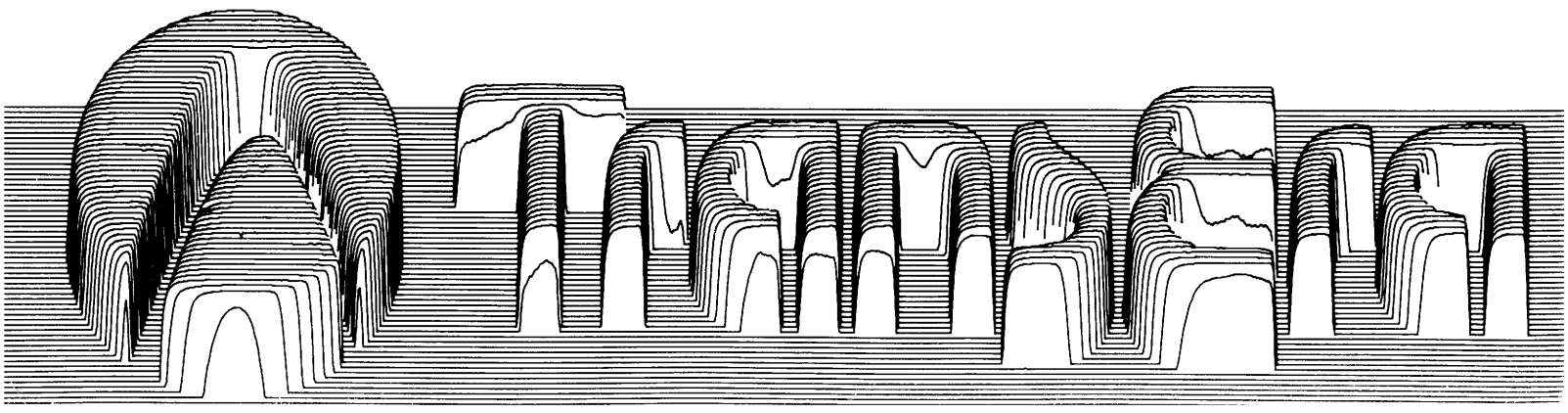
TransEra

MODEL 6400
AUXILIARY MEMORY
REFERENCE MANUAL



MODEL 6400
AUXILIARY MEMORY
REFERENCE MANUAL

TransEra Corporation
3707 North Canyon Rd.
Provo, Utah 84604
Tel: 801-224-6550

Manual Part No. 070-6400-03

Printed in the United States of America
Revision 03, November 1983

NOTICE

Auxiliary Memory
Reference Manual
070-6400-03

Revision History:

Original Release	-	March 1979
First Revision	-	September 1980
Second Revision	-	June 1981
Third Revision	-	November 1983

PREFACE

This manual describes the TransEra 6400 Series Auxiliary Memory for use with the 4050 Tektronix Graphic Computer Systems. This release describes firmware level 5 of the Auxiliary Memory File Manager ROM. The manual includes installation instructions, starting tutorial, command and error message descriptions. It also includes a copy of the current firmware Release Notes. These Release Notes contain information that may not be found in the manual.

Section 1 provides a brief introduction and overview of the system including installation procedures for first time set up. Instructions are also provided for adding field installable memory cards, firmware upgrades, and the battery back-up option.

Section 2 provides an overview of the Auxiliary Memory file identifiers, file structures, and file types. It also includes is a getting started tutorial for helping first time users become familiar with the Auxiliary Memory file system.

Section 3 lists all Auxiliary Memory File Manager commands, statements, and functions alphabetically. Each is explained in detail and illustrated by an example. This includes all program and data file I/O, and general file maintenance commands. Also included are special purpose and utility routines.

Section 4 describes the DMA (Direct Memory Access) options to the Auxiliary Memory. The operation and installation details are included here along with descriptions of the routines used to operate the DMA.

Section 5 describes the Auxiliary Memory architecture, some advanced programing techniques, and the use of the primitive read/write commands used for data acquisition purposes.

Appendix A lists a summary of all the commands. Appendix B lists the error messages with causes and corrective actions. Appendix C describes the ROM Pack slot priority. Appendix D describes the hardware specifications. Appendix E is a glossary of terms. Appendix F contains a copy of the current Firmware Release Notes.

PLEASE NOTE: Conventions And Formats Used In This Manual

Most of the commands described in this manual are CALL routines. Some are BASIC key words. The BASIC key words in general can be abbreviated to 3 characters. The CALL names may be abbreviated to 6 characters. Any names shorter than this must be spelled out.

The syntax of BASIC key words is checked at time of entry. CALL statements are not checked beyond the CALL key word and the required name string. For this reason, you must exercise greater care when entering CALL statements both as to spelling of names, and listing of arguments.

Some command arguments are optional and some are required. Sometimes you have your choice of a required argument. In this case the choice might be listed in braces {}. For example: {a$,x} means you have your choice of **a$** or **x**, but you must choose one. Most of the time, however, the formats will be listed separately when there is a choice of required arguments.

The format used in this manual to designate optional arguments is to enclose them in brackets []. When you see this, don't enter the brackets, they only set off what's optional. Some optional arguments depend on the order of other arguments in the list.

For example: x,y[,a$],a[,b[,c]] is an argument list that shows both nesting and variable types to determine what optional arguments may be omitted. In this case **b** cannot be omitted unless **c** is also omitted.

However, **a$** can be omitted regardless of the other arguments. The rule is; you must eliminate everything within a 'pair' of brackets. Note that **a$** can be distinguished from the other arguments because it is a string.

Sometimes the semi-colon is used in argument lists to flag the presence of an optional argument. When you see this, you MUST include the semicolon as part of the syntax. Example: f,[r;]d1[,d2...] indicates that **r** is optional only if it is followed by a semicolon. Note also that this requires another argument to follow since a CALL statement cannot end with a semicolon. In all other cases, do not use semicolons in place of commas unless they are required.

Throughout this manual you will see arguments listed in lower case and some in upper case. Lower case means the argument is used to pass information TO the routine and may be a literal, an expression, or a variable. Upper case means the argument is for passing information back FROM the routine and it must, therefore, be a variable only.

A simple variable is a scalar, versus a dimensioned array or vector or array element. Numeric expressions are referred to throughout this manual and they are intended to include simple variables, and literals. In general, string arguments may be literals or variables. String expressions are not allowed as command arguments.

When you see three dots '...' in an argument list, this means that you may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

There is one special case in argument formats that is used in most of the commands throughout this manual. This is the argument for specifying file identifiers. It is listed as 'fi' in the command argument lists. Although this is not a valid variable format, it is used to denote the fact that in general, files may be referenced by number or name.

This format avoids using braces {f$,f} or listing the the two argument formats separately for every command to identify the two choices. Make sure you understand this designation.

End of Preface

Table of Contents

Section 1 - Introduction

Section 2 - Getting Started

Section 3 - Statements, Commands, and Functions

Section 3 - Statements, Commands, and Functions (continued)

Section 1

System Overview and Installation

About This Section

This section provides a brief introduction and overview of the system including installation procedures for first time set up. Instructions are also provided for adding field installable memory cards, the Battery Back-up Option 15, DMA Options 11 and 12, and firmware upgrades.

The installation guide should be followed the first time you set up the equipment and any time it is de-installed for purposes of moving.

Auxiliary Memory Overview

The TransEra Auxiliary Memory module is a high speed, random access storage system that is functionaly equivalent to a solid state disk. Both program and data files are allowed. Files created in binary format result in the fastest possible transfer rates. Data files may be either Random or Sequential.

Large application programs may be divided into overlay segments and saved in separate files in the Auxiliary Memory. These program overlays may then be rapidly accessed and linked or appended into user memory as they are needed.

The Auxiliary memory also allows DMA (Direct Memory Access) from certain peripheral devices such as the A/D converter. Data may also be stored or retrieved by the user in an unformatted mode that ignores the file structure. This is useful for certain data acquisition applications or for special diagnostics.

The Auxiliary Memory is available in various memory sizes from 128K bytes to 1024K bytes. It is field upgradeable in increments of 128K bytes with plug-in cards.

Equipment supplied with each Auxilary Memory System:

1) 6400 Memory Box with 8 memory slots

2) 1-8 Memory Cards

3) Memory File Manager ROM Pack

4) Ribbon Cable Interconnect (34 Conductor) with Connectors

5) Power Cord

6) Reference Manual

6400 Auxiliary Memory Installation Instructions

1. If it is not already connected, the 34 pin D-type connector on the end of the ribbon cable must be connected to the back of the memory box.

2. The power cord to the Auxiliary Memory may be attached to the receptacle at the back of the Memory Box. The memory can operate from either a 120V 60Hz or a 240V 50Hz power source. Make sure the proper power is applied to the memory or it will not operate properly and may cause damage. A small PC card located immediately below the fuse may be removed and oriented in another direction to change the voltage setting.

3. After switching the 4050 system power OFF, the ROM pack file manager/interface may be plugged into an available slot in the firmware back-pack. The ROM Pack must be securely seated in the receptacle connector.

   NOTE: If a ROM expander unit is being used, it is best to try and locate the memory interface ROM pack in the other back-pack slot rather than in the ROM expander, as this will minimize data transmission problems. Although the memory may work fine from the ROM expander in most instances, there have been reported problems, especially with 4054's where the internal cabling is inherantly longer.

4. The auxiliary memory box may now be placed at any convenient location within reach of the cable such as on top of the 4050 cabinet so long as the cable is positioned such that it will receive a minimum of mechanical disturbance. It is also best to avoid close proximity with any severe sources of electrical noise.

