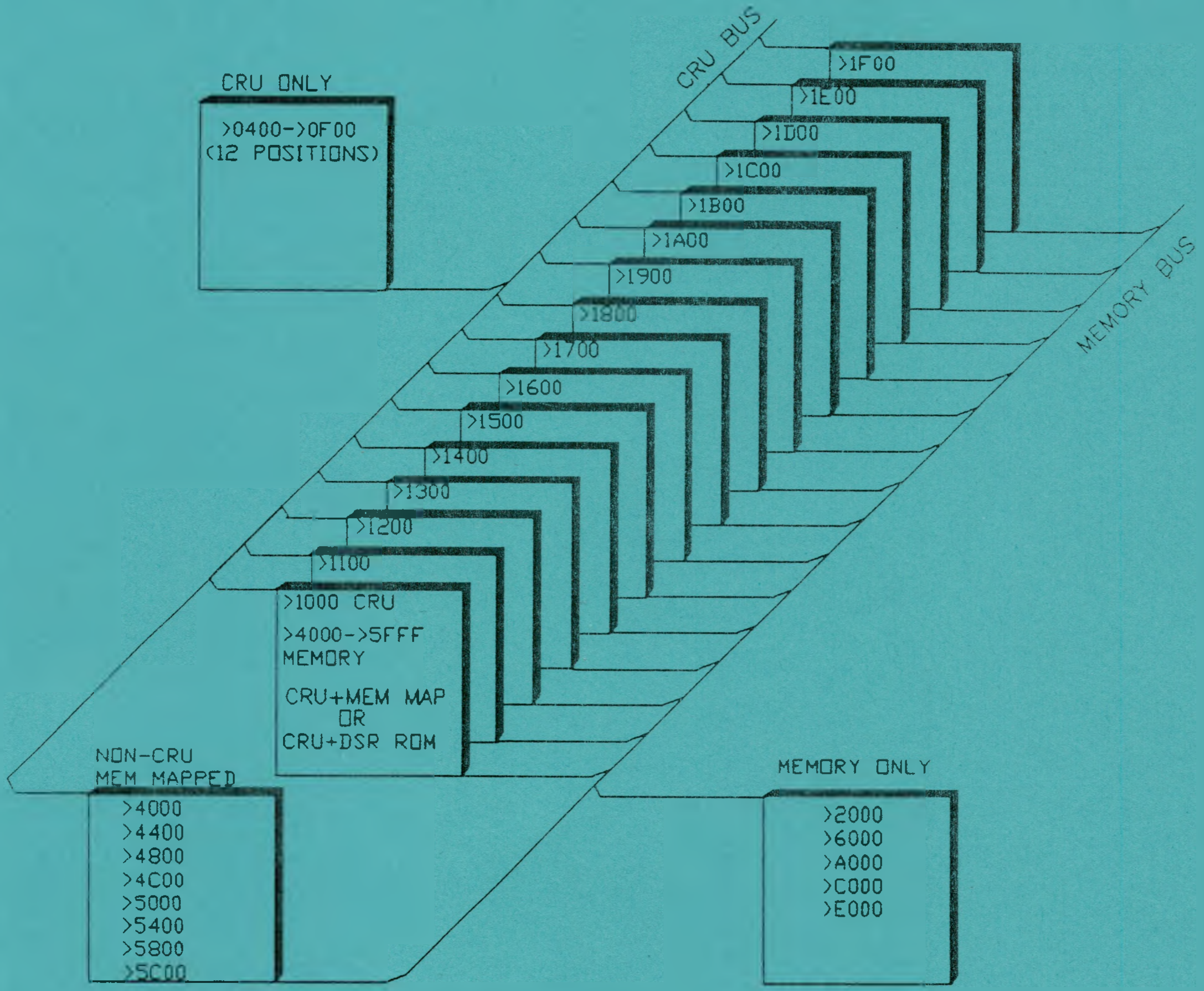
INTERFACE STANDARD & DESIGN GUIDE

for TI 99/4A Peripherals

SECOND EDITION

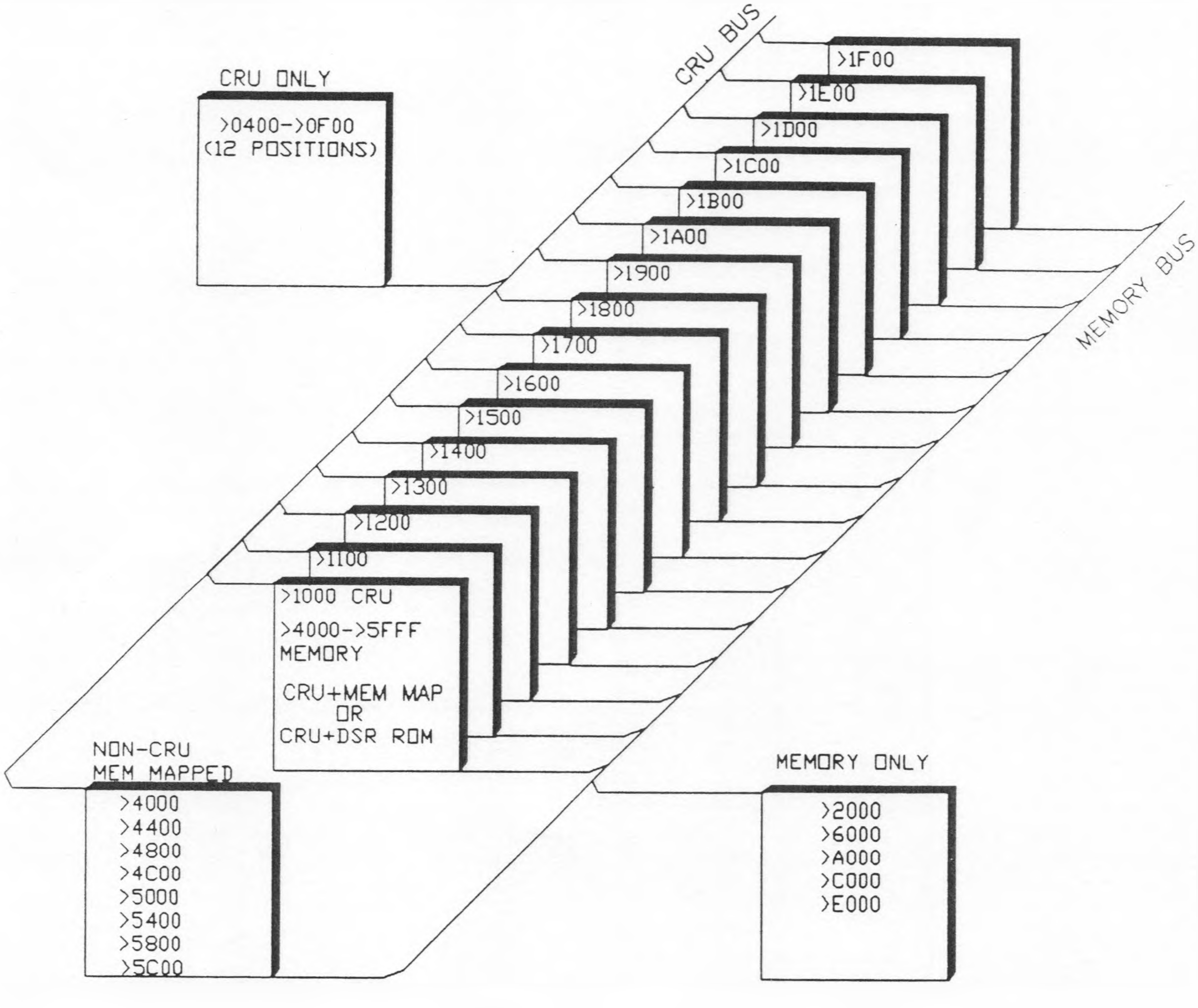


TONY LEWIS, PE

INTERFACE STANDARD & DESIGN GUIDE

for TI 99/4A Peripherals

SECOND EDITION



TONY LEWIS, PE

DISCLAIMER

DISCLAIMER ON CONTENTS

The following should be read and understood before purchasing and/or using this Interface Standard/Design Guide.

The technical information contained in this document is accurate, to the best of the knowledge of the author and the reviewers. However, given the volatile nature of the computer and electronics industry, along with the lack of access to original design documentation on the 99/4A system, there is no warranty that the contents of this manual will be free from error or will meet the specific requirements of the purchaser. The purchaser assumes complete responsibility for any decision made or actions taken based on information obtained using the contents of this manual. Any statements made concerning the utility of the contents are not construed as expressed or implied warranties.

The author reserves the right to revise any of the contents at any time without notice. However, registered owners will be notified if significant revisions or additions to the contents are made. Purchasers are encouraged to notify the author of any errors found in the text or graphics.

The author makes no warranty, either expressed or implied, including but not limited to any implied warranties of fitness, operability, or validation of design, regarding the contents or any information derived therefrom, and makes all contents available solely on an "as is" basis.

In no event shall the author or the reviewers be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of the purchase or use of this literature and the sole and exclusive liability of the author, regardless of the form of the action, shall not exceed the purchase price of this manual. Moreover, the author shall not be liable for any claim of any kind whatsoever by any other party against the user of this manual.

Texas Instruments did not contribute to, commission, nor approve the creation of the Interface Standard/Design Guide. All information used in deriving the contents of this manual are available in the public domain. "TI", "TI 99/4A", and "99/4A" are registered trademarks of Texas Instruments.

RESOURCES

This document was created on TIWriter Version 4.2, by R. A. Green Software. Graphics were created with AutoSketch 2.0.

Disassembly of existing console and DSR code was accomplished via Millers Graphics EXPLORER and DISkASSEMBLER programs. The program "GPLDIS" was used to disassemble the GPL code in GROMs 0-2. Reference 9 provided input on console routines. References 9, 10, and 14 were used to compile information on use of console RAM locations as they pertain to DSRs.

References 1, 2, 4, and 5 provide information on the TMS9900 microprocessor and related system design. References 1, 8 and 14 are excellent sources on assembly programming. Reference 13 provides completely disassembled and commented DSR codes for several peripherals. Reference 15 provides information on the Graphics Programming Language, which was useful in disassembling code in GROMs 0-2.

All references listed are recommended as excellent sources of information for the 99/4A and its Peripheral Expansion System, and the reader is encouraged to consult them for more information. Where possible, consult with the local library about obtaining references via the InterLibrary Loan System, using the ISBN number. Below is a list of current addresses for some of the references:

Millers Graphics
1475 W. Cypress Ave.
San Dimas, CA 91773

Texas Instruments Inc.
Data Book Marketing
PO Box 117692
Carrollton, TX 75011-7692
(800) 232-3200

The Bunyard Group
PO Box 62323
Colorado Springs, CO 80962-2323

LL Conners Enterprises
Computer and Electronics
1521 Ferry Street
Lafayette, IN 47904

Readers interested in obtaining copies of Technical Data and the GPL manual may contact the author directly for more information.

UTILITY PROGRAM COMMENTS

UTILITY PROGRAMS

A 5-1/4" single sided, single density diskette containing utility programs is provided with the manual to assist the peripheral developer in creating DSRs and application programs. The purpose of including these programs with the manual is to provide (in the author's opinion) the "best" utility programs available to the developer such that DSRs may be quickly written, debugged and released. Disassemblers are also included for reverse engineering console/peripheral code as needed to insure compatibility. [Use of Millers Graphics DISkASSEMBLER for disassembly is highly recommended; DISkASSEMBLER is available from several sources.]

Each of the programs took several days of development by their authors to complete. Responsible purchasers are obligated to forward a contribution to the software authors to acknowledge the usefulness of their products, and to encourage development of future products.

HOTBUG is not fairware; consult the documentation for proper registration of ownership. The README file on the disk contains a list of the utility programs, program description, recommended contribution amount, author name and current address.

REFERENCES

The following sources were used as references for this manual:

1. Microprocessors/Microcomputers System Design, Texas Instruments, McGraw-Hill Book Company, 1980. QA76.S.T49. ISBN 0-07-0637558-X.

2. 16 Bit Microprocessor Systems, Texas Instruments, McGraw-Hill Book Company, 1982. TK7895.M5.C35. ISBN 0-07-063760-1.

3. TI 99/4A Console and Peripheral Expansion System Technical Data, Texas Instruments, 1983.

4. Hardware Manual for the TI 99/4A Home Computer, Micheal Bunyard, PE, 1986.

5. TMS9900 Data Manual, Texas Instruments, 1985.

6. TTL Cookbook, Don Lancaster, Howard Sams Co., 1974. ISBN 0-672-21035-5.

7. TI 99/4A Peripheral Schematics: RS232-1039308; Memory Expansion-1039330, Disk Controller-1039340.

8. Software Development Handbook, 2nd Edition, Texas Instruments, 1981. ISBN 0-904047-31-8.

9. TI 99/4A INTERN, Heiner Martin, Verlag fur Technik und Handwerk GmbH, 1985. ISBN 3-88180-009-3.

10. Explorer Technical Manual, Millers Graphics, 1985.

11. DiskAssembler Technical Manual, Millers Graphics, 1986.

12. PEB ProtoBoard Manual, Scott Coleman and John Willforth, 1988.

13. Technical Drive, Monty Schmidt, 1987.

14. Editor/Assembler Manual, Texas Instruments, 1982.

15. Texas Instruments Graphics Programming Language User's Guide, Personal Computer Division, Texas Instruments, Dec. 1979.

16. Horizon RAMDisk Source Code and Technical Manual, Horizon Computer Limited, 1986.

INTRODUCTION

The purpose of this manual is to consolidate all information available in the public domain on the design and development of peripherals for the TI 99/4A computer into one reference. There are several excellent documents on the hardware and software of the console and its peripheral system available; however, this manual has been specifically written for designer/developers who wish to create new hardware and/or software for TI 99/4A peripherals.

The manual is an intermediate level text in that it is assumed that the reader is familiar with the TMS9900, its assembly language, the 99/4A peripheral system, the File Management System, and general computer and electronics concepts. Readers who are novices in any of these areas should consult the appropriate references before using this manual. Although some overlap of information exists between this manual and the references, the reader is urged to consult the references as needed for information not included in the Interface Standard/Design Guide.

As the title implies, this manual is meant to provide a consistent basis, or standard, for designers to create peripherals that will be compatible not only with the TI 99/4A, but with other peripherals as well. Basic information on hardware and software techniques is also provided for use by the developer.

Sections A-H cover the hardware aspects of the console and peripherals, and includes design information on chips and circuits. New peripheral types are defined in Section C, and existing peripheral locations are assigned in Section G. Section I covers the basics of Device Service Routine (DSR) construction. Section J discusses how the routines built into the console access peripherals and their DSRs. Where ever possible, examples are given of hardware and software concepts to assist the reader.

The author hopes that all readers will find the Interface Standard/Design Guide useful and informative. The author would also like to thank the following people who reviewed and commented on the original draft of the manual:

John Willforth
Matt Beebe
Jim Reiss
Peter Hoddie
Paul Carlton
Mike Dodd
Barry Boone
Mid-Atlantic 99ers
John Johnson

TABLE OF CONTENTS

SECTION PAGES

TABLE OF CONTENTS

SECTION | PAGES

SECTION A: 9900 SIGNALS/INTERFACING

1.0 Introduction

The TMS9900 microprocessor has 64 pins, 49 of which are used as signals to communicate with other chips and the outside world. These signals can be grouped into two basic sets, or buses: memory bus and CRU bus. The memory bus can be further divided into three types of signals: address, data, and control. This section will briefly discuss the function of the TMS9900 signals, and how they are used and modified by the /4A system. References 1, 2, and 5 contain more detailed descriptions of the 9900 signals.

2.0 Memory Bus

The memory bus is used to communicate with memory chips (or memory mapped devices) by selecting an address, then reading or writing data to or from the address. The address bus provides signals to select individual addresses, while the data bus provides a two way communication path for information to travel. The control bus signals coordinate action between the microprocessor and other devices. Below is a brief description of the functions of the memory bus.

2.1 Address Bus (A0-A14)

The 9900 has 15 address signals, A0 - A14, with A0 the most significant bit (MSB). These signals are driven out of the micro, and are used to select the address of information to be read or written. It is assumed that the memory system will have a 16 bit word design, and not a byte (8 bits) wide data bus, because the 9900 addresses 32K words (16 bits) for each read or write function. The 9900 cannot address individual bytes because there is no A15 signal to discriminate between even and odd bytes. To perform byte reads, the 9900 will read two bytes simultaneously, discarding the information in the unused byte. To perform byte writes, the 9900 first reads two bytes, alters the byte being written to, and then writes both bytes back to memory.

2.2 Data Bus (D0-D15)

The data bus is 16 bidirectional signal lines used to read or write information from other devices. Since the data bus is 16 bits wide instead of eight, the 9900 can access twice as much information per unit of time than a similar micro with an 8 bit data bus.

2.3 Control Signals (DBIN, -WE, etc.)

Control signals are used to synchronize the operations of the 9900 with the devices that it is communicating with. The 9900 control signals are summerized below; consult Reference 1 or 5 for more detail on these signals.

I/O	in /4A?	Signal	Description
0	Y	-MEMEN	used to enable memory accesses, differentiates between memory bus and CRU bus activity
0	Y	DBIN	data bus direction, determines the direction of data (in or out) for the 9900
0	Y	-WE	write enable, denotes writes to memory
0	Y	IAQ	Instruction Acquisition Status, denotes that the microprocessor is obtaining an instruction from memory
I	Y	READY	memory ready status, informs micro that system memory is ready to be accessed.
0	N	WAIT	Ready acknowledge, status signal that 9900 acknowledges memory not ready to be accessed.
I	N	-HOLD	HOLD process, when active, puts 9900 signals in inactive state. Memory bus may now be driven by another device.
0	Y	HOLDA	HOLD Acknowledge, informs external device that 9900 acknowledges receipt of HOLD request.
I	Y	-RESET	Reset input, resets micro to initial state
I	Y	-LOAD	nonmaskable interrupt, forces 9900 to branch to address >FFFC for new program counter and workspace values
I	N	IC0-IC3	interrupt code 0-3, inputs for up to 16 maskable interrupts
I	Y	-INTREQ	Interrupt Request, informs 9900 that an interrupt code is valid on IC0 - IC3

[signals may be either available externally, or used only internally by the console; signals may be altered or unused in current design]

3.0 CRU Bus

The input/output bus on the 9900 is known as the Communication Register Unit (CRU) bus. The CRU bus is similar in concept to the memory bus, with the following exceptions:

1: The memory bus can communicate in words (16 bits) with a set of odd and even addresses. The CRU bus associates one bit per address accessed by the 9900.

2: The CRU address space is limited to >0000 to >1FFF, where as the memory bus can address >0000 to >FFFF. These are separate and distinct addresses; control signals are used to differentiate between memory address space and CRU address space.

3: The CRU bus is used primarily to control peripherals (on/off) versus communication of data because the memory bus transfers more bits per access than the CRU bus.

4: The CRU bus does not have as many control signals as the memory bus, sometimes causing design concerns when developing circuitry.

The CRU bus consists of the following signals, all of which are used in the /4A system:

A3-A14	(out)	These lower order address lines define the CRU space >0000 to >1FFF. Same lines as used by memory bus.
CRUCLK	(out)	CRU Clock, used during CRU output to inform external device that address bus and CRUOUT output bit signals are stable.
CRUOUT	(out)	CRU Output Data, outputs value of bit when CRUCLK is active.
CRUIN	(in)	CRU Input Data, inputs bit value into 9900

References 1 and 2 have excellent discussions of the CRU bus as implemented by the 9900.

4.0 Interfacing with the /4A

The signals available for the 9900 are used in various combinations to allow it to interface to external devices. This section will cover the relationships between the signals on the memory and CRU buses as they are presented to peripheral devices by the /4A system. Not all 9900 signals are available in the /4A system for use with peripherals; likewise, the relationship and timing of some signals are radically modified by the /4A system and do not conform to the original 9900 signal format. Most notable is the fact that the /4A system has an 8 bit peripheral data bus, not 16 bits.

4.1 Memory Bus Interfacing

The /4A system will read an odd and even byte within a word boundary by reading the odd byte first, then the even one. Control logic circuitry internal to the /4A allows it to read the first byte, then the second, and then reassembles them into a word before presenting it to the 9900. Figure A.1 shows the appropriate signal timing and relationships.

The control logic is:

IF -MEMEN is low AND DBIN is high, THEN a memory READ is occuring.

There is no need to include -WE in decoding for a Read. The /4A system automatically inserts two wait states (333 ns each) for each byte access. Allowing for 100 ns settling time for the address lines to become valid after -MEMEN goes low, a peripheral has up to 650 ns to provide valid data on the data bus.

A0-A14 are held constant per memory read - - - only A15 changes state during a memory access for a Read. A15 is generated by the /4A system to differentiate between odd and even bytes, and is not produced by the 9900.

The /4A system writes information to an odd and even byte within a word boundary with timing similar to a Read operation. Figure A.2 shows signal timing and relationships for a Write operation. The control signal logic is:

IF -MEMEN is low AND DBIN is low, THEN a memory Write is occuring.

Data on D0-D7 is valid when -WE goes low.

Data is presented on the data bus and is valid (-WE goes low) approximately 333 ns after -MEMEN and DBIN are both low; -WE remains low for 578 ns, typically. One -WE pulse is generated per byte Write, whereas the 9900 generates only one -WE pulse per word. Recall also that the 9900 always performs a Read operation to a word boundary prior to a Write operation; this is true of the /4A system also (ie-the /4A reads two consecutive bytes, even when performing a single byte write operation).

4.2 CRU Bus Interfacing

Input and output on the CRU bus is more simplistic but also can create problems for designers if certain relationships are ignored. The CRUCLK signal of the 9900 is inverted by the /4A system to produce -CRUCLK. This is used to strobe a CRU bit out of the /4A via the CRUOUT line, similar to the way -WE strobes data from the D0-D7 lines. The timing relationship fo a CRU output series is shown in Figure A.3. The control signal logic is:

IF -MEMEN is high AND -CRUCLK is low THEN a CRU bit is output on CRUOUT.

The -MEMEN signal allows the /4A to multiplex the A15 and CRUOUT on the same pin; the pin is for "A15" if -MEMEN is low, and for "CRUOUT" if -MEMEN is high.

Input on the CRU bus is accomplished by establishing a valid address on A0-A15, then reading the bit value on the CRUIN line 400 ns after the address is valid.

No other control signals are needed to define a CRU Read operation. Unlike the memory bus operation, external devices have no warning that an operation on the CRU bus is about to occur (-MEMEN going low notifies the system that a memory bus is going active; there is no corresponding "-CRUEN" signal). Designers of peripherals utilizing the CRU bus must be aware of this restriction.

FIG A.1: READ CYCLE TIMING

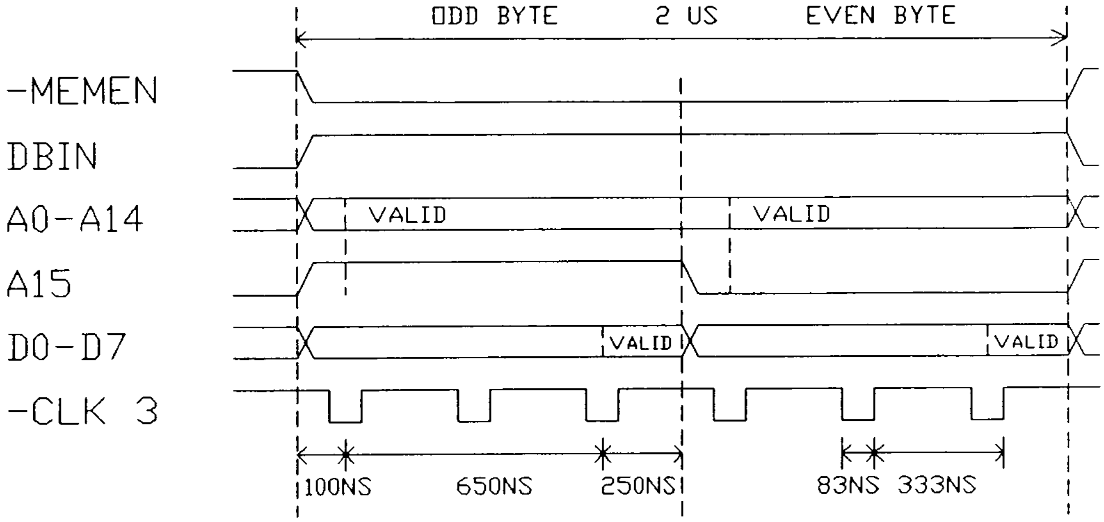


FIG. A.2: WRITE CYCLE TIMING

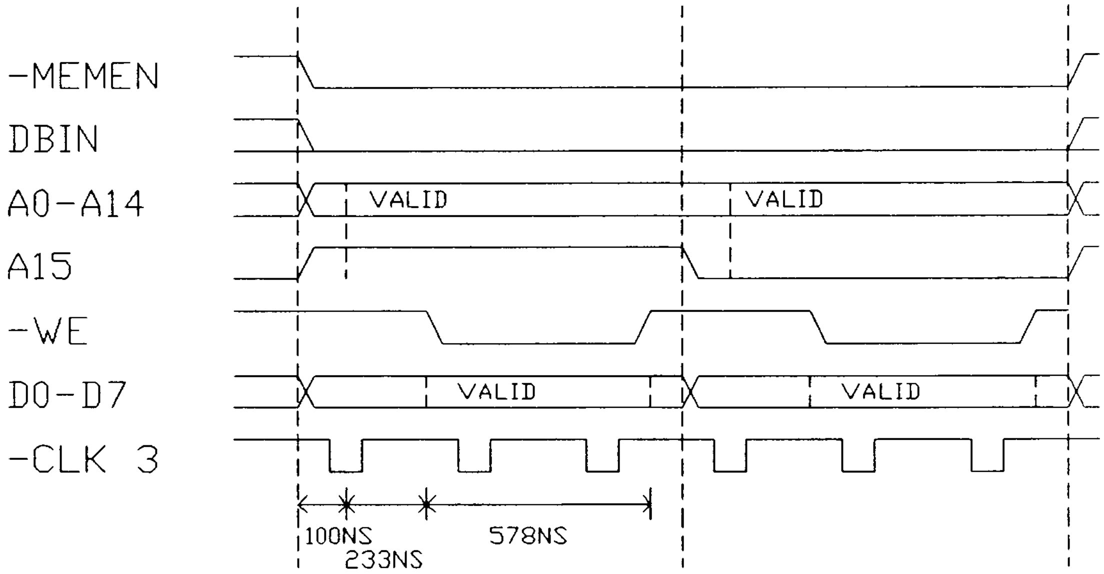


FIG. A.3: CRU OUTPUT

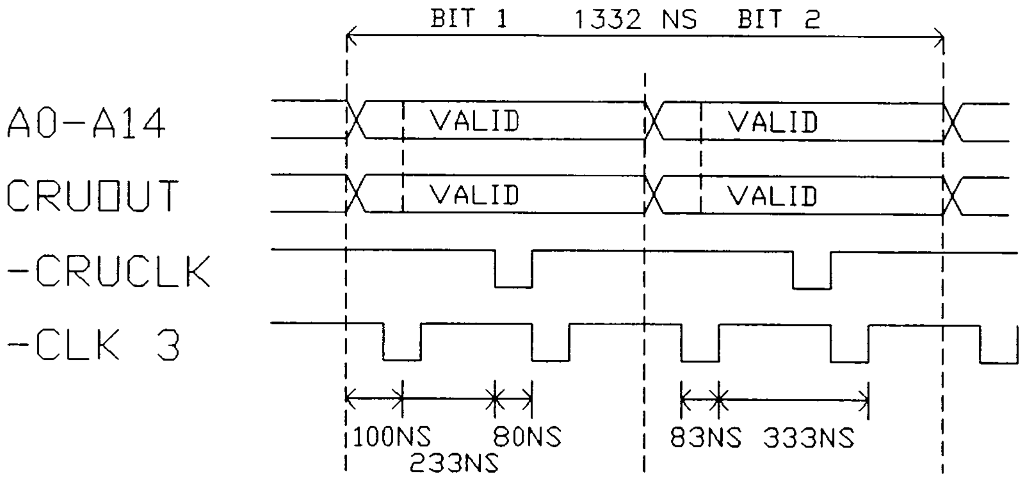
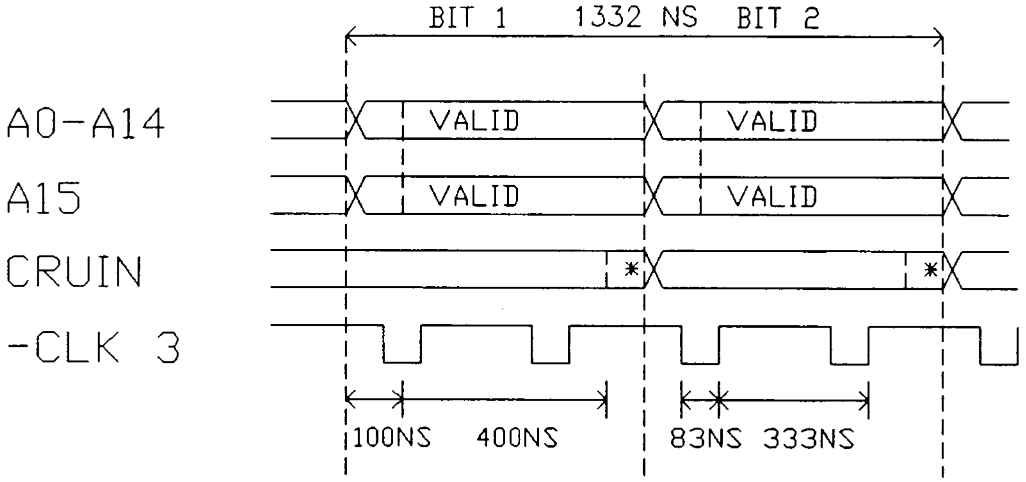


FIG. A.4: CRU INPUT



* INPUT ON CRUIN LINE IS VALID

SECTION B: CONSOLE (44 PIN) AND PBOX (60 PIN) CONNECTORS

1.0 Introduction

Not all of the signals from the 9900 are made available to the outside peripherals. The side connector on the console has 44 pins, 38 of which are signals (the rest are power and ground pins). The PBox bus has a 60 pin connector, which has 12 power and ground pins, 7 unused (currently) signals, and 41 active signals. The signals for the console and PBox are listed below, along with comments of their intended functions. Figures B.1 and B.2 show the console and PBox connectors as viewed looking into the connector.

