is to be defined into RAM.

Typical code may appear as:-

```

RAM:      NSEG 2000H      ;Define segments
ROM:      NSEG 0000H      ;
;
segment   BEGIN:         MVI A,3FH      ;Code goes into ROM      which was last
                                LXI HL,1757      ;defined. Now to use a RAM variable which
                                STA TEMP1         ;has not yet been defined!
                                SHLD VARIABLE      ;
;
                                RSEG RAM          ;Define the ram based variable here so that
TEMP1:    DS 1            ;it will be kept local to the module that
VARIABLE: DS 2            ;uses it.
;
                                RSEG ROM          ;That done, resume code into ROM
                                LDA VARIABLE      ;
                                etc

```

This saves the programmer from having to keep track of addresses, and allows RAM storage to be defined with the code that uses it.

Optional code inclusion. The IF-ELSE-ENDF construct allows a source file to contain code which may be included depending upon some option or other.

As an example assume that two slight variations of some programme are required, one used in Machine A and the other in Machine B. The sections of code unique to each machine could then be coded into the same file as shown below, which saves the problem of maintaining near duplicate files and keeping both updated.

For example:

```

MACHINE:  SET 'A'          ;(set machine being used)
;
;      ( common code for all options )
;
IFZ MACHINE-'A'          ;test for machine A
;
;      ( code for machine A )
;
ELSE
;
IFZ MACHINE-'B'          ;test for machine B
;
;      ( code for machine B )
;
ELSE
;
;      ( this code if neither A or B )
;
ENDIF          ;end of inner IF-ELSE
ENDIF          ;end of outer IF-ELSE

```

Notes:

- a) IFx-ELSE-ENDF constructs can be nested to a maximum depth of 10 levels.
- b) The ELSE clause is optional. An IFx-ENDF construct is valid.

Multiple File handling. Since the assembler is an absolute assembler and does not produce relocatable output code, all the source code associated with any programme must be assembled at once to maintain correct symbol linkage. On large programmes this creates large source files which are unwieldy and also makes it difficult to maintain unique meaningful labels. The MERG function provides a simple solution to these problems by allowing source code to be spread across multiple source files which can then all be included in the assembly at assembly time only.

A useful feature of the MERG command when used in conjunction with the SET command is that it allows the same file to be used several times with different variables if several slightly different copies of a certain file are needed. An example of this is where the same I/O port handler is needed for each of several identical ports.

For Example:

```

^PORT:      SET 0D0H           ;Set address for port 1
            MERG &PHNDL
^PORT:      SET 0D2H           ;Then set for port 2
            MERG &PHNDL
            etc

```

where &PHNDL may look like:

```

BEGIN:      MVI A, DATA
            OUT ^PORT          ;Output data to port defined
            RET                 ;by previous SET directive.
DATA:       DB 0AH
            END

```

An added feature of the MERG option is that *all symbols defined within the scope of a file* are accessible only to instructions within that file unless *specifically defined as being global* to all files. To assign a label as a global symbol, the label name must be preceded by a carat symbol '^'. This then allows code in any file to reference the symbol.

For Example:

Assume a programme consists of several subroutines each stored in separate files. The files may then look like this:

File 1 &MAIN:SC:CR

```

8085,LE "SAMPLE PROGRAM"
;
^RAM:      NSEG 2000H           ;Start RAM segment — make it global
^BUFF:     DS 32               ;define a global data buffer
          SPC 2
^ROM:      NSEG 0000H           ;Start ROM segment — also global
BEGIN:     LXI SP,^RAM+256      ;define a stack — local label BEGIN

          "
          CALL ^SUB1            ;call subroutine from other files
          CALL ^SUB2
          JMP BEGIN
          MERG &SUBR1:SC:CR      ;merge in the subroutine files
          MERG &SUBR2:SC:CR
          END

```

File 2 &SUBR1:SC:CR

```

^SUB1:     LDA ^BUFF            ;Start of subroutine — global label
          STA TEMP              ;saves into local variable
          RET
          RSEG ^RAM
TEMP:      DS 1                 ;Now define the local variable in RAM
          RSEG ^ROM              ;segment and restore ROM segment
          END

```

File 3 &SUBR2:SC:CR

```

^SUB2:     LDA TEMP             ;Start of SUB2 — label must be global
          STA ^BUFF
          RET
          RSEG ^RAM
TEMP:      DS 1                 ;Define local variable in RAM segment
          RSEG ^ROM              ;and restore ROM segment. NOTE that
          END                   ;this is a different TEMP to that in
                                ;SUB1 since it's local to its own file