5. The 4050 system power may now be applied.

Field Installation of Memory Cards

1. Disconnect all power cords and cables.

2. To remove the chassis cover on units with serial numbers 2000 and above, remove the screw at the back of the memory box, slide the cover gently towards back of the memory box and pull up, then skip to step 8. For serial numbers below 2000, follow steps 3-7.

3. With the memory box resting on its top (on a table or work surface), remove the four screws that are closest to the rubber feet on the bottom of the Memory box. (See figure 1.)

4. Lift bottom cover upward to remove.

5. Remove the four (two each side) 1/4" nuts from the chassis. (See figure 2.)

6. Lift chassis upward (lift one side then the other) to remove from top cover.

7. Turn chassis over so the inside circuit boards are showing.

8. Locate an empty card slot next to an existing memory card. The new memory boards must be installed in order sequentially without skipping any slots. Note that the bottom bus board is labeled by each connector, MEMORY CARD 0 to MEMORY CARD 7.

   Slide the new RAM board(s) into the black card guides and align the pins on the connector pressing down firmly, but slowly until the pins are seated.

9. You may at this point wish to test the memory to make sure all the connections were properly made before re-assembling. You can do this by attatching the power cord making sure that everything is well insulated and doing a CALL "MCHECK". If MCHECK does not return the correct memory size, then double check the seating and position on the pins to connectors on all boards.

10. Reverse the dis-assembly steps above to re-assemble making sure the vent holes in the top and bottom cover are toward the back of the module.

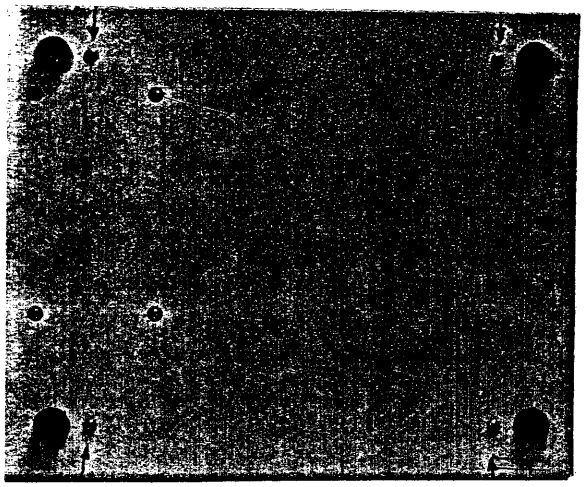


Figure 1. Bottom of Memory Module

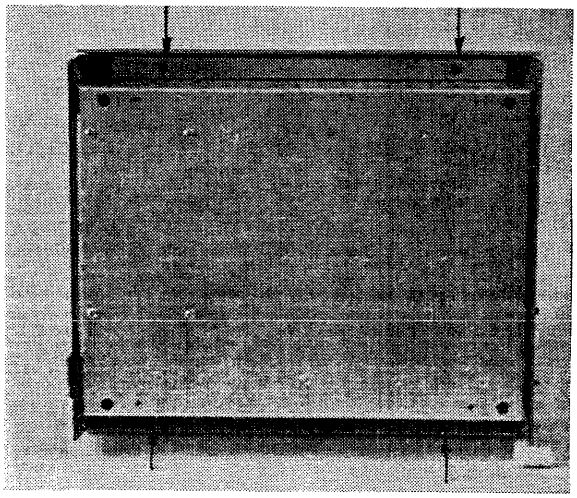


Figure 2. Chassis with Bottom Removed

Option 15 - Battery Back-up Description

The Battery Back-up provides power to the Memory during brief power interruptions. Minimum support time for a fully configured memory is 50 minutes. This allows the user to save programs and data during short power outages. The Battery Back-up has five 1.2 volt batterys in series for a total charged voltage of between 6.8 to 7.5 volts. When a black-out or brown-out occurs, the battery takes over and provides power to the Memory.

If the power fails to come back on within the time the batteries are able to sustain the installed memory, the battery back-up circuit will shut itself off to protect the batteries from completely draining. When the power is finally restored, the charging circuit senses the power increase and starts charging the batteries for the next use of the batteries.

Operating Instructions

It is recommended that once the Option 15 Battery Back-up is installed in the 6400 Memory, that it be powered at all times to ensure that the battery is fully charged in the event of a power failure. This also allows you to have programs or data accessible at any time.

Specifications

- Minimum Hold Time For Fully Configured Memory: 50 minutes
- Maintenance-Free: The batteries have a sealed construction, contain no free electrolyte, and require no service or maintenance except recharging.
- Overcharge Protection: Nickel-cadmium cells can be continuously overcharged, without noticeably affecting life.
- Cycle Life: 300 to 1000 cycles of discharge, or 5 years of standby power is common to NICAD cells.
- High-Rate Charging and Discharging: Cells can be charged in 3 to 6 hours minimum time and 14 hours maximum.

Field Installation of Battery Pack Option 15

Photo #1 shows the inside of the TransEra memory with cover removed and without a battery pack.

Step 1. Remove screw No. 1 holding red wire (+DC) and screw No. 2 holding green wire (–DC). (See separate instructions for removal of memory cover).

Photo #2 shows battery pack partially in place with red wire No. 1 protruding at right and green wire No. 2 protruding at left.

Step 2. Insert battery pack as shown with red and green wires protruding toward the rear of the memory case.

Step 3. Fasten these two wires where the screws were removed by placing them over the original two wires and replacing the screws (red to red and green to green).

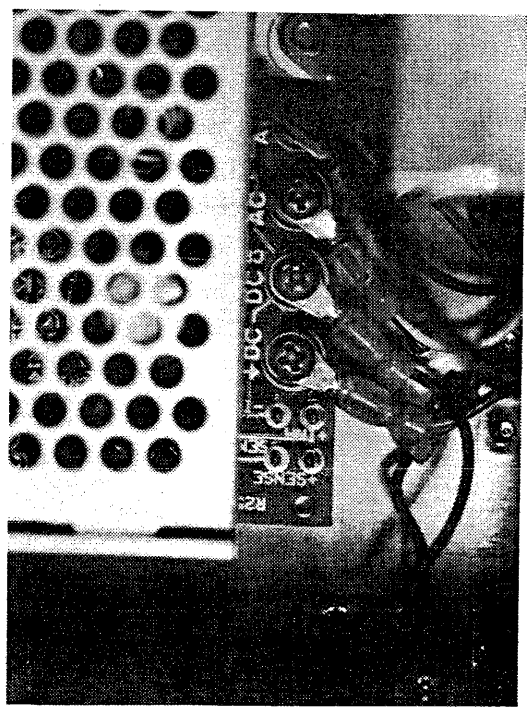


Photo #1

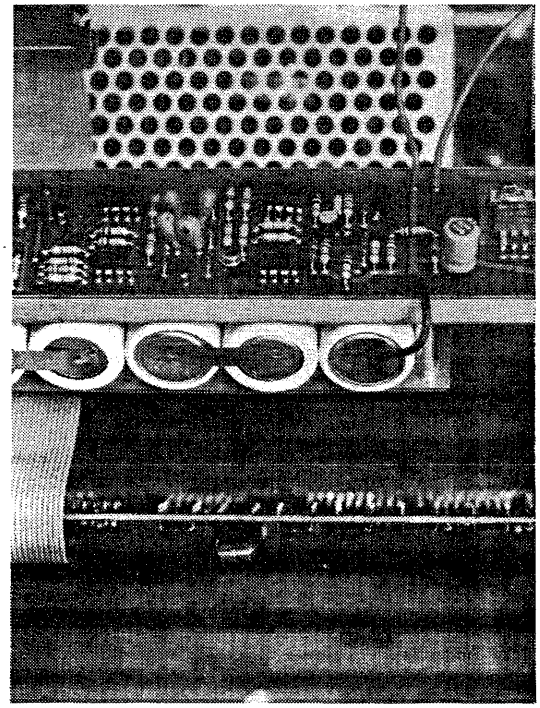


Photo #2

Photo #3 shows battery pack in place with wires attached. The Battery pack should be down as close as possible to perforated power supply housing.

Photo #4 shows screws through side of memory case holding battery pack.

Step 4. Place four screws (two on each side) through slots in memory case to screw bolts in memory pack to hold battery pack firmly in place.

Step 5. Insert fuse into Battery pack fuse holder (thus enabling the battery to power up the memory). Test battery backup operation by plugging the power cord in. If the LED (on Printed Circuit Board) comes on when the memory is plugged in, then the Battery pack is being charged.

Step 6. Reassemble case (see instructions) and memory is ready for operation.