2.0 Console 44 Pin Connector

The side connector on the console is an edgecard type with 44 pins spaced 0.10" pin to pin spacing. The signals, functions and pin numbers are as follows:

Signal	Pin	I/O	Comments
A0 (MSB)	31	0	Address bus signals
A1	30	0	"
A2	20	0	"
A3	10	0	"
A4	7	0	"
A5	5	0	"
A6	29	0	"
A7	17	0	"
A8	14	0	"
A9	18	0	"
A10	6	0	"
A11	8	0	"
A12	11	0	"
A13	15	0	"
A14	16	0	"
A15/ CRUOUT	19	0	A15 is created by logic internal to the console, not by the CPU. CRUOUT is gated A15, and is not active unless -MEMEN is high.
D0	37	I/0	Bidirectional data bus
D1	40	I/0	"
D2	39	I/0	"
D3	42	I/0	"
D4	35	I/0	"
D5	38	I/0	"
D6	36	I/0	"
D7	34	I/0	"

Signal	Pin	I/O	Comments
-MEMEN	32	0	Same as for 9900
DBIN	9	0	"
-WE	26	0	This is highly modified from the original 9900 signal into two active low -WE signals per cycle (one per byte)
-MBE	28	0	Memory Block Enable. Created by console logic; device enable signal for the >4000->5FFF memory block. Convenient for side mounted peripherals. Signal not transmitted to PBox bus.
-CRUCLK	22	0	Phase 3 clock, inverted
CRUIN	33	I	Same as for 9900
READY	12	I	" ", with pull up resistor
IAQ	41	0	Not transmitted to PBox bus
-LOAD	13	I	"
-RESET	3	0	This is output, and cannot be used to input a -RESET signal
-EXT INT	4	I	External Interrupt, active low, used by peripherals to indicate an interrupt request to the 9900
-PH 3	24	0	Phase 3 of the 9900 4 phase clock, inverted to active low.
SBE	2	0	Speech Block Enable; indicates access to speech memory at >9000/>9400
AUDIO IN	44	I	Input for audio from speech module to sound chip
+5V	1		Supply voltage for speech module
-5V	43		" *Not connected to PBox or interface cable. DO NOT use for side peripherals, or damage to console power supply may occur.*
GROUND	21,23,25,27		Ground

3.0 PBox Bus Signals - 60 Pin Connector

The PBox bus uses 60 pin female connectors with pins spaced 0.10" pin to pin spacing. Not all of the signals available from the 44 pin connector are available in the PBox bus. The Interface Card sold with the PBox determines which signals were transferred. The PBox end of the cable also holds some (currently) unused signals high, by tieing them to a 5V source via a resistor. The signals, functions, and pin numbers are as follows:

Signal	Pin	I/O*	Comments
A0.A	43	I	Address bus signals; "A" suffix denotes PBox signal
A1.A	44	I	
A2.2	41	I	"
A3.A	42	I	"
A4.A	39	I	"
A5.A	40	I	"
A6.A	37	I	"
A7.A	38	I	"
A8.A	35	I	"
A9.A	36	I	"
A10.A	33	I	"
A11.A	34	I	"
A12.A	31	I	"
A13.A	32	I	"
A14.A	29	I	"
A15/CRUOUT.A	30	I	"
AMA.A	46	HIGH	Extended address bit, held high by interface card
AMB.A	45	HIGH	
AMC.A	48	HIGH	"
D0	28	I/0	Data bus signals
D1	25	I/0	"
D2	26	I/0	"
D3	23	I/0	"
D4	24	I/0	"
D5	21	I/0	"
D6	22	I/0	"
D7	19	I/0	"
-MEMEN.A	56	I	Same as 44 pin side port signals
DBIN.A	52	I	"
-WE.A	54	I	"
-CRUCLK.A	51	I	"
CRUIN	55	0	"
READY.A	4	0	"
IAQHA	14	N/C	IAQ and Hold Acknowledge gated together. For use with 9995 based machines as Hold Ack.
-LOAD	18	N/C	Not used with /4A
-RESET	6	I	
-INTA	17	0	-EXT INT
-CLKOUT	50	I	-PH 3
AUDIO	10	0	
SCLK	8	N/C	System clock. Use is not defined with /4A
-LCP	9	N/C	9995 indicator. Low=9995 machine, high=/4A. Possible use to switch peripherals to faster speed.
PCBEN	12	HIGH	Enables cards in PBox. Low disables all cards.
-HOLD	13	N/C	Active low HOLD request for 9995 based machines

Signal	Pin	I/O*	Comments
-SENILA	15	HIGH	Interrupt level A and B Sense Enable. Allows computer to quickly identify peripheral interrupt. Not used by /4A system.
-SENILB	16	HIGH	
-RBDENA	11	0	Active low remote data bus driver enable line. Each peripheral that utilizes the DATA bus must generate an -RDBENA signal when accessing the data bus. This signal enables the LS245 transciever in the console end of interface card.
GROUND	3,5,7,20,27 47,49,53		Ground
UNREG 8V	1,2		Used to supply unregulated voltages to voltage regulators mounted on peripheral cards.
UNREG -16V	57,58		
UNREG +16V	59,60		

[*Either input into the PBox bus, or output to the console]

4.0 General Notes

4.1 The interface cable shares a common ground between the console and PBox. Positive and negative voltages are not interconnected between the PBox and the 44 pin console connector.

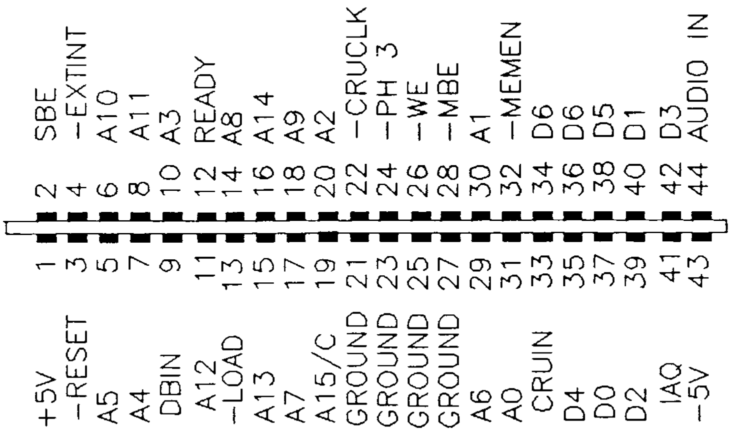
4.2 -RBDENA is not needed for peripherals that do not use the data bus (D0-D7). If used, it should be active low with the chip enable signal for the data bus transciever for the peripheral.

4.3 Unregulated +8V, +16V, and -16V sources are provided to allow for voltage regulators (as needed) on each peripheral card. Temporary voltage transients on an individual card will not affect the other peripheral cards.

4.4 Signals held high (+5V) by the Interface Card cannot be used unless the Interface Card is removed, modified or replaced with a different interface card.

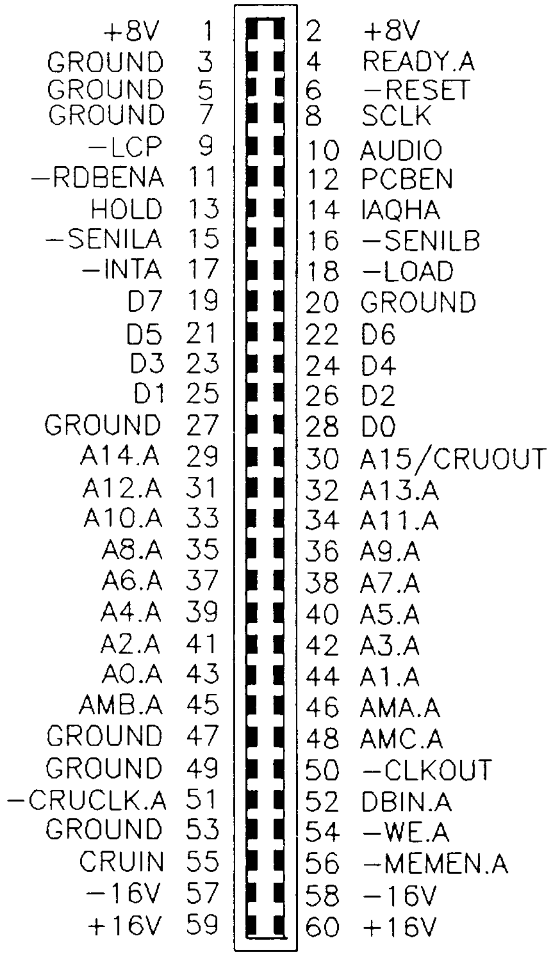
FIG. B.1: 44 PIN SIDE EDGEBOARD CONNECTOR

(VIEW LOOKING INTO CONSOLE SIDE)



1	+5V	2	SBE
3	−RESET	4	−EXTINT
5	A5	6	A10
7	A4	8	A11
9	DBIN	10	A3
11	A12	12	READY
13	−LOAD	14	A8
15	A13	16	A14
17	A7	18	A9
19	A15/C	20	A2
21	GROUND	22	−CRUCLK
23	GROUND	24	−PH 3
25	GROUND	26	−WE
27	GROUND	28	−MBE
29	A6	30	A1
31	A0	32	−MEMEN
33	CRUIN	34	D6
35	D4	36	D6
37	D0	38	D5
39	D2	40	D1
41	IAQ	42	D3
43	−5V	44	AUDIO IN

FIG B.2: 60 PIN PBOX CONNECTOR SOCKET



+8V	1	2	+8V
GROUND	3	4	READY.A
GROUND	5	6	−RESET
GROUND	7	8	SCLK
−LCP	9	10	AUDIO
−RDBENA	11	12	PCBEN
HOLD	13	14	IAQHA
−SENILA	15	16	−SENILB
−INTA	17	18	−LOAD
D7	19	20	GROUND
D5	21	22	D6
D3	23	24	D4
D1	25	26	D2
GROUND	27	28	D0
A14.A	29	30	A15/CRUOUT
A12.A	31	32	A13.A
A10.A	33	34	A11.A
A8.A	35	36	A9.A
A6.A	37	38	A7.A
A4.A	39	40	A5.A
A2.A	41	42	A3.A
A0.A	43	44	A1.A
AMB.A	45	46	AMA.A
GROUND	47	48	AMC.A
GROUND	49	50	−CLKOUT
−CRUCLK.A	51	52	DBIN.A
GROUND	53	54	−WE.A
CRUIN	55	56	−MEMEN.A
−16V	57	58	−16V
+16V	59	60	+16V

(VIEW LOOKING INTO PBOX)

SECTION C: PBOX CARD ELECTRONIC FEATURES

1.0 Introduction

Each peripheral card designed to be used with the PBox will have certain features that will not only allow the device to work properly, but will keep it from interfering with other devices and the computer. Due to the diverse nature of possible peripheral devices, not all of the following electronic features will be implemented on every device. The peripheral designer is responsible for insuring that his or her design utilizes the appropriate buffering and device selection techniques to prevent bus contention between devices.

2.0 Interfacing Notes

As noted previously, the PBox has two basic bus systems, memory and CRU. The CRU bus is used to control most peripherals, while the memory bus is used to transfer data to and from the peripheral. The CRU bus can also be used to transfer data to and from a peripheral, but a slower rate since the CRU bus transfers information at the rate of one bit per cycle, whereas the memory system transfers one byte (8 bits) each cycle.

These two buses can be used in several designs to communicate between the computer and the peripheral. However, most interface designs can be grouped into one of five categories:

1) CRU only (serial)
2) Memory only
3) CRU and Memory Mapped (non-DSR)
4) CRU, DSR ROM, and Device
5) Non-CRU Memory Mapped

Each of these categories are discussed below. Note that the fourth category is the most common design, and the last category has not been defined until now.

2.1 CRU Only

In many ways, interfacing the computer to a peripheral via the CRU bus (only) is the simplist design of all. A peripheral that uses only the CRU bus to communicate has only one constraint: the CRU address(es) used by the peripheral must not be used by any other device. This is extremely important because most of the peripheral devices and the 9901 which drives the keyboard already have several CRU addresses assigned to them. Attempting to use any of these CRU addresses will result in contention on the CRU bus, and possible activation of other peripherals on the memory bus. [Following sections will explain how the CRU bus is used to poll and activate peripherals.] Section G contains the CRU map for the /4A. Each >0100 CRU address block has 128 addresses. Each peripheral space in the 16 locations reserved for peripheral devices (from >1000 to >1F00) has 128 CRU addresses available; however, usually no more than the first 8 CRU bits are used. The TMS9901 utilizes 32 CRU bits, starting at address >0000. The CRU space from >0400 to >0FFE is unassigned and has 12 sets of 128 CRU addresses available. It is recommended that peripherals based upon using the CRU bus only should be located within the >0400 to >0FFE CRU address range, with 128 bits available per CRU-only peripheral space. Table C.1 defines these twelve peripheral blocks. If more than 128 bits are required, then sequential peripheral blocks should be utilized. Any device that utilizes one or more of the peripheral blocks in Table C.1 shall have the CRU addresses clearly identified in the device documentation, and noted on the device itself, if possible. None of the CRU-only peripheral blocks are currently defined.

TABLE C.1
CRU-ONLY PERIPHERIAL BLOCKS

Block	CRU Address Range
1	>0400->04FE
2	>0500->05FE
3	>0600->06FE
4	>0700->07FE
5	>0800->08FE
6	>0900->09FE
7	>0A00->0AFE
8	>0B00->0BFE
9	>0C00->0CFE
10	>0D00->0DFE
11	>0E00->0EFE
12	>0F00->0FFE

Note: While it is possbile to utilize CRU addresses within the sixteen polled peripheral spaces, it is not recommended since these bits may be used in existing or future devices. The twelve CRU-only peripheral blocks defined in Table C.1 should provide adequate space for development of these types of peripherals.

2.2 Memory Only

As seen in the /4A memory map, there is a total of 48K possible RAM space available, consisting of the following 8K blocks: >2000, >4000, >6000, >A000, >C000, and >E000. Utilization of these spaces is discussed below as they pertain to use in the PBox.

2.2.1 32K Design

The TI 32K RAM peripheral card covers the >2000 and >A000->FFFF memory spaces. Since the operating system of the /4A was designed to utilize RAM in these memory blocks, there is no need for special controls (such as CRU) to activate this memory device, only simple address decoding. (See Section E.2 for more details.) The original TI memory card utilizes dynamic RAM; subsequent third party devices use more commonly available static RAM. Low power CMOS RAM is used in some designs along with batteries to retain data when the PBox power is off.

2.2.2 >4000 Space

The memory space from >4000 to >5FFF is reserved for paging in various peripheral devices and for memory mapped devices. See Sections 2.3 to 2.5.

2.2.3 >6000 Space

The space >6000->7FFF is traditionally not accessed from devices in the PBox because the /4A system assumes that it will be accessed from a cartridge in the 36 pin module port. The signal -ROMG on pin 34 is used to activate the 8K block at >6000. Bus contention will occur if a device in the PBox contains RAM/ROM at >6000, and a plug-in cartridge contains RAM/ROM/GROM at the same location. Peripheral devices containing memory in this 8K location are acceptable only if the memory is inactive upon powerup, and is activated by the user via hardware (switch) or software (CRU activation). This places the burden upon the user to activate this RAM space only after confirming that no module with memory in the >6000 space is inserted in the console. If software checking is used, a powerup routine that looks for "AA" at byte >6000 can be used to confirm that the space is not free for use.

2.2.4 Bank Switching

Bank switching via CRU control is acceptable for the >2000, >6000 and >A000->FFFF memory spaces. However, most applications programs, especially BASIC utilizes these areas in predefined routines, particularily the >2000 block. Therefore, bank switched RAM blocks are useful for programs specifically designed to utilize them. As with other concepts, the designer must insure that two RAM blocks do not occupy the same address space simultaneously. Bank switching circuitry should be disabled by powerup or RESET activation.

2.2.5 Extended Address Lines

The address lines AMA, AMB, and AMC are provided in the PBox bus to increase the linear address space of the system from 64K to 512K. As with bank switching, use of these lines to extend the available memory space is acceptable, but useful only for programs specifically designed to utilize them. A different interface card is required for the /4A system to allow use of AMA-AMC, since the card holds these signals high. Any memory device that uses these lines must make sure that AMA-AMC are high (=1) when accessing the "normal" 32K. Also, the designer should note that most of the TI produced peripheral cards are not activated if either AMA, AMB, or AMC are low.

2.2.6 Memory Mapping

An advanced technique for extending memory for peripherals is referred to as memory mapping. This technique is similar to bank switching, but utilizes a specialized LSI chip, the 74LS612 memory mapper to control generation of address lines beyond A0-A15. The '612 can be utilized to expand the address lines to accomodate up to 16 Meg bytes, without utilizing AMA-AMC. If a memory peripheral is designed to address more than 512K, then it is recommended that a memory mapper be located on the peripheral to generate local extended addresses for that device. [A description of the '612 and an application report is given in TI's "LSI Logic Data Book", 1986]

2.3 CRU Select and Memory Mapped (non-DSR ROM)

The /4A system is designed to sequentially poll 16 peripheral spaces, all located in the >4000 memory space. The CRU bus is used to select and activate these perpherials one at a time to prevent bus contention. The system and the 16 peripheral spaces are described in Section 4.0. This section covers memory mapped devices that are placed in one of the 16 polled peripheral spaces. These devices may or may not also have applications ROM/RAM; but it does not contain a valid Device Service Routine program. Section 2.5 covers memory mapped devices that are not polled by the /4A system, and do not have separate applications programs within their assigned memory space.

Memory mapped devices are accessed at only one address, or a small series of addresses. An existing example of a memory mapped device is the 9918A video chip. For the purposes of this section, it is assumed that these devices do not require a Device Service Routine (DSR) ROM to properly operate. An applications ROM or RAM of up to 8K length may be located within the same peripheral space, as long as it does not place the value "AA" in the first byte, and its assigned address range does not include the memory mapped address(es). This type of device may be activated by the /4A peripheral polling system, but will not respond since it does not have a valid DSR header. This type of peripheral is different from the standard polled peripheral, described in Section 4.0, in that it does not need a powerup, interrupt or applications program that uses the /4A polling and PAB access system, but does need valid addresses for memory mapped devices, and possibly an applications program to run. Since it would be located in the >4000 block, it can not be activated while the polled peripherals are being accessed.

For this type of peripheral, the following requirements must be met:

a) The memory mapped device must be located in the range of >4002->5FFF; it cannot be located at either >4000 or >4001 since it might be accidently activated by the polling system.

b) The memory mapped address(es) must not overlap with any ROM/RAM activated by this device.

c) The memory mapped address decoder chips, data buffers and ROM/RAM select chips are to be activated only when that peripheral space is selected by the CRU bus by the calling program, and be deactivated when the peripheral is not selcted.

d) The peripheral must be activated by writing a high CRU bit (=1) to the card, and deactivated by writing a low value (=0) to the same bit. To avoid spurious activation by the /4A polling system, it is recommended that the activation CRU bit not be at CRU bit 0 for that peripheral space.

e) If the memory mapped device requires interrupts to communicate with the console, or a powerup/reset program, then it must have a valid DSR ROM (Section 2.4).

f) Applications software must be provided to properly activate the peripheral and insure that it is deactivated when communication is complete. Due to potential bus contention from another interrupt driven peripheral (which would automatically activate the polling system), all interrupts should be suspended via a LIMI 0 command by the applications software, then restored when the device is deactivated.

g) The device should utilize one of the peripheral spaces and its assigned CRU bit address range from the table in Section 4.0.

An example of this type of peripheral would be a Real Time Clock (RTC) that is periodically read by an applications program, and does not generate interrupts. The applications program would convert the RTC's output into the desired format, and place the time on the screen. When the applications program reads the RTC, or writes to set the time, external interrupts are suspended, the peripheral device is activated and accessed, then deactivated and interrupts are reactivated.

2.4 CRU Select, Device and DSR ROM

This type of peripheral is similar to those described in Section 2.3, except that a ROM device with a valid DSR must be included for the device to properly respond to the /4A polling system. The software section of this manual covers requirements for creation of DSR software. ROM/RAM up to 8K in length may be located in the >4000->5FFF space, and must not overlap with any other device on the peripheral that is memory mapped or uses other address decoding schemes.

For this type of peripheral, the following requirements must be met:

a) The DSR memory must be located starting at address >4000, and may extend to >5FFF.

b) Permanent memory (ROM, PROM, EPROM, EEPROM) is recommended for holding the DSR. RAM may be used, if loaded after powerup. Use of RAM for holding the DSR prevents use of the peripheral until the DSR is loaded. Non-DSR RAM (for scratchpad or data storage) may be used as long as the total memory (DSR + non-DSR) is 8K or less.

c) Any other devices on the peripheral must not share the same address space as the ROM/RAM.

d) The DSR memory must be designed such that the data buffers, DSR ROM/RAM select chips and any other device requiring address decoding are to be activated only when that peripheral space is selected by the /4A polling system. This system requires that the first CRU bit of that peripheral space activate the peripheral by writing a high value (=1) when it is selected, then deactivate the peripheral by writing a low value (=0) to the same CRU bit.

e) If the peripheral utilizes interrupts, then it must have an open collector driver connected to ground that can be cleared by the applications software once the peripheral is accessed.

An example of this type of peripheral is the RS232 cards, which are located at >1300 and >1500, and contain both DSR ROM and other chips, like the TMS9902 UART. See Section 4.0 for more details on how these peripherals are accessed by the /4A system.

2.5 Non-CRU Memory Mapped Devices

One of the drawbacks of the /4A's memory map is its utilization of the 8K memory space at >8000 to >9FFF. This memory space is assigned to the internal RAM and seven memory mapped devices, all of which are block decoded in 1K increments. Therefore, the RAM and memory mapped devices will respond whenever an access is made to an address within the assigned 1K block.

To allow for implementation of memory mapped devices NOT accessed internally by the /4A system, the memory space of >4000->5FFF is assigned for non-CRU memory mapped devices. This space can be accessed only when <u>none</u> of the 16 polled peripherals are paged in by the CRU bus.

For this type of peripheral, the following requirements must be met:

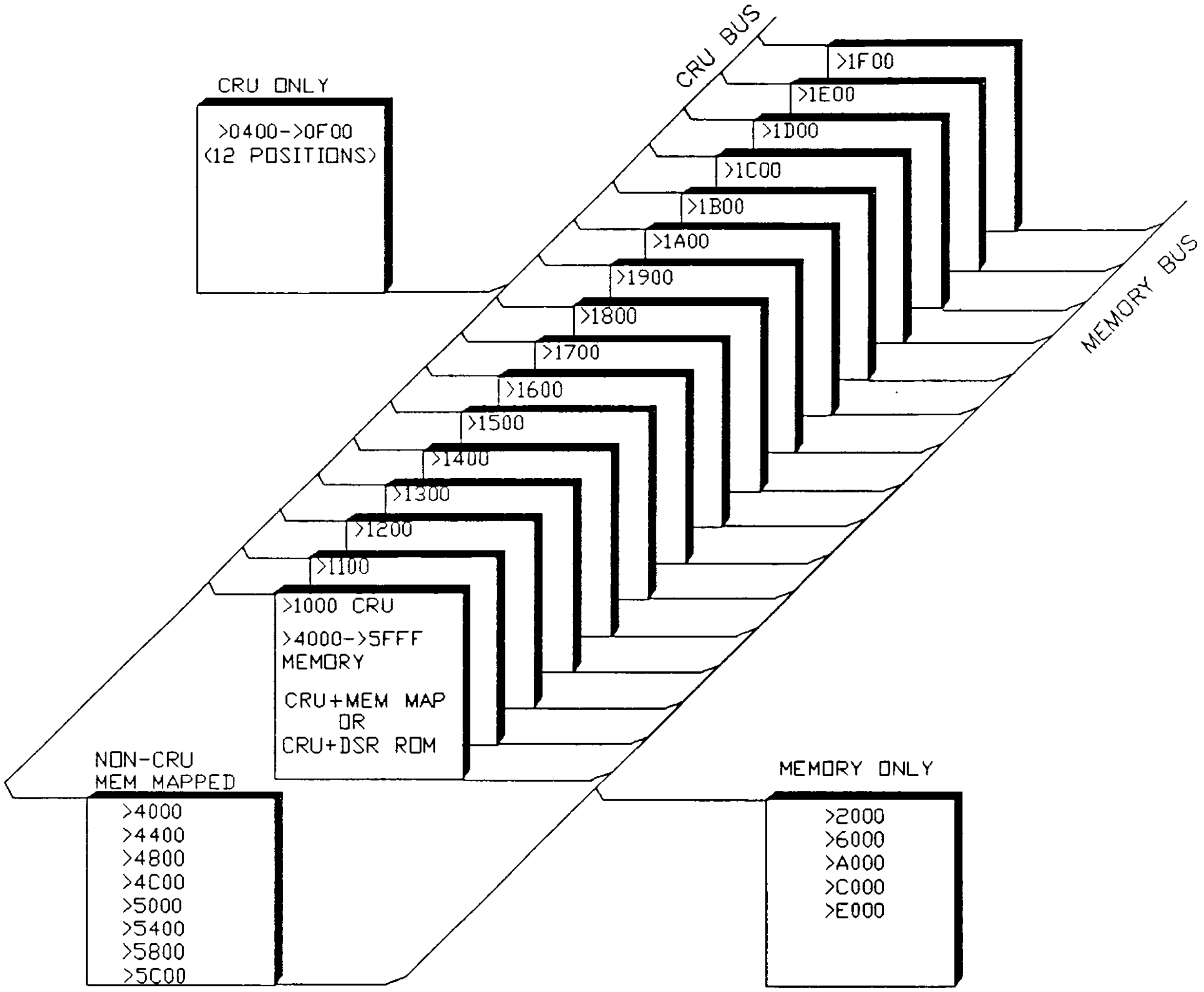
a) The peripheral must be disabled whenever a CRU access is made to one of the 16 polled peripherals, and enabled only when none of the polled peripherals is active.

b) The peripheral must be decoded to respond within one of the 8 1K blocks as defined below:

Memory mapped space	Address
1	>4000->43FF
2	>4400->47FF
3	>4800->4BFF
4	>4C00->4FFF
5	>5000->53FF
6	>5400->57FF
7	>5800->5BFF
8	>5C00->5FFF

If more than one address is needed, it shall be within the assigned 1K block.

FIG. C.1: PERIPHERAL TYPES



3.0 General Notes on Buffering, Activation, and Misc. Signals

To insure that peripheral devices do not cause bus contention, and are accessed properly, certain design features must be incorporated. Most devices will have to be buffered from the memory bus, activated only when that device is selected, and be capable of generating signals to the console of its status. The following sections discuss these design features.