```

J.7 Summary of Assembler Instructions

ARITHMETIC AND LOGICAL GROUP				DATA TRANSFER GROUP											
Add*		Increment**		Logical*		Move		Move (cont)		Move Immediate					
ADD	A 87	INR	A 3C	ANA	MOV	A,A 7F	MOV	E,A 5F	MVI	A, byte 3E					
	B 80		B 04			A,B 78		E,B 58		B, byte 06					
	C 81		C 0C			A,C 79		E,C 59		C, byte 0E					
	D 82		D 14			A,D 7A		E,D 5A		D, byte 16					
	E 83		E 1C			A,E 7B		E,E 5B		E, byte 1E					
	H 84		H 24			A,H 7C		E,H 5C		H, byte 26					
ADC	L 85	INX	L 2C	XRA	MOV	A,L 7D	MOV	E,L 5D	LXI	L, byte 2E					
	M 86		M 34			A,M 7E		E,M 5E		M, byte 36					
	A 8F		B 03			B,A 47		H,A 67		Load Immediate					
	B 88		D 13			B,B 40		H,B 60							
	C 89		H 23			B,C 41		H,C 61							
	D 8A		SP 33			B,D 42		H,D 62		B, dble 01					
SUB	E 8B	Decrement**		ORA	MOV	B,E 43	MOV	H,E 63	Load/Store	D, dble 11					
	H 8C	DCR	A 3D			B,H 44		H,H 64		H, dble 21					
	L 8D		B 05			B,L 45		H,L 65		SP, dble 31					
	M 8E		C 0D			B,M 46		H,M 66							
	A 97		D 15			C,A 4F		L,A 6F							
	B 90		E 1D			C,B 48		L,B 68							
SBB	C 91	DCX	H 25	CMP	MOV	C,C 49	MOV	L,C 69	LDAX B 0A						
	D 92		L 2D			C,D 4A		L,D 6A		LDAX D 1A					
	E 93		M 35			C,E 4B		L,E 6B		LHLD adr 2A					
	H 94		B 0B			C,H 4C		L,H 6C		LDA adr 3A					
	L 95		D 1B			C,L 4D		L,L 6D		STAX B 02					
	M 96		H 2B			C,M 4E		L,M 6E		STAX D 12					
Double Add †	A 9F	Specials		Arith & Logical Immediate	MOV	D,A 57	MOV	M,A 77	SHLD adr 22						
	B 98	DAA*	A 27			D,B 50		M,B 70							
	C 99		CMA 2F			D,C 51		M,C 71		STA adr 32					
	D 9A		STC† 37			D,D 52		M,D 72							
	E 9B		CMC† 3F			D,E 53		M,E 73							
	H 9C					D,H 54		M,H 74							
RST	L 9D	Rotate †		CPI byte FE	XCHG	D,L 55	EB	M,L 75							
	M 9E					D,M 56									
	B 09		RLC 07												
	D 19		RRC 0F												
RST	H 29	Restart	RAL 17	Call											
	SP 39		RAR 1F												
BRANCH CONTROL GROUP					I/O AND MACHINE CONTROL										
Return				Jump				Control				Stack Ops			
RET C9				JMP adr C3				DI F3				PUSH			
RNZ C0				JNZ adr C2				EI FB				B C5			
RZ C8				JZ adr CA				NOP 00				D D5			
RNC D0				JNC adr D2				HLT 76				H E5			
RC D8				JC adr DA								PSW F5			
RPO E0				JPO adr E2								B C1			
RPE E8				JPE adr EA								D D1			
RP F0				JP adr F2								H E1			
RM F8				JM adr FA								PSW* F1			
				PCHL E9											
Restart				Call				New Instructions (8085 Only)				POP			
0 C7				CALL adr CD				RIM 20				B C1			
1 CF				CNZ adr C4				SIM 30				D D1			
2 D7				CZ adr CC								H E1			
3 DF				CNC adr D4								PSW* F1			
4 E7				CC adr DC											
5 EF				CPO adr E4											
6 F7				CPE adr EC											
7 FF				CP adr F4											
				CM adr FC											
												XTHL E3			
												SPHL F9			

Courtesy of INTEL Corporation 8085 Reference Manual

J.8 Assembler Structure

This section covers the actual structure of the assembler programme and describes how it was implemented. The source code was written in HP ALGOL, a sub-set of ALGOL 68, and contains comments to cover the data structures used. However since ALGOL is an excellent language from a self documenting point of view, there were few comments needed to describe programme flow.

Opcode Lookup Table

One of the original reasons for writing this assembler was to achieve faster assembly times compared to a slow, somewhat limited, assembler available previously. Thus the opcode lookup table was implemented using a simple hashing algorithm. Since an opcode table is a fixed table, a small programme was written to vary the size of the lookup table and the hashing function until an optimal table was achieved. The final table resulted in 96 out of the 103 instructions achieving a 'hit' on the first hash, and the remaining 7 instructions (all infrequently used pseudo-ops) only suffering a single clash.