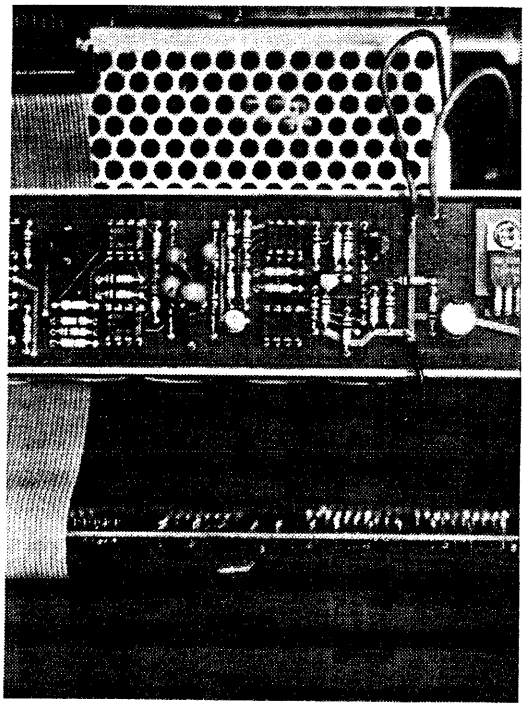


Photo #3

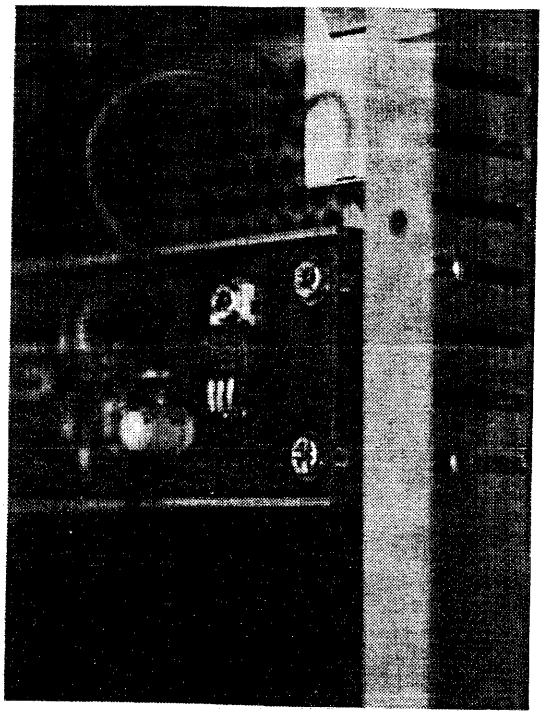


Photo #4

Options 11 and 12 - DMA Board Installation Instructions

1. Disconnect all power cords and cables.

2. To remove the chassis cover on units with serial numbers 2000 and above, remove the screw at the back of the memory box, slide the cover gently towards back of the memory box and pull up, then skip to step 8. For serial numbers below 2000, follow steps 3-7.

3. With the memory box resting on its top (on a table or work surface), remove the four screws that are closest to the rubber feet on the bottom of the Memory box. (See figure 1.)

4. Lift bottom cover upward to remove.

5. Remove the four (two each side) 1/4" nuts from the chassis. (See figure 2.)

6. Lift chassis upward (lift one side then the other) to remove from top cover.

7. Turn chassis over so the inside circuit boards are showing.

8. Remove the cover plate on back of memory for the DMA connector and install the DMA connector.

9. Install the DMA card, in any of the three 564 BUS slots.

10. You may at this point wish to test the memory to make sure all the connections were properly made before re-assembling. You can do this by attatching the power cord making sure that everything is well insulated and doing a CALL "MCHECK". If MCHECK does not return the correct memory size, then double check the seating and position on the pins to connectors on all boards.

11. After memory is functioning properly, test DMA (see Section 4).

12. Reverse the dis-assembly steps above to re-assemble making sure the vent holes in the top and bottom cover are toward the back of the module.

Refer to Section 4 for operating instructions for the DMA options.

Installing Firmware Upgrades

Before attempting to upgrade your ROM pack, first read the the Firmware Release Notes. All changes are documented only in the release notes. This will inform you of any changes that will affect your data, programs or mode of operation. Then proceed with the following steps:

1. Remove the 4 screws and the ROM pack lid.

2. Lift the printed circuit board from the plastic box and hold the board in front of you, component side up, with the gold printed-circuit connector fingers at the bottom.

3. Carefully remove the old EPROM(s) by prying up from the edges. A small screw driver works well.

4. Insert the new EPROM(s) into the socket(s). Please observe the proper orientation, and pin locations. The EPROM is inserted into its socket with the small notch in the EPROM case pointed to the left as you hold the ROM pack board upright with the gold printed circuit board connector fingers. at the bottom.

   If you are supplied with more than one EPROM, they should be marked with Roman Numerals. The EPROM socket nearest the bottom is for the EPROM marked with Roman numeral I. The next socket location up is for number II, and so on.

   Note: You may have to apply considerable pressure to seat the chip and it may snap into position. Do not be afraid of doing this so long as you have carefully lined up all the pins with the socket and you are applying even pressure. Also, make sure all the pins on the chip are straight and not bent over from shipment or handling before pressing the EPROM into position.

5. Replace the circuit board into the plastic case and line up the screw holes.

6. Put the ROM pack lid back on and fasten it with the four screws.

After you have the new firmware working and have had a chance to test your programs, please return the old EPROM(s) at your earliest convenience.

Section 2

Getting Started

About This Section

This section provides an overview of the Auxiliary Memory file structures, file names, and command usage. It also includes is a getting started tutorial for helping first time users become familiar with the Auxiliary Memory file system.

You should read the following descriptions of the file identifier, and the file structures used including the various file types from program to random and sequential data files. This will help you to understand the rest of the manual and generally improve your ability to program efficiently by adding to your overall understanding of the memory system.

File Identifers

File Identifiers consist of a file number and an optional file name. Before explaining file names, it must be understood that each file that is created has a number assigned to it. It is ultimately by this file number that the file is accessed. Unlike some other file management systems, this arrangement precludes many of the formalities of creating, opening, and closing files. There is no maximum number of files that can be open for access at any one time. All files are inherantly open.

Both program and data files can be created 'automatically' upon the first output reference, without formally using the CREATE command. Files may, however, be created with optional characteristics when the defaults are not applicable. This gives you the flexiblity when you need it without imposing a mandatory complexity when you don't.

File names are purely optional and may be used on selected files or on all files. They may be used strictly for labeling purposes for viewing in the directory listings, or they may be used to actually reference the files in the various commands. Since files have inherantly assigned numbers, they are the primary means of referencing files. They offer the greatest speed and efficiency in program execution since the file manager knows instantly where to find a file referenced by number.

The file name, therefore, is mainly for convenience purposes since names are usually easier to remember than numbers. File names may be from 1 to 28 characters in length. The file manager is very permissive, allowing almost any valid characters in names and retaining upper and lower case. Duplicate file names are also allowed if they are used only for labeling purposes.

The user is therefore responsible for avoiding duplicate names (if he intends to reference those files by name), and for choosing appropriate characters for names, remembering to enter the proper case (upper/lower) when referencing files. If duplicate file names are assigned and later referenced, only the first occurence or lowest numbered file will be found.

Routines are provided for finding file numbers based on assigned names (see MOPEN), and for finding the next lowest file number available (see NXTFIL). It is best to choose file numbers for new files that are as low as possible in order to conserve memory and speed access. Refer to the next discussion on File Structures to better understand why this is so.

Also, refer to the Preface under the title, "Conventions And Formats Used In This Manual", for a description of the format and syntax used to describe file identifiers throughout this manual.

File Structures

The Auxiliary Memory allocates memory in chunks of 256 bytes. This is referred to as a block. Upon power-up, or after a CALL "INIT" is executed, the file manager will reserve a few blocks of memory for its use. The first two blocks (512 bytes) are reserved for the directory. The first 16 bytes of the directory are actually reserved for file manager system information.

Each directory entry consists of a two byte pointer which points to the first block assigned to a new file when it is created. This means that there is enough reserved directory space for 248 files. If more files than this are created, or if file numbers higher than this are used, then another block must be allocated to the directory.

Following the two reserved directory blocks, there is a link map which contains a two byte pointer for every block in memory. This means that 4 blocks of link map are required for every 128K bytes of memory installed. The amount of memory is determined on power-up and the appropriate number of blocks are reserved for the directory and link map respectively.

Since it is possible to select file numbers in any order and you are not prevented from choosing arbitrarily high numbers, you must understand the consequences. There is not too much of a speed penalty in referencing high file numbers since the random access memory is inherhantly fast.

However, unnecessarily high numbers will cause additional directory memory to be allocated. Also, you may notice slight delays in directory listings since every directory entry must be checked between the first file and highest number. If you had two files, 1 and 1000, then the directory lister would be forced to scan 998 empty directory entries.

There are three basic file types in the Auxiliary Memory. One is a program file, and the other two are data files, random, and sequential.

Each of these files has a file header of 44 bytes. This inlcudes space for a 28 character file name. Other bytes are used for file type information, record size, current item pointer, and last item pointer. For more details on this and general memory architecture, refer to Section 5 of this manual.

Program Files

Programs are saved in a binary format. This is a partially compiled format which the 4050 uses as its internal format for programs. This is why loading and storing programs in this format is much faster than the ASCII format which must be fully interpreted and broken down to the internal binary structure. All that is necessary in transferring binary programs is to convert variable names to table pointers or vice-versa depending on the direction of transfer.

When a program is saved, it is written to the first block of memory assigned to the file starting after the 44 byte header. When that

block is filled, another block is requested and chained to the preceding block through the link map. In this way the file dynamically expands using as many blocks as are required. When a program is re-saved, its file is implicitly deleted, releasing all blocks assigned to it. Therefore, if the program is any smaller, it will only take as many blocks as required each time it is saved.

Random Files

Random files are the most flexible data file types and are required by many of the utility or special purpose routines. It is therefore the default file type created when writing to a previoulsy un-referenced file.

The random file is assigned a specific record size. This is the space allocated for each individual data item. This is different from some file systems which allow more than one data item per record. By allowing only one data item per record, numeric and string arrays become very easy to manage. You are allowed to perform array reads and writes and the item pointer will automatically move through the file.

Since each record has a fixed length, they may be randomly accessed very rapidly by computing their position. Also, selected items may be overwritten, moved, or sorted without affecting neighboring records.

Random files can be created with any record size from 1 to 255. Every record in a particular file will have the same length. The default record size for numerics is 8 bytes. This is the space required by the 4050 system representation for full precision floating point numbers. The default for strings is 72 bytes. The type of data written in a record is not saved with each data item. You must keep track of what was written and where in order to properly read it back.

