TI-99/8 HOME COMPUTER

BASIC INTERPRETER

Design Specification

(SECTION 3 ONLY)

Consumer Group
Mail Station 5890
2301 N. University
Lubbock, Texas 79414

TEXAS INSTRUMENTS
INCORPORATED

Date: August 3, 1983
Version 1.0

TABLE of CONTENTS

Paragraph Title

SECTION 3 GENERAL DESCRIPTION

LIST of TABLES

LIST of FIGURES

SECTION 3

GENERAL DESCRIPTION

The BASIC interpreter included in the TI-99/8 console is an integral part of the system software included in the console. The interpreter is intended to be ANSI and TI Standard compatible and is intended to provide access to some of the unique features of the TI-99/8's hardware. The interpreter provides several enhancements above the ANSI nucleus to access the color graphic capability and to access the sound generators contained in the machine.

3.1 Use of System Resources

This section describes how the interpreter utilizes the memory resources of the TI-99/8 console, including the RAM and ROM contained within the system. The interpreter resides in approximately 32K of ROM and approximately 18K of GROM. As much of the speed critical code as possible was put in the high speed ROM to increase the speed of the interpreter.

BASIC utilizes the CPU RAM for program storage, symbol table storage, Peripheral Access Blocks, string space, crunch buffer, and the floating-point stack. The CPU RAM is also used to maintain all of the necessary pointers and temporary variables involved with editing, prescanning, and executing a BASIC program.

BASIC utilizes the VDP RAM for screen I/O.

3.1.1 ROM Usage.

The ROM contained in the console contains many of the most frequently executed parts of the BASIC interpreter. The ROM code portion of the interpreter is contained in 19 separate assembly modules. Two of these modules are entitled PARSE and BASSUP. PARSE contains the statement dispatcher, the parser front-end and ending code, and some of the most frequently used NUD, LED, and statement handlers (see section 4.3). It also contains the CPUSH and CPOP routines which get values on and off the value stack. BASSUP contains several of the most frequently used support routines such as the symbol table search routine, symbol

assignment routine, and several small routines to access the VDP RAM.

In order to put together all of the ROM code for the entire system, each of the separate modules of the system must be included in a link-edit. The modules contained in the TI-99/8 console are:

INTSIM - The GPL interpreter
XOPS - The memory mapper utilities
BIPHAS - The cassette device service routine
FLTPT - The floating point package
CSN - The convert string to number routine
SCREEN - The screen handler
NEWXML - The XML table
X2SUBS - The utility routines callable from GPL, or as an XOP/XML2 call
X3SUBS - The utility routines callable from GPL, or as an XOP/XML3 call
TRINSIC - The intrinsic functions
CNS - The convert number to string routine
STRING - The string package
BASSUP - The BASIC support package
FLMGRSUP - The file manager support routines
PARSE - The BASIC parser
FORNEXT - The FOR/NEXT command
SCAN99X - The prescan routine
SUBPRGX - The subprogram routines

Since the assembly language portions of the console can be link-edited there are no problems in putting all of the modules together, as there are with the GPL portions of the console.

3.1.2 GROM Usage.

The GROM portion of the interpreter contains part of the BASIC interpreter. The line input, crunching, program editing, and some of the statement NUD and LED handlers are contained in GROM. All input and output routines, and all of the GPL subprograms such as SOUND, COLOR, and KEY are located in GROM. The error messages' text as well as the error handling routine are also located in GROM code.

The GROM portion of the interpreter is contained in five separate assembly modules. These modules are entitled: EDIT, PSCAN, EXEC, FILEMGR, and GSUB. These mnemonics describe the portions of BASIC handled by each of these assemblies. EDIT handles mainly the editing, crunching, and top-level portions of BASIC. PSCAN contains the error handling routine as well as

numerous subroutines such as the line input routine. EXEC contains all of the GPL portion of execution except the screen/file management portions which are contained in the FILEMGR assembly. GSUB contains all of the BASIC provided GPL subprograms, such as CLEAR, SOUND, and COLOR.

In order to include all the separate assemblies of the console together, a GPL object code linker is available. The linker is simply a file concatenator. There is no link-editing capability, so linkage from one assembly to another is fairly complicated. Using BASIC as an example, the way to get from one assembly to another is to set up a branch table at the beginning of each assembly and 'ORG' it to a permanent location. In order to tranfer control between routines in different assemblies, the address of the branch-table entry for the destination routine is used as an equate in the assembly of the source routine. An example is:

```
*       ASSEMBLY 1
*       BRANCH TABLE
        GROM 7              GROM 7 starts at >E000
        ORG >40             Offset of >40 bytes
        BR   SCANKB         >E040 - Link to scan routine
        BR   WRITE          >E042 - Link to write routine

*       ASSEMBLY 2
SCANKB  EQU  >E040          Address in ASSEMBLY 1 of link to SCANKB
WRITE   EQU  >E042          Address in ASSEMBLY 1 of link to WRITE
        CALL SCANKB         Call SCANKB routine in ASSEMBLY 1
        B    WRITE          Goto WRITE routine in ASSEMBLY 1
```

In this example ASSEMBLY 2 has equates for SCANKB and WRITE, which are routines in ASSEMBLY 1. The equates are used in the CALL and Branch statements in ASSEMBLY 2. It should be noted that this causes one extra instruction for each routine (the table entry Branch instruction in ASSEMBLY 1), but this is a small price to pay for the convenience of having BASIC in manageable pieces.

When ASSEMBLY 2 encounters the CALL SCANKB instruction, control transfers to a predetermined address in ASSEMBLY 1, >E040. The instruction at this address then transfers control to the routine SCANKB in ASSEMBLY 1. When the SCANKB routine is complete, control comes back to the B WRITE instruction in ASSEMBLY 2. This instruction causes control to go to the ASSEMBLY 1 address >E042, which then transfers control to the ASSEMBLY 1 WRITE routine.