The hashing function consists of two divisions and one addition to determine the index key for a 367 element sparse lookup table which provides the index of the opcode in the opcode table. When a clash occurs, the preceding value in the index table is taken instead of performing a re-hash. The resultant sparse index lookup table approach required a table with 264 zero entries, but the waste of 264 words of memory was considered a small price to pay for the gain in speed.

Symbol table hashing

The symbols (or labels) are hashed into a symbol table which was made as large as possible with the proviso that its length is always a prime number. This proviso ensures that during re-hashing on clashes, no step value could ever cause cycling by being a factor of the table length. The actual hashing function involved reducing each word (character pair) of the label modulo the table length and then adding this to the next word before performing the next modulo operation. Each label name comprises 6 words, 5 to hold the 10 character name itself and one to hold a file pointer for the local/global feature. Thus hashing involves 6 divisions and 6 additions to produce the hash key into the symbol table.

To accommodate clashes which are bound to occur, a hash step is calculated in a similar fashion to the hash key only using a different value for the modulo operation. This ensures that different labels which produce the same hash key are unlikely to produce the same hash step value and so should not clash on a re-try.

Re-tries are implemented by successively adding the hash step value to the hash key modulo the table length until such time as the symbol is found, or a clash depth of twenty tries has been achieved. This limit to the clash depth results in a lower utilization of the symbol table (about 85% maximum) but does limit the speed degradation that will occur as the table becomes full.

Intermediate data storage

During pass one, source lines are read in from the source file, and unpacked in R1 format into a 1024 word buffer. The line is then fully parsed and the pointers to all the fields are also stored in the same buffer. Finally the opcode is decoded from the lookup table and its type, length, execution time and value are stored in the buffer. All lines are stored sequentially into this buffer until it is filled when it is then written to a scratch file on disc. This is read back in pass two making pass two execution much faster than would have been the case if intermediate storage had not been used and all the line decoding had had to be performed again in pass two.

The unpacking of characters from A2 to R1 format was performed only once rather than doing character fetches from an A2 array every time a character is checked. This process speeds up execution but doubles the amount of temporary disc storage used. This could have been overcome by repacking the 1K buffer prior to storage, but this was felt an unnecessary waste of time.

Multiple file handling

Two major tables were used to cope with the multiple file facility. An array FNAMES was used to hold the names of all the files used in the assembly, with each new file being added to the end of the list. This list limits the maximum number of files to 63, and the entry number of the file in this list is used as the 6th word of a label when storing the label into the symbol table, so giving the local/global feature.

To hold the status of open files a large packing buffer (FILES) was used to hold four file data control blocks (DCBs), allowing a maximum nesting depth of only four files. For each file that is open, the packing buffer also holds the current line number, a pointer to the file name in the FNAMES list and a type indicator to indicate whether the file is a disc file or not, and whether it is open for scanning or must be merged completely into the assembly (see section J.6).

Conditional assembly options

To cope with the 'IF-ELSE-ENDF' conditional assembly feature, a 10 word stack was used allowing up to 10 levels of nesting. For every new level of 'IF' nesting encountered a new element of +1 for a false 'IF' condition or -1 for a true 'IF'

condition is pushed onto the stack, and a flag (SKIPFLAG) is set accordingly causing all subsequent code after a false 'IF' to be ignored. When SKIPFLAG is true, each opcode is checked to see if it is an 'IF', an 'ELSE' or an 'ENDF' statement and if none of these, then the line is excluded from further assembly by not adding it to the intermediate packing buffer.

When an 'ELSE' is encountered, the top stack element is changed from -1 to +2 or +1 to -2, thus indicating an 'ELSE' condition and changing the state of the SKIPFLAG. The value change from 1 to 2 allows checking for the illegal condition of two 'ELSE' clauses in succession with no intervening 'ENDF'.

Finally an 'ENDF' statement is used to pop the top element off the stack irrespective of whether it has an absolute value of 2 from following an 'ELSE' statement, or an absolute value of 1 from an 'IF' statement. The assembler checks for excess 'ENDF' statements, unclosed constructs and incorrectly nested statements, printing appropriate error messages in each case.

General

The assembler was written in a very modular form with all specific processor references being confined to as few procedures as possible. This enabled the bulk of the code to be usable in constructing cross assemblers for other micro-processors, and it has already been used as the core of a Z80 cross assembler.

Appendix K

The RMUX Users Manual

K.1 Introduction

This appendix contains summarised information for two distinct types of users.

For the System Manager, the process of connecting the interface to terminals and of generating the software into the system is described, followed by the required boot up configuration procedures.

For the terminal user, those features which affect keyboard operation are described, particularly those which differ from the standard driver DVR05 features. The ways in which the terminal can be re-configured may be of value to the more advanced terminal user.