Strings are delimited by a zero byte, or by the end of the record. Numerics are always 8 bytes or less. In larger record sizes, only the first 8 bytes are read into target numeric variables. In records smaller than 8 bytes, a reduced precision format is used with 1-3 byte records being integer format.

Sequential Files

The sequential files main virtue is more efficient memory usage when variable lengthed strings are used. Each data item begins where the previous item left off, with no wasted space in between. Each item in a sequential file starts with two bytes that contain its length. When an item in a sequential file is randomly accessed, its position is found by starting with the first item and chaining forward using the length pointers until the item addressed is found.

Because a current item pointer position is maintained, this process can be slightly more efficient if the referenced item number is greater than the last referenced position represented by the current item pointer. In this case, the forward chaining process can be relative to this position rather than the file's first item.

This item location process is transparent to the user and the same read/write commands are used in exactly the same way to access both random and sequential files. In sequential files, however, you cannot write to the middle of the file without forcing a new end of file marker. This is because the variable length fields may cause one item to write over its neighbor, or fall short of the next chain (length) pointer.

Getting Started Tutorial

This section gives a brief overview of some of the more common Memory File Manager commands. It gives examples and shows the file manager's response. It is recommended reading if you are unfamiliar with the 6400 Memory File Manager.

Let's start with a simple program that accesses the memory and that we can save to the memory as well. Assume that we have just powered up first the Auxiliary Memory, and then the 4050. Then enter the following:

```
100 CALL "WRITE",1,1,2,3,4,5,6,7,8,9,10
110 CALL "SETIP",1,1
120 FOR X=1 TO 10
130 CALL "READ",1,D
140 PRINT D;
150 NEXT X
160 END
```

Now run the program and type the following statements:

```
RUN

1 2 3 4 5 6 7 8 9 10

CALL "MSAVE",10
CALL "DIR"

 No.      Name                    Type         Rec Siz    Size

   1      Noname                  Random       8          124
  10      Noname                  Program      -          514

127234 Bytes Free

DELETE ALL
CALL "MOLD",10
RUN

1 2 3 4 5 6 7 8 9 10
```

```
LIST

100 CALL "WRITE",1,1,2,3,4,5,6,7,8,9,10
110 CALL "SETIP",1,1
120 FOR X=1 TO 10
130 CALL "READ",1,D
140 PRINT D;
150 NEXT X
160 END
```

Now let's examine what we've done. The first line of this program is a WRITE statement. Note that we did not have to formally create a data file. The first argument in the list is a 1. This is the file number. The remaining arguments are numbers from 1-10. These are data items to be written to file 1, which is automatically created as a random file with a record length of 8, and the 10 data items are written to this file when line 100 is executed.

After running the program we can see that the data items were read back out of the file and printed to the screen with the READ statement after having reset the item pointer to 1 in line 110.

We next saved the file using the MSAVE command in file number 10 as the file number. These two files are then listed by the directory command. To convince ourselves that we have properly saved the program, we next delete the program and data from memory and load it back in with the MOLD command. We then run the program and list it back out noting that everything is the same.

One thing is different, however. In running the program the first time, the current item pointer was left at position 11 after reading the 10 items. So, in running the program a second time after loading it back, that position was retained and 10 more items were written to file starting at 11 for a total of twenty items.

Before reading begins the second time, the item pointer was set back to 1. This means that if we run this program a few more times, it will not add 10 items each time but will continue to over-write the second 10. This is because each reading loop leaves the item pointer at position 11.

You may want to insert CALL "GETIP" and CALL "LSTIP" after each access to the memory in the above program to see the what happens to the current item pointer and last item pointer positions after each operation. If these statements are entered without arguments the pointer positions will be printed to the screen.

Now let's try some arrays to check out the speed and storage capability of the memory by entering the following;

```
DELETE ALL
CALL "INIT"
DIM A(3000)
A=1
CALL "WRITE",1,A,A,A,A
```

You have just transferred 96,000 bytes to the memory. This is more than can be resident in user memory at one time. Did you notice how fast this happened? Now let's read it back and verify the contents, again noting the speed.

```
DELETE A
DIM A(3500)
CALL "READ",1,1;A
SUM(A)

 3500
```

This shows that correct values were read back. Note that the item pointer was set to record position 1 in the READ routine by the value of 1 following the following the file number and delimited by the semicolon. Now check the current and last item pointers.

```
CALL "LSTIP"
 12000
CALL "GETIP"
 3501
```

The last item pointer reflects the total number of items written to the file from the four 3000 element arrays. The current item pointer position is the result of having read 3500 elements from the beginning and then being ready to read the next location.

You should now be able to do quite a lot both in saving and olding programs and in reading and writing data files. But before we quit, let's try one thing more that uses some specialty routines and some of what we already know. This will be a random number plot.

```
DELETE ALL
CALL "INIT"
DIM A(1000)
A=-1
A=RND(A)
CALL "WRITE",1,A
CALL "SCALE",1,100,3
CALL "MPLOT",1
```

This should produce a random line plot that fills most of the screen. The SCALE routine uses the mulitply code of 3 and a value of 100 to scale the random data in file 1 up to the window size. The MPLOT routine then plots this data to the screen using two numbers at a time as a pair of x,y coordinates directly from the file.

You should now feel somewhat comfortable in working with the Auxiliary Memory file system and be anxious to learn some of the more advanced features awaiting you on the following pages.

Section 3

STATEMENTS, COMMANDS, AND FUNCTIONS

INTRODUCTION

This section describes all of the statements, commands, and functions implemented in the Auxiliary Memory File Manager ROM pack. They are listed in alphabetical order. The format for each is given followed by a detailed description including exmples.

APLOT Plots absolute format UDU data directly from a file.

Format

CALL "APLOT",fi[,[-]io[,i1[,i2]]] format for x,y pairs in 1 file
CALL "APLOT",f1,f2;[-]io[,i1[,i2]] format for x & y in 2 files

where: fi = file identifer for x,y values
f1 = file number for x values
f2 = file number for y values
[-]io = I/O device number (optional)
i1 = starting item position (optional)
i2 = ending item position (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal. This file contains x,y coordinate pairs.
file number for x	a numeric expression designating the file containing the x values to be plotted.
file number for y	a numeric expression designating the file containing the y values to be plotted.
device number	a numeric expression designating either the screen (device 32), or an external plotting device on the GPIB.
starting position	a numeric expression that specifies the starting item position to be plotted.
ending position	a numeric expression that specifies the last item position to be plotted.

Why Use It?

If you have data already expressed in standard UDU format that extends into the negative quadrants, APLOT will allow you to transfer this data to files 'as is' for direct file plotting.

This is different from MPLOT in that APLOT uses the full range of UDU data, whereas MPLOT treats each negative valued data point as a move.

What It Does

The APLOT routine plots graphical information expressed in user definable units directly from the specified file(s) to the specified I/O device or by default to the screen. The first coordinate plotted can be flagged as a move by the sign of the I/O device number, where a negative value indicates the first coordinate should be a move.

How To Use It

Data to be plotted can be stored in one of two ways. A single file may contain a series of x,y coordinate pairs. In this case, the APLOT routine requires a single file identifier, which may be a name or number. All other arguments are optional. If the I/O device number is omitted, then the plot, by default, will be made to the screen.

The second way is where the x and y coordinates are stored in separate files. In this case, the files must be referenced by number, where the first file contains x values and the second file contains y values, and the second file argument MUST be followed by a semicolon. This is necessary as it is the only way to ditinguish between single file format and dual file format. Also, since the CALL statement cannot end with a semicolon, the I/O argument which comes next is not optional in this case. Therefore, when a plot is targeted to the screen, the device address of 32 must be inserted following the y file's semicolon.

The optional range specifiers are the same for both file formats. If the optional starting position is omitted, the default starting position will be the first item, and the ending position will be the last item in the file. If only the ending position is omitted, the starting point can still be specified and the ending point will be the last item position.

Examples

```
100 PRINT "ENTER FILE NAME TO BE PLOTTED: ";
110 INPUT F$
120 CALL "APLOT",F$,-32
```

The contents of the file named in F$ are plotted to the screen. The negative sign on the screen address specifies that the first coordinate plotted will be interpreted as a move. Data in this file is stored in a series of x and y coordinate values.

```
100 DIM X(100),Y(100)
110 WINDOW -1,1,-1,1
120 VIEWPORT 15,115,0,100
130 I=0
140 FOR A=0 TO 2*PI STEP 2*PI/99
150 I=I+1
160 X(I)=COS(A)
170 Y(I)=SIN(A)
180 NEXT A
200 CALL "WRITE",1,X
210 CALL "WRITE",2,Y
220 CALL "APLOT",1,2;-32
```

This program computes 100 coordinates on a unit circle and saves the x and y coordinates in separate files. Note that the first y coordinate is made negative before the data is written to the file. This will cause the first point to be a move. This circle is then plotted to the screen, centered and expanded by the WINDOW and VIEWPORT.

Note that if it is desired to MOVE to location (0,0), it is not possible to make either coordinate negative to flag the move since both are zero. In this case you may solve the problem by making one value a small negative number near zero, such as 1.0E-300.

```
100 MOVE 0,0
110 CALL "APLOT",10,1,51
```

This statement will plot x,y data from file number 10 to the external GPIB address 1. The starting data location within the file is at record 51. The first move is made in line 100 before APLOT is called.

CONCAT Combines two files into one by concatination.

Format

CALL "CONCAT",f1,f2

where: f1 = target file
f2 = source file

Arguments

target file	a numeric expression that specifies the file to be added to. Must be an existing random file.
source file	a numeric expression that specifies the file to be copied from. Must be an existing random file.