3.1 Buffers

Devices which utilize the data bus must have a bidirectional driver, such as a 'LS245, with direction control and enable signals. The bus driver chip should be activated by the peripheral activation CRU bit (see 3.2) if a polled peripheral; other peripheral types will utilize other activation schemes. Other signals (address, control) should be driven by a 'LS244 (except as noted in 3.4), which is always active.

3.2 CRU Peripheral Card Activation

Polled peripherial cards in the PBox are activated by writing a high value (=1) to the first CRU bit of the assigned CRU peripheral space. For example, to activate the peripheral card at location >1500, CRU bit >1500 is turned 'on' (=1) via the SBO command. The first CRU bit shall be used to enable the data buffer, DSR ROM (if used), and indicator LED. CRU activated cards without DSRs should use a CRU bit other than bit 0 for activation. This bit may be used to enable any other chips located on the peripheral card; thereby reducing power requirements when the card is not selected. An indicator LED shall be provided to give the user visual feedback that the peripheral card is active; yellow LEDs are recommended for consistancy. The CRU bit is not the only enable signal for the DSR ROM; see section 3.3. The CRU bit must be latched by a flip flop, 'LS259, 9901, or similar device that is capable of storing the status of the CRU bit. Provisions must also be made in the design for use of the -RESET signal to clear the latch whenever -RESET goes low.

For non-CRU memory mapped devices, the device must be selected only when the assigned address is selected and none of the first bits of the 16 peripheral spaces is activated. Circuitry must be provided to track the status of the peripheral activation bits, and to deselect the memory mapped device if one of these CRU bits is activated. Provisions must also be made in the design for use of the -RESET signal to clear the non-CRU activation circuitry whenever -RESET goes low.

3.3 Memory Activation

All memory devices, whether for general storage or memory mapped devices, must use address decode circuitry to insure that the device will be activated at its assigned address(es). General storage memory (at the >2000, >6000, or >A000->E000 blocks), must be activated by use of upper address lines and -MEMEN. No CRU bits are required to activate general storage memory locations.

Likewise, non-CRU memory mapped devices, CRU memory mapped devices and DSR ROM/RAMs are activated by use of appropriate address lines, -MEMEN, and the CRU bus. These devices must be located in the >4000 to >5FFF address range. DSR ROMs must start at >4000; other devices are not required to start at address >4000.

As noted previously, the READ access time for memory devices is 650 ns, which is extremely generous, and should allow use of 'LS type decoders for use with the /4A system.

3.4 Miscellaneous Signals

Peripheral cards must use certain signals to communicate with the console and other peripherals. These signals and design notes are discussed below.

3.4.1 READY: System Ready signal

Used to put the 9900 in a WAIT state during initialization, or to extend a memory access cycle for slow memory devices. If used by the peripheral card, it must be an open collector driver (like an 'LS125) that is tied to ground. Note- activation of the READY signal is the sole responsiblity of the individual peripheral and not the console. Failure to deactivate the READY signal will result in an inoperative system. Note also that this signal is driven out of the card to the console.

3.4.2 -RESET: active low console driven Reset signal

This signal should be used on peripheral devices to clear the CRU activation bit, as well as any other CRU bits, and any other device that must be reinitialized to function properly after a low -RESET signal. It should be driven into the peripheral by an 'LS244 or similar chip.

3.4.3 PCBEN: active high PCB Enable

This signal is gated with other signals to activate a peripheral card. It can be driven into the peripheral by an 'LS244, or taken directly into the PCB with no driver chip.

3.4.4 -RBDENA: active low Remote Data Bus driver

-RBDENA must be provided to indicate to the Interface Card that a memory cycle (Read/Write) is needed for a peripheral in the PBox. It enables the 'LS245 on the console end of the cable. It is recommended that an open collector signal ('LS125) or tri-state gate ('LS244) tied to ground be used to drive the signal, with the gate controller tied to the same signal used to activate the data bus driver.

3.4.5 CRUIN: CRUIN signal

Usually does not have buffer drivers. Signal sent unbuffered directly to CRUIN pin of the 9900.

3.4.6 -LOAD: console LOAD input

-Load should not be used by a peripheral for the /4A system and standard Interface Card. The Interface Card sold by TI did not connect the -LOAD signal in the PBox to the console. Use of the -LOAD signal assumes use of 32K memory exapansion, since the LOAD interrupt vectors are at >FFFC and >FFFE. Therefore, peripherals for the PBox cannot use the -LOAD signal with the /4A system and interface as released by TI.

Use of the -LOAD signal is permitted with non /4A systems, or /4A systems modified to properly use the signal. If used, an open collector ('LS 125) tied to ground should drive the signal. Note- the TI-released disk drive controller drives the -LOAD signal periodically. Any new peripherals designed to utilize the -LOAD signal must either acknowledge the presence (and possible conflict) of the disk drive card, or require the user to disable the -LOAD driver on that card. TI has stated that the use of the LOAD signal on the disk drive controller card was for use with an unreleased console, the /4B.

3.4.7 -INTA: external interrupt to console

This signal informs the console that the Interrupt Service Routine in the peripheral's DSR ROM must be serviced. The signal must be driven by an open collector ('LS125) tied to ground. The gate controller must be activated and deactivated by the peripheral. The signal must be activated only when an interrupt is requested. The signal must be deactivated only after the interrupt service routine has been accessed by the console.

3.4.8 -SENILA, -SENILB: Interrupt Sense Enable Levels A + B

Values for these lines are set by the Interface Card as a high level (+5V). If the peripheral is to be used with a non /4A system, utilizing these signals, then -SENILA and -SENILB must be driven into the card by an 'LS244 or similar driver. As noted earlier, -SENILA enables 8 of 16 peripherals to drive one of 8 bits on the data bus low, while -SENILB causes the other 8 peripherals to place a unique interrupt code on the data bus; this allows the system to rapidly identify the peripheral.

If the Interrupt Sense Enable system is implemented at a later date, then each of the 16 polled peripherals may be assigned one bit on the data bus for interrupt identification as shown in Table C.2.

TABLE C.2 INTERRUPT IDENTIFICATION BITS

-SENILA Active			-SENILB Active		
CRU Address	Device	Data Bit Active	CRU Address	Device	Data Bit Active
>1300	RS232-1	D0	>1000	Unassigned	D0
>1300	RS232-2	D1	>1100	Disk drive	D1
>1400	Unassigned	D2	>1200	Unassigned	D2
>1600	"	D3	>1700	"	D3
>1500	RS232-3	D4	>1900	"	D4
>1500	RS232-4	D5	>1B00	"	D5
>1A00	Unassigned	D6	>1D00	"	D6
>1C00	"	D7	>1F00	"	D7

The RS232 positions were established by TI; see section F 2.0 for details.

To allow for development of future peripherals with this capability from various developers, the following guidelines are recommended:

1) Identify on the peripheral card and in the documentation that the device will utilize the "A" and "B" interrupt sense levels.
2) Provide a switch or jumper on board to allow the user to disable the -SENILA/B circuits.
3) Provide a switch or jumper on board to allow the user to assign the ID bit to the peripheral to match individual system hardware/software needs.

4.0 Peripheral Polling System

Several peripheral concepts for the /4A system have been discussed in this chapter- CRU-only, non-CRU memory mapped, memory only, CRU and non-DSR, and CRU and DSR. Of these, only the last (CRU and DSR) is automatically polled by the /4A operating system. Polling is a technique whereby the console will use the CRU bus to activate one of 16 peripheral locations in the >4000->5FFF memory block, and perform a function. Activation of a peripheral also activates its DSR ROM, which contains the software program(s) used with that peripheral.

As noted in Section I, the peripherals may automatically be polled by the /4A under the following conditions:

1) INITIALIZATION (RESET): Some devices require initialization of registers or other functions when the system is first activated, or following a software reset.
2) INTERRUPT: Devices that use interrupts must be polled to determine if an interrupt has occured; the interrupt must be cleared after processing.
3) DEVICE ROUTINE: This is the application program that is used to make the peripheral work. When requesting a certain device ("PIO", etc.), the console will search for the DSR that corresponds to that

device and execute the program.

4) BASIC SUBPROGRAMS (CALLS): Likewise, when BASIC or EXTENDED BASIC make subroutine CALLs, the /4A will search the available DSR ROMs for the corresponding program.

In each of the four categories, the /4A operating system performs the peripheral polling within the 16 locations defined at >0100 intervals between >1000 to >1F00. The /4A will search for the routine, starting at location >1000. If it does not find what it is looking for there, it checks the peripheral at >1100, and so on, incrementing the CRU address by >0100 until the peripheral at >1F00 is checked. If no corresponding routine is found, an error message is returned.

Peripheral devices in the other categories are not polled by the operating system, and will not be checked automatically (unless directed by a DSR in one of the 16 locations that directs the console to check a non-DSR device). It is recommended that any peripheral device that requires initialization, interrupts, device routines (independent of the 32K RAM space) or BASIC CALL subroutines, be placed in a polled peripheral space.

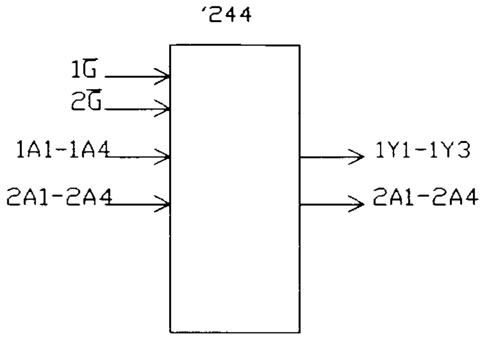
SECTION D: TYPICAL CARD CHIPS

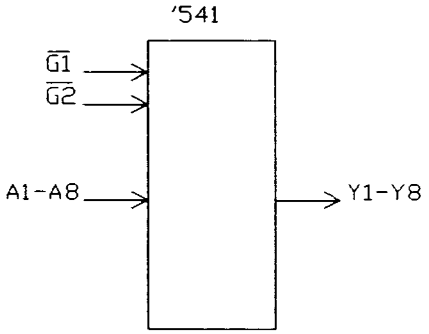
1.0 Introduction

This section is provided to assist the designer with a quick reference to integrated circuit chips commonly used in peripheral devices for the /4A system. The chips listed are not 'all' of the chips that could be utilized; the designer should have access to data books such as "Standard TTL, Volumes 1 + 2", "LSI Logic Book", "ALS/AS Logic Book", and "Interface Circuits Data Book" from Texas Instruments. A complete list of logic data books is available from Texas Instruments.

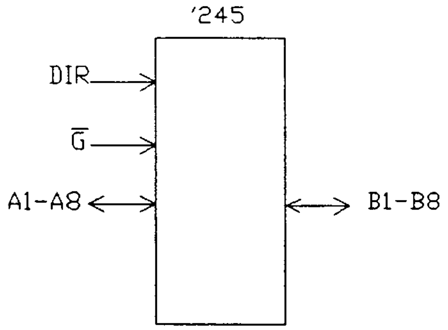
The chips are grouped into four types: drivers, logic, decode and CRU. Only basic information is given about the chips. Consult the data books for more detailed information, such as power requirements and propogation delays.

FIG. D.1: TYPICAL DRIVER CHIPS

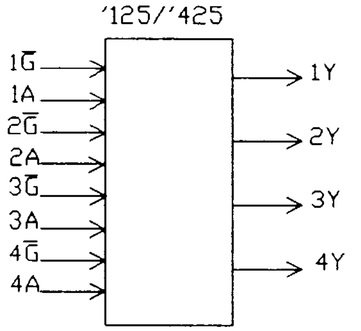




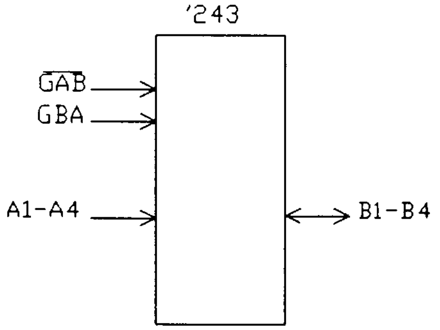
OCTAL DRIVERS



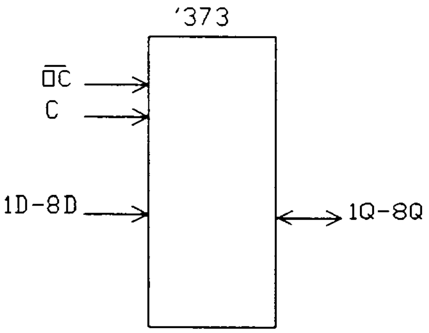
OCTAL TRANSCEIVER



QUAD DRIVER, INDEPENDENT
OUTPUTS, OPEN COLLECTOR

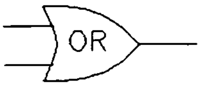


QUAD TRANSCEIVER



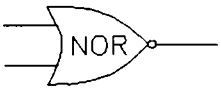
OCTAL TRANSPARENT
LATCHES

FIG. D.2: LOGIC CHIPS



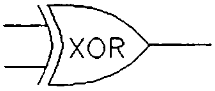
A	B	OUT
L	L	L
L	H	H
H	L	H
H	H	H

'32 QUAD 2 INPUT



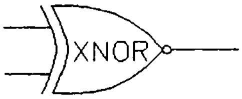
A	B	OUT
L	L	H
L	H	L
H	L	L
H	H	L

'02 QUAD 2 INPUT
'27 TRIPLE 3 INPUT
'25 DUAL 4 INPUT W/STROBE



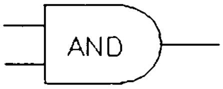
A	B	OUT
L	L	L
L	H	H
H	L	H
H	H	L

'136 QUAD 2 INPUT, OC OUTPUT



A	B	OUT
L	L	H
L	H	L
H	L	L
H	H	H

'135 QUAD XOR/XNOR



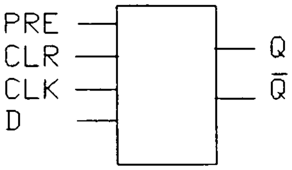
A	B	OUT
L	L	L
L	H	L
H	L	L
H	H	H

'08 QUAD 2 INPUT
'11 TRIPLE 3 INPUT
'21 DUAL 4 INPUT

A	B	OUT
L	L	H
L	H	H
H	L	H
H	H	L

'00 QUAD 2 INPUT
'10 TRIPLE 3 INPUT
'20 DUAL 4 INPUT
'30 8 INPUT
'133 13 INPUT

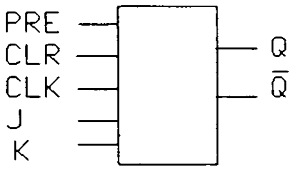
D FF



PRE	CLR	CLK	D	Q	Q
L	H	X	X	H	L
H	L	X	X	L	H
H	H	↑	H	H	L
H	H	↑	L	L	H

'74 DUAL D TYPE

J-K FF



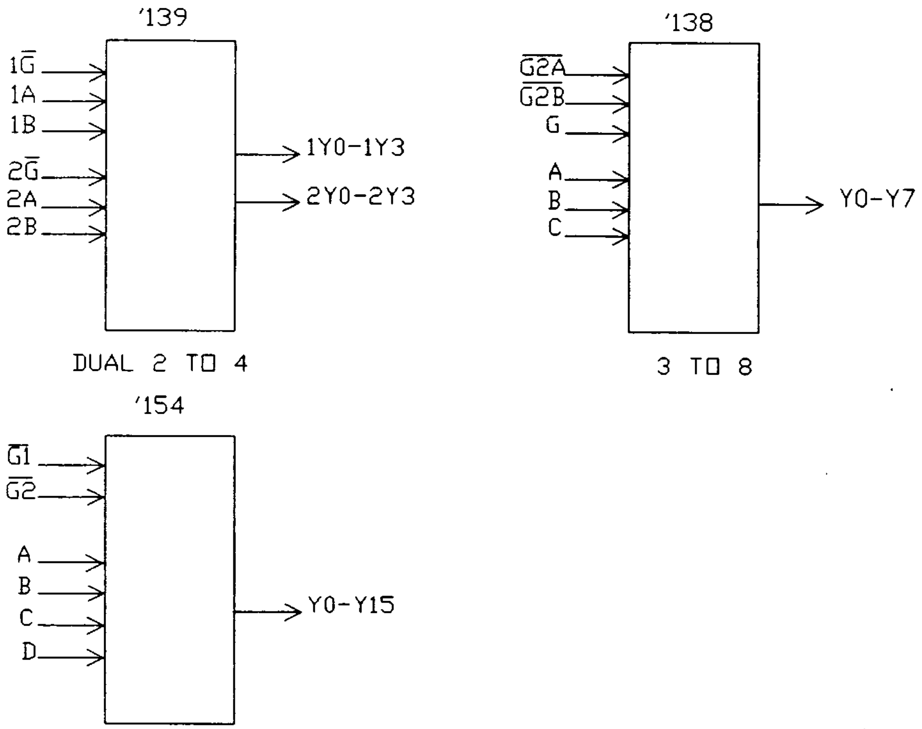
CLR	CLK	J	K	Q	Q
L	X	X	X	L	H
H	⎍	H	L	H	L
H	⎍	L	H	L	H
H	⎍	H	H	TOG.	

'73/'70 POSITIVE EDGE
'73A NEGATIVE EDGE

PRE	CLR	CLK	J	K	Q	Q
L	H	X	X	X	H	L
H	L	X	X	X	L	H
H	H	⎍	H	L	H	L
H	H	⎍	L	H	L	H
H	H	⎍	H	H	TOG.	

'76/'71/'72 MASTER/SLAVE
'78 DUAL W/COMMON
CLK & CLEAR

FIGURE D.3: DECODER CHIPS

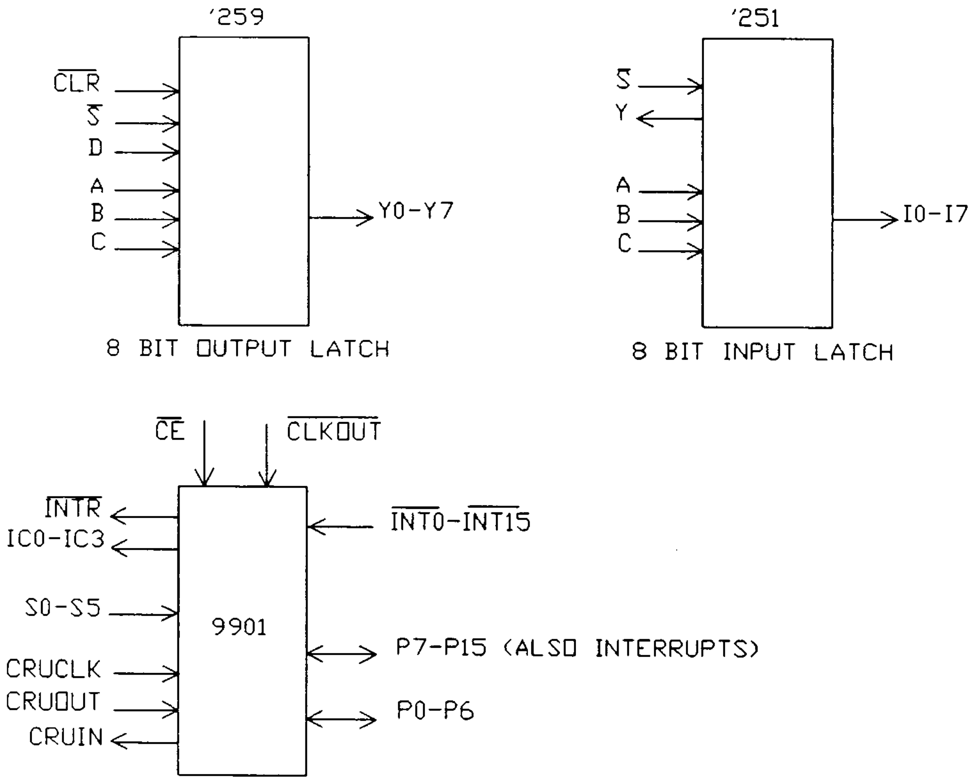


DUAL 2 TO 4

3 TO 8

4 TO 16

FIGURE D.4: CRU CHIPS



8 BIT OUTPUT LATCH

8 BIT INPUT LATCH

SECTION E: TYPICAL CIRCUIT EXAMPLES

1.0 Introduction

Peripheral devices for the /4A system can vary greatly in complexity and function. However, most of them will have to be interfaced to the memory bus, CRU bus, or both. The following sections present some typical circuit examples to demonstrate how interfacing may be accomplished on the /4A system. As with Section D, Typical Chips, these circuits are not necessarily optimal for all peripheral devices, but are presented here as a reference for the designer.

2.0 Memory Interface

This example shows how a device incorporating the 32K RAM could be assembled. The integrated circuits used are summerized below, along with their function. This example has address decoding, data bus buffering, and generating of the -RDBENA signal; these circuits are common with most peripheral devices and are not repeated for the other examples.

Chip	Function
244	address and control signal drivers for A0-15, -WE, DBIN
245	data bus transceiver for D0-7
138	decode A0-A2 into 8K chip select banks; selects >2000, >A000, >C000 and >E000 8K blocks
21	4 input AND gate, activates the -RDBENA and 245 chip enable, inputs are from '138 chip select signals
04	1 of 6 hex inverter, used to convert DBIN to -DBIN (-OE for 8K RAM and 245)

3.0 CRU Interface

CRU interface can be implemented rather easily with relatively common chips. Figure 3a shows how an 'LS259 is used to latch up to 8 individual CRU bits. The 'LS138 is used as an address decoder, while -CRUCLK is used as an enable signal (to prevent activation during a normal memory bus access). The 8 individual bits are selected by address lines A12-A14, and the CRUOUT line inputs the value of the bit (0 or 1). The -RESET line clears all the bits when the system is reset. The 'LS259 is used in the Latch mode which means that the value of the CRUOUT line at the time of access is held constant until it is either rewritten or reset. Software must be written to insure that CRU bits are not accidently left on when a peripheral device is no longer accessed.

Figure 3b demonstrates how to input data via the CRU bus. Once again, the 'LS138 is used for address decoding. The 'LS251 is used as a 1 of 8 data input, with address lines A12-A14 selecting the input line. Once the line is selected, the CRUIN line reads the value (0 or 1). The -CRUCLK signal is not used, since it is for data output by the CRU bus only. Data at the 8 inputs to the 'LS251 must be valid before being read by the CRUIN line.

The most versatile, and often underused, chip for CRU interfacing is the TMS9901. The 9901 can provide 6 dedicated interrupts, 7 dedicated I/O CRU bits, and 9 lines that can individually be programmed as either interrupts or I/O bits. In addition, it has a 16 to 4 interrupt prioritizer (which is not used in peripheral designs for the /4A, due to the limited interrupt structure), and a built-in programmable timer that can be preset to interrupt at a specified interval. Figure 3c shows the basic interconnect for a 9901 to the PBox bus. Address decode circuitry selects the chip, while the full CRU bus is directly connected to the 9901. The lower address bits A10-A14 are used to select the I/O and interrupt pins. When an interrupt occurs, the -INTREQ line drives the -XINT line low, and it is up to the software to read the interrupts internally to determine which one was active. -CRUCLK must be inverted back to positive CRUCLK for the CRUOUT line to function properly.

4.0 Memory mapped interface

Memory mapped decoding is similar to regular memory interfacing, in that various address lines are used to select a particular device. In Figure 4a, two 'LS138s are used to decode the 6 most significant address lines. With this scheme, the second '138 provides 8 select lines that will activate individual 1K blocks. These 8 1K blocks reside within the 8K block chosen by the first '138 decoder. When used in a polled peripheral, this 8K block would be >4000->5FFF. CRU decoding is used to select the peripheral space and activate the device (or DSR ROM) only if both the address is valid, and the first CRU bit in the CRU peripheral space is set.

Figure 4b also uses two chips, but can decode all 16 bits to an individual address. The first '688 compares the first 8 MSB (A0-A7) to a value set by an 8 switch DIP set. When these 8 bits are equal, they enable the second '688, which does a similar comparison for the lower 8 address bits. The second '688 produces a low true signal when the 16 address bits equal the value set on the DIP switches. Once again, this is gated with the appropriate CRU bit to activate the peripheral.

The non-CRU memory mapped decode circuitry is more complex, as shown in Figure 4b. Once again, an 'LS138 is used to decode A0-A2 to enable the >4000 8K block. The 'LS154 further decodes A3-A7 into all 16 possible polled peripheral locations (>1000->1F000). All 16 outputs are routed to two 'LS21s (4 input AND gate) which are connected such that the final AND gate output goes low if any of the 16 inputs goes low. This, along with other signals is latched by a 259. This circuit will provide a master output signal, -Qo, that follows the following logic:

IF -Qo is low THEN a polled perpiherial is active and the non-CRU memory mapped device CANNOT BE ACTIVE.

IF -Qo is high THEN no polled peripheral is active and the non-CRU memory mapped device CAN BE ACTIVE.

This circuit must be combined with other address decode circuitry to select the peripheral within the >4000 block.