In order to keep track of the amount of GROM used and which parts are used by which sections, a GROM map is kept. The GROM

map reflects the organization and usage of the console GROMs. The following figure is the GROM map for the latest revision of the console GROM code.

TI-99/8 GROM MEMORY MAP
REVISION x
RELEASED xx/xx/xx
xx/xx/xx xx:xx

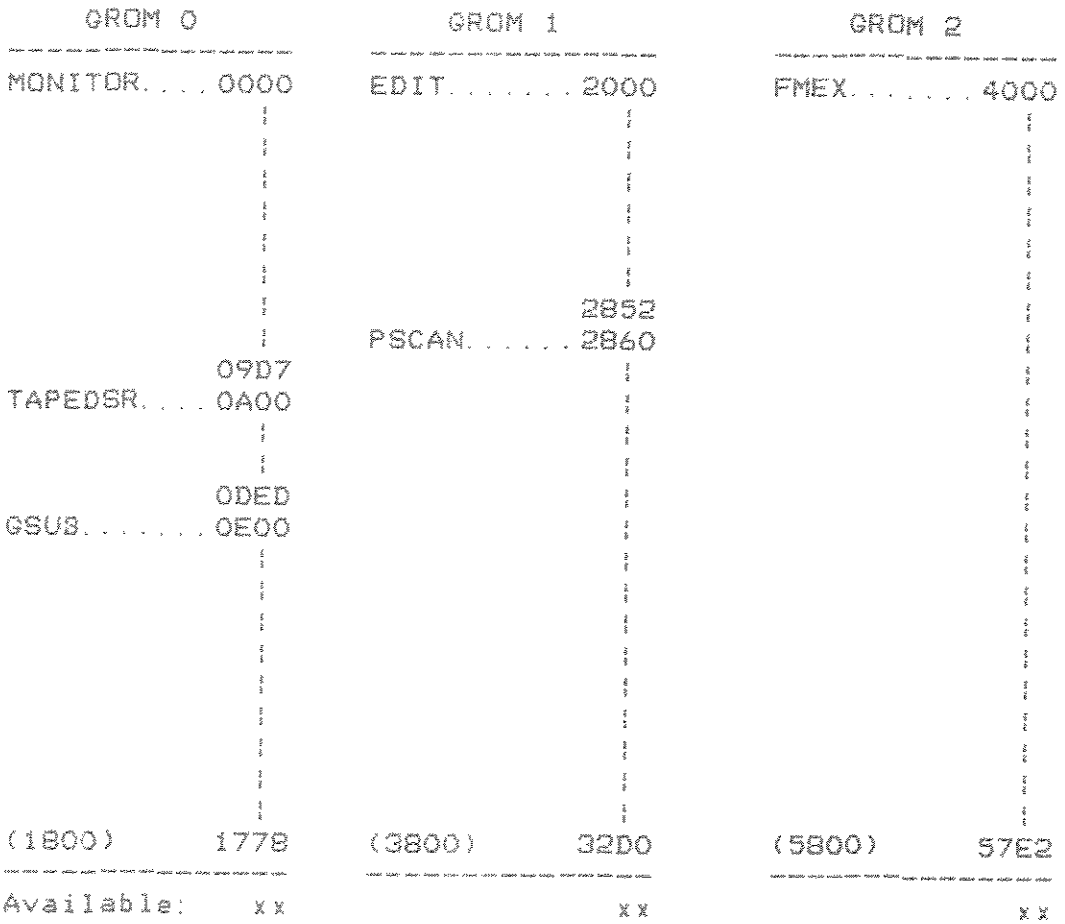


Figure 3-1 TI-99/8 GROM Memory Map

Each GROM holds 6K worth of information and has been fully utilized.

3.1.3 CPU RAM Usage.

The 2K bytes of on-board RAM are used exclusively for the Scratch Pad RAM. The Graphics Language interpreter, various interrupt routines, and peripheral devices use 142 bytes from >8372 to >83FF. The rest of the 2K bytes is used by the BASIC interpreter. This section itemizes BASIC's usage of the Scratch Pad RAM and generalizes the GPL interpreter's usage of the remaining bytes. Note that the GPL interpreter's workspace (>83E0 to >83FF) is also used by the assembly language portions of BASIC, so it is important to preserve information that the GPL interpreter needs. This information includes workspace registers 13, 14, and 15.

The following figure shows the layout of the Scratch Pad RAM using logical addresses.