K.2 Generation of RMUX into RTE 4B

To include an RMUX interface into RTE 4B, two software modules must be generated into the system, while the third module may be generated in or loaded on line. These are included in the generator answer file at the relocate phase as:-

```
REL,    %DVX05::CR      ,* Main RMUX driver
REL,    %$DVM5::CR      ,* RMUX mapping handler
REL,    %4AUTX::CR      ,* Power fail – auto restart for RMUX

      * %4AUTR must be removed from the relocate phase to enable %4AUTX to load.
      * %4AUTX can be loaded on line if desired.
```

At the parameter input phase, force \$DVM5 into TA1 with:-

```
$DVM5,15                      ,* Force map handler into TA1
```

At the EQT definition phase, assign 4 contiguous equipment table entries, all pointing to the RMUX card select code as follows:-

E.G. Assigning EQT 12-15 to RMUX interface in slot 17B

```
17,DVX05,B                    ,* EQT 12 RMUX Port 0
17,DVX05,B                    ,* EQT 13 RMUX Port 1
17,DVX05,B                    ,* EQT 14 RMUX Port 2
17,DVX05,B                    ,* EQT 15 RMUX Port 3
```

Then in the device reference table definition phase allocate the four ports unit numbers:-

E.G. Assigning Lu's 20-23 to these Eqt's

12,0,	* LU 20 RMUX Port 0
13,0,	* LU 21 RMUX Port 1
14,0,	* LU 22 RMUX Port 2
15,0,	* LU 23 RMUX Port 3

Lastly in the interrupt table definition assign the select code of the interface to programme PRMPT:-

17, PRG, PRMPT	*RMUX quad terminal interface
----------------	-------------------------------

The reason for a new auto-restart programme is that after a power failure the interface will have lost it's configuration setting which must be re-established with a CN,lu,30B,nnnn (configure word) call. The modified version of AUTOR performs a CN,lu,30B,0 call on each terminal before sending the usual power fail message. This causes DVX05 to use the configure value last sent to the interface, (stored in EQT 12), and hence re-configure the interface correctly. Until this is done, the interface will ignore all requests and will not process them.

K.3 Configuration Options for RMUX ports

After a boot-up or a power failure the interface must be initialised by using the configure call (control 30B). Until a port has been initialised, all requests to the port will be ignored.

The configure control call (CN,lu,30B) uses a single 16 bit variable to define all the terminal operating characteristics (see figure K.1) as well as defining the relationship between port number and equipment table entry.

Once the first configure call has been executed, subsequent calls allow all terminal characteristics to be altered, except the EQT-port relationship. A control 30B call with a zero parameter will cause DVX05 to use the same configuration parameter that was sent previously. This is normally used after a power failure by the auto-restart programme AUTOR.