Why Use It?

You may wish to combine 2 files into one in order to perform some common operation on them such as scaling, sorting, plotting, or location of min/max values.

What It Does

CONCAT allows two random data files to be combined together into one file. The second file is added to the end of the first file (f1=f1+f2). The second file remains unaffected.

How To Use It

Both files must be Random files with the same record lengths. The first file indicated in the CONCAT command is the target file. The second file is added to the end of the first file. The second file will then remain unaffected and the first file will contain the information from both. The Last Item Pointer will be appropriately updated in the first file to reflect the new total item count.

Example

```
100 CALL "CONCAT",8,4
```

File 8 is the target file. File 4 is joined to the end of file 8 and file 4 remains unchanged while file 8 now has data from both.

COPY Makes a copy of an existing file by creating a new one.

Format

CALL "COPY",f1,f2

where: f1 = source file
f2 = target file

Arguments

source file	a numeric expression that specifies the file to be copied. Must be an existing random file.
target file	a numeric expression that specifies the file to be copied to. Must be an existing random file.

Why Use It?

You may want a copy of a file that you can alter without affecting the original file. You might also want a back-up copy of important files.

What It Does

Makes a copy of an exisiting file by creating a new file and duplicating the contents of the first file to the new file. The first file specified is the file copied, the second file is the new file (f1-->f2).

How To Use It

Any file type can be copied. The original file remains unaffected. The destination file can be any valid file number that is currently unused. If the file copied was write protected, then the new file will also be write protected.

Example

```
100 CALL "COPY",3,4
```

File 3 is copied to file 4, which was a previously an un-used file number.

DIR Prints a listing of all files currently in the directory.

Format

CALL "DIR"[,i0]	Call format
DIR[ECTORY] [i0]	Key word format

where: i0 = I/O device number (optional)

Arguments

i/o device	a numeric expression that designates some I/O device connected to the system such as a printer. If omitted, the directory will be printed on the screen.

Why Use It?

To print a directory of all files created to either the screen or a selected I/O device, such as tape drive or printer. This will enable you to see what files have been created, how many there are, how much space each file has used, how much memory is left, and the names, numbers, and record sizes assigned to each.

What It Does

DIR prints a listing of all files currently in the directory to the 4050 screen or to the optionally specified I/O address. This listing inlcudes: file number, file name, file type, record size, and file size. Write protected files are also marked with a 'wp'. There are 3 file types: Random, Sequential, and Program. The record size, for random files, indicates the maximum string length allowed, and if less than 8, the numeric format and accuracy. The record size does not apply to Sequential or Program file types and is therefore listed as '–' in the directory. The file size indicated is in bytes and includes 44 bytes which is used as a file header on each file.

How To Use It

This command has both a CALL format and a key word format. Either may be used if no other file manger is present. If there is another file manager present in the system, then the CALL format may always be used exclusively, or the DIR key word may be used if the appropriate protcol is observed. Refer to the description of MUNIT and DUNIT for an explanation of this protocol.

The I/O device is the only argument for this command and it is optional. The default device is the 4050 system screen. Other devices you may wish to use would be a printer, internal or external mag-tape, or other GPIB device.

Examples

```
100 CALL "DIR",51
```

A directory is printed to the printer located at I/O device address 51.

```
DIR
```

No.	Name	Type	Rec Siz	Size
1	ADDRESS LIST	Random	72	18044
2	STOCK NUMBERS	Random	8	68
5	COMCODES	Random	-2	1044
7	GRAPH DATA	Random	3	80
8	TITLES	Sequential	-	2000
9	GRAPHPROG	Program	-	8112

The directory is printed to the screen.

EXCLUDE Sets the exclusion level for remarks in programs loaded.

Format

CALL "EXCLUDE",I

where: I = exclusion level

Arguments

exclusion level	a numeric expression representin an integer value of 0, 1, or 2 which sets the exclusion for remarks when loading program files.

Why Use It?

This routine allows you to load program files from auxiliary memory excluding remarks that are in the originally saved program. This may be important to conserve user memory for additional data space.

What It Does

The EXCLUDE routine allows you to set the mode of exclusion to be used prior to a program load instruction from the auxiliary memory such as OLD. A value of zero will turn off the exclusion process allowing all remarks present in the saved program to be loaded into user memory.

A value of one for the EXCLUDE argument, will cause REM statements to be trimmed of any remark and keep only the line number and the REM code.

A value of two will cause any lines containing remark statements to be entirely eliminated as the program is loaded.

This function is similar to the A series disk file manager function but will work on all non-A machines including 4051.

How To Use It

This routine must be called prior to the next program load instruction such as OLD, LINK, or APPEND. If the program contains REMark statements that are referenced by flow control statements such as GOTO, GOSUB, IF THEN, etc., then an exclusion level of 1 should be used so that these statement numbers will be retained for reference but trimmed of remark contents.

If there are no referenced remarks, then exclude level 2 may used which will eliminate all remark statements and save the most amount of user space.

The program as saved in the file will still contain all the remarks and may be loaded into user memory with remarks later for editing purposes.

Examples

```
900 CALL "EXCLUDE",2
910 CALL "MLINK",13,100
```

These program lines load the program file number 13 from the auxiliary memory with all remark statements excluded.

```
CALL "EXCLUDE",0
OLD "P-CALC"
```

These statements load the file named P-CALC into user memory with all remarks intact and ready to be edited and re-saved.

FILHDR Returns directory listing for specified file.

Format

CALL "FILHDR",fi,H$

where: fi = file identifier
H$ = target string

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
target string	a string variable to receive directory information about the specified file. Must not be dimensioned smaller than 72.

Why Use It?

Use FILHDR to get the directory information on a single file, or on a selected range of files. Since it returns the information in a string variable, it may also be used to extract certain directory information under program control.

What It Does

FILHDR returns the same information that would be output to the screen or optional I/O device by the DIRECTORY command. The information, however, is placed in a string variable and only for one selected file at a time. If the file on which information is requested does not exist, then a null string will be returned.

How To Use It

Two arguments must be supplied with this function. The first argument specifies the file desired either by name, or by number. The second argument is a string variable which must be dimensioned large enough to receive the information about the selected file. This function may be incorporated into a loop to produce a directory listing over a selected range. This is easily achieved both because no error is generated by invalid file numbers (a null string is returned) and because the string returned has a carriage return appended if valid. This allows the string to be printed with a semicolon to suppress additional carriage returns and produce listings without blank lines and without checking for null strings. This is illustrated below.

Examples

```
100 FOR F=10 TO 35
110 CALL "FILHDR",F,F$
120 PRINT F$;
130 NEXT F
```

Produces a directory listing for any exisiting files between 10 and 35.

```
100 PRINT "ENTER FILE NAME YOU WISH TO CHECK FILE SIZE OF: ";
110 INPUT A$
120 CALL "FILHDR",A$,B$
130 C$=SEG(B$,66,6)
140 IF C$="" THEN 100
150 PRINT "FILE SIZE =";VAL(C$)
160 END
```

Prints the file size for the user selected file.

FILTYP Returns a code for the file type of a specified file.

Format

CALL "FILTYP",fi,T

where: fi = file identifier
T = target variable

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
target variable	a numeric variable to receive the file type code for the specified file.

Why Use It?

It may be more convenient to get file type information about a selected file in a numeric variable rather than a string as with FILHDR, or by the DIRECTORY command. This could be valuable to determine if a certain file number is available. Zero is returned for files that don't exist.

What It Does

FILTYP returns one of 4 codes 0, 1, 2, or 3 indicating file type. Zero indicates the file has not yet been created and is available. A value of 1 = RANDOM, 2 = SEQUENTIAL, and 3 = PROGRAM file types. These values correspond with the values used in the MCREATE command to select the file type.

How To Use It

This function requires two arguments, one to select the file to be typed, and the second to receive the numeric value of the file type. If the value returned is zero, then the file specified does not exist. After CALLing this function, it may be followed by a conditonal IF/THEN statement, or a computed GOTO, or the value may simply be reported back.

Example

```
100 FOR F=1 TO 10
110 CALL "FILTYP",F,T
120 IF T>0 THEN 140
130 CALL "MCREATE",F,1,8
140 NEXT F
```

This program creates any files between 1 and 10 that are not already created.

FPLOT Fast plots GDU format data directly from a file.

Format

CALL "FPLOT",fi[,i1[,i2]] format for x,y pairs in 1 file
CALL "FPLOT",f1,f2;i1[,i2] format for x & y in 2 files

where: fi = file identifer for x,y values
f1 = file number for x values
f2 = file number for y values
i1 = starting item position (optional)
i2 = ending item position (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal. This file contains x,y coordinate pairs.
file number for x	a numeric expression designating the file containing the x values to be plotted.
file number for y	a numeric expression designating the file containing the y values to be plotted.
starting position	a numeric expression that specifies the starting item position to be plotted.
ending position	a numeric expression that specifies the last item position to be plotted.

Why Use It?

Use FPLOT to plot data expressed in graphic defined units (GDU's), which are not subject to WINDOW and VIEWPORT scaling. This is convenient for placement of forms or other graphical information which is totally independant of the window and viewport scale and offset.

Since GDU data is positive only, negative values may be used to designate moves so that entire plots can be output from a file with a single command having any number of embedded moves. This format will also result in faster plotting speeds than if individual MOVE and DRAW statements are utilized.

What It Does

The FPLOT routine plots graphical information expressed in Graphic Display Units directly from the specified file(s) to the screen only. The GDU format ignores any window or viewport definitions. Both the window an viewport are effectively set at 0,130,0,100 or the actual screen size.