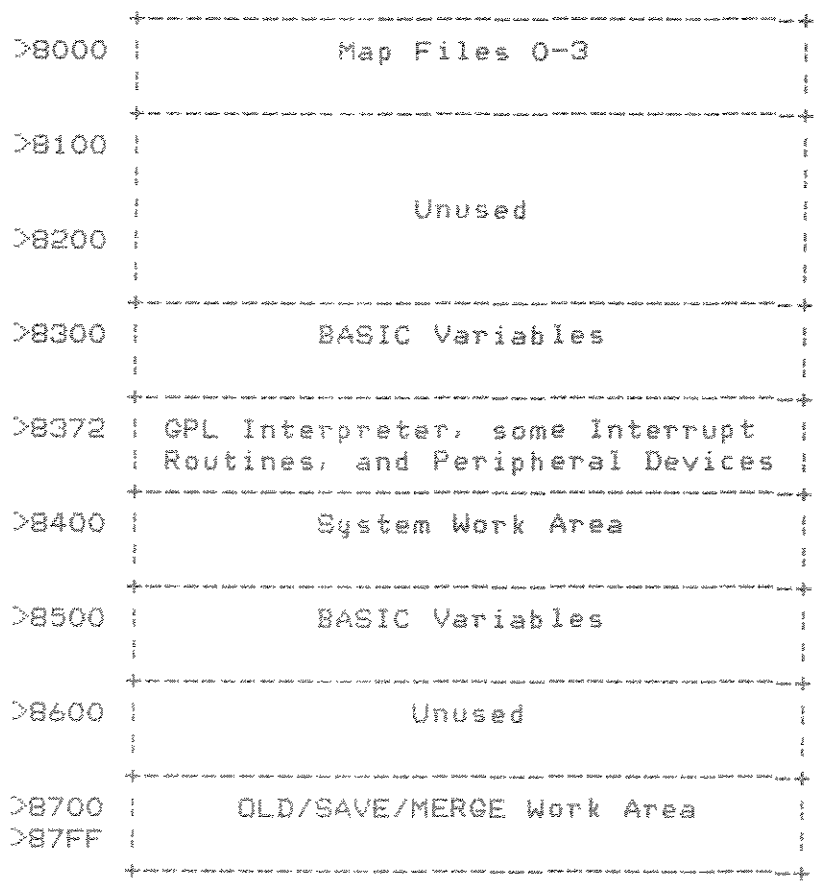


Figure 3-2 Scratch Pad RAM Description

3.1.4 VDP RAM Usage.

The VDP RAM is used as the primary memory for the screen, color table, character generator table, sprite information, and sound list. The current graphics mode determines how the VDP RAM is partitioned for use by the BASIC interpreter. More information on the data structures used in these areas can be found in the particular sections of this document which describe them. The following tables describe VDP RAM usage for each graphics mode.

Table 3-1 VDP RAM Usage for Pattern Mode

Addresses	Use
0000-02FF	Screen (768 bytes)
0300-037F	Sprite Attribute List (128 bytes)
03C0-03DF	Color Table (32 bytes)
03E0-03FF	Sound List (32 bytes)
0780-07FF	Sprite Velocity List (128 bytes)
0800-0FFF	Character Generator Table (2K bytes)
0800-0FFF	Sprite Generator Table (2K bytes)

Text mode usage of VDP RAM is similar to that of Pattern mode. The difference is that since the screen takes up more memory in Text mode, the Sprite Attribute List is at address >0400 instead of >0300.

Table 3-2 VDP RAM Usage for Text Mode

Addresses	Use
0000-038F	Screen (960 bytes)
03C0-03DF	*Color Table (32 bytes)
03E0-03FF	Sound List (32 bytes)
0400-047F	*Sprite Attribute List (128 bytes)
0780-07FF	*Sprite Velocity List (128 bytes)
0800-0FFF	Character Generator Table (2K bytes)
0800-0FFF	*Sprite Generator Table (2K bytes)

* The sprite- and color-related tables are maintained even though sprites are not displayed in text mode. This is to insure that a CALL SPRITE command will have a safe place to write data.

The two Split-Screen modes and the High-Resolution mode, use completely different VDP RAM locations than the Pattern and Text modes.

Table 3-3 VDP RAM Usage for Split-Screen: Text-High Mode

Addresses	Use
0000-07FF	Sprite Generator Table (2K bytes)
0000-17FF	Character Generator Table (6K bytes)
1800-1AFF	Screen (768 bytes)
1B00-1B7F	Sprite Attribute List (128 bytes)
1B80-1BFF	Sprite Velocity List (128 bytes)
1C00-1C1F	Sound List (32 bytes)
2000-37FF	Color Table (6K bytes)

The two Split-Screen modes are the same except for the location of the Sprite Generator Table.

Table 3-4 VDP RAM Usage for Split-Screen: Text-Low Mode

Addresses	Use
0000-17FF	Character Generator Table (6K bytes)
1000-17FF	Sprite Generator Table (2K bytes)
1800-1AFF	Screen (768 bytes)
1B00-1B7F	Sprite Attribute List (128 bytes)
1B80-1BFF	Sprite Velocity List (128 bytes)
1C00-1C1F	Sound List (32 bytes)
2000-37FF	Color Table (6K bytes)

The High-Resolution mode has the Sprite Generator Table at >3800, which is different from both the Split-Screen modes. This mode also has an I/O Screen which is unique to the High-Resolution and Multicolor modes. The I/O Screen is the location where the results from the ACCEPT, DISPLAY, INPUT, LINPUT, and PRINT statements are placed. This causes the results from these statements to be invisible to the user.

Table 3-5 VDP RAM Usage for High-Resolution Mode

Addresses	Use
0000-17FF	Character Generator Table (6K bytes)
1800-1AFF	Screen (768 bytes)
1B00-1B7F	Sprite Attribute List (128 bytes)
1B80-1BFF	Sprite Velocity List (128 bytes)
1C00-1C1F	Sound List (32 bytes)
1D00-1FFF	I/O Screen (768 bytes)
2000-37FF	Color Table (6K bytes)
3800-3FFF	Sprite Generator Table (2K bytes)

The Multicolor mode is similar to Pattern mode, except the Sprite Generator Table is located at address >1000, rather than at >0800. This mode also has an I/O Screen which is unique to the High-Resolution and Multicolor modes. The I/O Screen is the location where the results from the ACCEPT, DISPLAY, INPUT, LINPUT, and PRINT statements are placed. This causes the results from these statements to be invisible to the user.

Table 3-6 VDP RAM Usage for Multicolor Mode

Addresses	Use
0000-02FF	Screen (768 bytes)
0300-037F	Sprite Attribute List (128 bytes)
03C0-03DF	*Color Table (32 bytes)
03E0-03FF	Sound List (32 bytes)
0780-07FF	Sprite Velocity List (128 bytes)
0800-0FFF	**Character Generator Table (2K bytes)
1000-17FF	Sprite Generator Table (2K bytes)
1D00-1FFF	I/O Screen (768 bytes)

* The Color Table is not useful in this mode, but it is defined.

** The Character Generator Table is actually used for color information in this mode.

3.1.5 Physical Memory Usage.

The following figure is a graphic representation of how physical memory is partitioned for use by the BASIC interpreter.