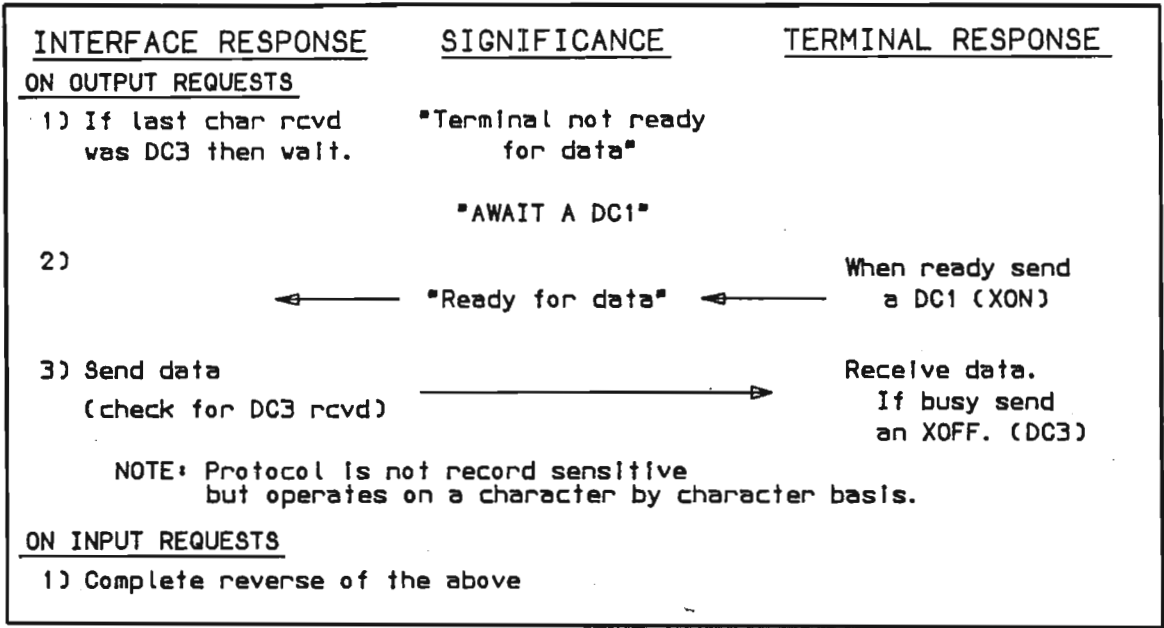


Figure K.2 XON/XOFF style handshake protocol

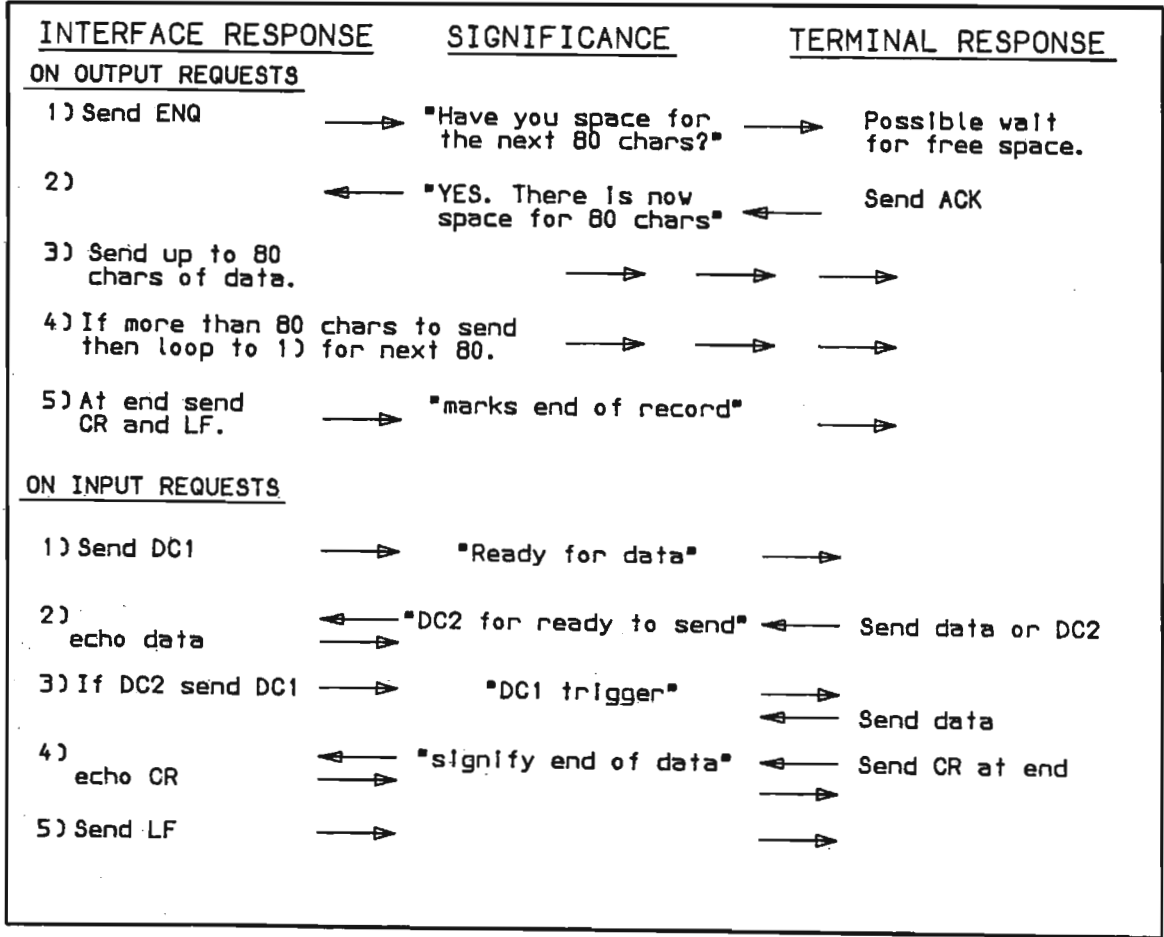


Figure K.3 HP style handshake protocol

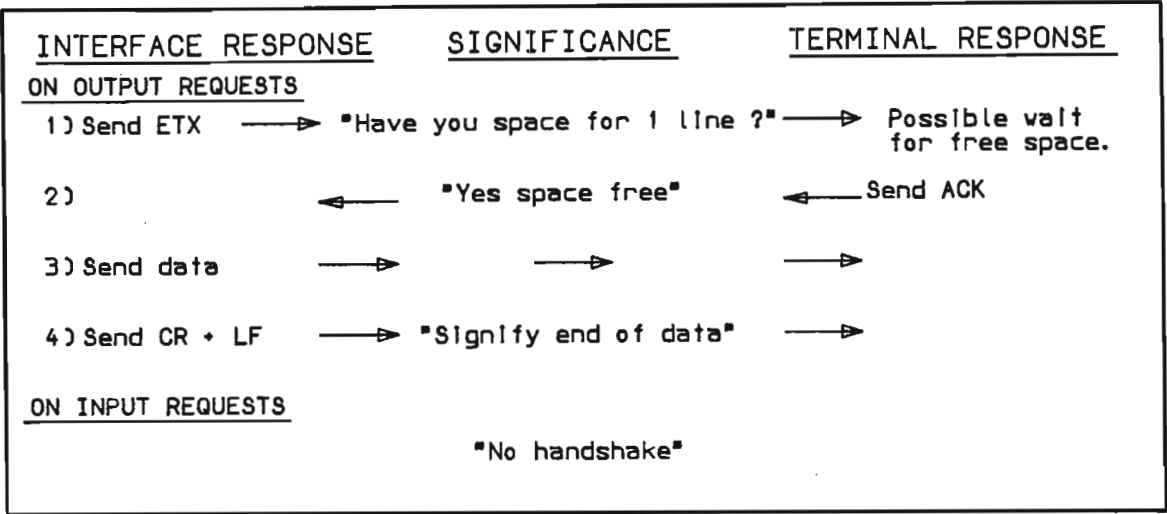


Figure K.4 QUME style handshake protocol

Since each port must be initialised after boot up, the configuration calls should be placed into the WELCOM file [38] so that the terminals will always be configured correctly after boot up. An example of typical entries for a WELCOM file is shown below:

```
:CT,20,30B,112534B
:CT,21,30B,112535B
:CT,22,30B,112536B
:CT,23,30B,112537B
:CT,20,20B,, Terminal 20 enabled (9600 baud, odd parity, 7Data, 1sb)
:CT,21,20B,, Terminal 21 enabled (9600 baud, odd parity, 7Data, 1sb)
:CT,22,20B,, Terminal 22 enabled (9600 baud, odd parity, 7Data, 1sb)
:CT,23,20B,, Terminal 23 enabled (9600 baud, odd parity, 7Data, 1sb)
```

The configuration parameter 112534B sets up LU20 to correspond to port zero with terminal parameters set to odd parity, 9600 baud, 1 stop bit and 7 data bits using an HP style handshake.

The :CT,lu,30B calls are configuration calls to set up the interface cards operating conditions while the :CT,lu,20B calls enable the terminals to operate in interactive terminal mode with break mode access, this being a standard RTE IVB control call.

K.4 RMUX features available from the Keyboard

Record length. The RMUX interface will accept data from the keyboard in either character or block mode, providing each record does not exceed 255 characters.

Break mode access. To gain the attention of the break mode processor PRMPT, strike the 'space' bar at any time except when the interface is performing input. Alternatively enter the cancel character (ASCII 'CAN' or CNTL X) at any time, even during input to gain PRMPT's attention. This is very useful for eliminating programmes that dominate terminal input and cannot be terminated from within the programme. *Note that when not in input mode the interface will ignore all characters except 'SPACE' and 'CAN'.*