FIG. E.2: MEMORY INTERFACE EXAMPLE

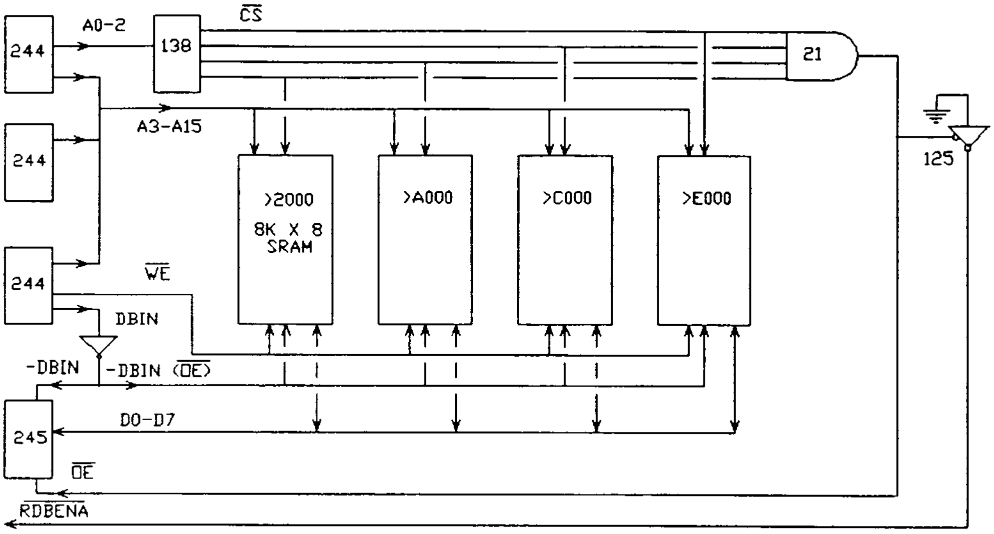


FIG. E.3: CRU INTERFACE EXAMPLES

FIG. E.3A SIMPLE OUTPUT

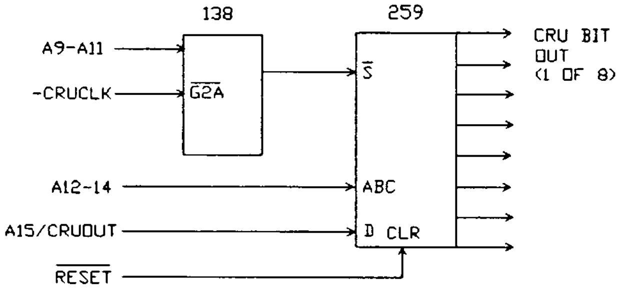


FIG. E.3B SIMPLE INPUT

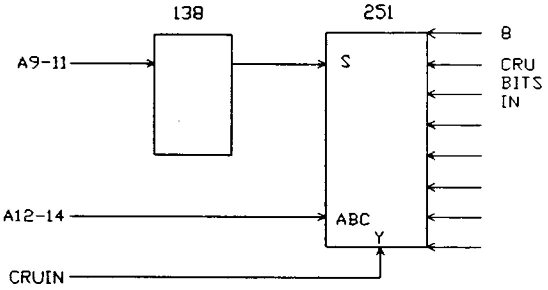


FIG. E.3C 9901 EXAMPLE

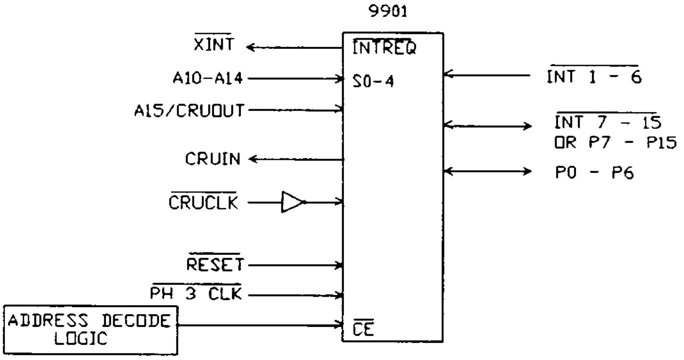
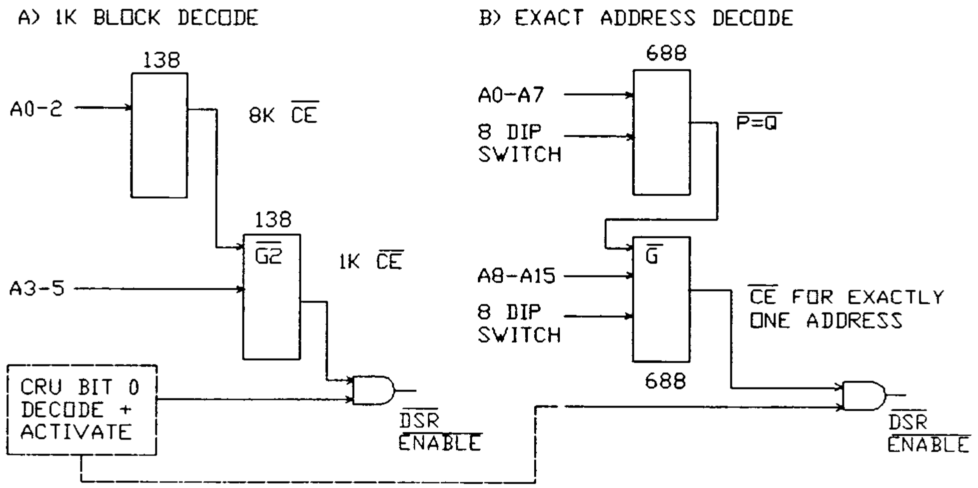
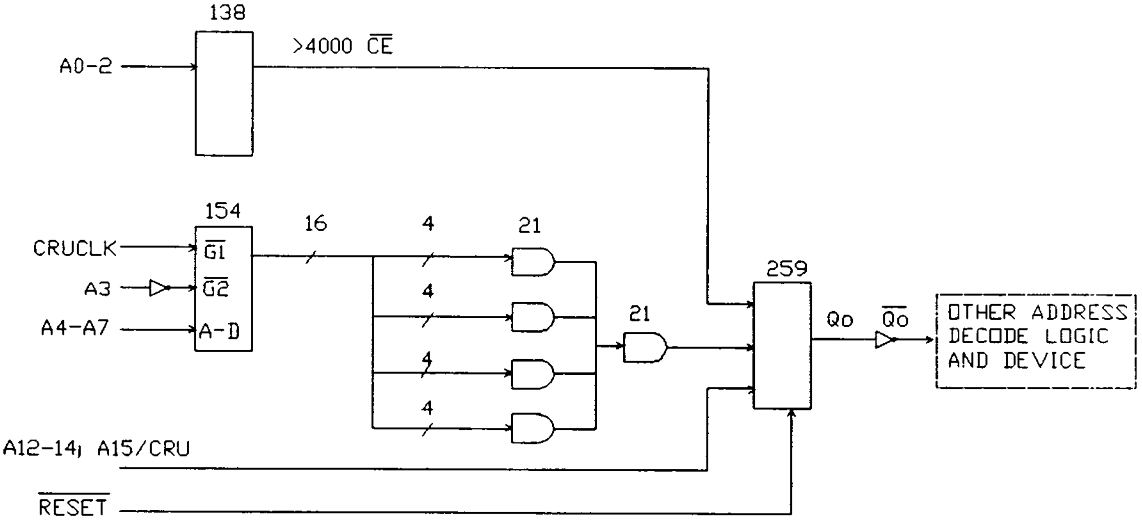


FIG. E.4: MEMORY MAPPED EXAMPLE


A) 1K BLOCK DECODE

B) EXACT ADDRESS DECODE

FIG. E.5: NON-CRU MEMORY MAPPED



SECTION F: TI DEVELOPED CARDS

1.0 Introduction

Texas Instruments developed and sold 4 cards for the peripheral expansion box: the RS232, 32K memory, disk drive controller and P-code card. Reference 4, the Bunyard Manual, has a very detailed explanation of how each of these cards are put together. For purposes of comparison of design methodology (this document vs. TI), a brief description of the interface circuitry for the RS232, 32K memory and disk drive controller is provided.

2.0 RS232 card

'LS244s are used to drive A0-A15, AMA-AMC, -CLKOUT, -MEMEN, -WE, -CRUCLK and DBIN. An 'LS245 is used for D0-D7, with DBIN determining the data flow direction. An 'LS125 is used to drive pin 17, -XINT, and -RDBENA. CRUIN is brought on the board unbuffered, as well as PCBEN. The majority of the address decode is done by a Programmable Array Logic (PAL) chip. On the RS232, as well as the other cards, that AMA-AMC and PCBEN must be active high to select the card. The RS232 card can also drive D0 and D1, or D4 and D5 if an interrupt occurs, and -SENILA is brought low. The Bunyard Manual notes that the CRU address of the card can be changed by moving one resistor.

3.0 32K memory card

'LS244s are used to drive A0-A15, AMA-AMC, -MEMEN, and DBIN. Neither CRUCLK or -WE are used by this card. -RDBENA is driven by an 'LS125 when the card is selected, and an 'LS245 controls the data on D0-D7, with DBIN controlling the direction. The interesting item on this card is the lack of use of the -WE signal. A PAL controls the 32K worth of dynamic RAM, which is much more complicated than need be if static RAM were used. Also, PCBEN is not used in the decode logic.

4.0 Disk drive controller card

Once again, 'LS244s drive A0-A15, -MEMEN, -WE, -CRUCLK, DBIN, AMA-AMC, and -CLKOUT. PCBEN and CRUIN are unbuffered, as well as -RESET. An 'LS245 carries the data on D0-D7. -RDBENA and READY are driven by an 'LS125. A PAL is used for most of the address decode logic. The disk drive controller does not drive -XINT, but does have an active, but unused interrupt signal -INTRQ. This originates from the WD1771 controller chip, and is connected to pin 18, -LOAD on the PBox bus. An 'LS125 also drives D0 low if -SENILB is low.

SECTION G: PERIPHERIAL LOCATION ASSIGNMENTS

1.0 Introduction

As noted previously, there are 16 CRU defined locations available for peripherals for the /4A system. Of these 16, four locations are assigned to existing products released by TI. Other locations were reserved by TI, but the planned peripherals were not released. Table G.1 lists the 16 peripheral spaces by their CRU address, indentifies the address values to decode that space, and defines the assignment of all spaces. The functions were arbitrarily assigned, but were done so to help developers determine where their product should be located in the peripheral space, thereby minimizing conflicts between different devices. It is recommended that peripheral developers include circuitry on their devices (similar to the examples in Section E) to allow the user to select the CRU location for the device in their individual systems. While this design feature could result in conflicts between two devices accidently assigned to the same peripheral space, it does allow maximum flexibility for the end user. Any documentation accompanying the device should clearly identify the recommended CRU location.

The space assignments are choose to correspond with individual peripheral products currently available with various computer systems. Availability of any peripheral product listed in Table G.1 is dependent upon the efforts of the hardware developer, and does not necessarily imply that such a product exists, or will at a later date. Two peripheral spaces are left undefined to allow for prototype projects, or future devices whose function is not clearly defined elsewhere in Table G.1

Table G.1
Peripheral Location Assignments

Peripheral Space	Addr. Lines	Established Function*	Assigned Function	Notes
	A34567			
>1000->10FE	10000	-	mass storage	1
>1100->11FE	10001	disk controller	disk controller	
>1200->12FE	10010	(home security)	math coprocessor	2
>1300->13FE	10011	RS232-1	RS232-1	
>1400->14FE	10100	(internal modem)	internal modem	3
>1500->15FE	10101	RS232-2	RS232-2	
>1600->16FE	10110	(digital cassette)	prototype low	4
>1700->17FE	10111	Hex Bus	attached computer	5,6
>1800->18FE	11000	thermal printer	MIDI/music	5,7
>1900->19FE	11001	(eprom programmer)	programmer	8
>1A00->1AFE	11010	(student typing)	speech/DSP	9
>1B00->1BFE	11011	(debugger card)	Utility card	10
>1C00->1CFE	11100	video	video	
>1D00->1DFE	11101	IEEE 488 control	real time clock	5,11
>1E00->1EEF	11110	-	prototype high	4
>1F00->1FFE	11111	P-code	P-code	

*Items in parenthesis denote third party or unreleased TI devices.

Table G.1 Notes

1) Mass storage is currently defined as RAM disks, but also includes other media such as CD-ROM.

2) Position >1200 is reserved for high speed math coprocessors, which may be interrupt driven and require quick response times.

3) An internal modem is defined as a standalone peripheral with direct connection to telephone lines, with no interface to the RS232 devices.

4) Prototype locations (low and high) are left undefined for prototype circuits and undefined future products.

5) These card positions were defined by TI; however, the peripheral either was not released, or is seldom used. Therefore, this peripheral space was reassigned.

6) Attached computer or microprocessor refers to a self contained computer system with its own microprocessor. This space can also be utlitized for interfacing to independent computers.

7) This space is reserved for electronics music devices.

8) This space is reserved for programmers of various devices, such as PROM, E(E)PROM, PAL, etc.

9) This space is reserved for speech and/or digital signal processing devices.

10) A utility card refers to peripherals designed to enhance or supplement development of assemlby or other advanced program applications.

11) Real time clock peripheral space; also used as RTC space by some third party products.

SECTION H: MISCELLANEOUS DESIGN CONSIDERATIONS

1.0 PBox Peripheral Card Dimensions and Layout

Figure H.1 gives the physical dimensions of a printed circuit board designed to fit in the /4A PBox. These dimensions are taken from the prototype board, and assume that the card will be bare (ie- will not use a 'clamshell' cover like the original TI cards). These dimensions are extremely useful for designers who are planning to produce PCBs for kits or final projects, or for the hobbiest who constructs his or her own one-of-a-kind PCBs. The extension section in the rear is optional, and is designed to extend outside of the PBox; it is not needed unless external connections are used by the card. Positioning of the indicator LED is relatively critical - it must line up with the built-in lens of the PBox to give the user a good strong light signal.

Drivers and buffers must be physically located as close to the 60 pin edgeboard as possible. Unregulated +8V, +16V, and -16V are provided at opposite ends of the card for input to voltage regulators. If the +16V and -16V pins are not used, it is recommended that their edge connectors not be put on the PCB. This will eliminate accidental shortage of the unregulated voltage with adjacent signals, such as -MEMEN. Voltage regulators may be mounted directly to the PCB for heat sinking purposes. Voltage regulators should have heatsinks with heat sinking compound in most designs, particularily if the circuit draws more than one-half of the rated output of the regulator. Regulators should also have enough de-spiking capacitors to ensure reliable performance.

Layout of other components on the PCB should not be critical. PCB traces with signals or power feeds should not be routed near the front or back, where they could accidently short to ground against the PBox chassis. All PCB areas not utilized for traces should be left unetched (ie- solid copper), and tied to ground to act as a ground plane, and minimize external signal interference. Any connecting hardware such as plugs, sockets, etc., that will have cables inserted and removed should be bolted to the board to prevent damage to the PCB from repeated insertion/removal.

Each peripheral card should not use more than the following maximum power on the three unregulated power buses:

500 ma on 8V
250 ma on 16V
30 ma on -16V

This is a function of the PBox power supply, split over a maximum of seven cards. If more power is required by an individual card, an independent power supply with common ground should be used.

2.0 Prototype Board

Designers wishing to test their peripheral circuits prior to manufacturing PCBs can utilize a prototype PBox board, which is currently available from LL Conner Enterprises. This is a wire wrap type board with all signals brought on board from the bus, and positions for bus drivers and buffers provided. Multiple positions for memory and general purpose chips are also provided, as well as for the voltage regulators.

If the prototype board is unvailable, then prototypes can be constructed from breadboard/wirewrap board with a 60 pin plug that is compatible with the PBox bus (TI part # L21111121-30).

3.0 Extender cable

Access to circuit boards installed in the PBox is extremely limited. To facilitate easier testing and troubleshooting of prototype devices, the hardware designer may wish to construct a bus extender cable. A bus extender cable may be constructed by connecting a 60 pin edgeboard plug to a 60 pin edgeboard connector (0.1" pin to pin spacing for both) with two 30 conductor ribbon cables. A maximum of two feet of cable is usually desirable to allow for ease in placing the prototype card in a convenient location. Use of shielded cable is recommended to minimize EMI/RFI interference.

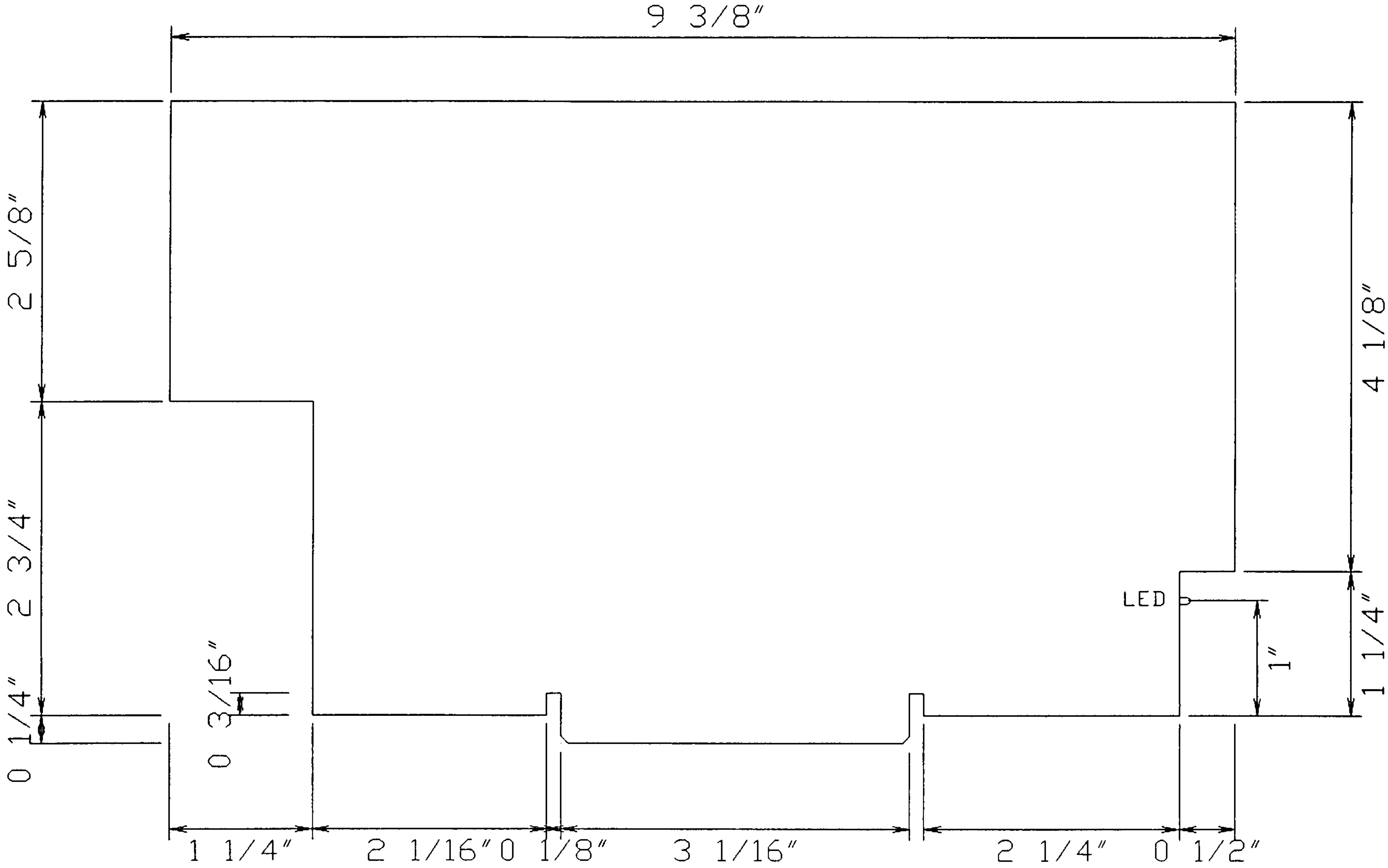
4.0 Modified Interface Card

As noted in several previous sections, the Interface Card sold with the PBox limits the use of some signals in the PBox bus. Figure H.2a is a simplified diagram of a modified interface card that duplicates the function of the original interface card, but also allows use of some of the restricted signals.

AMA-AMC, -SENILA/B, and HOLD are connected to +5V via resistors, but also have a DIP switch set that allows these lines to float (neither +5V or ground), like SCLK, IAQHA, etc. PCBEN must be held to 5V to allow the TI developed cards to operate, and does not have a switch. -LCP is tied to the chip enables for the address and data drivers such that a low value will disable the interface card. The design shown in Figure H.2a would be useful in implementing a system that has an independent computer/microprocessor in a peripheral space that occasionally takes over the PBox bus from the /4A. Software between the two systems would be responsible for disabling and enabling the interface card. [This system would not be required if HOLD and -HOLDA were available at the /4A 44 pin side port.]

A slightly different design for the interface card is shown in Figure H.2b. Here, signals such as AMA-AMC, -SENILA/B, etc., are taken to the console end of the interface cable, where an undefined 'black box' is used to generate signals under software control. As an alternative, the black box circuitry could be on the interface card in the PBox. This design assumes the /4A will generate all new signals.

FIG. H.1 PE BOX CARD DIMENSIONS



ALL DIMENSIONS IN INCHES

FIG. H.2A: MODIFIED INTERFACE - INDEPENDENT MICRO IN PBOX

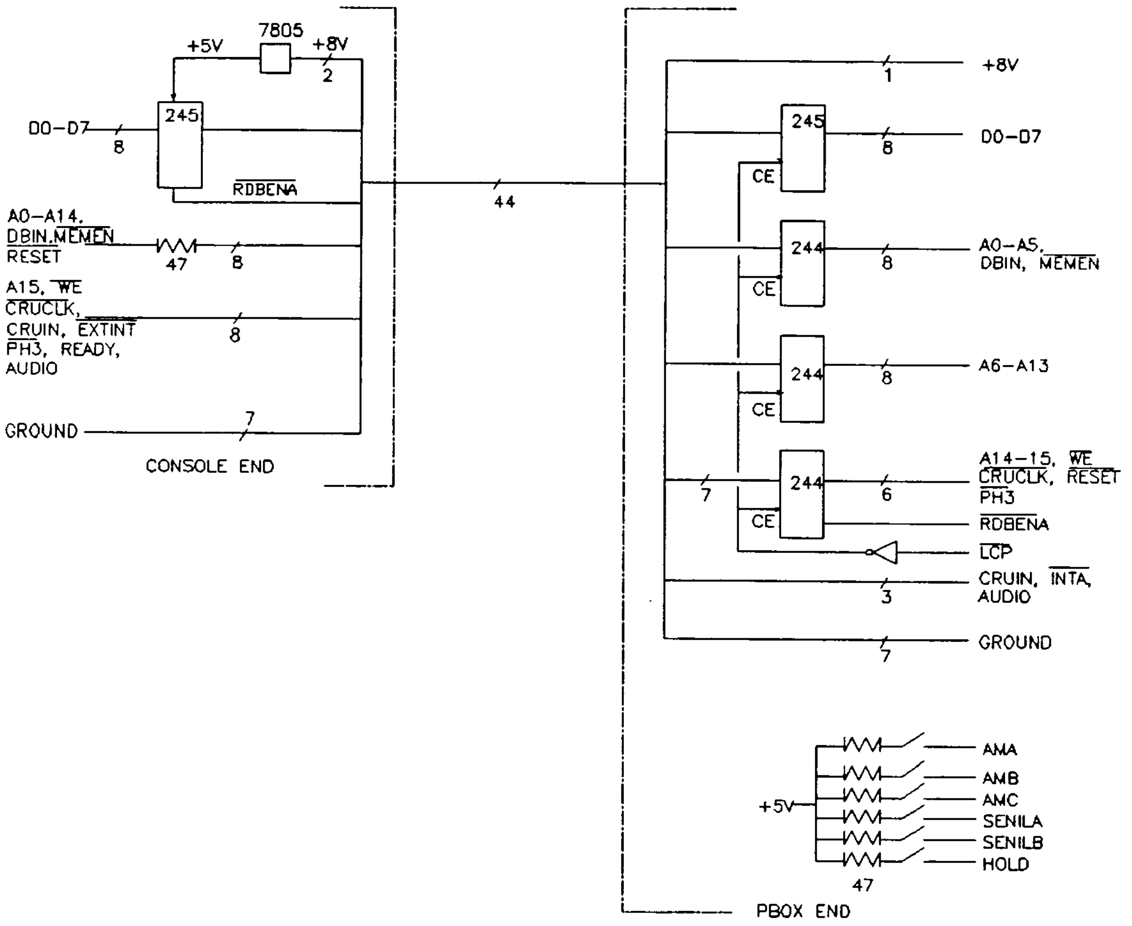
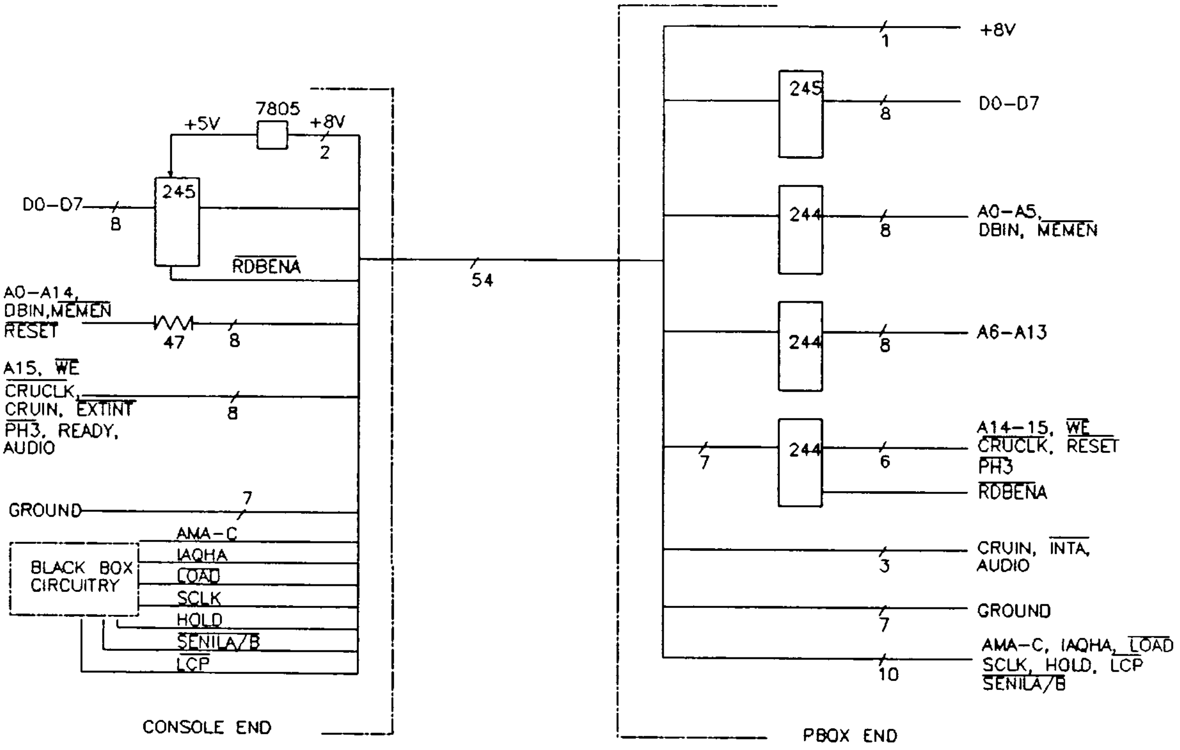


FIG.2B: MODIFIED INTERFACE-ALL SIGNALS ACTIVE



SECTION I: DSR ARCHITECTURE

1.0 Introduction

This section briefly covers the software required to make peripherals compatible with the /4A and its File Management System (FMS). Software for a standard peripheral is referred to as a Device Service Routine, or DSR. Section I discusses the basic parts of a DSR and gives coding examples. Section J examines how the console accesses DSRs. Section K contains miscellaneous information on direct access of peripherals and the non-DSR peripherals from a application program, and information on Peripheral Access Blocks (PABs) and how they interact with DSRs.

2.0 Device Service Routines

Device Service Routines are included on many peripherals to allow the /4A to communicate with the device(s) located on the peripheral. To simplify the addition of new peripherals, the /4A has a defined protocol for interaction with peripherals during inital powerup, interrupts, and main device programs. Peripherals may also add new subprograms (CALL XXX) to BASIC and XBASIC. The /4A communicates with each peripheral in exactly the same manner; it is up to that peripheral's DSR to define how the peripheral operates. This way, new peripherals may be added without altering the console routines. More information on the /4A and its File Management System can be found in the Editor/Assembler manual, or /4A Peripheral Technical Data manual.

DSRs can be located in either GROM or ROM (PROM, EPROM, EEPROM, battery backed SRAM) devices. Since GROMs are beyond the scope of this manual, further discussion of DSRs will be limited to ROM-type applications only. DSRs can be composed of several different kinds of routines, depending on what functions the peripheral is to perform. Six types of DSRs are defined: power up, user application, main device service, subroutine links, BASIC subprogram libraries, and interrupt service programs. The /4A finds and executes these programs by searching the ROM header, which is in the first 10 bytes of the ROM. The console identifies valid DSR ROMs by checking that byte 0 is "AA". [Note: non-DSR peripherals may have applications programs in ROM at the same location, but must be called by an independent program in the console, and must not have the validation byte (0) set to "AA".] Byte 1 contains the version number of the DSR. The remaining bytes in the header identify the entry points for the various programs used by the DSR. Table I.1 identifies the contents of the DSR header.

Table I.1: DSR Header

Location	Size	Contents
>4000	byte	>AA valid ID
4001	byte	version number
4002	byte	number of application programs, set to zero
4003	byte	reserved, set to zero

4004	word	address of first power up header
4006	word	address of first user program header*
4008	word	address of first main device header
400A	word	address of first subroutine link header
400C	word	address of first interrupt link
400E	word	address of first BASIC subprogram library*

[*Only in GROM or at >6000 location]

The address of any routine types should be >0000 in the ROM header if there are no routines of that type in the DSR. The number of application programs and version number values (bytes 1 and 2) are ignored by the /4A system. Program entry addresses may be placed anywhere within the >4010 - >5FFF range, and multiple routines per program type are allowed (ie- there may be more than one main device routine, interrupt, etc.). The address of the program type given in the header is the link to the next routine of that type. At the first address is the linking address of the next routine; the word immediately after this address is the entry point of the first routine. If the linking address is zero, then no more routines of that program type are available. See Examples I.1 - I.5.

Not all peripherals require all types of routines; it is up to the designer/programmer to determine which routines the peripheral's DSR will require. As noted in Table I.1, there are four types of programs available for DSRs in ROMs: power up, interrupt, main device routine and subroutine link [also referred to as BASIC CALL (sub), or low level routines]. These program types are discussed next.

2.1 Power up routines

Some peripherals require intialization upon power up or following a system reset. Or a power up routine may be included simply to flash the peripheral indicator light to let the user know that the device is active. In either case, they will require a power up routine in the DSR. The /4A initializes the console upon power up, then searches and executes all peripheral DSR power up routines.

Each power up routine can use R0-R10 of the GPLWS. R12 will be set up with the proper CRU address for the peripheral (which is also the CRU address used to enable the peripheral). R11 contains the return address. R13 and R15 contain the memory mapped addresses of GROM Read Data and VDP Write Address, respectively. All VDP and GROM operations can be indexed from these two registers. R14 contains the status flags and should not be altered. The power up routine may use VDP RAM from >0000 to the address pointed to by >8370 [note that the VDP and its memory are not completely initiated at this point]. It may also use all console scratch pad RAM except >8355->836D and >83C0->83DF. Errors are not assumed to occur during execution of power up routines, and there are not provisions in the FMS for identifying power up errors. The power up routine may print an error message on the title screen to let the user know of a problem with the peripheral. If there are no errors, the power up routine returns with a B *R11.