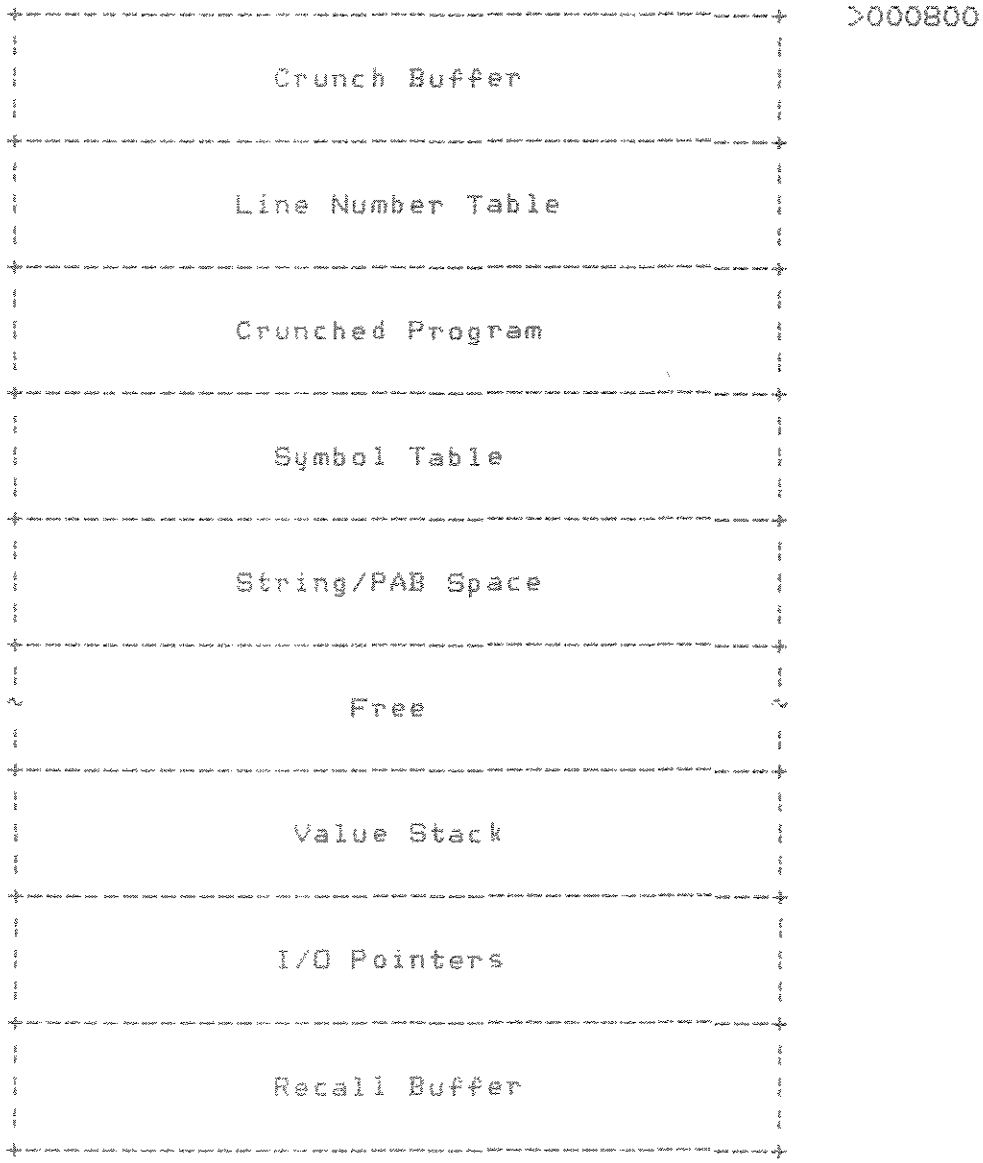


Figure 3-3 TI-99/8 Mapped Memory Usage

3.2 System Description

The TI-99/8 system has a 9995 processor with 64K of physical memory in the console and is expandable to 16 MEG of physical memory. There are two possible modes: /8 mode and /4A mode. Pascal uses the /8 mode and BASIC uses the /4A mode. This document only describes the /4A mode.

Since there are 16 address lines from the 9995 and 24 address lines to physical memory, there is a need for intermediary addressing logic. The Address Decoder and the Memory Mapper chip are the intermediary components for the TI-99/8. The following figure shows an overview of the system.