Changing the ECHO. The special character 'SUB' (CNTL Z) can be used during normal input to toggle the echo flag. This character is not passed through to the input buffer, but if echo was enabled on a terminal prior to a CNTL Z being entered, it will be disabled after the CNTL Z and vice versa. This feature has its value for entering secret information such as passwords which should not be displayed on the terminal screen.

EOT status. The special character 'EOT' (CNTL D) when entered during the normal input process, will cause DVX05 to execute an immediate return to the caller with a zero transmission log and the EOT bit (BIT 5) of the status word set. The interface will not send a return or a line feed (CR or LF) and any input characters entered into the source record prior to the CNTL D will be lost. Entering a return (CR) as the first character in any record will also execute an immediate return with a zero transmission log except that in this case the EOT bit will not be set. There are several other special characters which are processed in a special fashion and figure K.5 summarises these and their effect.

Character	Key in	Effect
CR	Return	Terminates an input record. CR, LF echoed.
LF	Line feed	Echoed. excluded from request buffer.
US	CNTL _	Echoed. excluded from request buffer.
RS	CNTL ^	- as for CR -
EOT	CNTL D	Immediate return, EOT bit set, zero trans. log.
DEL	Delete	Send CR,Esc,K,\. Clears line and leaves prompt
BS	Backspace	Back up character pointer. If first character in the line then transmit the delete sequence.
SUB	CNTL Z	Toggle ECHO state on/off
CAN	CNTL X	Set break bit in status to schedule PRMPT.

Figure K.5 Special character processing for normal keyboard input

ASCII/BINARY bit. In ASCII mode input is only terminated by receiving a return (CR) upon which a (CR, LF) is echoed. Should the input string be longer than the request buffer, then the buffer will be filled and all subsequent characters ignored until the 'CR' character is detected.