EXAMPLE I.1: POWER UP ROUTINES

```
        AORG >4000  start of DSR
        BYTE >AA    validation byte
        BYTE 1      version number
        DATA 0      reserved
        DATA PU1    first power up link
        -
        -
PU1     DATA PU2    link to second power up
        DATA PU1EN  entry point of 1st power up routine
        BYTE 0      name length, set to zero
        EVEN        reset pointer to word boundary
PU2     DATA >0000  no more power up routines
        DATA PU2EN  entry point of second power up
        BYTE 0
        EVEN
        -
        -
PU1EN   $           entry of 1st power up routine
        -
PU2EN   $           entry of 2nd power up routine
        -
EXIT    B *R11      return to console
```

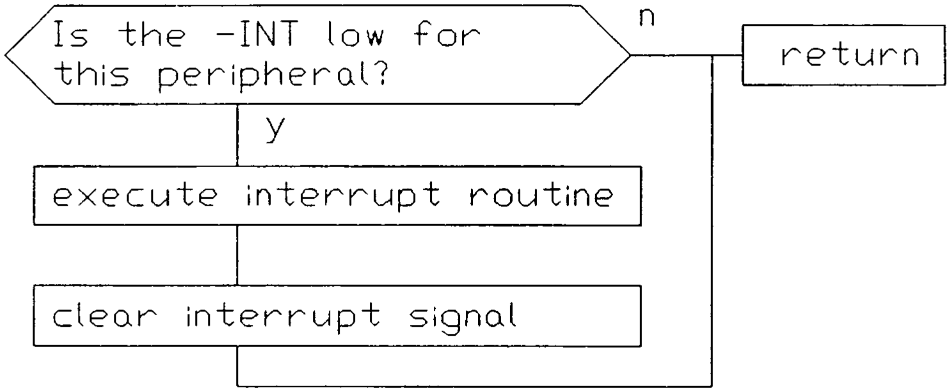
2.2 Interrupt routines

Interrupt routines are required for peripheral devices that generate interrupts, and are connected to the -INTA line. Since the /4A system scans all peripheral interrupt routines to determine which DSR caused the interrupt, the DSR must be capable of checking the peripheral to determine if it generated the interrupt. The interrupt routine must clear the interrupt when the routine is complete. Generally, peripheral devices will generate their own -INT signal, which is cleared by DSR software. Other designs may incorporate a TTL latch or flip flop that is cleared by CRU or other signal. In any case, the -INT line must be cleared prior to exiting, or the console will continue to branch to that interrupt routine indefinitely. The flow diagram for the interrupt program logic is shown in Figure I.1.

The interrupt routine can use R1-R10 of GPLWS, except for R9. The contents of R13 - R15 are the same as for the power up routines; R11 contains the return address. Because of the execution of an interrupt routine only as part of a DSR, the DSR and interrupt routine can split the allocation of scratch pad RAM from >834A to >836D. An interrupt routine is always exited by a B *R11. As with power up routines, no provisions are made for reporting errors that occur during an interrupt routine.

The RS232 card as released by TI assumes that no other peripheral will cause an interrupt while it is in use. Interrupt driven peripherals must have routines that either acknowledge this feature, or can function around the RS232 card. Refer to the RS232 interrupt routine as disassembled and commented in "Technical Drive".

FIG. I.1: INTERRUPT ROUTINE



EXAMPLE I.2: INTERRUPT ROUTINE

```
          AORG >4000

          BYTE >AA
          BYTE 1
          DATA >0000
          DATA -----            powerup routine
          DATA >0000
          DATA -----            main device routine
          DATA -----            subroutine link
          DATA INT1             1st interrupt link
          DATA >0000
          -
          -
INT1      DATA INT2             link to 2nd interrupt routine
          DATA INTEN1           entry point of 1st routine
          BYTE 0                name length set to zero
          EVEN                  reset WP
INT2      DATA >0000            no more int. routines
          DATA INTEN2           entry point of 2nd routine
          BYTE 0
          EVEN
          -
          -
INTEN1    $                     entry of 1st device
          [check for interrupt; if none go to END]
          [interrupt service routine]
          [go to CLEAR]
          -
INTEN2    $                     entry of 2nd device
          [check for interrupt; if none go to END]
          [interrupt service routine]
          [go to ClEAR]
          -
CLEAR     $                     clear interrupt
          [reset -INT signal]
          -
END       B *R11                return
```

2.3 Main Device Routine

The Main Device Routine(s) defines the function(s) of the peripheral, and must be included on all peripherals with devices to be accessed. As with the power up and interrupt routines, there may be multiple main device routines for one peripheral, such as for the RS232 card. Main device routines are called via the File Management System in the BASIC/XBASIC environment, which establishes PABs in VDP memory for each opened file. The device name is located in VDP RAM, and is pointed to by a word value at >8356. The device name and character count byte is also included in the PAB. The main device routine is called by either the File Management System, or by DSRLNK in an applications program.

R12 will contain the CRU address of the peripheral being addressed, and R11 contains the return address. Registers R0-R10 can be used by the routine as well as >834A ->836D. If an error occurs, the DSR must set the error codes in the PAB, as defined in Section K, and perform a B *R11. If no errors occur, the routine must increment R11 by two prior to exit. However, if the peripheral is not interested in responding to the call (ie- the same device name may be on more than one peripheral), it may return via B *R11.

The main device routines often require extensive coding due to numerous housekeeping responsibilities that they must perform. These responsibilities include:

1) Maintain interface with FMS

Device routines are accessed in terms of files and records. [See Section 18 of the Editor/Assembler manual]. The designer/programmer must determine what data format(s) is appropriate for interfacing with the FMS. Changes to the I/O status of a peripheral must also be handled by the device routine.

2) Respond to STATUS I/O opcode

As noted in Section K, the DSR should be capable of responding to a STATUS I/O request by updating byte 8 of the PAB. This byte is used to determine the current status of the peripheral.

3) Report any errors

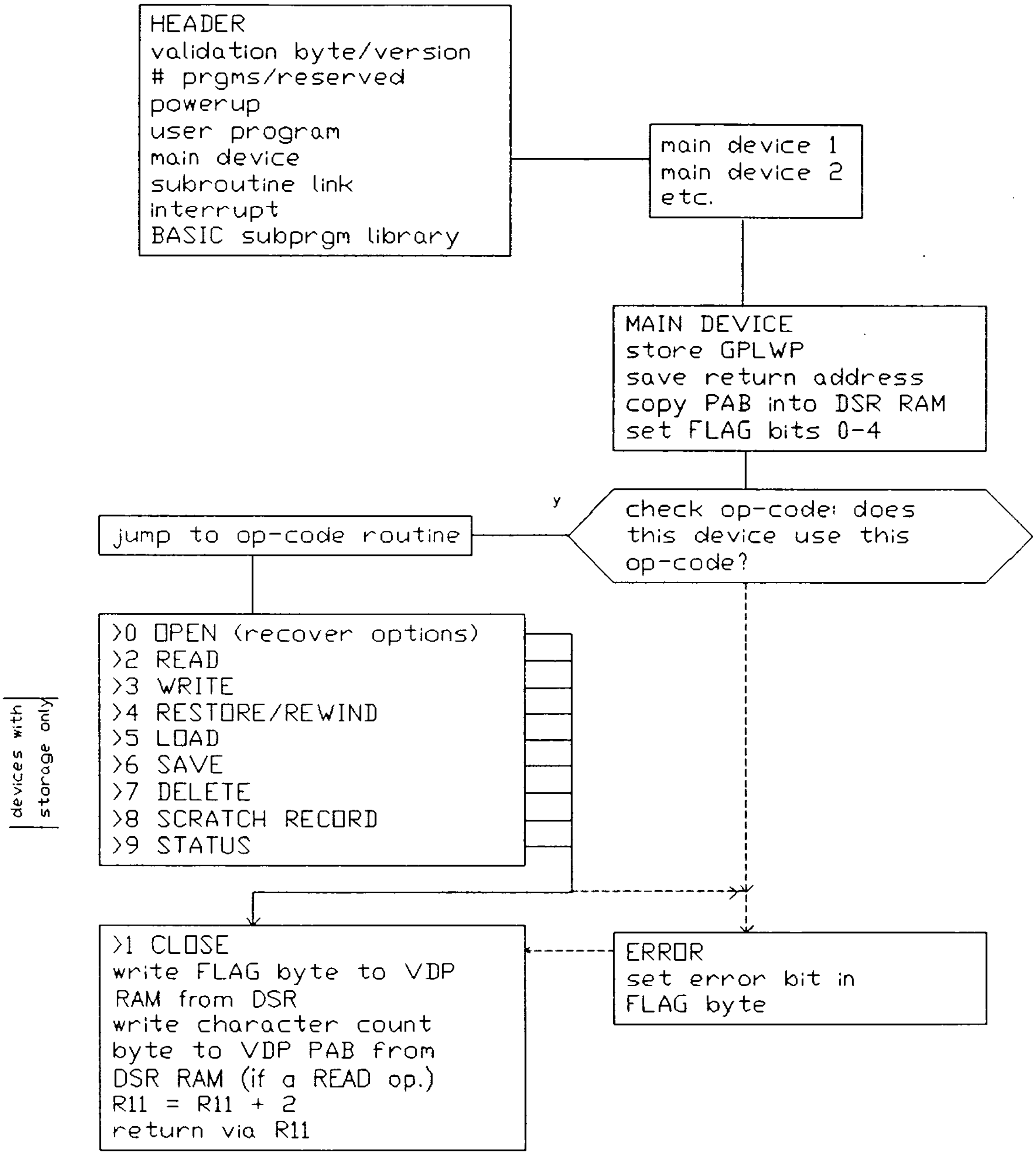
The DSR also reports errors that occur during processing of main device routines before returning. Section K defines the error codes and their meaning. Errors are reported in the FLAG byte of the PAB.

4) Maintain device housekeeping

The DSR must also tend to any requirements of devices located on the peripheral while active. The DSR must also properly disable the device prior to terminating access.

The basic flowchart for a main device routine is given in Figure I.2. It is based upon a review of the CLOCK, RS232, and Disk Drive Controller DSRs. The programmer is responsible for determining what I/O opcodes are used with the peripheral, and developing the appropriate code. Example I.3a is a typical main device routine example. Example I.3b is the disassembled and commented DSR for the CORCOMP 9900 Clock card (from "Technical Drive", by Monty Schmidt, reprinted with permission). The programmer may consider using a similar structure in developing main device routines for new peripherals. The programmer is strongly encouraged to review the disassembled and commented RS232 and Disk Drive Controller DSRs given in the manual "Technical Drive".

FIG. 1.2: MAIN DEVICE ROUTINE



EXAMPLE I.3a: MAIN DEVICE ROUTINES

```
        AORG >4000
        BYTE >AA
        BYTE 1
        DATA >0000
        DATA ----
        DATA >0000
        DATA DSR1       link to 1st device routine
        DATA ----
        DATA ----
        DATA >0000
        -
        -
DSR1    DATA DSR2       link to next DSR
        DATA DSREN1     entry point of 1st device
        BYTE 4          name length of 1st device
        TEXT 'DEV1'     name of 1st device
        EVEN
DSR2    DATA >0000      no more DSRs
        DATA DSREN2     entry to 2nd device
        BYTE 4          name length of 2nd device
        TEXT 'DEV2'     name of 2nd device
        EVEN
        -
        -
DSREN1  $               entry point of device 1
        -
DSREN2  $               entry point of device 2
        -
        -
        -
ERROR   $               entry point of error reporting routine
        [set error bits]
        -
        B *R11          do not increment R11
        -
OKEND   $               no errors return
        INCT R11        increment R11 by two
        B *R11          return
```

```
**************************************************************************
*                                                                        *
*                Source code for CORCOMP 9900 Clock card                 *
*                                                                        *
*                  Dissassembled and commented 3/20/86                   *
*                                                                        *
*                           by Monty Schmidt                             *
*                                                                        *
**************************************************************************

4000 AA01  DATA >AA01                   ** Valid DSR identifier and version
4002 0000  DATA >0000                   ** Not used in DSR calls
4004 4038  DATA >4038                   ** Address of Power up link
4006 0000  DATA >0000                   ** Not used in DSR calls
4008 403E  DATA >403E                   ** DSRLNK address
400A 0000  DATA >0000                   ** Not used in DSR calls
400C 0000  DATA >0000                   ** INTLNK zero, no interrupt rtn.
400E 0000  DATA >0000                   ** Not used in DSR calls
4010 0460  DATA >0460                   ** ???? Test routine perhaps
4012 430C  DATA >434C                   ** Points to infinite loop Test rtn?
4014 2843  TEXT '(C) COPYRIGHT 1985'
4016 2920
4018 434F
401A 5059
401C 5249
401E 4748
4020 5420
4022 3139
4024 3835
4026 2042  TEXT ' BY CORCOMP, INC.'
4028 5920
402A 434F
402C 5243
402E 4F4D
4030 502C
4032 2049
4034 4E43
4036 2E00  EVEN
4038 0000  DATA >0000                   ** Linkage set to 0: only 1 power up
403A 4062  DATA >4062                   ** Entry point of power up routine
403C 0000  DATA >0000                   ** Name length set to 0
403E 0000  DATA >0000                   ** Linkage to next device field-none
4040 40A4  DATA >40A4                   ** Entry point of device
           BYTE >05                     ** Name length of device
4042 0543  TEXT 'CLOCK'                 ** Device name
4044 4C4F
4046 434B
           BYTE >01                     ** ????????????
4048 0113  BYTE >13                     ** Number of characters to read
           BYTE >08                     ** Mask byte for write
404A 0804  BYTE >04                     ** Maximum allowable opcode
           BYTE >30                     ** ASCII offset for numbers
404C 30C0  BYTE >C0                     ** Enable byte
           BYTE >10                     ** Mask byte for write
404E 1040  BYTE >40                     ** ????????????
           BYTE >60                     ** ????????????
4050 60E0  BYTE >E0                     ** Mask for status and enable byte
4052 2C2F  TEXT ':/,'
```

```
4054 3A00  EVEN
4056 000A  DATA 10                         ** Constants for write routine
4058 0004  DATA 4                          **
405A 40F2  DATA >40F2                      ** Open routine address
405C 4080  DATA >4082                      ** Close routine address
405E 423A  DATA >423A                      ** Read routine address
4060 4134  DATA >4134                      ** Write routine address

** Power up routine **

4062 C18C  MOV  R12,R6                     ** Copy CRU address into R6
4064 045B  RT                              ** Return


** Error Codes **

4066 0201  LI   R1,>4000                   ** Bad Open Attribute
4068 4000
406A 1008  JMP  $+>12          >407C
406C 0201  LI   R1,>6000                   ** Illegal Operation
406E 6000
4070 1005  JMP  $+>0C          >407C
4072 0201  LI   R1,>8000                   ** Out of Table or Buffer space
4074 8000
4076 1002  JMP  $+>06          >407C
 4078 0201  LI   R1,>C000                   ** Attempt to read past end of file
407A C000
407C F901  SOCB R1,@>FF6B(R4)              ** Set Status Bit in DSR area
407E FF6B


** Close Opcode Routine **

4080 06A0  BL   @>4112                     ** Set up address to PAB Status Byt
4082 4112
4084 4001  DATA >4001
4086 DBE4  MOVB @>FF6B(R4),@>FFFE(R15) ** Write DSR Status Byte to VDP PA
4088 FF6B
408A FFFE
408C 06A0  BL   @>4112                     ** Set up address to PAB Char count
408E 4112
4090 4005  DATA >4005
4092 DBE4  MOVB @>FF6F(R4),@>FFFE(R15) ** Move DSR char cnt to PAB char c
4094 FF6F
4096 FFFE
4098 05E4  INCT @>FF86(R4)                 ** INCT the return address
409A FF86
409C 04C8  CLR  R8
409E C2E4  MOV  @>FF86(R4),R11             ** Move return address into R11
40A0 FF86
40A2 045B  RT                              ** and go back!

** DSR Routine **

40A4 02A4  STWP R4                         ** Store GLPWSP pointer in R4
40A6 C90B  MOV  R11,@>FF86(R4)             ** Save return address in DSR area
40A8 FF86
40AA C184  MOV  R4,R6                      ** Move GPLWSP pointer into R6
40AC 0226  AI   R6,>FF78                   ** Make it point to >8358 of DSR
40AE FF78                                  ** area
```

```
40B0 0205  LI    R5,>0007                ** Clear out 7 words of the DSR area
40B2 0007
40B4 04F6  CLR   *R6+
40B6 0605  DEC   R5                      ** Done yet?
40B8 16FD  JNE   $->04         >40B4     ** Nope?, do it again
40BA 06A0  BL    @>4112                  ** Set VDPWA to beginning of PAB
40BC 4112
40BE 0000  DATA  >0000
40C0 0205  LI    R5,>000A                ** We're going to get 10 bytes
40C2 000A
40C4 C184  MOV   R4,R6                   ** Put GPLWSP in R6
40C6 0226  AI    R6,>FF6A                ** Point to DSR area in Scratch Pad
40C8 FF6A
40CA DDAF  MOVB @>FBFE(R15),*R6+         ** Move byte from PAB to Scratch Pa
40CC FBFE
40CE 0605  DEC   R5                      ** Done Yet?
40D0 16FC  JNE   $->06         >40CA     ** nope?, Do it again
40D2 5920  SZCB @>4051,@>FF6B(R4)        ** Clear out bottom 5 bits of stat
40D4 4051                                ** byte. Set to defaults
40D6 FF6B
40D8 9824  CB    @>FF6A(R4),@>404B       ** Is this valid opcode?
40DA FF6A
40DC 404B
40DE 1202  JLE   $+>06         >40E4     ** Yes, then keep going
40E0 0460  B     @>406C                  ** Nope!, Return an error
40E2 406C
40E4 D164  MOVB @>FF6A(R4),R5            ** Put opcode in MSbyte of R5
40E6 FF6A
40E8 0985  SRL   R5,8                    ** Put it in low byte
40EA 0A15  SLA   R5,1                    ** Multiply it by 2
40EC C165  MOV   @>405A(R5),R5           ** Get address from opcode table
40EE 405A
40F0 0455  B     *R5                     ** Jump to the correct opcode rtn.

** Open Opcode Routine **

40F2 D0A4  MOVB @>FF6E(R4),R2            ** Move @ DSR Logical length into
40F4 FF6E
40F6 1609  JNE   $+>14         >410A     ** If its not 0 then don't alter it
40F8 06A0  BL    @>4112                  ** Set up VDP pointer to logical
40FA 4112                                ** length
40FC 4004  DATA  >4004
40FE 0202  LI    R2,>1300                ** 19 Chars for length
4100 1300
4102 D902  MOVB R2,@>FF6E(R4)            ** Put it in DSR logical length byt
4104 FF6E
4106 DBC2  MOVB R2,@>FFFE(R15)           ** Put it in PAB logical length byt
4108 FFFE
410A D064  MOVB @>FF6B(R4),R1            ** Move status byte to R1
410C FF6B
410E 0460  B     @>4080                  ** Return
4110 4080


** This routine sets up the VDPWA for a read or write from the PAB.  A    **
** data statement is passed and used as follows: >40 in MSByte is a       **
** write, >00 is a read, LSByte is offset into PAB.                       **

4112 C064  MOV   @>FF76(R4),R1           ** Put device pointer in R1
4114 FF76
```

```
4116 6064  S    @>FF74(R4),R1           ** Subtract the device name length
4118 FF74
411A 0221  AI   R1,>FFF6                ** Subtract 10 (Point to start
411C FFF6                               ** of PAB)
411E A07B  A    *R11+,R1                ** Add data statement to R1
4120 D7E4  MOVB @>0003(R4),*R15         ** Move LSByte of R1 into VDPWA
4122 0003
4124 1000  NOP                          ** Wait
4126 D7C1  MOVB R1,*R15                 ** Move MSByte of R1 into VDPWA
4128 045B  RT                           ** Go back


** This routine is a delay routine. The data statement passed is the      **
** number of times to execute the delay loop.                             **

412A C07B  MOV  *R11+,R1                ** Get number of times to loop
412C 1000  NOP                          ** Wait
412E 0601  DEC  R1                      ** Done yet?
4130 16FD  JNE  $->04          >412C    ** No?, loop again.
4132 045B  RT                           ** Go back

** Write Opcode Routine **

4134 C064  MOV  @>FF6C(R4),R1           ** Data Buffer Address Pointer
4136 FF6C                               ** into R1
4138 06A0  BL   @>4120                  ** Enter in middle of sub to set
413A 4120                               ** VDPWA to the data buffer in VDP
413C D1E4  MOVB @>FF6F(R4),R7           ** Move Chr count to MSbyte R7
413E FF6F
4140 0987  SRL  R7,8                    ** Put it in LSbyte R7
4142 0287  CI   R7,>0013                ** Compare to See if 19 bytes
4144 0013
                                        ** Where's the Jump if Greater?


** Get data from PAB for write

4146 D1AF  MOVB @>FBFE(R15),R6          ** Get day of week
4148 FBFE
414A 06C6  SWPB R6                      ** Switch em
414C D1AF  MOVB @>FBFE(R15),R6          ** Read another
414E FBFE
4150 91A0  CB   @>4052,R6               ** Check to see if it's a ','
4152 4052
4154 168E  JNE  $->E2          >4072    ** No? Then return an error
4156 06C6  SWPB R6                      ** Put day back in High Byte
4158 D906  MOVB R6,@>FF7E(R4)           ** Put it in DSR area >835E
415A FF7E
415C D92F  MOVB @>FBFE(R15),@>FF82(R4)  ** Get first digit of month
415E FBFE
4160 FF82
4162 D92F  MOVB @>FBFE(R15),@>FF81(R4)  ** Get second digit
4164 FBFE
4166 FF81
4168 D1AF  MOVB @>FBFE(R15),R6          ** Get rid of slash
416A FBFE
416C 1000  NOP                          ** wait
416E D92F  MOVB @>FBFE(R15),@>FF80(R4)  ** Get first digit of day
4170 FBFE
4172 FF80
```

```
4174 D92F  MOVB @>FBFE(R15),@>FF7F(R4) ** Get second digit of day
4176 FBFE
4178 FF7F
417A D1AF  MOVB @>FBFE(R15).R6         ** Get rid of slash
417C FBFE
417E 1000  NOP                         ** wait
4180 D1AF  MOVB @>FBFE(R15).R6         ** Get first digit of year
4182 FBFE
4184 D906  MOVB R6,@>FF84(R4)          ** Move it into DSR area
4186 FF84
4188 0986  SRL  R6,8                   ** Move it into 16 bits
418A 39A0  MPY  @>4056,R6              ** Multiply by 10
418C 4056
418E D16F  MOVB @>FBFE(R15),R5         ** Get second digit of year
4190 FBFE
4192 D905  MOVB R5,@>FF83(R4)          ** Put it in DSR area
4194 FF83
4196 0985  SRL  R5,8                   ** Put it in 16 bits
4198 A1C5  A    R5,R7                  ** Add it to First digit * 10
419A 04C6  CLR  R6                     ** Set up for divide
419C 3DA0  DIV  @>4058,R6              ** Divide by 4
419E 4058
41A0 C1C7  MOV  R7,R7                  ** Check to see if a leap year
41A2 1603  JNE  $+>08          >41AA   ** If its not then jump
41A4 F920  SOCB @>404B,@>FF80(R4)      ** Set bit 5 of first digit of day
41A6 404B                              ** byte
41A8 FF80
41AA D1AF  MOVB @>FBFE(R15),R6         ** Get the comma
41AC FBFE
41AE 91A0  CB   @>4052,R6              ** Is it a comma?
41B0 4052
41B2 1302  JEQ  $+>06          >41B8   ** Yes?, Keep going
41B4 0460  B    @>4072                 ** Report the error
41B6 4072
41B8 D92F  MOVB @>FBFE(R15),@>FF7D(R4) ** Get first digit of hour
41BA FBFE
41BC FF7D
41BE F920  SOCB @>404A.@>FF7D(R4)      ** Set most significant bit of the
41C0 404A                              ** low nybble
41C2 FF7D
41C4 D92F  MOVB @>FBFE(R15),@>FF7C(R4) ** Get second digit of hour
41C6 FBFE
41C8 FF7C
41CA D1AF  MOVB @>FBFE(R15),R6         ** Get rid of colon
41CC FBFE
41CE 1000  NOP                         ** Wait
41D0 D92F  MOVB @>FBFE(R15),@>FF7B(R4) ** Get first digit of minutes
41D2 FBFE
41D4 FF7B
41D6 D92F  MOVB @>FBFE(R15),@>FF7A(R4) ** Get second digit of minutes
41D8 FBFE
41DA FF7A
41DC D1AF  MOVB @>FBFE(R15),R6         ** Get rid of colon
41DE FBFE
41E0 1000  NOP                         ** Wait
41E2 D92F  MOVB @>FBFE(R15).@>FF79(R4) ** Get first digit of seconds
41E4 FBFE
41E6 FF79
41E8 D92F  MOVB @>FBFE(R15),@>FF78(R4) ** Get second digit of seconds
41EA FBFE
```

```
41EC FF78
41EE 0206  LI    R6,>000D                   ** 13 Bytes to write
41F0 000D
41F2 C144  MOV   R4,R5                      ** Duplicate GPLWS in R5
41F4 0225  AI    R5,>FF84                   ** Point to >8364 in DSR area (bytes
41F6 FF84                                   ** we want to put in clock)
41F8 0208  LI    R8,>5040                   ** Point to enable byte
41FA 5040
41FC D620  MOVB  @>404D,*R8                 ** Enable the device
41FE 404D
4200 06A0  BL    @>412A                     ** Delay >10 times
4202 412A
4204 0010  DATA  >0010
4206 D0D5  MOVB  *R5,R3                     ** Move the byte into R3
4208 0243  ANDI  R3,>0F00                   ** Mask out bits except for Low
420A 0F00                                   ** nybble of MSByte
420C C1C6  MOV   R6,R7                      ** Copy R6 into R7
420E 0607  DEC   R7                         ** Subtract one from R7
4210 0A87  SLA   R7,8                       ** Put it in MSByte
4212 F1E0  SOCB  @>404D,R7                  ** Set Bits 0 and 1 of MSByte
4214 404D
4216 D607  MOVB  R7,*R8                     ** Move to enable byte
4218 D803  MOVB  R3,@>5000                  ** Move to data byte
421A 5000
421C 0247  ANDI  R7,>4F00                   ** Mask out bits 0,2 and 3 of MSByte
421E 4F00                                   ** and all of LSByte
4220 D607  MOVB  R7,*R8                     ** Move it to enable byte
4222 F1E0  SOCB  @>404E,R7                  ** Set bit 3 of MSByte
4224 404E
4226 D607  MOVB  R7,*R8                     ** Move it to enable byte
4228 51E0  SZCB  @>404E,R7                  ** Clear all but bit 3 of MSByte
422A 404E
422C D607  MOVB  R7,*R8                     ** Move it to enable byte
422E 0605  DEC   R5                         ** Subtract one from pointer to data
4230 0606  DEC   R6                         ** Done yet?
4232 16E9  JNE   $->2C            >4206     ** Nope?, Do it again
4234 D606  MOVB  R6,*R8                     ** Disable write byte
4236 0460  B     @>4080                     ** Go back!
4238 4080

** Read Opcode Routine **

423A 0206  LI    R6,>000D                   ** We're going to read 13 bytes
423C 000D
423E C144  MOV   R4,R5                      ** Put GPLWSP in R5
4240 0225  AI    R5,>FF84                   ** Point to >8364 in DSR area
4242 FF84
4244 D920  MOVB  @>4049,@>FF6F(R4)          ** Put 19 into chr count of DSR ar
4246 4049
4248 FF6F
424A 0208  LI    R8,>5040                   ** Enable Byte address in R8
424C 5040
424E D620  MOVB  @>404D,*R8                 ** Put >C0 into Enable byte
4250 404D
4252 06A0  BL    @>412A                     ** Delay >10 times
4254 412A
4256 0010  DATA  >0010
4258 D620  MOVB  @>4051,*R8                 ** Let it know we're going to read?
425A 4051
425C C1C6  MOV   R6,R7                      ** Copy R6 into R7
```

```
425E 0607  DEC  R7                          ** Subtract 1
4260 0A87  SLA  R7,8                        ** Put it in MSByte
4262 F1E0  SOCB @>4051,R7                   ** Set 3 MSBits
4264 4051
4266 D607  MOVB R7,*R8                      ** Tell it we want another byte
4268 1000  NOP                              ** Wait
426A 1000  NOP
426C D0E0  MOVB @>5000,R3                   ** Get numeric value from data byte
426E 5000
4270 F0E0  SOCB @>404C,R3                   ** Add ascii offset for numbers
4272 404C
4274 D543  MOVB R3,*R5                      ** Move it into DSR area
4276 0605  DEC  R5                          ** Dec DSR area address pointer
4278 0606  DEC  R6                          ** Done yet?
427A 16F0  JNE  $->1E            >425C      ** Nope?, do it again
427C D606  MOVB R6,*R8                      ** Put 0 into enable byte
427E C064  MOV  @>FF6C(R4),R1               ** Address to Pab Data buffer in R1
4280 FF6C
4282 06A0  BL   @>411E                      ** Set up VDPWA
4284 411E
4286 4000  DATA >4000                       ** Gonna write
4288 D1A4  MOVB @>FF7E(R4),R6               ** Get Day of week from DSR area
428A FF7E
428C DBC6  MOVB R6,@>FFFE(R15)              ** Put it in Pab Data buffer
428E FFFE
4290 DBE0  MOVB @>4052,@>FFFE(R15)          ** Put a ',' in there!
4292 4052
4294 FFFE
4296 DBE4  MOVB @>FF82(R4),@>FFFE(R15) ** Move number of month in there
4298 FF82
429A FFFE
429C DBE4  MOVB @>FF81(R4),@>FFFE(R15) ** Second digit of month
429E FF81
42A0 FFFE
42A2 DBE0  MOVB @>4053,@>FFFE(R15)          ** Put in a '/'
42A4 4053
42A6 FFFE
42A8 D1A4  MOVB @>FF80(R4),R6               ** Get the day number
42AA FF80
42AC 0246  ANDI R6,>3300                    ** Mask out anything greater than
42AE 3300                                   ** ASCII 3
42B0 DBC6  MOVB R6,@>FFFE(R15)              ** Put in PAB
42B2 FFFE
42B4 DBE4  MOVB @>FF7F(R4),@>FFFE(R15) ** Put in the second digit
42B6 FF7F
42B8 FFFE
42BA DBE0  MOVB @>4053,@>FFFE(R15)          ** Put in another '/'
42BC 4053
42BE FFFE
42C0 DBE4  MOVB @>FF84(R4),@>FFFE(R15) ** Put in the year
42C2 FF84
42C4 FFFE
42C6 DBE4  MOVB @>FF83(R4),@>FFFE(R15) ** Second digit of year
42C8 FF83
42CA FFFE
42CC DBE0  MOVB @>4052,@>FFFE(R15)          ** Time for another ','
42CE 4052
42D0 FFFE
42D2 D1A4  MOVB @>FF7D(R4),R6               ** Get top digit of hour
42D4 FF7D
```

```
42D6 0246   ANDI R6,>3300                 ** Mask out anything greater than
42D8 3300                                 ** ASCII 3
42DA DBC6   MOVB R6,@>FFFE(R15)           ** Put it in PAB
42DC FFFE
42DE DBE4   MOVB @>FF7C(R4),@>FFFE(R15)   ** Put in second digit of hour
42E0 FF7C
42E2 FFFE
42E4 DBE0   MOVB @>4054,@>FFFE(R15)       ** Put in a ':'
42E6 4054
42E8 FFFE
42EA DBE4   MOVB @>FF7B(R4),@>FFFE(R15)   ** Put in Minutes
42EC FF7B
42EE FFFE
42F0 DBE4   MOVB @>FF7A(R4),@>FFFE(R15)   ** Put in second digit of minutes
42F2 FF7A
42F4 FFFE
42F6 DBE0   MOVB @>4054,@>FFFE(R15)       ** another ':'
42F8 4054
42FA FFFE
42FC DBE4   MOVB @>FF79(R4),@>FFFE(R15)   ** Put in seconds
42FE FF79
4300 FFFE
4302 DBE4   MOVB @>FF78(R4),@>FFFE(R15)   ** Put in second digit of seconds
4304 FF78
4306 FFFE
4308 0460   B    @>4080                   ** Go back
430A 4080
430C 02E0   LWPI >8300                    ** Load GPLWSP   Test routine
430E 8300                                 ** perhaps?
4310 0460   B    @>430C                   ** Keep looping???
4312 430C
```