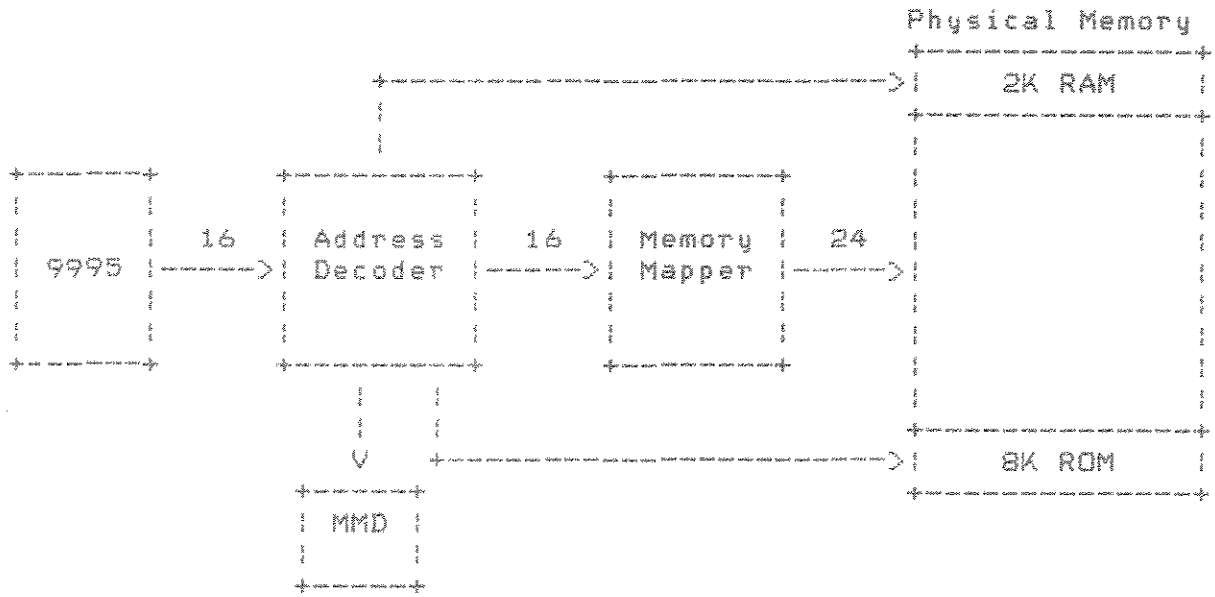


Figure 3-4 System Overview

Memory is generally described in two ways: Logical Address Space (LAS) and Physical Address Space (PAS). The LAS is merely an aid for the programmer and is a logical representation of how the PAS is being used. Valid logical addresses range from >0000 to >FFFF. The PAS is the actual memory which includes console memory and expanded memory. Valid physical addresses range from >000000 to >FFFFFF. The following figure describes the layout of physical memory.

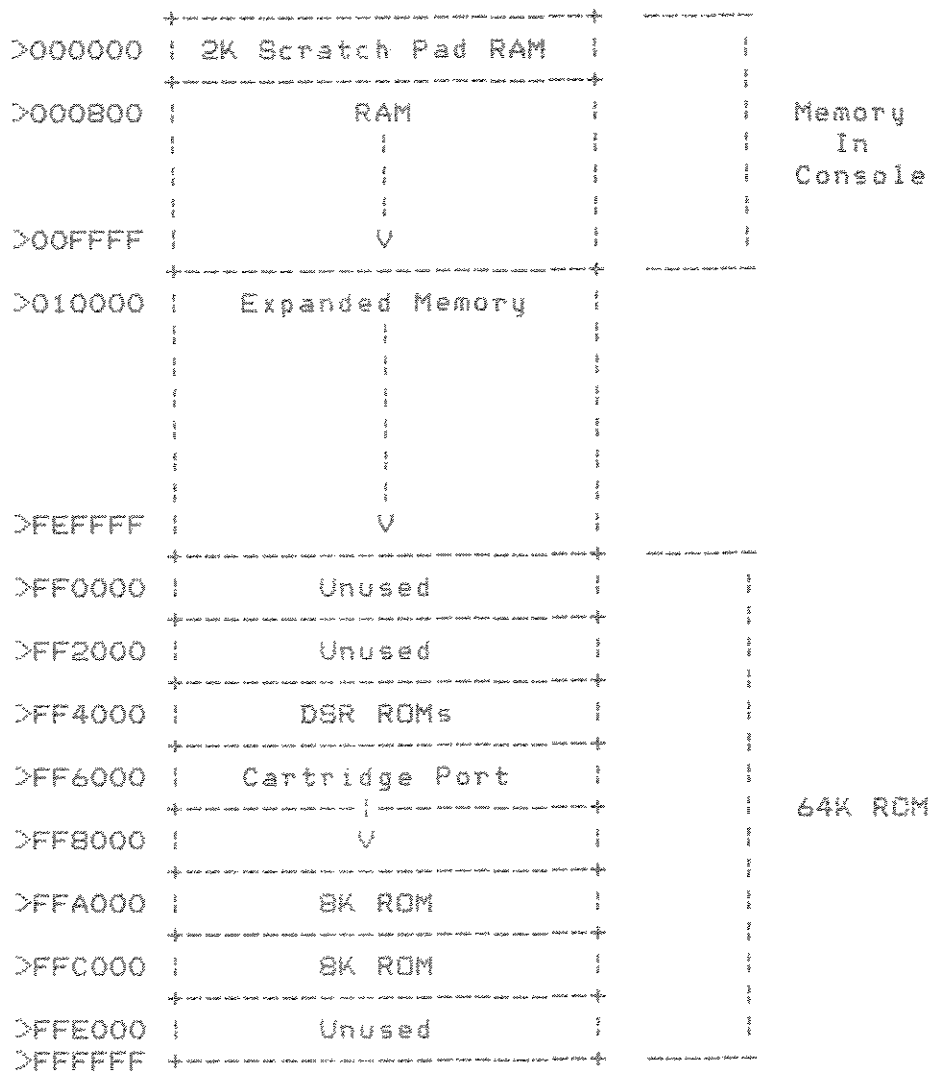


Figure 3-5 Physical Memory

3.2.1 The Address Decoder.

The Address Decoder takes logical addresses and routes them to the appropriate place. Logical addresses >0000 through >1FFF go to the system ROM. (The system ROM, or ROM 0, is not in the physical address space.) Logical addresses >8000 through >87FF are translated to physical address >000000 through >0007FF respectively, and sent to the 2K Scratch Pad RAM. Logical addresses >8800 through >9FFF are sent to the appropriate Memory

Mapped Device (MMD). All other logical addresses are sent to the Memory Mapper.

3.2.2 The Memory Mapper.

The Memory Mapper translates logical addresses from the Address Decoder into physical addresses. To do this, the Memory Mapper uses the value in the most significant nibble of the logical address as the address of one of its 16 registers. It then uses the value in the least significant 3 nibbles of the logical address as a displacement, and adds it to the contents of that register. The result is the desired physical address and is placed on the 24-bit address bus to physical memory.

3.2.3 Map Files.

In order to change the way PAS is partitioned, it is necessary to define a map file. A map file contains information which describes the 4K sections of PAS which are to be mapped into the LAS. A map file can only describe PAS sections which are 4K bytes in length.

There are eight map files numbered 0 to 7. Multiple map files may be defined, but only one map file at a time can be used to map in the PAS. The following table shows the logical address of each map file.