ASCII output is terminated by a newline pair (CR,LF) unless the special case occurs where the last character in the record is an underline ('_'). In this case the underline will not be output, nor will a newline (CR,LF).

In BINARY mode there is no special character processing and all input is stored in the buffer until the buffer is full, at which time a return to the caller is made. If an odd number of characters is requested, then in Binary mode the last word is right padded with a null whereas in ASCII mode it is padded with an ASCII 'space'. Binary mode output causes all characters (bytes) to be sent with no special processing and no CR or LF.

ECHO bit. If the ECHO bit is set then all input is echoed back to the terminal otherwise no echo occurs. The state of this bit can be toggled during an input line by using the CNTL Z key. This function is independent of the ASCII/Binary bit and is ignored during output.

HONESTY mode. In HONESTY mode input, (IE. B10 set, B9 clear), all special characters are stored in the buffer without any special processing. However input still terminates upon receipt of a return character and not a on buffer full condition.

Output in HONESTY mode causes all characters in the record to be output irrespective of value. No new line characters (CR,LF) are sent to the terminal at the end of the record. This bit is only effective in ASCII mode.

BLOCK READ mode. In BLOCK READ mode (Both B9 and B10 set) the DC1 which precedes any read request (HP handshake only) will trigger a block of data from the terminal. This data is accepted without any special processing, is not echoed, but is terminated by the transmission of a CR,LF pair. This mode is used exclusively for HP terminals strapped for line mode block reads and is only effective for reads. For write requests it is treated as Honesty mode since bit 10 is set.

K.6 Signal Connections and Wiring Tables

Each port has eight signals plus a common, and these are all buffered through RS232 line drivers or receivers and brought out via a 48 way edge connector. The signal pin numbers are shown in table K.1.

Table K.1 Signal Pinouts on terminal interface

TRACK SIDE	PIN No.		COMPONENT SIDE
COMMON	A	1	COMMON
DSR 3	B	2	CTS 3
CLKIN 3	C	3	/RXD 3
CTS 2	D	4	DTR 3
DSR 2	E	5	CLKOUT 3
/RXD 2	F	6	RTS 3
DTR 2	H	7	CLKIN 2
/TXD 3	J	8	RTS 2
CLKOUT 2	K	9	-
-	L	10	/TXD 2
-	M	11	-
-	N	12	-
CTS 1	P	13	-
-	R	14	DSR 1
-	S	15	-
CLKIN 1	T	16	/RXD 1
RTS 1	U	17	DTR 1
-	V	18	-
/TXD 1	W	19	CLKOUT 1
/RXD 0	X	20	CLKIN 0
/TXD 0	Y	21	RTS 0
CLKOUT 0	Z	22	DTR 0
DSR 0	AA	23	CTS 0
COMMON	BB	24	COMMON

** This is the EPROM end of the connector **

* the '/' sign indicates inversion or negative true logic

K.7 Definition of the RS 232 – C Signals

/TXD Transmit Data. This is the negative true transmitted data signal with voltage levels according to EIA RS232 standards.

/RXD Received Data. Negative true received data input line.

RTS Request to Send. This positive true output will be set true (positive) whenever the interface desires to transmit data. It will be maintained true for the entire duration of a record, and then will be set false again after the last character of the record has been sent.

CTS Clear to Send. Positive true input signal to the interface. This signal *must* be true for any data to be transmitted. Normally CTS is set true in response to RTS whenever the modem is ready to accept data. If CTS is set false in the middle of transmitting a character, then transmission will terminate one character after the character which was being transmitted when CTS went false. This is due to the double buffering feature of the interface output USART ^[39]. When CTS is once again set true, transmission *resumes with the re-transmission* of the last character – IE. the one that was sent while CTS was false. If the interface is driving a terminal directly and not through a modem then RTS and CTS should be connected together.

DTR Data Terminal Ready. Positive true output set true prior to the first interface transaction and maintained true continuously thereafter.

DSR Data Set Ready. Positive true input, the state of which is indicated in the EQT status word bit 4. It has no effect upon the interface operation.

CLKOUT Programmable RS-232 level output clock running at 16 times the baud rate set by the last configure command (see figure K.1).

CLKIN This input must be driven by an RS-232 level compatible square wave clock with a frequency of 16 times the nominal baud rate. The most common connection for this signal is to connect it to CLKOUT enabling the user to set the terminal baud rate under programme control. However, if connected to a modem this could be driven by the modem clock.

Table K.2 below gives the connections of each signal to an RS-232 compatible connector – a D25 type. The wiring shown is for the interface to appear as Data Terminal Equipment (DTE) as defined by the RS-232 standard [40].