2.4 Subroutine links

Subroutine links (also known as low level routines or BASIC CALL subprograms) are the least documented program types available on DSRs. However, they offer the possibility of adding powerful new features to BASIC and XBASIC on the /4A system. Subroutine links may be of two types: low level routines or BASIC subprograms. Low level routines may be accessed via DSRLNK with a DATA >A (instead of >8) statement, as shown in the example code. Low level routines are not accessed by the FMS in the console, as are the main device routines. BASIC subprograms (ie- CALL FILES(x)), are accessed from the command mode of BASIC or XBASIC, and from running BASIC programs. The search DSR routine for subprograms in XBASIC searches only GROMs and not peripheral ROMs when the XBASIC program is running. Both of these routine types can be included in peripheral DSRs to enhance or add features not available in other applications. An example of this is the Disk Drive Controller which has seven low level routines for disk/file access and adds "CALL FILES(x)" to the BASIC environment. Each of the two routine types is discussed further below.

2.4.1 Low level routines

Low level routines are accessed only by DSRLNK. The PAB values to be set are name length (1 byte) and the routine number (>10->FF). The low level routine may use registers R0 - R10. Registers R11 - R15 have the previously defined meanings. Upon entry into a low level routine, R11 should be saved. Data or parameters may be passed through the FAC RAM area or at a known offset in the VDP from the PAB. The low level routine must include an error reporting subroutine that places the 3 bit error codes used by main device routines in byte >8350. Table I.2 lists the memory locations and contents.

TABLE I.2: LOW LEVEL ROUTINE PARAMETER ADDRESSES

Address	Contents
>834A (FAC)	data I/O, parameter passing
834B	"
834C	"
834D	"
834E	"
834F	"
8350	MSB contains error codes upon return
8352	data I/O

It is the responsibility of the application program to prepare data for transfer in the available addresses, prepare for the DSRLNK, and read the error codes and any other data in the >834A->8352 addresses returned by the DSR. Example I.4a lists partial code for a low level routine. Example I.4b lists partial code for accessing a low level routine. Low level routines can be used in conjunction with an application program to add utility or special purpose functions that: 1) require speed (in assembly instead of BASIC), and 2) are versatile enough for use in more than one program, thereby justifying inclusion in ROM.

EXAMPLE I.4a: LOW LEVEL ROUTINE

```
          AORG >4000
          BYTE >AA
          BYTE 1
          DATA >0000
          DATA ----
          DATA >0000
          DATA ----
          DATA LLR10          link to 1st low level routine
          DATA ---
          DATA >0000
          -
          -
LLR10     DATA LLR20           link to 2nd low level routine
          DATA LLR10E          entry point of 1st low level routine
          BYTE 1               name length
          BYTE >10             low level routine number
          EVEN
LLR20     DATA >0000           no more low levels
          DATA LLR20E          entry point for 2nd low level routine
          BYTE 1
          BYTE >20             low level routine number
          EVEN
          -
          -
LLR10E    $                    entry point of low level >10
          -
LLR20E    $                    entry point of low level >20
          -
ERROR    [set 3 MSbits of >8350]
ERRTN     B *R11               return
RETURN    INCT R11
          B *R11
```

EXAMPLE I.4b: LOW LEVEL ROUTINE ACCESS

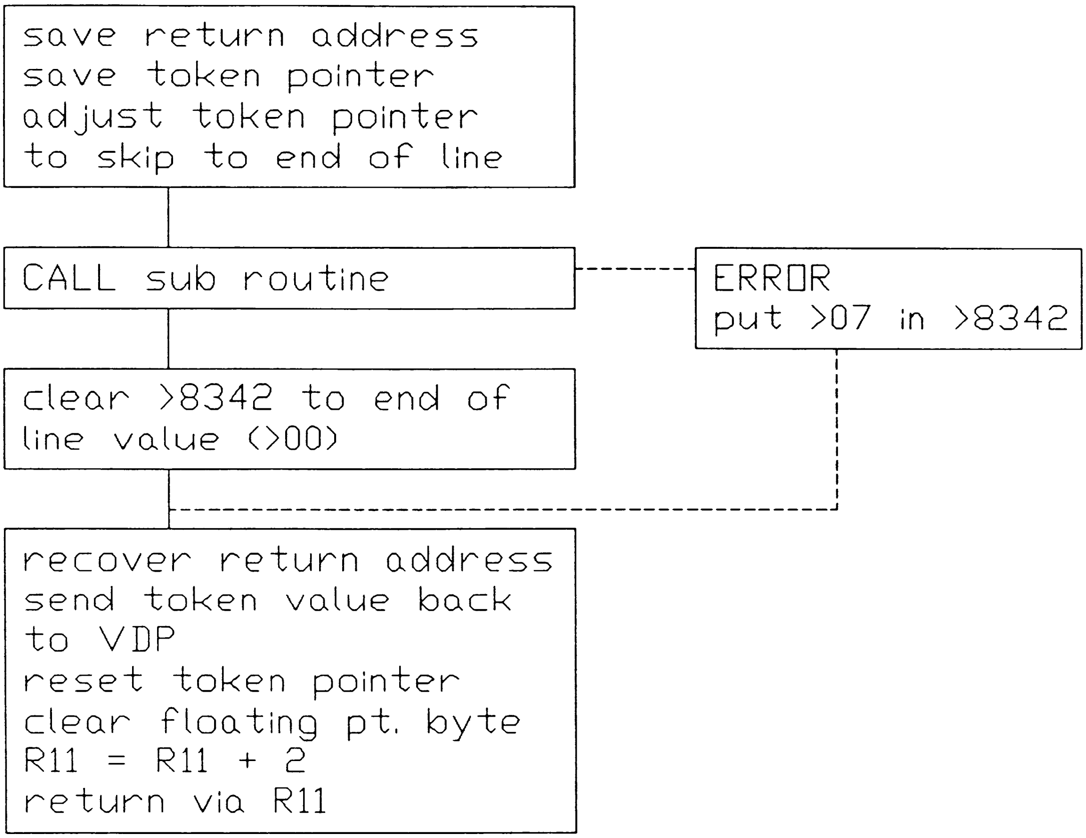
```
         DEF ACCESS
         REF DSRLNK,VMBW
PAB      DATA >0110         name length 1, routine >10
DATBUF   EQU >1000          pointer to data buffer in VDP RAM
PABPTR   EQU >0F80          pointer to PAB in VDP RAM
         -
ACCESS   LI R0,>XXXX        load data for transfer
         MOV R0,@>834A      put in 1st address
         -                  (load other data in FAC if needed)
         LI R0,PABPTR       put PAB in >0F80 of VDP
         LI R1,PAB
         LI R2,2            2 bytes to write to VDP
         BLWP @VMBW         write PAB to VDP
         MOV R0,@>8356      put pointer to PAB in >8356
         BLWP @>DSRLNK      access to low level
         DATA >A
         -
RETURN   [check >8350 for error bits]
         [recover data in FAC]
         -
         B *R11             return
         END
```

2.4.2 BASIC CALL subprograms

BASIC CALL subprograms are accessed from BASIC or XBASIC, and not through the FMS. Because of this, special care must be taken in interaction between the 'assembly' environment of the DSR and the 'GPL' environment of BASIC. The header for CALL subprograms is similar to that of low level routines. Errors may be returned by setting the BASIC token value at >8342 to >07. This will return an "INCORRECT STATEMENT" error in the BASIC program. DATA may be transferred from the BASIC environment by this program type (for example, CALL FILES(1)). The subprogram must manipulate the BASIC token pointer (>832C) and the current token (>8342) to retrieve data and reset the current BASIC line token before returning to GPL. Other error bytes must be checked and cleared prior to returning also. The routine is responsible for keeping track of the current token pointer position, and adjusting it as needed.

CALL subprogram routines may use the registers as defined for the low level routines. Errors can be returned as INCORRECT STATEMENT. The routine should not utilize GROM access, as this may disrupt GROM address settings assumed by the BASIC calling program. If GROM access is required, then the GROM address should be saved and then restored. As with all DSR accesses, it is the responsibility of the programmer to ensure that the routine does not disrrupt the BASIC/XBASIC environment while executing, or upon its return. Figure I.3 is a flowchart of a CALL subprogram. Example I.5 demonstrates how a routine may be coded. The disassembled and commented CALL FILES of the Disk Drive Controller (from "Technical Drive") is also included. More documentation on these routines is available in the source code and technical manual for the Horizon RAMdisk. The source code file for the original version of the RAMdisk ROS section of CALL subs is also included, and demonstrates how to recover input parameters from the BASIC environment, and handle errors.

FIG. 1.3: CALL SUBROUTINE



EXAMPLE I.5: CALL SUBPROGRAM

```
          AORG >4000
          BYTE >AA
          BYTE 1
          DATA >0000
          DATA ----
          DATA >0000
          DATA ----
          DATA CALL1         link to 1st CALL sub
          DATA ----
          DATA >0000
          -
          -
CALL1     DATA CALL2         link to 2nd CALL sub
          DATA CALL1E        entry point to 1st CALL sub
          BYTE 3             name length
          TEXT 'FPT'         name of routine
          EVEN
CALL2     DATA >0000         no more CALL subs
          DATA CALL2E        entry point of 2nd CALL sub
          BYTE 5             name length
          TEXT 'MACRO'       name of routine
          EVEN
          -
          -
CALL1E    $                  entry point of 1st CALL sub
          MOV R11,R7         save return address
          MOV @>832C,R8      put token pointer in R8
          AI R8,3            add name length to get to end of line
          -
          [routine for FPT]
          -
          B @ERROR           only if error
          B @OKRTN           return with no error
CALL2E    $                  entry point of 2nd CALL sub
          MOV R11,R7         save return address
          MOV @>832C,R8      put token pointer in R8
          AI R8,5            add name length to get to end of line
          -
          [routine for MACRO]
          -
          B @ERROR           only if error
          B @OKRTN           return with no error
          -
          -
ERROR     LI R0,>0700        incorrect statement token
          MOVB R0,@>8342     put it in current token address
          JMP EXIT
OKRTN     SZCB @>8342,@>8342 makes end of line (>00) current token
EXIT      MOV R7,R11         recover return address
```

```
MOV R8,R1          set token pointer
SWPB R1            LSB first
MOVB R1,@>8C02     move to VDP write
SWPB R1
MOVB R1,@>8C02     MSB next
MOV R1,@>832C      reset token pointer
INCT R11           no other errors
SZCB >4000,@>8354  clear floating point error
B *R11             return
END
```

BASIC CALL FILES ROUTINE FOR THE TI DISK DRIVE CONTROLLER

```
AORG >5D5A

MOV  R11,R7                  *save return address                  5D5A
BL   @>4724                  *Init routine                         5D5C
MOV  @>002C(R9),R8           *token code pointer in R8             5D60
AI   R8,>0007                *add 7 to skip 'FILES('               5D64
BL   @>4B76                  *get 2 bytes afer it                  5D68
CI   R0,>C801                *check for unquoted string,           5D6C
                             *length of 1 (tokenized code)
JNE  AA                      *no?, then jump                       5D70
INCT R8                      *point to ASCII #                     5D72
BL   @>4B76                  *get it from VDP                      5D74
SWPB R0                      *put ASCII # in low byte,             5D78
                             *>B6 in high byte
AI   R0,>49D0                *mask out ASCII offset                5D7A
CI   R0,>0009                *check if non-numeric                 5D7E
JH   >5DAA                   *yes? then jump                       5D82
SWPB R0                      *put # of FCB's to reserve            5D84
                             *in MSByte
MOVB R0,@>004C(R9)           *put it in >834C for >16 return       5D86
BL   @>4658                  *do routine >16 (reserve buffers)     5D8A
DATA >5DB4                                                         5D8E
MOVB @>0050(R9),@>0050(R9)   *error?                               5D90
JNE  >5DAA                   *yes? then jump                       5D96
MOV  @>002C(R9),R8           *put token pointer in R8              5D98
AI   R8,>000C                *point to end of statement            5D9C
MOV  R8,@>002C(R9)           *put it in token pointer              5DA0
SZCB @>0042(R9),@>0042(R9)   *put >00 at >8342 (current token)     5DA4
                             *(end of statement indicator)
B    @>4676                  *return                               5DAA
END
```

```
*********************************************
*                                           *
*       RAMDISK OPERATING SYSTEM            *
*                                           *
* COPYRIGHT 1985, HORIZON COMPUTER, LIMITED *
*                                           *
*       -- ALL RIGHTS RESERVED --           *
*                                           *
*********************************************
*                                           *
* This code consists of all of the CALL sub-*
* programs accessable from TI BASIC and     *
* Ext. BASIC command mode except for CALL DM*
* The code resides in the second 2K block.  *
*                                           *
*********************************************
*
       AORG >5800
WTPR   EQU  >415E
MAXSEC EQU  >4010
SV1    EQU  >419A
TEN    EQU  >41D0
ONE    EQU  >40D0
SVADR  EQU  >41AA
FAC    EQU  >834A
DRIVE  EQU  >400E
FCB    EQU  >41D8
SAVADR EQU  >41A8
VRWA   EQU  >4290
VDPRD  EQU  >8800
FCBDIF EQU  >41D4
NUMDRV EQU  >4014
*
*
*-------------------------------------
*
*           TI BASIC CALLS
*
*-------------------------------------
*
* CHANGE DRIVE NUMBER
*
*
DRCNG  MOV  R11,@SVADR
       MOV  @>832C,R1      GET TOKEN POINTER
       AI   R1,4           POINT TO TOKEN FOR TRANSFER
       BL   @CLRD          CALL ROUTINE DIGIT READ
       MOV  @FAC+2,@FAC+2  ERROR?
       JEQ  DRN01
       B    @DRERR
DRN01  A    @FAC+4,R1
       MOV  @FAC,R2
```

```
       JGT  DRN02           >0?
       B    @DRERR
DRN02  CI   R2,7            <7?
       JLT  DRN03
       B    @DRERR
DRN03  MOVB @>8800,R0       GET PARENTHESIS AND EOL BYTE
       INC  R1              R1 POINTS TO EOL BYTE
       SWPB R0
       MOVB @>8800,R0
       CI   R0,182          RIGHT PAREN? )
       JEQ  DRN04
       B    @DRERR
DRN04  MOVB @DRIVE+1,R0
       BL   @DROPEN
       MOV  @FCB,@FCB
       JEQ  DRN05
       B    @DRERR
DRN05  MOV  R2,R0
       SWPB R0
       BL   @DROPEN
       MOV  @FCB,@FCB
       JEQ  DRN06
       B    @DRERR
DRN06  MOV  R2,@DRIVE
       JMP  CALLRT
DRERR  LI   R0,>0700
       MOVB R0,@>8342
       JMP  CALLER
CALLRT SZCB @>8342,@>8342
CALLER MOV  @SVADR,R11
       SWPB R1
       MOVB R1,*R15
       SWPB R1
       MOVB R1,*R15         RE-LOAD WRITE REGISTER
       MOV  R1,@>832C       RESET TOKEN POINTER
       INCT R11
       SZCB @>4014,@>8354
       B    *R11            RETURN
DROPEN MOV  R11,R9
       MOV  @>8370,R8
       AI   R8,6
       MOV  R8,@FCB
       DEC  R8
       BL   @VRWA
       MOVB @VDPRD,R4          MAX # OF OPEN FILES
       SRL  R4,8
DROP1  MOV  @FCB,R8
       AI   R8,5
       BL   @VRWA
       MOVB @VDPRD,R5          DRIVE #
       CB   R5,R0
       JNE  DROP3
```

```
       MOVB @VDPRD,R3          1ST CHAR FILE NAME
       JEQ  DROP3
       B    *R9
DROP3  A    @FCBDIF,@FCB
       DEC  R4
       JNE  DROP1
       CLR  @FCB
       B    *R9
*
* SET MAXIMUM SECTOR SIZE
*
MAXSC  MOV  R11,@SVADR
       MOV  @>832C,R1
       AI   R1,4
       BL   @CLRD           GET NUMBER
       MOV  @FAC+2,@FAC+2   ERROR?
       JEQ  MAX01
       B    @DRERR
MAX01  A    @FAC+4,R1
       MOV  @FAC,R2
       JGT  MAX02
       B    @DRERR
MAX02  CI   R2,1441
       JLT  MAX03
       B    @DRERR
MAX03  MOVB @>8800,R0       GET PARENTHESIS AND EOL BYTE
       INC  R1              R1 POINTS TO EOL BYTE
       SWPB R0
       MOVB @>8800,R0
       CI   R0,182          RIGHT PAREN? )
       JEQ  MAX04
       B    @DRERR
MAX04  MOV  R2,@MAXSEC
       B    @CALLRT
*
* CALL EXECUTE ROUTINE
*
EXCUT  MOV  R11,@SVADR
       MOV  @>832C,R1
       AI   R1,4
       BL   @CLRD
       MOV  @FAC+2,@FAC+2
       JEQ  EX01
       B    @DRERR
EX01   A    @FAC+4,R1
       MOV  R1,@>8300
       MOV  @FAC,R2
       BL   *R2
       LWPI >83E0
       MOV  @>8300,R1
       SWPB R1
       MOVB R1,*R15
```

```
       SWPB R1
       MOVB R1,*R15
       INC  R1
       MOVB @>8800,R0
       SWPB R0
       MOVB @>8800,R0
       CI   R0,182
       JEQ  EX02
       B    @DRERR
EX02   B    @CALLRT
*
* READ A NUMBER FROM THE CALLING PROGRAM
*
*  R1 - POINTS TO THE STRING DESCRIPTOR TOKEN
*  R0-R3 ARE USED
*
*  FAC   RETURNS THE INTEGER
*  FAC+2 0 FOR NO ERROR, 1 FOR ERROR
*  FAC+4 NUMBER OF TOKEN POINTER MOVES
*  FAC+6 TEMP STORAGE OF R1
*
*  VDPWA/R1  LEFT POINTING TO NEXT CHAR/TOKEN
*
CLRD   MOV  R11,@SV1
       CLR  @FAC
       CLR  @FAC+2
       CLR  @FAC+4
       MOV  R1,@FAC+6
       SWPB R1
       MOVB R1,*R15
       SWPB R1
       MOVB R1,*R15
       CLR  R1
       MOVB @>8800,R1  CHECK TOKEN FOR UNQUOTED STRING
       INC  @FAC+4
       CI   R1,>C800
       JEQ  NM1
       B    @NMERR     IF NOT, RETURN AN ERROR
NM1    MOVB @>8800,R1  CHECK NEXT BYTE FOR LENGTH OF STRING
       INC  @FAC+4
       SRL  R1,8
       MOV  R1,R1
       JGT  NM2        >0?
       B    @NMERR
NM2    CI   R1,7       <7?
       JLT  NM25
       B    @NMERR
NM25   MOVB @>8800,R2  CHECK FOR MINUS SIGN
       INC  @FAC+4
       SRL  R2,8
       CI   R2,>002D
       JEQ  NM27
```

```
       CI    R1,6         ARE THERE <6 DIGITS?
       JLT   NM35
       B     @NMERR
NM27   MOV   @TEN,@FAC+2
       JMP   NM35
NM3    MOV   @>8800,R2    READ A DIGIT
       INC   @FAC+4
       SRL   R2,8
NM35   AI    R2,-48
       BL    @NMCK        A VALID DIGIT?
       MOV   R0,R0        R0=0 IF 0-9
       JEQ   NM4
       B     @NMERR
NM4    MOV   R2,R2        IS DIGIT 0?
       JEQ   NM6          IF SO, SKIP MPY
       CLR   R3
       MOV   R1,R0        COUNTER FOR MPY
NM5    DEC   R0
       JEQ   NM6          DONE?
       MPY   @TEN,R2
       MOV   R3,R2
       JMP   NM5
NM6    MOV   R2,R3
       A     R3,@FAC
       DEC   R1           MORE DIGITS?
       JNE   NM3
       C     @FAC+2,@TEN  NEGATIVE NUMBER?
       JNE   NM7
       NEG   @FAC
NM7    CLR   @FAC+2
       B     @NMRTN
*
* RETURN TO INTERNAL CALLER
*
NMERR  MOV   @ONE,@FAC+2
NMRTN  MOV   @SV1,R11
       MOV   @FAC+6,R1
       B     *R11
*
* CHECK DIGIT
*
NMCK   LI    R0,1
       MOV   R2,R2
       JLT   NMR
       CI    R2,9
       JGT   NMR
       CLR   R0
NMR    B     *R11
*
* PROTECTION ON
*
WRPON  MOV   R11,@SVADR
```

```
       MOV  @ONE,@WTPR
       MOV  @>832C,R1
       AI   R1,2
       JMP  WPN01
*
* PROTECTION OFF
*
WRPOFF MOV  R11,@SVADR
       CLR  @WTPR
       MOV  @>832C,R1
       AI   R1,2
WPN01  B    @CALLRT
*
* CARD CRU ON/OFF
*
CON    INCT R11
       MOV  R11,@SVADR
       MOV  @>832C,R1
       AI   R1,2
       JMP  WPN01
COFF   MOV  R11,@SVADR
       MOV  @>832C,R1
       AI   R1,2
       JMP  WPN01
*
* SET NUMBER OF DRIVES 1-4
*
SETNDR MOV  R11,@SVADR
       MOV  @>832C,R1
       AI   R1,4
       BL   @CLRD
       MOV  @FAC+2,@FAC+2
       JEQ  STDRN1
       B    @DRERR
STDRN1 A    @FAC+4,R1
       MOV  @FAC,R2
       JGT  STDRN2       >0?
       B    @DRERR
STDRN2 CI   R2,5         <5?
       JLT  STDRN3
       B    @DRERR
STDRN3 MOVB @>8800,R0
       INC  R1
       SWPB R0
       MOVB @>8800,R0
       CI   R0,182      RIGHT PAREN? )
       JEQ  STDRN4
       B    @DRERR
STDRN4 MOV  R2,@NUMDRV
       B    @CALLRT
       END
```

SECTION J: DSR ACCESS

1.0 Introduction

The purpose of this section is to demonstrate how the console can access DSRs. Subsection 2.0 outlines how the console accesses interrupts, power up (initialization) routines, BASIC subroutine links (CALL xxx), and access of main device routines as files. Subsection 3.0 provides a simple DSRLNK routine in assembly for use with XBASIC (which does not load the DSRLNK routine as does the E/A cartridge). This is also provided as an example of DSR access programming for application programs.

2.0 Console DSR Access

2.1 Interrupt access routine

The interrupt access routine is discussed first because it is a stand-alone 9900 assembly routine located in the console ROM; other DSR access routines in the console are also in ROM or in GPL in GROMS 0-2. The interrupt access routine flowchart is shown in Fig. 2.1. The disassembled and commented source code is given in Fig. 2.2. Commonly used address EQUates are given in Table J.1.

Upon entering the routine due to the detection of a low signal on the -INTREQ line, all interrupts are disabled to prevent further activation until the current interrupt is serviced. The GPL workspace is loaded, and the CRU register (R12) is cleared. The cassette and VDP interrupts are checked first. Then R12 is initialized and all peripherals from >1000 to >1F00 are checked. If no peripheral caused the interrupt, the CPU returns to the main program. When searching the DSR, the routine checks for the header "AA" at byte >4000, then checks for the first interrupt routine. If there is an interrupt routine, the CPU will first save the next interrupt routine address in R0, and execute the current interrupt routine. Upon return from the first interrupt routine, the CPU will continue to execute subsequent interrupt routines until none are found for that peripheral. Therefore, any peripheral may have an interrupt routine (if needed), and each peripheral may have multiple interrupt routines. It is the responsiblity of the individual peripheral to determine if it has caused an interrupt, and reset it when the routine is complete. All DSR interrupt routines are accessed whether or not they caused the interrupt, even after the one that caused the interrupt is found.