Table 3-7 Map File Locations

Map File	Logical Address
0	>8000
1	>8040
2	>8080
3	>80C0
4	>8100
5	>8140
6	>8180
7	>81C0

Each map file uses 64 bytes, and is divided into sixteen 2-word entries. Each entry of the current map file describes a "window" in the LAS. Information for each entry is stored as follows. The three most significant bits of the first byte are for the flags not used in the system. These three flags are the execute protect, the read protect, and the write protect flags. The remaining five bits of the first byte are unused. The

second, third, and fourth bytes (24 bits) store the physical address.

Only the first two map files, map file 0 and map file 1, are currently used by BASIC, but map file 2 and map file 3 are reserved for future use. The current definition does not reserve >8100 - >8200 for map files.

3.2.4 The Map Register.

Storing an appropriate value in the map register tells the Memory Mapper to LOAD or READ the current memory configuration described by the specified map file. This can be done as often as the programmer desires. The 8 bit map register is located at address >8810. The definition of each bit in the map register is described below.

U = Unused
A = Map File Number
L = 1 for LOAD a map file into the memory mapper
L = 0 for READ current memory mapper into a map file

Figure 3-6 Map Register Description

Specifying LOAD tells the memory mapper which 4K sections of PAS are to be mapped into the LAS. Specifying READ puts the sixteen 2-word physical address pointers of the currently mapped PAS into the specified map file.

The following table describes the results of storing various values in the map register.

Table 3-8 Map Register Contents

Map File	Map Register Contents		Meaning
0	0000 0000	or >00	READ Map File 0
0	0000 0001	or >01	LOAD Map File 0
1	0000 0010	or >02	READ Map File 1
1	0000 0011	or >03	LOAD Map File 1
2	0000 0100	or >04	READ Map File 2
2	0000 0101	or >05	LOAD Map File 2
3	0000 0110	or >06	READ Map File 3
3	0000 0111	or >07	LOAD Map File 3

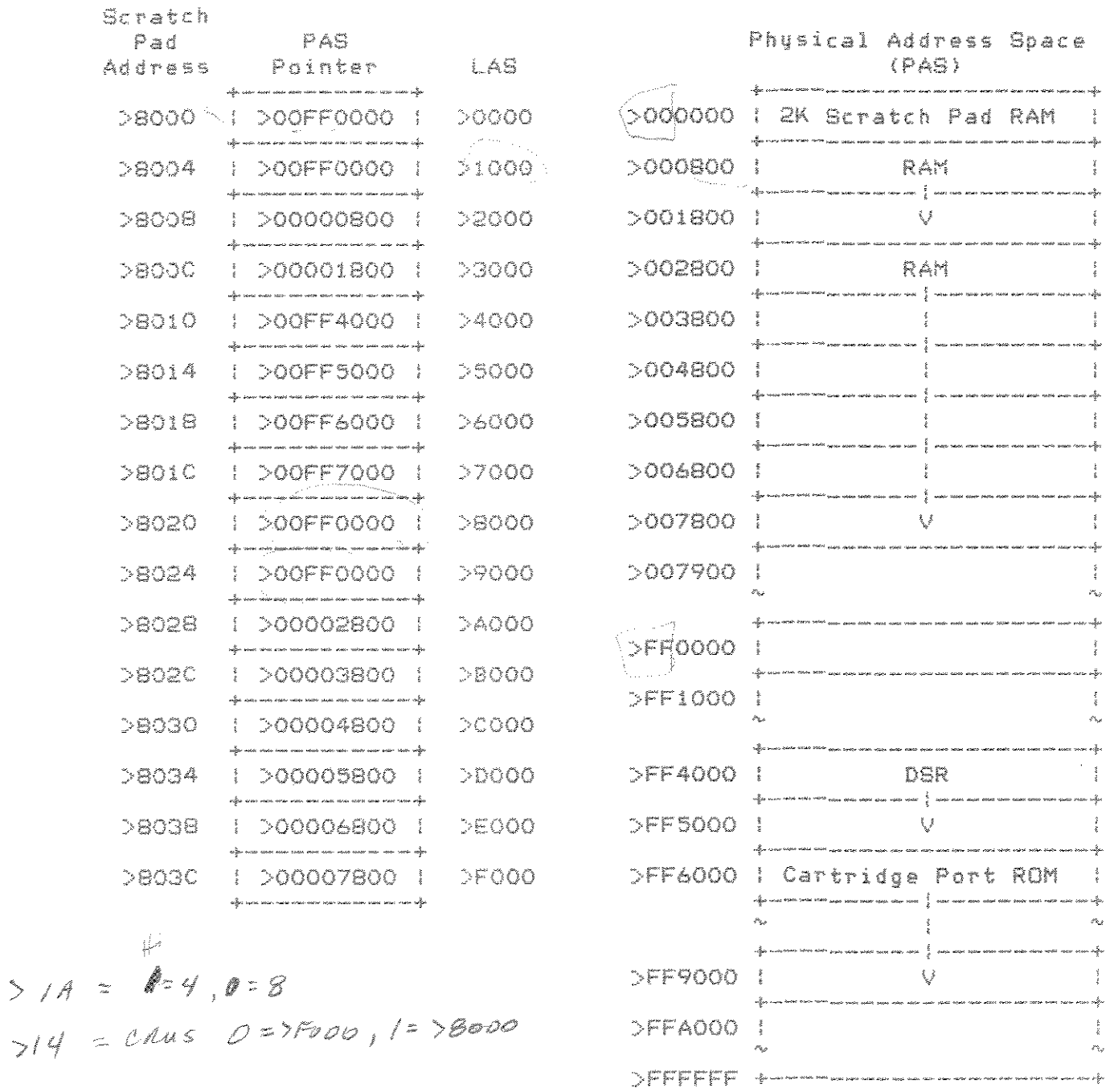
Examples of how to use the Map Register are described later in this section.