This allows graphical information with fixed positions to be easily placed on the screen using a single command plotting from a single data file (or 2 files for separate x,y storage).

How To Use It

Data to be plotted can be stored in one of two ways. A single file may contain a series of x,y coordinate pairs. In this case, the FPLOT routine requires a single file identifier, which may be a name or number, and all other arguments are optional.

The second way is where the x and y coordinates are stored in separate files. In this case, the files must be referenced by number, where the first file contains x values and the second file contains y values, and the second file argument MUST be followed by a semicolon. This is necessary as it is the only way to ditinguish between single file format and dual file format. Also, since the CALL statement cannot end with a semicolon, the starting position argument which comes next is not optional in this case.

In the single file format if the optional starting position is omitted, the default starting position will be the first item, and the ending position will be the last item in the file. If only the ending position is omitted, the starting point can still be specified and the ending point will be the last item position.

Examples

```
100 CALL "FPLOT","FORMS OVERLAY"
```

The contents of the file named "FORMS OVERLAY" are plotted to the screen. Data in this file is stored in a series of x and y coordinate values.

```
200 CALL "SCALE",1,1,10
210 CALL "SCALE",2,1,20
220 CALL "FPLOT",1,2;1
```

This program plots GDU format data with x values conained in file 1, and y values conained in file 2. Before the files plotted, each axis is shifted with the SCALE routine which adds 10 to every x coordinate in line 100 and 20 to every y coordinate in line 110.

Note that if it is desired to MOVE to location (0,0), it is not possible to make either coordinate negative to flag the move since both are zero. In this case you may solve the problem by making one value a small negative number near zero, such as 1.0E-300.

```
CALL "FPLOT",F1,S,S+999
```

This statement plots 500 x,y pairs from the file specified by F to the screen. The starting position of the first data pair is specified by the value in S.

GETADD Gets the address of the first data byte in a specified file.

Format

CALL "GETADD",fi,a

where: fi = file identifier
a = target variable for address

Arguments

file identifier	a numeric expression that specifies the file whose address is to be returned.
target variable	a simple variable that will be assigned the value of the address of the first data byte in the specified file.

Why Use It?

Use the GETADD routine to find the physical memory address of the first data byte location of a given file. It is the first byte beyond the 44 byte file header. This address may be used by the primitive routines RBYTES and WBYTES, as well as the DMA routines, and data acquisition ROM packs such as the A/D converter.

What It Does

GETADD returns the address of the first data byte in a file. The first data byte is located just beyond the file header which is 44 bytes long. This memory address is assigned to the target variable supplied and passed back to the user or the program. If a file that does not exist is referenced, an error will be produced.

How To Use It

GETADD must reference a valid file and provide a simple variable as a target for the starting file address that is to be returned. If the file header is to be accessed, the address returned minus 44 bytes must be used to point to the beginning of the file header. Otherwise the address will point to the first data byte which is just beyond the header.

Examples

```
100 CALL "GETADD",11,A
110 CALL "RBYTES",A$,A-44,44
```

These 2 lines will get the starting data address of file 11 and then read the 44 byte file header of that file into A$.

```
100 CALL "GETADD",F,X
110 CALL "DMA8",X
```

These statements find the address of file F and passes that address to the DMA option ready to take data at that address.

GETIA Gets the physical address of a data item in a file.

Format

CALL "GETIA",fi,a

where: fi = file identifier
a = target variable for item address

Arguments

file identifier	a numeric expression that specifies the file whose current item address is to be returned.
target variable	a simple variable that will be assigned the address of the current item pointer for the specified file.

Why Use It?

Use GETIA to get the physical memory address for the current item pointer of a specified file. This will enable you to access that item location with the primitive read and write routines. The item address may also be used for DMA or data acquisition operations.

What It Does

GETIA uses the file identifier to look up the requested file and locate its internal current item pointer address. This value is then assigned to the simple variable provided as a target and is passed back to the user or to the program.

How To Use It

To use GETIA, simply specify a file either by name or be number and provide a target variable for the current item address to be passed back in. You must provide a valid file specifier or an error will result.

Examples

```
100 CALL "GETIA",F,A
110 CALL "RBYTES",A$,A,8
```

This program reads the address of the current item in line 100 and then uses it to read the contents of that item location with RBYTES into a string variable.

```
100 CALL "SETIP",1,10
110 CALL "GETIA",1,N
120 A$="TEST"
120 CALL "WBYTES",A$,N
130 CALL "READ",1,10;B$
140 PRINT A$,B$
```

This program sets the current item position in file 1 to 10. It then gets the address of that item and uses it to write the string "TEST" assigned to A$ to that item location using the WBYTES routine. It then uses the READ routine to read back the contents of item 10 into B$ and prints both A$ and B$ to the screen to verify results.

GETIP Gets and returns a file's current item position.

Format

CALL "GETIP",f[,I]

where: fi = file identifier
I = target for current item pointer (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
item pointer	must be a simple variable that can receive the value of the current Item Pointer. It will be printed to the screen if omitted.

Why Use It?

Since the current item pointer position is automatically advanced by one for each item read or written, it might be valuable after a series of reads or writes to be able to read the current position. This might be used to ascertain the position of an item after meeting a certain criteria from a search.

Since a seperate item pointer is maintained for each file created, this means that it is not necessary for the user to maintain a separate set of variables to save the current item position for each file when accessing a number of files at once.

What It Does

GETIP returns the position of the current Item Pointer for the specified file. The current item pointer is positioned at the address of the next item to be read or written. The value of the item pointer is the position of that data item starting from the first item at position 1, the second item at position 2, and so on.

This value will be assigned to the target variable, if one is supplied, or printed to the screen if no target variable is supplied.

How To Use It

The initial value of the Item Pointer for a newly created file is 1. This value is updated each time a READ or WRITE of an item occurs. A separate item pointer is maintained for each file. The file referenced may be either Random or Sequential. Any array reads or writes will advance the pointer by 1 for each element in the array that is transferred.

If you want to see the position of the item pointer for a given file, you may issue the command with no target variable and the value will be printed on the screen. If the position is to be used by the program, then the value can be returned in a variable by supplying the optional target variable as the second parameter in GETIP following the file specifier.

Examples

```
100 CALL "GETIP",6,I
```

The position of the current Item Pointer for file 6 will be assigned to the variable I.

```
CALL "GETIP",10
```

The position of the current Item Pointer for file 10 is printed on the screen.

INIT Initializes the Auxiliary Memory, releasing all memory.

Format

CALL "INIT"

Arguments

none.

Why Use It?

Use it as a convenient way to delete all files and free all memory when none of the existing files are to be preserved. It may also be used to re-initialize if a corrupt memory is suspected.

What It Does

INIT deletes all existing files, re-formats the directory structure, re-initializes the link table, makes all memory available for use.

How To Use It

This command should only be used when none of the currently open files are to be preserved. It has no arguments and is simply CALLed by name. Be sure to exercise caution when using this routine, as all files will be deleted regardless of write protection and are not recoverable.

Example

```
100 CALL "INIT"
```

All files are deleted from the auxiliary memory and all available memory is made free.

INSERT Inserts a blank record by expanding the data items in a file.

Format

CALL "INSERT",fi,i

where: fi = file identifier
i = item number to be inserted

Arguments

file identifier	a numeric expression that specifies the file that is to be expanded by inserting a record.
item number	a numeric expression that specifies the position for the blank record to be inserted.

Why Use It?

You may want to add a data item to list of items that is already in sorted order. This routine will automtically expand the existing list by copying every record ahead one position from the specified position, leaving that record open for a new data item. You may then add a data item at the specified record location keeping the list in sorted order.

What It Does

The INSERT routine allows you to insert a data item into a random file. This is done by moving each item forward one position starting with the last item and making it the new last item. The next to last item is then moved to the old last item spot. This operation continues backwards from there until the record at the specified item number location is moved forward making that location free to accept a new data item.

After moving the records forward, the item pointer is set to the location specified so that it may be followed immediately by a write operation without separately using SETIP.

How To Use It

To use INSERT, simply specify a file and the record position within that file where a new data item is to be added. The records from there forward will be moved up one position so that a new data item may be written to the specified location without destroying any of the existing data.

Each time INSERT is called, the last item position will increase by one.

Examples

```
100 CALL "INSERT",8,21
110 CALL "WRITE",8,A$
```

This program inserts the string data item contained in A$ at record position 21 in file 8 after having moved all other data items in file 8 forward one position starting with record 21.

KILL Deletes a specified file or list of files.

Format

KILL f$	Key word format
CALL "KILL",fi	Call format
CALL "KILL",f1[,f2..]	Call format with file list

where: f$ = file name
fi = file identifier
f1,f2,... = list of file numbers

Arguments

file name	must be a string of 1 to 28 characters expressed either as a variable or a literal.
file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
file list	must be one or more numeric expressions referencing files that are to be deleted.

Why Use It?

Use KILL to delete files that are no longer needed in order to clear them from the directory listings and make the memory used by them available for new files.

What It Does

Kill deletes the specified file(s) by removing them from the directory list and then adding any memory blocks used by them back into the free block list. If a specified file does not exist or has already been killed, then it will be ignored and no error message will be generated.

How To Use It

Once a file has been KILLed, it is not possible to recover it. The memory occupied by that file will be released for use by other files and that file number may be re-used for another new file. Any attempt to KILL a file that does not exist will be ignored.

To verify the existance of a file before KILL is attempted, the FILTYP command may be used. If a non-zero file type is returned by FILTYP, then that file identifier may be assumed valid and used in the KILL command. This is only necessary if there is any question as to the validity of the file identifier before KILL is used. FILHDR may also be used for this purpose.