2.2 GPLDSR access routine

Access by the console to other DSRs is via routines in GPL in the console GROMs or assembly in the console ROMs. Each of these access routines (power up (ROM), CALL subprogram and main device/files (GROM)) use a routine referred to as GPLDSR. The GPL interpreter accesses this assembly language subroutine via an XML>19 command. The

FIG. J.2.1: INTERRUPT ACCESS ROUTINE

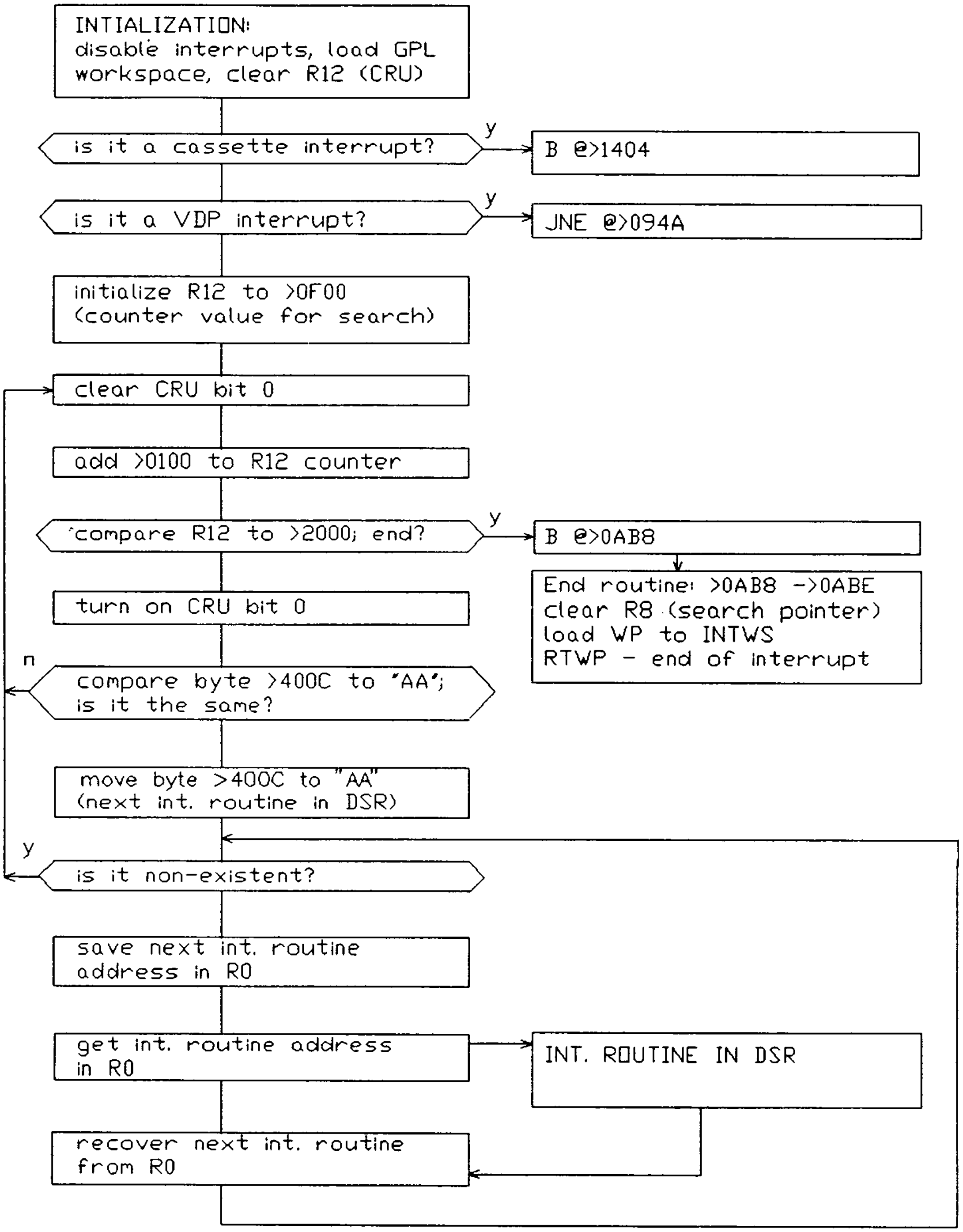


FIGURE J.2.2: INTERRUPT ACCESS ROUTINE CODE

```
       AORG >0900
                                                         0900
       LIMI >0000        *disable interrupts             0900
       LWPI >83E0        *load GPLWS                     0904
       CLR  R12          *clear CRU register             0908
*
       COC  @>0032,R14   *is it the cassette int.?       090A
       JNE  NOCAS        *no, jump                       090E
       B    @>1404       *jump to cassette routine       0910
*
NOCAS  TB   >02          *is it the VDP int.?            0914
       JNE  AB           *yes, jump to routine           0916
*
       LI   R12,>0F00    *load initial search value
       SBO  >00          *turn on card                   091C
*
LOOP   SBZ  >00          *turn off card                  091E
       AI   R12,>0100    *add >0100 to search reg.       0920
*
       CI   R12,>2000    *are we at the end?             0924
       JEQ  END          *yes, jump to end               0928
*
       SBO  >00          *turn card on                   092A
*
       CB   @>4000,@>000D *check for valid "AA" header   092C
       JNE  LOOP         *no, start over                 0932
*
       MOV  @>400C,R2    *save entry addr. in R2         0934
*
NXTDSR JEQ  LOOP         *no address, start over         0938
*
       MOV  R2,R0        *save next addr. in R0          093A
       MOV  @>0002(R2),R2 *get addr. for routine         093C
*
       BL   *R2          *branch and link to ISR         0940
*
       MOV  *R0,R2       *recover next ISR addr.         0942
       JMP  NXTDSR       *check and execute              0944
*
END    B    @>0AB8       *end, return                    0946
       END
*************************************************

       AORG >0AB8

       CLR  R8           *clear ROM search pointer       0AB8
       LWPI >83C0        *load WP to INTWS               0ABA
       RTWP              *return                         0ABE
```

GPLDSR routine starts at >0AC0 and runs to >0B20 in the console ROM. Although similar to the interrupt access routine, it is used only from the GPL environment. The purpose of GPLDSR is to search all DSRs, find and execute the requested routine type as needed. Since the remaining DSR access routines utilize GPLDSR, it is outlined first.

Figure 2.3 is the flowchart for GPLDSR. Initially, R1 is cleared and is used as a DSR version counter. R2 is used as storage for the address table value. >83D0 is a predefined ROM/GROM address which may be set by the calling routine if the peripheral location is already known. >83D2 is used to save the program address while >836D has an increment, or table jump value set by the calling program to instruct GPLDSR which address table value in the DSR to branch to.

Upon entry into GPLDSR, R1 is cleared and CRULST is checked. If a non-zero value is given in CRULST (ie, the peripheral CRU value is known, and does not have to be searched for), then the routine branches to that peripheral CRU value directly. If CRULST is zero, then the GPLDSR routine steps through each peripheral (>1000 - >1F00), and checks for "AA" on the first byte. It then gets the first table entry value, and checks to see if it is non-zero. If non-zero, the address is saved in >83D2. The program entry point is stored in R9. The name of the routine being searched for is checked by the routine NAMEMATCH (Section 2.2.1). If there is no match, the search routine continues. If there is a match, R1 is incremented, and the routine is executed by a BL *R9. When GPLDSR is complete, it branches to a subroutine @>0842 which changes the value of CRULST before returning to the calling routine. If no matches are found in any peripheral, GPLDSR branches to an error handler at >006A.

2.2.1 NAMEMATCH routine

The NAMEMATCH routine is at >0BE8 - >0C08 in the console ROM, and is used by either ROMs or GROMs to match device names. CPU RAM >8354 is SCLEN, and the device name length; >834A is NBA, the name buffer address (see Figures 2.5 and 2.6).

Upon entry into NAMEMATCH, the device name length is retrieved from SCLEN and placed into R5 where it is used as a counter. If it is zero, then the CPU returns to the calling routine. The value of R5 is compared to the value given by the address in R2; if not equal then a return is made directly via R11, indicating a problem. If the length is OK, then R5 is adjusted to the right byte, and R6 is loaded with the name buffer address (NBA). Next, NAMEMATCH checks to see if it is searching a GROM. If not, the counter value of R2 is incremented. The character (byte) pointed to by R6 is compared to that pointed to by R2. If not equal, a return with error is made. If the characters match, the loop continues until all characters are checked. If the device name length is correct and all characters match, then R11 is incremented by 2 (indicating no errors on return) and the CPU returns to the calling routine.

2.3 Power up (initialization) access routine

The power up access routine is in GPL in GROM 0 at >018B (Figure

FIG. J.2.3: GPLDSR ROUTINE

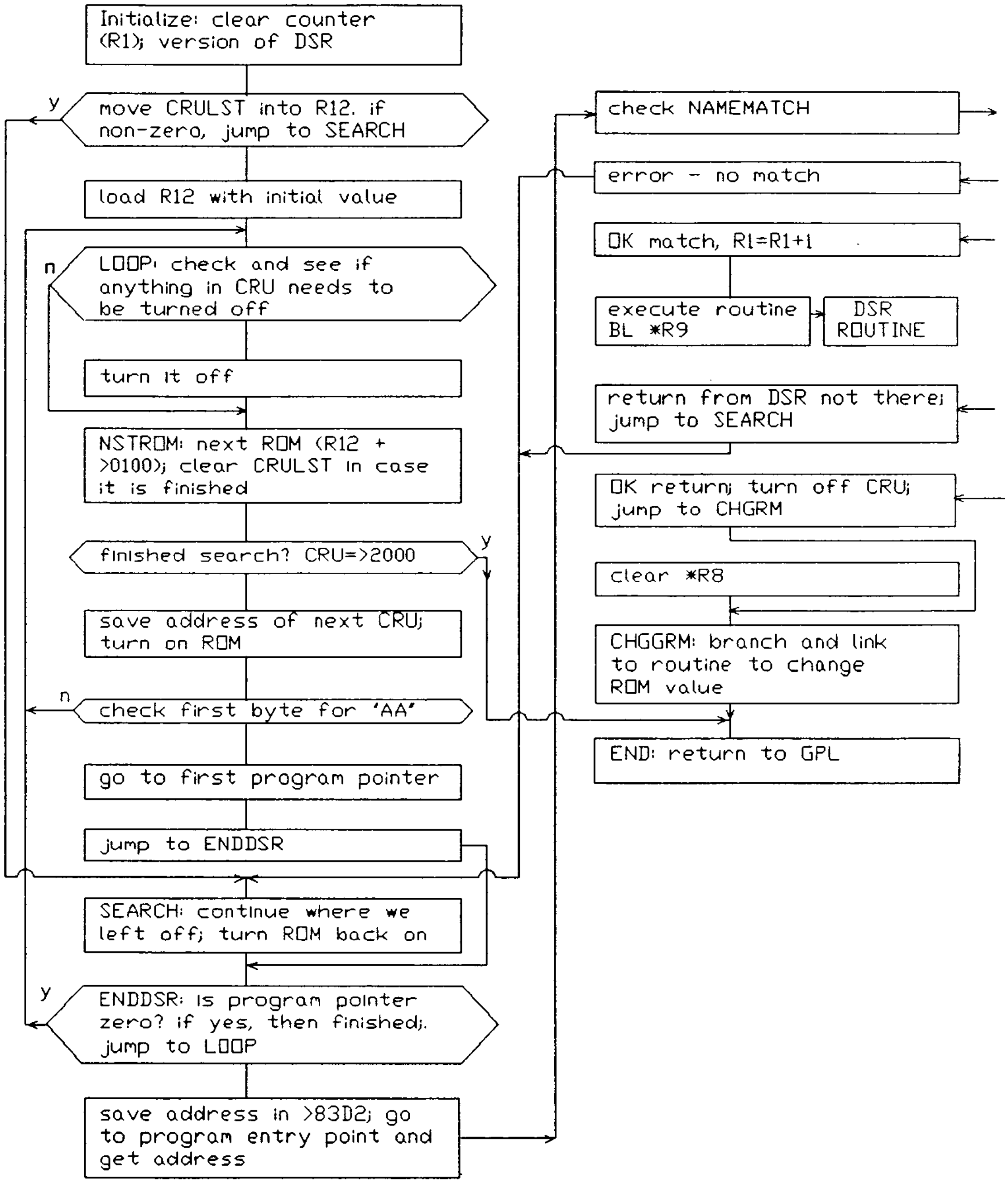


FIGURE J.2.4: GPLDSR ROUTINE CODE

```
       AORG >0AC0
                                                                  0AC0
       CLR  R1                  *clear counter R1                 0AC0
*
       MOV  @>83D0,R12          *get ROM search pointer (CRULST)  0AC2
       JNE  SEARCH              *if none, jump to SEARCH          0AC6
*
       LI   R12,>0F00           *load initial value               0AC8
*
LOOP   MOV  R12,R12             *check CRULST                     0ACC
       JEQ  NXTROM              *if none, jump to NXTROM          0ACE
*
       SBZ  >00                 *turn off card in case done       0AD0
*
NXTROM AI   R12,>0100           *add >0100 to CRULST              0AD2
       CLR  @>83D0              *clear search pointer             0AD6
*
       CI   R12,>2000           *at the end?                      0ADA
       JEQ  END                 *yes, jump to end                 0ADE
*
       MOV  R12,@>83D0          *move R12 into CRULST             0AE0
       SBO  >00                 *turn on card                     0AE4
       LI   R2,>4000            *set R2 to >4000 (ROM addr)       0AE6
*
       CB   *R2,@>000D          *valid "AA" header?               0AEA
       JNE  LOOP                *no start over                    0AEE
*
       AB   @>836D,@>83E5       *calculate program entry addr.    0AF0
       JMP  ENDDSR              *jump to ENDDSR                   0AF6
*
SEARCH MOV  @>83D2,R2           *move save addr. into R2          0AF8
       SBO  >00                 *turn on card                     0AFC
*
ENDDSR MOV  *R2,R2              *no entry address given?          0AFE
       JEQ  LOOP                *start over                       0B00
*
       MOV  R2,@>83D2           *move R2 into save addr.          0B02
       INCT R2                  *add 2 to R2                      0B06
       MOV  *R2+,R9             *point to name                    0B08
*
       BL   @>0BE8              *check name via NAMEMATCH         0B0A
*
       JMP  SEARCH              *error, no match return           0B0E
*
       INC  R1                  *OK return, increment version no. 0B10
*
       BL   *R9                 *execute routine                  0B12
*
       JMP  SEARCH              *return if wrong peripheral       0B14
*
```

```
       SBZ  >00                   *correct peripheral return, turn
                                   off card
       JMP  CHGGRM                *jump to CHGGRM                 0B18
*
       CLR  *R8                   *return for other routines      0B1A
CHGGRM BL   @>0842                *change ROM search value routine  0B1C
*
END    B    @>006A                *return to GPL                  0B20
       END
```

==============================

```
       R1= DSR version number
       R2= address table value
       >83D0= predefined ROM/GROM value
       >83D2= save address
       >836D= jump increment for DSR entry table
       >83E5= R2 LSB
```

FIG. J.2.5: NAMEMATCH ROUTINE

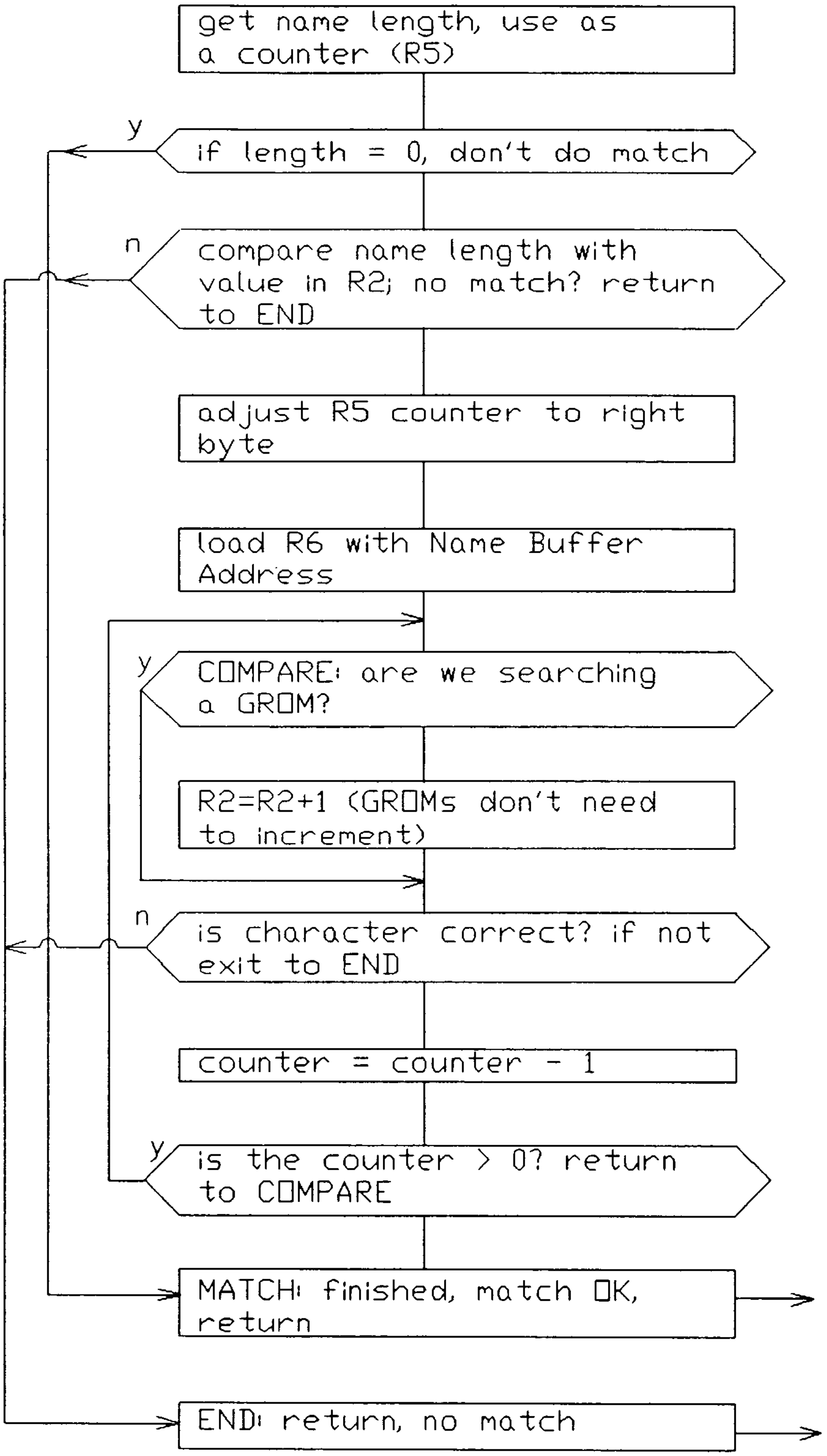


FIGURE J.2.6: NAMEMATCH ROUTINE CODE

```
       AORG >0BE8
                                                               0BE8
       MOVB @>8355,R5         *get name length, put in R5 for  0BE8
*
       JEQ  ENDOK             *END if zero                     0BEC
*
       CB   R5,*R2            *compare name length with value in 0BEE
                               R2
       JNE  ENDBAD            *END if bad                      0BF0
*
       SRL  R5,8              *adjust to right byte            0BF2
       LI   R6,>834A          *put NBA in R6                   0BF4
*
COMPAR CI   R2,>9800          *searching a GROM?               0BF8
       JHE  CHKCHR            *yes, jump                       0BFC
*
       INC  R2                *no, this is ROM, increment addr by 0BFE
                               1
*
CHKCHR CB   *R6+,*R2          *compare characters              0C00
       JNE  ENDBAD            *no match, end bad               0C02
*
       DEC  R5                *counter=counter-1 (R5)          0C04
       JNE  COMPAR            *keep going                      0C06
*
ENDOK  INCT R11               *good return                     0C08
*
ENDBAD B    *R11              *error return                    0C0A
       END
```

2.7 and 2.8). This will cycle through all peripherals and execute any power up routines found. The power up access routine clears CRULST (>83D0) since a specific device or subprogram name is not being searched for, and SCLEN (>8354) is also set to zero. A jump value of >04 is stored at >836D to tell GPLDSR where to start in the DSR access table. Then a branch to XML >19 (GPLDSR) is made. This access routine is made very early during the power up sequence and allow initialization of the peripherals before they are accessed by the console.

2.4 CALL subprogram access routine

When a CALL subprogram is executed from a running BASIC program, or in the command mode of BASIC or XBASIC (DSRs in peripheral ROMs are not searched by XBASIC programs during execution), then the GPL code at >50DB - >5110 is executed (see Fig. 2.9 and 2.10). First, >830C is cleared, and the contents of >832C are set equal to those of >8356. The CPU checks to see if the subprogram is in GROM or ROM. If in ROM, a CALL G@>0010 is executed, and the GPL DSRLNK routine is accessed. A data value of >0A is used as a jump value for proper entry in the DSR address table.

The GPL DSRLNK routine is at >03DC - >0415; the flowchart is shown in Fig. 2.9. Upon entry, the jump value is recovered and put into >836D. The MSB of the SCLEN (>8354) is cleared and the device/subprogram name length is placed in >8355. The name length variable for this routine (>8358) is cleared. The word value in >8356 is moved to >8352, which is used as a counter. The main loop compares the device name length in >8355 to >8358 to see if it is equal; if so, an exit is made. The character value is checked to see if it is a "."; if so, the loop is exited. Otherwise, the value of >8358 is incremented until it equals SCLEN or a "." is reached. This loop allows the device name to be shortened to characters left of a ".", thereby eliminating device options not needed in the search. An example of this would be "PIO.CR.LF" becomes "PIO.". The GPL DSRLNK routine is also used by the Main Device search routine as noted in Subsection 2.5). Once the loop is exited, the name length is checked to see if it is zero; if so, an error is returned. The new device name length is now moved from >8358 into >8355. If it is greater than 8 characters long, an error is returned. Now the MSB of SCLEN is cleared (again), the ROM search pointer CRULST (>83D0) is cleared, the subroutine pointer at >8356 is incremented by two, and the device name is moved to the buffer at >834A (NBA). The word at >8356 is moved to >8354 and now >8355 is the new device name length. Then GPLDSR is called via a XML >19.

2.5 File/Main Device access routines

BASIC and XBASIC can access peripherals through the console File Managment System which utilizes Peripheral Access Blocks or PABs. The File Managment System and PABs are covered in Section K. Because of the design of the File Management System, all peripherals' main device routines are accessed as files via different commands (or modes) such

FIG. J.2.7: POWERUP ROUTINE

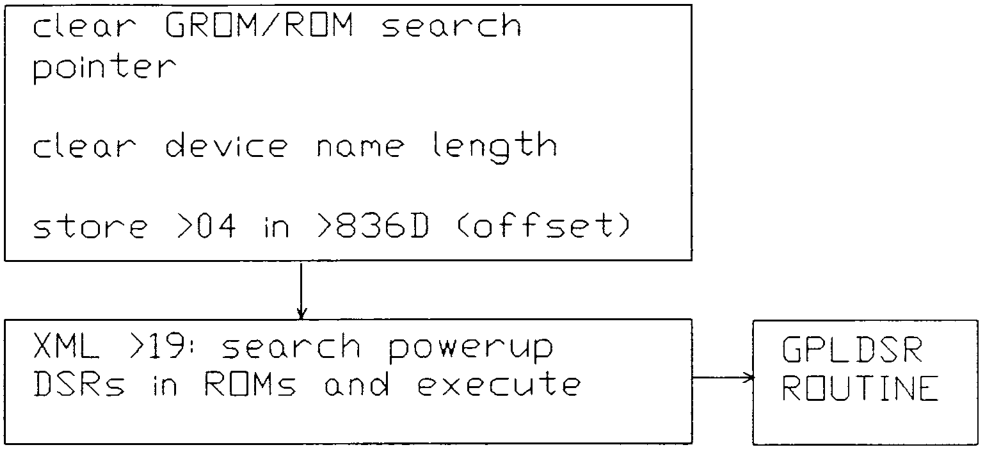
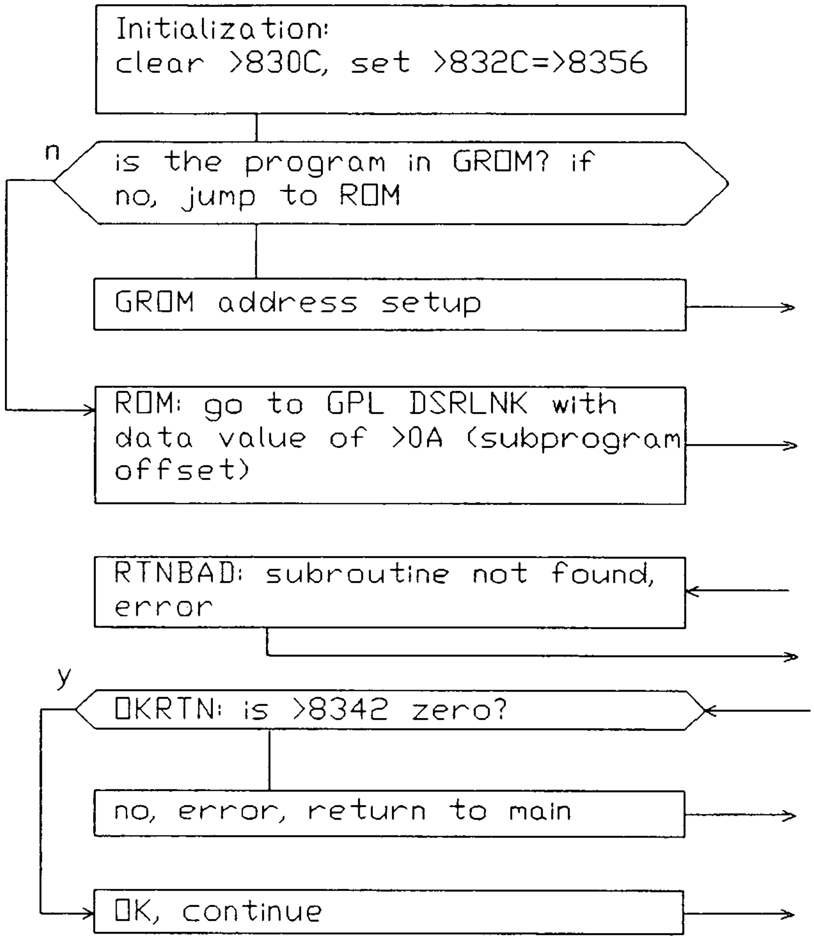


FIGURE J.2.8: POWERUP ROUTINE CODE

```
     DCLR  @>83D0        *clear ROM search pointer 0183
     CLR   @>8355        *clear device name length byte         0186
     ST    @>836D,>04    *store >04 in counter >836D            0188
*
     XML   >19           *go to GPLDSR                          018B
     END
```

FIG. J.2.9: SUBPROGRAM ACCESS ROUTINE



Note:
>832C=program text or token code pointer
>8356=subroutine pointer
>830C=temp. variable

FIGURE J.2.10: CALL SUBPROGRAM ROUTINE CODE

```
     CLR   @>830C            *clear >830C temp. variable          50E0
     DST   @>8356,@>832C     *set token code pointer = subroutine 50E2
                              pointer
*
     CZ    @>8389            *is the subprogram in GROM?          50E5
     BS    G@>5101           *no, in ROM, jump                    50E8
*
     MOVE  >0001 BYTE TO @>830D FROM G@>0000(@>832C)  *GROM access 50EA
     INC   @>830D                                     *code        50F1
     MOVE  *>830C BYTE TO V@>0320 FROM G@>0000(@>832C)*            50F3
     DST   @>8356,>0320                               *            50FA
     DADD  @>832C,@>830C                              *            50FE
*
ROM  CALL  G@>0010           *go to GPL DSRLNK                    5101
     DATA  >0A               *with data value of >0A (entry offset 5104
                              value)
*
     BS    G@>510C           *error return                        5105
*
     CZ    @>8342            *good return, check for end of line  5107
     BR    G@>5671           *no, error                           5109
*
     CONT                    *continue                            510B
     END
```

as OPEN, CLOSE, DELETE, etc. The access routines are in GPL in GROM 2; all I/O mode routines utilize the GPL code from >4CB9 - >4CE9 with three entry points: >4CB9, >4CC0, and >4CC6. Below is a list of access modes and their associated entry points (for console BASIC only):

I/O mode	Entry point
OPEN	>4CC6
DELETE	4CB9 DATA >07
CLOSE	4CC6
RESTORE	4CB9 DATA >04
INPUT	4CC0
LOAD	4CC6
SAVE	4CB9 DATA >06

[Note: The entire file access system logic is beyond the scope of this manual. Only the common access routines are provided.]

Figure 2.11 is the flowchart of the files access routines from >4CB9 - >4CE9. At entry point >4CB9, the opcode is recovered from the DATA statement following CALL G@>4CB9 and is written to the VDP. Entry point >4CC0 does not assume use of the DATA statement, and directly calls the last routine at entry point >4CC6. Upon return from >4CC6, the subroutine may either branch to an error message, or a normal return. Entry point >4CC6 installs the screen offset value and saves the FAC. The pointer for the DSR value is recovered, and an offset of >0D is added to provide proper table entry into the DSR. Then GPL DSRLNK is called with a data value of >08. Upon completion, the FAC is restored, checking is done for errors, and a proper return is made.

A point of interest is the GPL code from >4BA1 - >4BFB. This section apparently is used by the console in creating PABs.