3.2.5 Memory.

The Logical Address Space is the only memory which is directly addressable by the processor. The LAS contains 16 "windows". Each "window" corresponds to a logical address as follows: Window 0 is address >0000, Window 1 is address >1000, Window 2 is address >2000, etc. Each "window" contains a physical address which points to the top of a 4K section of PAS and is defined by the current map file. So if the current map file is Map File 0, the contents of Window 0 corresponds to information stored at logical address >8000, and the contents of Window 1 corresponds to information stored at logical address >8004, etc. Note that the contents of Window 0, Window 1, Window 8, and Window 9 can not be changed.

In order to access physical memory, it is necessary to do memory mapping. In other words, it is necessary to map physical addresses into logical address windows.

Map file 0 starts at Scratch Pad address >8000. Each of the sixteen 2-word entries in the map file contain information about the PAS and correspond to part of the LAS. Figure 3-7 shows the contents of Map File 0, the corresponding logical addresses, and the layout of physical memory at power-up time.



>1A = #=4, 0=8

>14 = CRUS 0=>F000, 1=>8000

Figure 3-7 System Memory at Power-up Time

The following figure shows the contents of Map File 0, the corresponding logical addresses, and the layout of how the power-up code maps the PAS for BASIC.

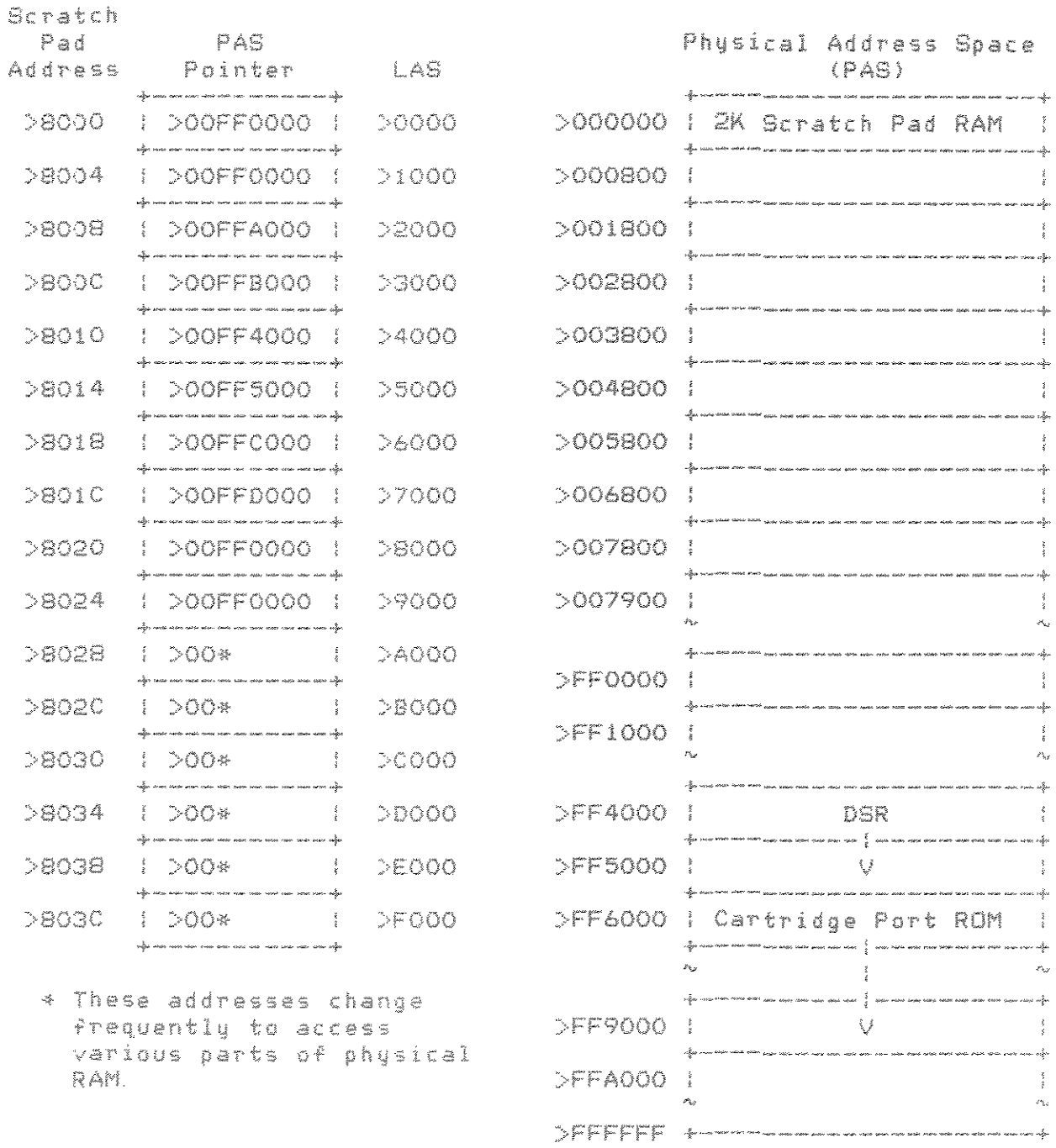


Figure 3-8 Memory Map of BASIC

The following paragraphs describe how to access the Memory Mapper from Assembly Language and from GPL.

3.2.6 Accessing the Memory Mapper From Assembly Language.

The following code assumes FAC through FAC+3 contain the 2-word physical address to be mapped in at window >D of map file 0.

```
WINDOW EQU  >8034              >8000+(4*>D)
LOADO  BYTE >01                00 (for MAP FILE 0) !! 1 (for LOAD)
MAPREG EQU  >8810              THE MAP REGISTER ADDRESS
*
       MOV  @FAC,@WINDOW       STORE 1ST HALF OF 2-WORD PHYSICAL
*                              ADDRESS TO MAP IN MAP FILE
       MOV  @FAC+2,@WINDOW+2   STORE 2ND HALF OF 2-WORD PHYSICAL
*                               ADDRESS TO MAP IN MAP FILE
       MOVB @LOADO,@MAPREG     SEND COMMAND TO LOAD MAP FILE 0
```

The two MOV instructions need to be repeated for each physical address to be mapped in. The MOVB instruction only needs to be executed once, since it will load (or read) the entire specified map file.