Example

```
100 PRINT "Enter file number to kill: ";
110 INPUT F
115 IF F=0 THEN 180
120 CALL "MOPEN",F,F$
125 IF F$="" THEN 100
130 CALL "MSPACE",M1
140 CALL "KILL",F
150 CALL "MSPACE",M2
160 PRINT F$;" deleted. ";M1-M2;" bytes made free for use."
170 GOTO 100
180 END
```

This program asks for a file number and verifies its existance by finding its name. It then deletes the file, checking the memory space before and after to report the amount memory freed up.

LSTIP Returns position of the last item pointer in a file.

Format

CALL "LSTIP",f,[l]

where: fi = file identifier
l = target for last item pointer (optional)

Arguments

file identifier — can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.

item pointer — must be a scalar varriable that can be assigned the value of the current Item Pointer. It will be printed to the screen if this variable is omitted.

Why Use It?

You may want to know how many total items are in a file. You may want to use this value in the program to reset the current item pointer to add more data to the file at the very end.

What It Does

It returns the position of the last item in the specified file. It is also a total count of items written to the file. The value of the last item pointer's position will be printed to the screen if the optional target varaible is omitted. The value will be assigned to a target variable if one is supplied.

How To Use It

The initial value of the Last Item Pointer for a new file is 0. This pointer is updated by 1 each time a new item is written to the file beyond the old last item position. A separate Last Item Pointer is maintained for each file. The file referenced may be either Random or Sequential.

To add data to the end of file, the current item pointer must be moved to last item pointer value plus one. Setting the item pointer to the value of the last item pointer will allow you to read the last item, but any attempt to write from that position will over-write the last item. Therefore, the next item position beyond the last is available for writing, but not reading.

Examples

```
100 CALL "LSTIP",1,I2
```

The position of the last item in file 1 is assigned to I2.

```
CALL "LSTIP",2
```

The position of the Last Item in file 2 is printed to the screen.

MAPPEN Appends a sub-program to the exisiting program.

Format

APP[END] f$;t[,i] Key word format
CALL "MAPPEN",fi,t[,i] Call format

where: f$ = file name
fi = file identifier
t = target line
i = renumber increment (optional)

Arguments

file name	must be a string of 1 to 28 characters expressed either as a variable or a literal.
file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
target line	specifies a line number in the current BASIC program where the new program is to be appended.
renumber increment	a numeric expression that specifies the increment to be used in renumbering the program. The default value is 10.

Why Use It?

Use MAPPEND (or APPEND) to add or insert a program segment or subroutine into the exisiting program currently resident in executable memory. Variables currently defined are not affected.

This allows you to modularize your programs and bring in segments as needed into the main program. This way, more than one main program can access the module and append it where appropriate.

What It Does

MAPPEND loads a program from the Auxiliary Memory into user memory by adding it to the currently resident program at a specified starting target line number. This target line is the only line from the original program that is deleted, as it becomes the new first line of the appended program.

The remainder of the main program is moved down to follow after the appended portion. This may produce a forced automatic re-numbering of both the appended portion and the remainder of the main program. The user may optionally specify a renumber icrement which will also force a renumbering to occur.

How To Use It

The MAPPEN routine allows a new program segment to be appended to the program currently in user memory. Renumbering occurs if the lines following the target line in the current BASIC program do not allow enough room for the incoming program segment. The renumbering increment may be specified in the MAPPEN command, or it will default to a value of 10. Since the target line is destroyed by the incoming program, it should be a dummy statement set up for that purpose.

Examples

```
100 CALL "MAPPEN",4,1000,5
```

The program contained in file 4 of the Auxiliary Memory is appended to the current program in user memory starting at line 1000, and the program is renumbered thereafter in increments of 5.

```
APPEND "Maximum Value";100
```

The program in the Auxiliary Memory in the file named "Maximum Value" is appended to the resident program starting at line 100. The renumbering increment default is 10, and renumbering occurs if the next line line number of the main program is not higher than the highest number of the appended program. The program must be started manually since the command was entered from the keyboard.

MAX1 Finds the maximum value and its postion in a file.

Format

CALL "MAX1",fi,M,P[,i1[,i2]]

where:
- fi = file identifier
- M = maximum value (target variable)
- P = position (target variable)
- i1 = starting item position (optional)
- i2 = ending item position (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
maximum value	must be a numeric variable that can be assigned the value of the resulting maximum number found.
position	must be a numeric variable that can be assigned the value of the corresponding item position of the maximum value found.
starting point	specifies the starting item position to be searched. May be expressed as a simple variable, a numeric expression, or a literal value.
ending point	specifies the last item position to be searched. May be expressed as a simple variable, a numeric expression, or a literal value.

Why Use It?

You may wish to locate the maximum value and its position from a large array of numbers very rapidly. This routine allows you to do so very easily with a single statement.

What It Does

The MAX1 routine finds the maximum value and its position in a file. The entire file can be searched or a selected range within the file.

How To Use It

The file specified must be a Random, numeric data file. Sequential files are not allowed. If the optional starting position is omitted from the argument field, the default starting position will be the first item and the ending position will be the last item in the file. If only the ending position is omitted, the starting point can still be specified, and the end point will default to the last item position.

The file may be designated either by its name or number. If numerous accesses are to be made to the file, however, the file number should be used instead of the file name in order to achieve a faster execution speed. If the file is remembered by its name rather than its number, then use the MOPEN command to find the number.

Examples

```
100 CALL "MAX1",3,M,P
```

File 3 is searched for the maximum value which is assigned to the variable M. The position in the file of this value is assigned to P.

```
100 CALL "MAX1",F,M,P,100,120
```

The file designated by F is searched from item 100 to item 120 for the maximum value which is then assigned to M and its position to P.

MCHECK Performs a diagnostic check of the memory.

Format

CALL "MCHECK"

Arguments

none.

Why Use It?

Use MCHECK to verify the integrity of the memory or the interfacing hardware. The test will over-write any data currently in the memory causing it to be permanently lost. Therefore, MCHECK should only be invoked after any important data has been backed-up.

What It Does

MCHECK performs a diagnostic check of the auxiliary memory by exercising each location and printing to the screen the number of bad locations found in each block of 256 bytes, and the block number they occured in.

If there are no errors, then the total number of free bytes will be printed at the end of the check. This is the same as the number of free bytes reported by the MSPACE command.

The test performed is called a 'memory march'. This consists of writing a binary value to a given location, incrementing that value by one, and writing to the next physical memory location. This continues until each location in memory has been written to. The process is then repeated, only this time reading each location and comparing it with the value that was previously written.

Although this method is able to catch most problems associated with a bad memory chip or location within a chip, it cannot catch all problems.

It is possible to find no errors with this routine and still have a problem with the hardware. Some 'hidden' problems can be the result of an intermittant problem, a speed related problem, a value that by chance works for a given location but does not for a different value, and some problems related to address lines or data lines from random noise or borderline threshold, or loading problems.

In these situations, more extensive testing may be required, some of which can be done with the MTEST routines. If this fails, then the user may have to consult the factory for help. The factory may supply a loaner during repairs, or supply a unit in trade to track down the problem.

How To Use It

This command takes approximately 2-4 seconds per 128K bytes of memory to run. It is for self diagnostic purposes only and is fatal to any existing files. This routine should only be invoked to verify the memory when hardware problems are suspected.

If intermittant problems are present, you may expect to get different results each time the test is performed.

If the interface cable is not plugged in, or the power cord is not plugged in, or the memory cards are not installed, or some other hardware or interface problem is preventing the file manger from communicating with the memory, then this test will produce a series of block locations each indicating 255 bad locations.

The reason why 255 bad values out of 256 are reported is that the sequential test pattern that repeats every 256 bytes will contain one value that compares correctly with the one bad value that is contiuously read from the bad interface. This value is usually hex FF (decimal 255) and indicates all data lines high.

This pattern may also be produced if the memory tries to address memory beyond the amount actually installed. In this case the block number will indicate that the supposedly bad memory is at a higher address than the total amount of installed memory. This may happen if for any reason, the ROM pack thinks there are more memory cards present, or the location in the 4050 that stores the memory size has been glitched or otherwise modified such that the MCHECK routine does not know where to quit.

MCREATE Creates a memory file with optional name and attributes.

Format

CALL "MCREATE",f[,n$],t[,r] new syntax
CALL "OPEN",f[,n$],t[,r] former syntax (for compatibility)

where: f = file number
n$ = file name (optional)
t = type of File
r = record size (optional)

Arguments

file number — a numeric expression with an integer value greater than zero that specifies the file number of the file to be created.

file name — must be a string variable or a literal containing a name of 1 to 28 characters.

type of file — a numeric expression with an integer value of 1, 2, or 3 for Random, Sequential, or Program file types respectively.

record size — a numeric expression with an integer value between 1–255. It is the number of bytes reserved for each item in a RANDOM file. If less than 8 bytes, it also specifies a reduced precision format for numerics.

The record size may be omitted for SEQUENTIAL or PROGRAM files. If included, it will be ignored.

Why Use It?

Use MCREATE to create sequential files, or to create random files with special record sizes. It is also used to assign names to files.

What It Does

MCREATE generates a new entry in the directory listing, and allocates the first block of memory (256 bytes) for the new file. As the file requires additional memory space, it will dynamically expand. For existing files, MCREATE allows the file's name and/or record size to be changed. The current item pointer is also reset to 1.

How To Use It

It is not necessary to CREATE a file to be used as a PROGRAM file except to reserve a particular file number for later use, since SAVE will automatically create and name the file. Also, RANDOM data files are created automatically by the WRITE statement. The default record size for a random file is 8 bytes if the first item written to the file is a numeric, and 72 bytes if the first item is a string.