Table K.2 DTE connections according to RS232C

PIN no.	SIGNAL
1	Protective ground – should connect to system earth
2	/TXD Transmitted data from Interface
3	/RXD Received data to Interface
4	RTS Request to Send
5	CTS Clear to Send
6	DSR Data Set Ready
7	COMMON
15	CLKIN Transmit clock from modem
17	CLKIN Receive clock from modem
20	DTR Data Terminal Ready
24	CLKOUT Clock signal to Modem

REFERENCES

- 1 Hewlett Packard Company, *A Pocket Guide to Interfacing HP Computers*. (1970)
- 2 Hewlett Packard Company, *HP12531 Buffered TTY Interface. Installation, Service and Reference Manual*. (1972)
- 3 Hewlett Packard Company, *HP12894A-E01/E02. Multiplexed Input/Output Accessory Kit. Operating and Service Manual*. (1976)
- 4 Sperry Univac Corporation, *PPLS 1100 System User Reference*. (1979)
- 5 INTEL Corporation, *8257/8257-5 Programmable DMA Controller*. p6-125, MCS80/85 Family User's Manual. (1979)
- 6 E.W.Dijkstra, *Co-operating Sequential Processes*, F.Genuys, ed., Academic Press, New programming Languages, York, NY. (1968)
- 7 Per Brinch Hansen, *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ. (1973)
- 8 R Peplow, *RMUX Application Code*. Source Code Unpublished. (1979)
- 9 Hewlett Packard Company, *HP2621b Interactive Terminal Owners Manual*. pp5-3 to 5-5. (1982)
- 10 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. pp3-26 to 3-30. (1980)
- 11 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. p3-26. (1980)
- 12 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. pp2-5 & 3-2. (1980)
- 13 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. pp2-5 & 2-6. (1980)
- 14 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. pp3-22 to 3-25. (1980)

- 15 Hewlett Packard Company, *HP12531 Buffered TTY Interface. Installation, Service and Reference Manual.* (1972)
- 16 Hewlett Packard Company, *RTE Driver DVR00 for Multiple Device System Control – Programming and Operating Manual.* (1975)
- 17 Hewlett Packard Company, *HP12620 Breadboard Interface Kit – Operating and Service Manual.* (1970)
- 18 Hewlett Packard Company, *A Pocket Guide to Interfacing HP Computers.* (1970)
- 19 Hewlett Packard Company, *HP12894A-E01/E02. Multiplexed Input/Output Accessory Kit. Operating and Service Manual.* (1976)
- 20 Intersil Inc., *IM6402/IM6403 Universal Asynchronous Receiver Transmitter (UART)* p2–3 Hot Ideas in CMOS. (1983/1984)
- 21 Hewlett Packard Company, *HP2754A/B Teleprinter – Operation and Service Manual.* (1970)
- 22 Honeywell Corporation, *Model 112 Lineprinter Maintenance Manual.* (Modified by R Peplow 1979)
- 22 Honeywell Corporation, *Model 112 Lineprinter Maintenance Manual* pp2–3 to 2–20. (Modified by R Peplow 1979)
- 22 Honeywell Corporation, *Model 112 Lineprinter Maintenance Manual* p3–45. (Modified by R Peplow 1979)
- 25 INTEL Corporation, *8237/8237–2 High Performance Programmable DMA Controller.* p6–101, MCS80/85 Family User's Manual. (1979)
- 26 Hewlett Packard Company, *RTE Operating System Driver Writing Manual.* (1980)
- 27 Hewlett Packard Company, *A Pocket Guide to Interfacing HP Computers.* (1970)
- 28 Hewlett Packard Company, *RTE Operating System Driver Writing Manual.* pp2–5 & 2–6. (1980)
- 29 INTEL Corporation, *8257/8257–5 Programmable DMA Controller.* p6–125, MCS80/85 Family User's Manual. (1979)
- 30 INTEL Corporation, *8257/8257–5 Programmable DMA Controller.* p6–123, MCS80/85 Family User's Manual. (1979)

- 31 Hewlett Packard Company, *A Pocket Guide to Interfacing HP Computers*. p3–16. (1970)
- 32 INTEL Corporation, *8257/8257-5 Programmable DMA Controller*. p6–125, MCS80/85 Family User's Manual. (1979)
- 33 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. pp3–26 to 3–30. (1980)
- 34 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. p2–3. (1980)
- 35 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. pp3–1 to 3–8. (1980)
- 36 Hewlett Packard Company, *RTE Operating System Driver Writing Manual*. p3–23. (1980)
- 37 INTEL Corporation, *8080/8085 Assembly Language Programming Manual*. (1979)
- 38 Hewlett Packard Company, *HP92068A RTE-IVB System Managers Manual* p5–18. (1980)
- 39 INTEL Corporation, *8251 Programmable Communication Interface*. p6–160, MCS80/85 Family User's Manual. (1979)
- 40 Electronic Industries Association, *EIA Standard RS-232-C*. (1969)