The following XOP 3 instruction accomplishes the same procedure. It is slower than directly accessing the mapper because it is a subroutine call, but it saves program steps. For the example as set up above:

```
WINDOW EQU  >8034       >8000+(4*>D)

       XOP  @FAC,3      FAC CONTAINS A 2-WORD ADDRESS TO MAP IN
       DATA WINDOW      PUT MAP FILE ADDRESS IN DATA STATEMENT
```

The XOP 3 instruction needs to be repeated for each physical address to be mapped in. The XOP 3 instruction automatically loads the entire map file 0, so this instruction is very slow when several physical addresses need to be mapped in.

It is not necessary to define every window with MOV instructions. The following example shows how to read the current description of physical memory into map file 2. It is then possible to change any windows and then to load the new map file.

```
READ2  BYTE >04                10 (for MAP FILE 2) !! 0 (for READ)
LOAD2  BYTE >05                10 (for MAP FILE 2) !! 1 (for LOAD)
MAPREG EQU  >8810              THE MAP REGISTER ADDRESS

       MOVB @READ2,@MAPREG     SEND COMMAND TO READ CURRENT PHYSICAL
```

```
*                                   MEMORY POINTERS INTO MAP FILE 2
*       Change any windows.
*
        MOVB @LOAD2,@MAPREG     SEND COMMAND TO LOAD MAP FILE 2
```

3.2.7 Accessing the Memory Mapper From GPL.

The following paragraphs describe GPL instructions which are used in memory mapping.

3.2.7.1 MAPIN.

The MAPIN routine fetches the physical address pointer from a Scratch PAD RAM address (>8300 - >87FF), and maps this address in at the top of the 4K logical address space window specified by the WINDOW parameter. The format for this instruction is:

```
DATA   XML2,MAPIN
DATA   #PHYADD,WINDOW
```

In MAPIN, PHYADD (Physical Address) is a 1-word pointer to the Scratch Pad RAM location containing a 2-word physical address. WINDOW is 1-byte parameter and contains the value of the desired window. (Recall that windows have values from >0 through >F.)

An example for this instruction is:

```
DATA   XML2,MAPIN
DATA   #FAC+4,>E
```

This example moves the physical address at FAC+4 through FAC+7 to Window E of map file 0.

3.2.7.2 GETP.

The GETP routine will fetch data from a physical address to a buffer in Scratch Pad RAM (>8300 - >87FF). The format for this instruction is:

```
DATA   XML2,GETP
DATA   #SRCPTR,#LOGDEST,BYTCNT
```

In GETP, SRCPTR (Source Pointer) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer to data located in physical memory. LOGDEST (Logical Destination) is a 1-word address of where in Scratch Pad RAM to put the data. BYTCNT is a 1-byte parameter which tells how many bytes of data to read.

An example for this instruction is:

```
DATA  XML2,GETP
DATA  #DATA,#CHAT,1
```

This example moves 1 byte from the physical address pointed to by DATA to CHAT.

3.2.7.3 PUTP.

The PUTP routine will fetch data from a Scratch Pad RAM address to a physical address. The format for this instruction is:

```
DATA  XML2,PUTP
DATA  #SRCPTR,#PHYDEST,BYTCNT
```

In PUTP, SRCPTR (Source Pointer) is a 1-word address of where in Scratch Pad RAM the data is located. PHYDEST (Physical Destination) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer of where in physical memory to put the data. BYTCNT is a 1-byte parameter which tells how many bytes of data to read.

An example for this instruction is:

```
DATA  XML2,PUTP
DATA  #FAC,#ARG,4
```

This example moves 4 bytes from FAC to the physical address pointed to by ARG through ARG+4.

3.2.7.4 MOVUP.

The MOVUP routine will move the contents of low physical memory to high physical memory. The format for this instruction is:

```
DATA  XML2,MOVUP
DATA  #SRCPTR,#PHYDEST,#LENGTH
```

In MOVUP, SRCPTR (Source Pointer) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer to the data located in physical memory. PHYDEST (Physical Destination) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer of the address of where in physical memory to put the data. Since MOVUP is a non-destructive move, it is necessary for SRCPTR to point to the high-end address of the buffer that is to be moved. Similarly, PHYDEST must point to the high-end address of the desired buffer location. LENGTH is a 1-word address of a length

variable.

An example for this instruction is:

```
DATA  XML2,MOVUP
DATA  #OLDBOTTM,#NEWBOTTM,#FAC
```

This example moves FAC number of bytes from the physical address pointed to by OLDBOTTM to the physical address pointed to by NEWBOTTM.

3.2.7.5 MOVDN.

The MOVDN routine will move the contents of high physical memory to low physical memory. The format for this instruction is:

```
DATA  XML2,MOVDN
DATA  #SRCPTR,#PHYDEST,#LENGTH
```

In MOVDN, SRCPTR (Source Pointer) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer to the data located in physical memory. PHYDEST (Physical Destination) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer of the address of where in physical memory to put the data. Since MOVDN is a non-destructive move, it is necessary for SRCPTR to point to the low-end address of the buffer that is to be moved. Similarly, PHYDEST must point to the low-end address of the desired buffer location. LENGTH is a 1-word address of a length variable.

An example for this instruction is:

```
DATA  XML2,MOVDN
DATA  #OLDTOP,#NEWTOP,#FAC
```

This example moves FAC number of bytes from the physical address pointed to by OLDTOP to the physical address pointed to by NEWTOP.