3.0 XBASIC DSRLNK

The XBASIC cartridge from Texas Instruments does not load DSRLNK when a CALL INIT is issued, and therefore it is not available when in the XBASIC environment. The following assembly program (provided by TI) demonstrates how a DSRLNK assembly program maye access the RS232 card, print a message, and return to the XBASIC calling program. The code and comments are provided to assist programmers in understanding how the console accesses DSRs; note that several routines are very similar to those presented in Subsection 2.0.

FIG. J.2.11: GPL DSRLNK ROUTINE

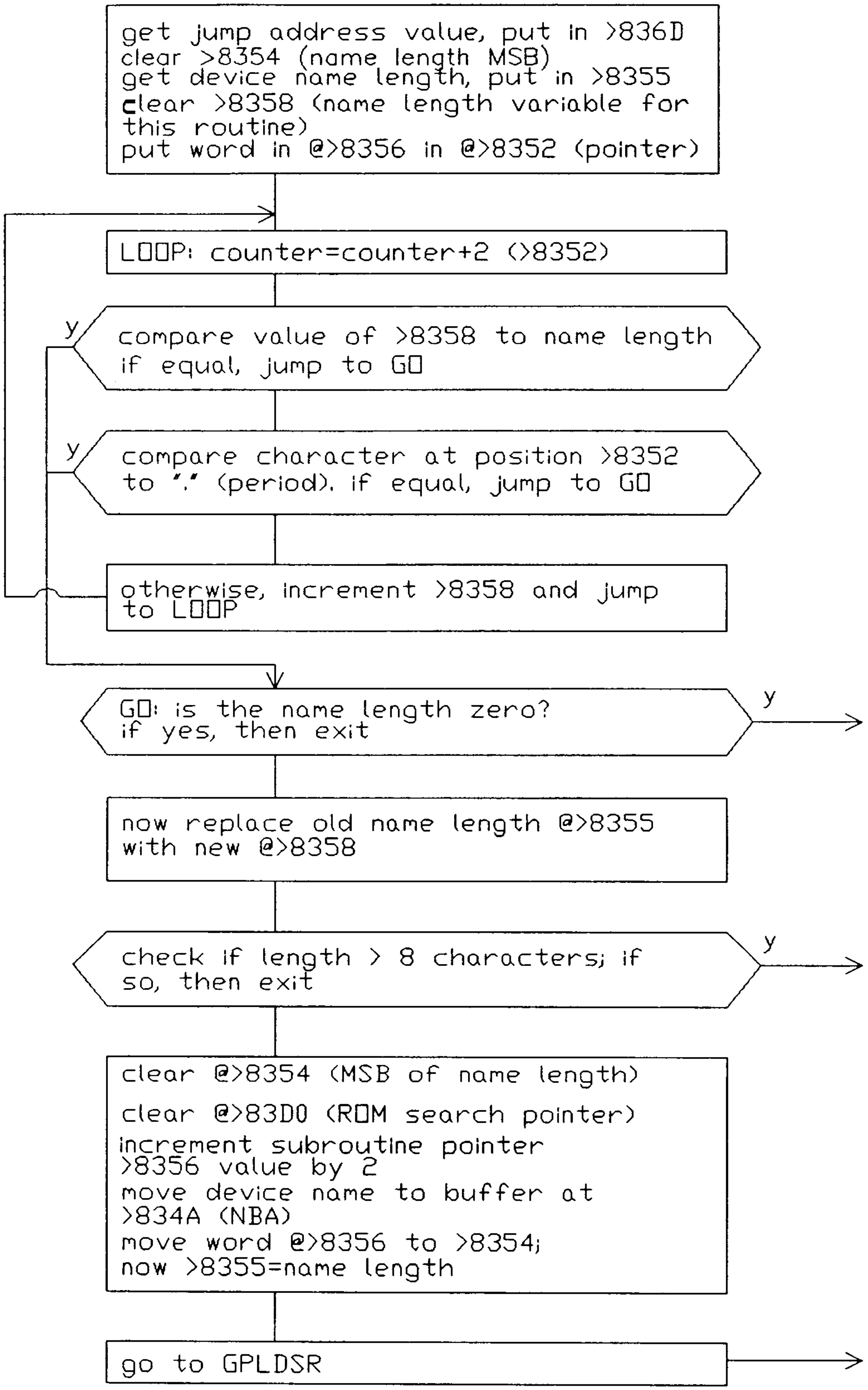


FIGURE J.2.12: GPL DSRLNK ROUTINE CODE

```
      FETCH @>836D            *get jump addr. value, put in >836D  03DC
      CLR   @>8354            *clear name length MSB               03DE
      ST    @>8355,V*>8356    *get device name length, put in >8355 03E0
      CLR   @>8358            *clear name length variable          03E4
      DST   @>8352,@>8356     *put pointer value in >8352          03E6
*
LOOP  DINC  @>8352            *counter=counter + 2                 03E9
*
      CEQ   @>8358,@>8355     *is name length equal?               03EB
      BS    G@>03FA           *yes, jump to GO                     03EE
*
      CEQ   V*>8352,>2E       *is the current character "."?       03F0
      BS    G@>03FA           *yes, jump to GO                     03F4
*
      INC   @>8358            *otherwise, increment >8358          03F6
      BR    G@>03E9           *and start again at LOOP             03F8
*
GO    CZ    @>8358            *is the name length zero?            03FA
      BS    G@>0438           *yes, then exit                      03FC
*
      ST    @>8355,@>8358     *replace old name length with value  03FE
                               in  >8358
*
      CGE   @>8355,>08        *is it greater than or equal to 8?   0401
      BS    G@>0438           *yes, then error exit                0404
*
      CLR   @>8354            *clear MSB of name length word       0406
      DCLR  @>83D0            *clear ROM search pointer word       0408
      DINC  @>8356            *increment subroutine pointer by 2   040B
      MOVE  *>8354 BYTE TO @>834A FROM V*>8356 *move new name to VDP 040D
                                                buffer
      DADD  @>8356,@>8354     *move word at >8356; now >8355=name  0412
                               length
*
      XML   >19               *go to GPLDSR routine                0415
      END
```

FIG. J.2.13: MAIN DEVICE/FILES ACCESS ROUTINES

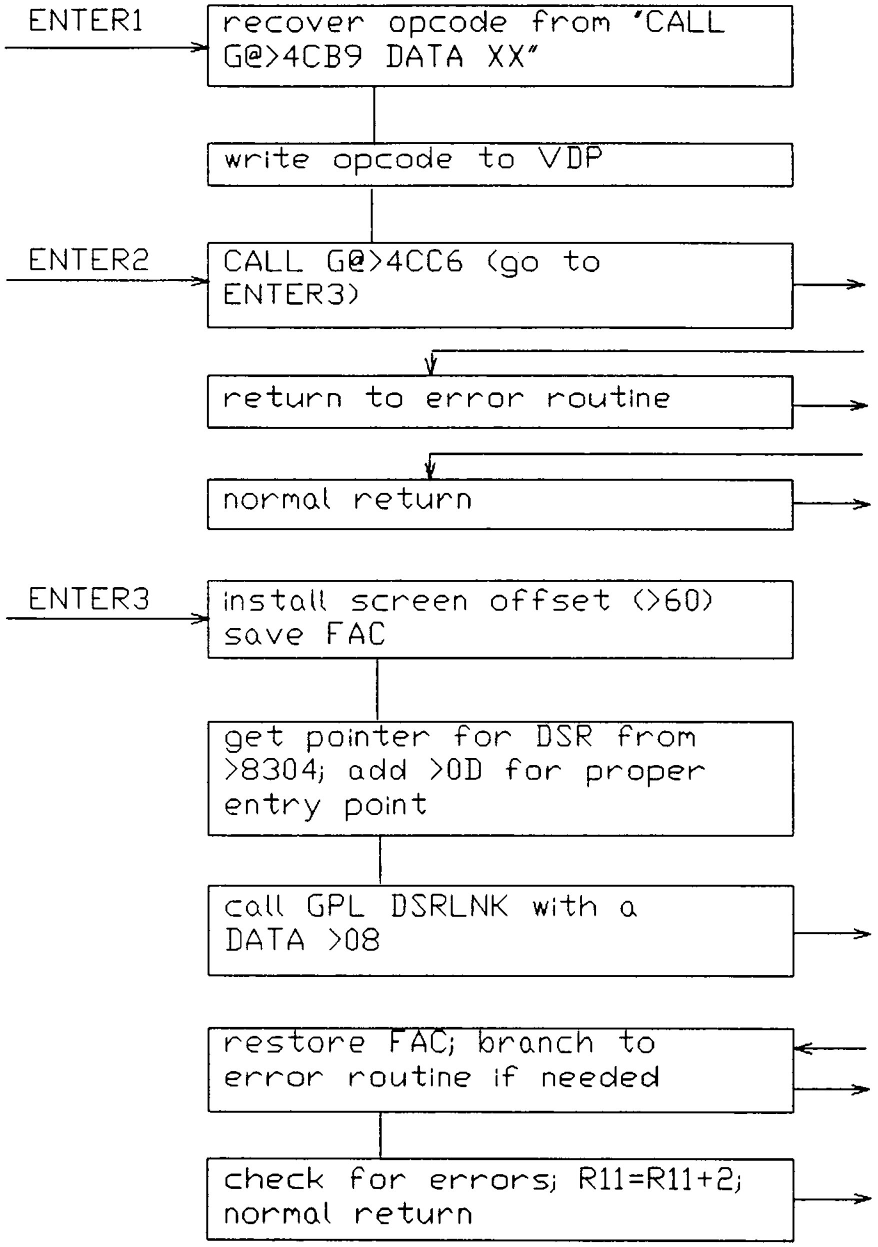


FIGURE J.2.14: MAIN DEVICE/FILES ROUTINE CODE

```
ENT1 FETCH @>8356                           *recover opcode from   4CB9
     ST    V@>0004(@>8304),@>8356           *calling routine (DATA 4CBB
*                                            >XX) and write to VDP
ENT2 CALL  G@>4CC6                          *call ENT3 routine     4CC0
     BR    G@>57C3                          *error return          4CC3
     RTN                                    *normal return         4CC5
*
ENT3 ST    V@>000C(@>8304),>60              *install screen offset 4CC6
     MOVE  >001E BYTE TO V@>03C0 FROM @>834A *save FAC              4CCB
     DST   @>8356,@>8304                    *get pointer from >8304 4CD1
     DADD  @>8356,>000D                     *add >0D for entry addr 4CD4
     CALL  G@>0010                          *call GPL DSRLNK       4CD8
     DATA  >08                              *with a DATA >08       4CDB
     MOVE  >001E BYTE TO @>834A FROM V@>03C0 *restore FAC           4CDC
     BS    G@>4CE9                          *branch to error if    4CE2
                                             needed
     CLOG  V@>0005(@>8304),>E0              *check for other error 4CE4
     RTNC                                   *return + 2 no errors  4CE9
     END

*************************************
build PAB is located at >4BA1 - >4BFB
```

```
      *********************************************
      *                                           *
      *  DSR ROUTINE FOR EXTENDED BASIC ENVIORNMENT *
      *                                           *
      *********************************************
*
SCLEN  EQU  >8355             NAME LENGTH
SCNAME EQU  >8356             NAME BUFFER POINTER
CRULST EQU  >83D0             CRU ADDR VALUE
SADDR  EQU  >83D2             SAVE ADDRESS BYTE
GPLWS  EQU  >83E0             GPL WORKSPACE
VDPRD  EQU  >8800             VDP READ ADDR
VDPWD  EQU  >8C00
VDPWA  EQU  >8C02
FLGPTR DATA 0                 POINTER TO FLAG BYTE IN PAB
SVGPRT DATA 0                 SAVE GPL RETURN
SAVCRU DATA 0                *CRU ADDRESS OF PERIPHERAL
SAVENT DATA 0                *ENTRY ADDRESS OF DSR
SAVLEN DATA 0                 SAVE DEVICE NAME LENGTH
SAVPAB DATA 0                 POINTER TO DEVICE NAME IN PAB
SAVVER DATA 0                *VERSION  OF DSR
DLNKWS DATA 0,0,0,0,0
TYPE   DATA 0,0,0,0,0,0,0,0,0,0,0,
*
***DATA
*
C100   DATA 100
H20    EQU  $
H2000  DATA >2000
DECMAL TEXT '.'
HAA    BYTE >AA
*
***UTILITY BLWP VECTORS
*
DSRLNK DATA DLNKWS,DLENTR     LINK TO DSR
*
***LINK TO DSR
*
DLENTR MOV  *R14+,R5          FETCH PROGRAM TYPE FOR LINK
       SZCB @H20,R15          RESET EQUAL BIT
       MOV  @SCNAME,R0        FETCH POINTER INTO PAB
       MOV  R0,R9             SAVE POINTER
       AI   R9,-8             ADJUST POINTER TO FLAG BYTE
       BLWP @VSBR             READ DEVICE NAME LENGTH
       MOVB R1,R3             STORE IT
       SRL  R3,8              MAKE IT A WORD
       SETO R4                INIT COUNTER
       LI   R2,NAMBUF         ADDRESS OF BUFFER TO HOLD NAME
LNK$LP INC  R0
       C    R4,R3             END OF NAME?
       JEQ  LNK$LN
```

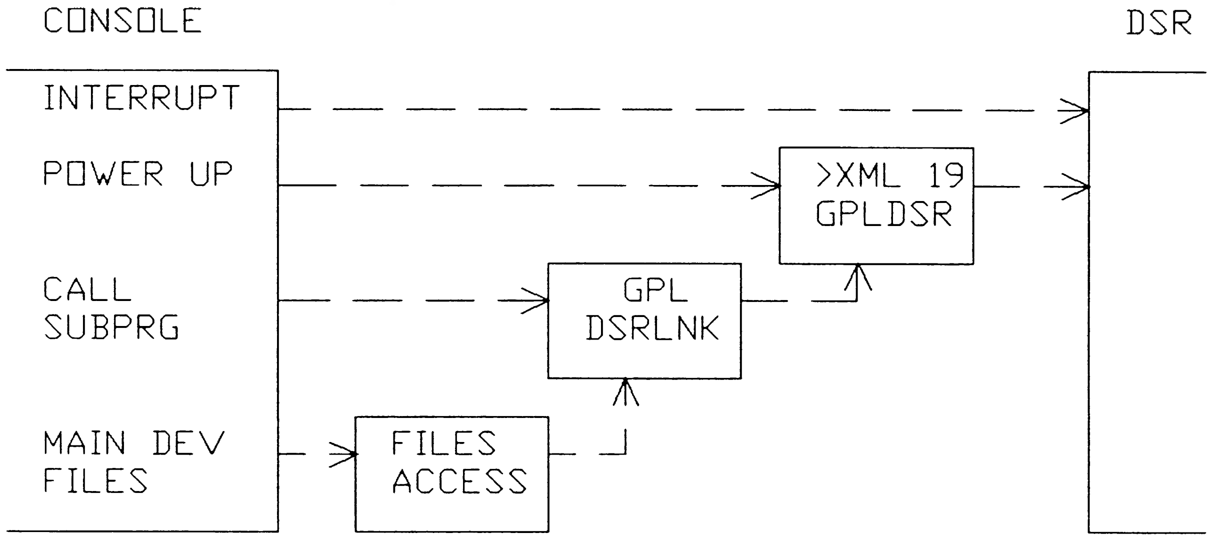
```
        BLWP @VSBR              READ A CHARACTER
        MOVB R1,*R2+            MOVE TO NAMBUF
        CB   R1,@DECMAL         A PERIOD?
        JNE  LNK$LP
LNK$LN  MOV  R4,R4              IS NAME LENGTH ZERO?
        JEQ  LNKERR             ERROR ROUTINE
        CI   R4,7               IS NAME LENGTH > 7?
        JGT  LNKERR             ERROR ROUTINE
        CLR  @CRULST
        MOV  R4,@SCLEN-1        STORE NAME LENGTH FOR SEARCH
        MOV  R4,@SAVLEN         SAVE LENGTH
        INC  R4                 NEXT AVAILABLE POSITION
        A    R4,@SCNAME         POSITION AFTER NAME
        MOV  @SCNAME,@SAVPAB    SAVE POINTER INTO DEVICE NAME
*
***SEARCH ROM FOR DSR
*
SROM    LWPI GPLWS              USE GPLWS FOR SEARCH
        CLR  R1
        LI   R12,>0F00          START
NOROM   MOV  R12,R12            ANYTHING TO TURN OFF?
        JEQ  NOOFF
        SBZ  0                  TURN OFF
NOOFF   AI   R12,>0100          NEXT ROM
        CLR  @CRULST            CLEAR IN CASE WE'RE FINISHED
        CI   R12,>2000          END OF CONSOLE ROM
        JEQ  NODSR              DIDN'T FIND DSR
        MOV  R12,@CRULST        NEXT CRU
        SBO  0                  TURN ON ROM
        LI   R2,>4000           START AT BEGINNING
        CB   *R2,@HAA           IS IT A VALID ROM?
        JNE  NOROM
        A    @TYPE,R2           GO TO FIRST POINTER
        JMP  SGO2
SGO     MOV  @SADDR,R2          CONTINUE WHERE WE LEFT OFF
        SBO  0                  TURN ON AGAIN
SGO2    MOV  *R2,R2             ZERO?
        JEQ  NOROM              NO PROGRAM
        MOV  R2,@SADDR          REMEMBER WHERE TO GO NEXT
        INCT R2                 GO TO ENTRY POINT
        MOV  *R2+,R9            GET ENTRY ADDRESS
*
***SEE IF NAME MATCHES
*
        MOVB @SCLEN,R5          GET LENGHT AS COUNTER
        JEQ  NAME2              ZERO LENGTH, NO MATCH
        CB   R5,*R2+            LENGTH MATCH?
        JNE  SGO
        SRL  R5,8               MAKE WORD
        LI   R6,NAMBUF          POINT TO BUFFER
NAME1   CB   *R6+,*R2+          CHARACTER CORRECT?
        JNE  SGO
```

```
       DEC  R5              MORE TO LOOK AT?
       JNE  NAME1
NAME2  INC  R1              NEXT VERSION FOUND
       MOV  R1,@SAVVER      SAVE VERSION NUMBER   * CAN BE USED *
       MOV  R9,@SAVENT      SAVE ENTRY POINT      *  TO AVOID   *
       MOV  R12,@SAVCRU     SAVE CRU ADDRESS      * SUBSEQUENT  *
       BL   *R9             CALL SUBROUTINE       *  SEARCHES   *
       JMP  SGO             NOT RIGHT VERSION
       SBZ  0               TURN OFF ROM
       LWPI DLNKWS          SELECT DSRLNK WORKSPACE
       MOV  R9,R0           POINT TO FLAG BYTE IN PAB
       BLWP @VSBR           READ FLAG BYTE
       SRL  R1,13           SAVE ERROR FLAGS
       JNE  IOERR           ERROR?
       RTWP
*
***ERROR HANDLING
*
NODSR  LWPI DLNKWS          DSRLNK WORKSPACE
LNKERR CLR  R1              CLEAR ERROR FLAGS
IOERR  SWPB R1
       MOVB R1,*R13         STORE ERROR IN R0 OF CALLER
       SOCB @H20,R15        INDICATE AN ERROR
       RTWP                 RETURN TO CALLER
```

TABLE J.1: DSR-RELATED RAM ADDRESSES

Address	Name	Description
>202E	FLGPTR	pointer to the flag in the PAB
>2030	SVGPRT	GPL return address
>2032	SAVCRU	CRU address of the peripheral
>2034	SAVENT	entry address of DSR or subprogram
>2036	SAVLEN	device or subprogram name length
>2038	SAVEPAB	pointer to the device or subprogram name in PAB
>203A	SAVVER	version number of the DSR
>8300 - >834F		used by BASIC and XBASIC for temporary storage; the following address apply to DSRs and/or PABs
>832C		pointer to token code
>833C		pointer to first entry in PAB list
>8342		current character/token
>834A - >836D		FAC (floating point accumulator)
>834A		PAB I/O code
>834B		flag/status
>834C		data buffer address
>834E		logical record length
>834F		character count
>8350		record number
>8352		screen offset
>8353		option length
>8354	SCLEN	device length
>8356	SCNAME	subroutine pointer/ 1st char after PAB in VDP
>8358 - >836D		DSR use
>836E - >837B		misc. GPL usage
>837C	STATUS	GPL status byte, set to zero for DSR call. Bit 2 is cond. bit; console turns this bit on to indicate non-existant file.
>8380 - >83BF		misc. GPL usage
>83C0 - >83DE		INTWS Interrupt workspace, also:
>83D0	CRULST	
>83D2	SADDR	
>83E0 ->83FF		GPLWS GPL workspace

FIG. 2.15: OVERALL ACCESS



SECTION K: MISCELLANEOUS ACCESS NOTES

1.0 Introduction

As the title implies, this section covers some miscellaneous tips and techniques on accessing peripheral devices. Subsection 2.0 briefly covers direct access of peripherals and their DSRs. Subsection 3.0 summerizes information on Peripheral Access Blocks (PABs), particularly as they apply to interaction with DSRs.

2.0 Direct Access

2.1 Memory access

Access to the DSR or peripheral (if memory mapped) by the console, or access to system memory by the DSR is straightforward. All possible memory locations fall between >4000 to >5FFF for peripherals; it is assumed that the proper CRU manipulations have been performed to page in the desired peripheral into the >4000 space prior to access. The CRU base address should not be changed by the DSR until the >4000 memory space is exited unless the contents of R12 are saved upon entry into the DSR..

Data may be moved between the peripheral/DSR and the system via MOV and MOVB instructions. MOV will transfer a 16 bit word, while MOVB transfers the MSB of a 16 bit word. Some examples of these instructions are as follows:

Operation	Example	Description
register to register	MOV R4, R0	move contents of R4 into R0
	MOVB R4, R0	move R4 MSB to R0 MSB
register to memory	MOV R4, @>40C0	move contents of R4 into address >40C0 and >40C1
	MOVB R4, @>40C0	move R4 MSB to byte >40C0
register indirect	MOV R4, *R0	move contents of R4 to address given in R0
	MOVB R4, *R0	move R4 MSB to MSB of address given in R0
reg. auto-increment	MOV R4, *R1+	move contents of R4 into R1 and increment R1 by two
	MOVB R4, *R1+	move R4 MSB into R1 MSB and increment R1 by one
indexed memory	MOV @>4033(R3), R2	move contents at address given in R3 plus >4033
	MOVB @>4033(R3), R2	move MSB between locations

Other transfer modes are possible. The Indexed Memory and Auto-Increment are very common, particularily with memory mapped devices, and VDP transfers. It is recommended that the programmer use labels for commonly referenced addresses. Care must be taken with devices such as GROMs, which autoinrement their addresses with each access.

2.2 CRU access

Individual CRU bits may be manipulated via the SBO and SBZ commands. For example, if register 12 contains the base CRU address, >1500 for example, then

```
SBO 0
```

would activate CRU bit >1500 to 1 or "on". Likewise, the command

```
SBZ 4
```

would set CRU bit >1508 to 0 or "off". SBO 0 and SBZ 0 are found in the DSR access routines and are used to activate the peripheral device and its DSR ROM. Other SBO/SBZ commands can be found in the DSR itself that manipulate the indicator light or other functions. The programmer is reminded that all DSR software should clear any CRU bits activated when exiting the DSR (the FMS will deactivate the 0 bit). Most CRU bits toggle latches or flip flops which may remain active unless its CRU bit is cleared, causing bus contention once the DSR is exited.

The command TB can be used to determine the status of a CRU bit. Following the previous example, if the CRU base is >1500, then

```
TB >5
```

would input the value (1 or 0) of CRU bit >150A. TB works only for peripherals that have an active CRUIN line to the console.

The multiple bit transfers are more involved than the single bit transfers. Register 12 contains the starting address of the CRU bit selected, and the count "n" in the instruction indicates the decimal number of successive bits to be transferred. For example:

```
LDCR @>4351,4
```

The command LDCR is an output operation. It transfers data from the memory location >4351 (example) to the 4 CRU bits, starting at the address that is the value of the contents of register 12. If n, the number of bits to be transferred, is less than 8 the address is a byte address. If n is 8 or more, the address represents a word address. If n is 0, 16 bits are transferred. If using register addressing, the byte addressed (for n less than 8) is the left byte of the register containing the data. These statements apply equally to the input operation STCR, which inputs n bits from the CRU device and stores them in the specified memory location. For example, to input 6 bits to the memory location addressed by the contents of register 10 would require

```
STCR *R10,6
```

Single bit operations are usually associated with standard DSR peripherals, while multiple bit CRU operations are useful for CRU-only peripherals, or ones with 9901 interface chips.

2.3 Programming examples

Some simple assembly programming examples for accessing various peripheral types are provided below. Note that all peripherals that do not use a DSR in ROM must be accessed directly by an applications program located in the system RAM, and cannot be accessed by the FMS of the /4A.

*CRU-only

```
        LI R12,>0600   set CRU base for >0600
        SBO 4          turn on >0608 bit
        -
        -
END     SBZ 4          turn off >0608 bit
        B *R11         return
```

*Non DSR peripheral
These peripherals are paged into the >4000 space by an independent applications program, and not the console.

```
        LI R12,>1500   set CRU base for >1500
        SBO 0          turn on bit 0. could be any bit within assigned
                       range of CRU bits
        -
        [program/device access]
        -
END     SBZ 0          turn off peripheral (bit 0)
        B *R11         return
```

*DSR peripheral
These peripherals are normally accessed by the console FMS. However, they can be accessed via an applications program using code similar to the previous example.

*Non-CRU memory-mapped
These devices are in the >4000 - >5FFF memory address space, but are paged in only when no other DSR peripheral is active. Although this feature is implemented in hardware (see Section C), the following code may be used to ensure that conflict does not occur.

```
        LIMI 0         disable interrupts
        LI R12,>0F00   initialize CRU counter
LOOP    AI R12,>0100   increment by >0100
        SBZ 0          turn off DSR peripheral
```

```
        CI R12,>2000   check if at end
        JEQ END        stop
        JMP LOOP       continue
   END  $
        -
        [access to device]
        -
        B *R11         return
```

3.0 Notes on PABs and File Management System

The Peripheral Access Blocks, or PABs are used in the BASIC/XBASIC environment to access the peripheral main devices via the File Management System (FMS). The Editor/Assembler manual, section 18, and Peripheral Technical Data Manual, section C, cover PABs and the FMS in great detail and contain programming examples. The following notes are provided to assist the developer/programmer when designing DSR peripherals that interface with the FMS.

A) Consider the type of peripheral vs. its access mode. Memory mapped devices will utilize sequential files. Mass storage devices may use either sequential or relative files. Some peripherals may only read, and are therefore limited to the INPUT mode. The DSR must clearly define which I/O modes it will operate in, and generate error codes for those that it does not implement.

B) DSRLNK is used to access the DSR in applications programs that bypass the FMS. XBASIC requires a routine like that in Section J. A DATA >8 directive after the BLWP @DSRLNK is for linkage to a main device routine, and >A for a low level routine.

C) DSR detected errors should be indicated in the flag byte (1) of the PAB. The DSR should save the least significant 4 bits of the I/O opcode byte or the most significant 5 bits of the flag byte.

Figure K.1: PAB Flag Byte

```
   bit  0  1  2  3  4  5  6  7
        X  X  X  X  X  E  E  E
                       |--|--|----3 BIT ERROR CODE
```

It is the responsibility of the DSR to set bits 5-7 in the flag byte to inidicate errors that occured within the DSR. Table K.1 defines the error codes.

D) The DSR is responsible for updating byte 8 (screen offset) of the PAB when a STATUS I/O opcode is issued. When a STATUS (>09) code is issued, the applications program can check on the current status of the peripheral. The first six bits have meaning regardless if the file is currently open or not. The last two bits are valid for open files only. Some DSRs such as the decoded CLOCK DSR in Section I do not implement the STATUS I/O response; it is recommended that all DSRs be capable of responding to this opcode.

TABLE K.1: ERROR CODES FOR DSRS

Error code	Meaning
000	no error or bad device name if bit 2 of >837C is set by FMS
001	device is write protected
010	bad open attribute or no records in relative file
011	illegal operation
100	out of buffer space on the device
101	attempt to read past EOF, or non existent relative record
110	device error
111	file error

TABLE K.2: STATUS BIT DEFINITION

Status bit	Meaning
7	logical end of file; 1=EOF, cannot read more
6	physical end of file; 1=end of physical media
5	record type; 1=variable 0=fixed
4	file type; 1=program 0=data
3	data type; 1=BINARY/INTERNAL 0=ASCII/DISPLAY
2	reserved set to zero
1	protect flag; 1=protected 0=not protected
0	non-existent file