Therefore, besides assigning a name to a file, the main function of the CREATE routine is to specify a SEQUENTIAL file type, or to specify a different record size for a RANDOM file. A RANDOM file with a record size of less than 8 bytes will cause a reduced precision format to be used for numerics with record sizes of 1-3 bytes being integer format (negative indicating signed integer format) and 4-8 being floating point format. Random records with more than 8 bytes may contain one numeric item only (of 8 bytes) and the remaining bytes will be unused.

The record size in a RANDOM file fixes the number of bytes allowed for both numerics and strings. Strings larger than the defined record size will be truncated. Only one data item per record is allowed even if the data item fills only part of the record. This makes RANDOM files faster access but less memory efficient. SEQUENTIAL files are more memory efficient but slower to randomly access.

Since both upper and lower case characters of the file name are retained, the name must be referenced accordingly to be subsequently recognized. If the optional file name is omitted, the name field in the directory will list the name as "Noname".

Since every file has a unique file number, the name is used primarily as a label in the directory listings to help identify the file usage. For this reason it is possible to have more than one file with the same name. The file manager will make no attempt to flag duplicate names. Any file references made by name will result in the first occurance of a file by that name being located. It is the responsablity of the user, therefore, to ensure no duplicate names if he intends to reference the files by name rather than by number.

Examples

```
100 CALL "MCREATE",1,"ADDRESS LIST",2
```

File 1 is created as a SEQUENTIAL file and assigned a name of "ADDRESS LIST".

```
100 CALL "MCREATE",f,1,30
```

File f is opened as a RANDOM file with a record size of 30 which is the space alloted for every data item, numeric or string. The numeric accuracy used is 8 bytes. The default file name is "Noname".

```
100 CALL "MCREATE",2,"Signed Data",1,-3
```

File number 2 is assigned a name of "Signed Data", and is created as a RANDOM file with a record size of 3 that will accomodate signed integers or strings of 1 to 3 bytes.

Table of record sizes and numeric accuracies

Record Size	Numeric Accuracy In Decimal Digits (Floating Point Range E±307)
8	14 digits
7	12
6	9.5
5	7
4	4.5
	Integer Value Range
3	0 to 16,777,215
2	0 to 65,535
1	0 to 255
-3	-8,388,608 to +8,388,607
-2	-32,768 to +32,767
-1	-128 to +127

MCROSS Locates values in a file that 'cross' a threshold.

Format

CALL "MCROSS",fi,v,l[,n[,i1[,i2]]]

where:
- fi = file identifier
- v = threshold value
- l = target for cross location
- n = number of crosses (optional)
- i1 = starting position (optional)
- i2 = ending position (optional)

Arguments

file identifier	a numeric expression or a string that specifies the file number or name.
threshold value	a numeric expression that specifies the crossing threshold to compare against.
target cross	a simple variable that is assigned the value of the interpolated crossing location.
number of crosses	a numeric expression that specifies the crossing location to return. The default is 1 if this argument is omitted.
starting position	a numeric expression that specifies the starting position within the file where differentiation is to occur. The first item is the default if this argument is omitted.
ending position	a numeric expression that specifies the last position within the file through which differentiation is to occur. The last item is the default position if this argument is omitted.

Why Use It?

Use this routine to determine the location of a threshold crossing within an array of numbers stored in a random file. This is useful for signal processing applications.

What It Does

CROSS searches for a value in the specified file that crosses a given threshold value. The crossing location is the first value that is greater than or equal to the threshold value, if the file values are increasing. It is the first value that is less than or equal to the threshold value, if the file values are decreasing.

If the threshold is crossed more than once in the file, you can use the optional fourth argument to specify which crossing location you want. If that argument is omitted, the location of the first crossing will be returned.

If no crossing is located, a value of -1 will be returned. When the threshold value is crossed between file items, interpolation will be performed to give a more accurate position.

How To Use It

You must specify the file to be searched, the threshold value to search for, provide a target variable to receive the location, and optionally a fourth argument which specifies which crossing location you want.

You may also optionally specify the range of items within the file to be searched by either starting location or a starting and an ending location.

Examples

```
100 CALL "WRITE",1,5,10,15,20,25,30
110 CALL "MCROSS",1,17.5,P
```

The crossing location found from the data in this program will be 3.5. This is the interpolated location between the values of 15 and 20 in the 3rd and 4th locations for the threshold value of 17.5.

```
CALL "WRITE",1,100,21,17,13,9,2
CALL "MCROSS",1,50,N,2
```

The value returned here will be −1, indicating that there was not a second crossing location, although there was one location between the 1st and 2nd elements.

MCSUM Performs a checksum on contents of the auxiliary memory.

Format

CALL "MCSUM",S,p

where: S = target for chekcsum remainder
p = page count

Arguments

checksum target	a simple variable to be used as a target for the checksum of specified portion of memory.
page count	a numeric expression that specifies how many pages of 65,536 bytes are to be checksummed starting from the first page.

Why Use It?

This routine can be used for diagnostic purposes to determine if the contents of memory have changed from one time to another. MCSUM is primarily intended to verify data after performing a tape back-up and restore operation. This will enable you to have confidence that the data is still intact and was properly recorded on tape.

It may also be used to check data over a period of time such as over night if there is any suspicion of noise or other problems causing data losses.

What It Does

The MCSUM routine sums each data byte in the range of pages specified or in the entire memory. The sum is maintained in a single byte with over flows being lost so that a remainder is effectively retained. This number will always be between 0 and 255.

Although it is possible to have more than one data byte change and produce the same checksum, the chances are small and this method should provide a reliable means of verifying memory validity.

How To Use It

You must supply MCSUM with two arguments: the target variable to receive the checksum value, and the page count indicating how many pages are to be summed. If the the page count equals 2, then the first two pages (128K bytes) will be summed.

Example

```
100 CALL "MCSUM",S,4
110 FIND 1
120 CALL "TBACK"
130 FIND 1
140 CALL "TRESTORE"
150 CALL "MCSUM",S2,4
160 PRINT S,S2
```

The checksum of the first 4 pages (256K bytes) is performed before and after a back-up/restore sequence and the two results are printed to the screen.

MDIF2 Performs 2 point differentiation on the specified file.

Format

CALL "MDIF2",fi[,i1[,i2]]

where: fi = file identifier
i1 = starting position (optional)
i2 = ending position (optional)

Arguments

file identifier	a numeric expression or a string that specifies the number or name of the file that is to be differentiated.
starting position	a numeric expression that specifies the starting position within the file where differentiation is to occur. The first item is the default if this argument is omitted.
ending position	a numeric expression that specifies the last position within the file through which differentiation is to occur. The last item is the default position if this argument is omitted.

Why Use It?

Use this routine to perform 2 point differention directly on data stored in a file. This is a function useful for signal processing applications. The direct file access is useful both in handling larger arrays than can be held in user memory and for processing data stored directly in the auxiliary memory by data acquistion peripherals.

What It Does

The DIF2 routine performs a 2 point differentiation on the specified file with the original contents of the file being replaced with the results. The difference calculation peformed is the following:

$$b(t) = a(t+1) - a(t) \qquad \text{for } t = 1,2,\ldots,n-1$$
$$b(n) = a(n-1)$$

where: a = original file data
b = resulting file data
n = number of elements in file (or specified range)

How To Use It

You must specify the file to be differentiated. It must be a random numeric data file with 3 or more elements. The range of items to operate on may optionally be specified. The default will be the entire file.

If you wish to maintain a copy of the original file data, you must make a copy before performing the differentiation. You may do this most easily using the COPY command.

Examples

```
100 FOR J=0 TO 2*PI STEP 2*PI/100
110 CALL "WRITE",1,SIN(J)
120 NEXT J
130 CALL "COPY",1,2
140 CALL "MDIF2",1
```

A file of 101 data elements is created here and a copy made before the differentiation is performed on the original data.

MDIF3 Performs 3 point differentiation on the specified file.

Format

CALL "MDIF3",fi[,i1[,i2]]

where: fi = file identifier
i1 = starting position (optional)
i2 = ending position (optional)

Arguments

file identifier	a numeric expression or a string that specifies the number or name of the file that is to be differentiated.
starting position	a numeric expression that specifies the starting position within the file where differentiation is to occur. The first item is the default if this argument is omitted.
ending position	a numeric expression that specifies the last position within the file through which differentiation is to occur. The last item is the default position if this argument is omitted.

Why Use It?

Use this routine to perform 3 point differention directly on data stored in a file. This is a function useful for signal processing applications. The direct file access is useful both in handling larger arrays than can be held in user memory and for processing data stored directly in the auxiliary memory by data acquistion peripherals.

What It Does

The MDIF3 routine performs a 3 point differentiation on the specified file with the original contents of the file being replaced with the results. The difference calculation peformed is the following:

```
b(1) = (-3*a(1)+a(2)-a(3))/2
b(t) = (a(t+1)-a(t-1))/2          for t=2,3,...,n-1
b(n) = (a(n-2)-4*a(n-1)+3*a(n))/2
```

where: a = original file data
b = resulting file data
n = number of elements in file (or specified range)

How To Use It

You must specify the file to be differentiated. It must be a random numeric data file with 3 or more elements. The range of items to operate on may optionally be specified. The default will be the entire file.

If you wish to maintain a copy of the original file data, you must make a copy before performing the differentiation. You may do this most easily using the COPY command.

The three-point differentiation should be used instead of the two-point on data where large transitions occur over intervals greater than 3 elements. This will result in the least error for the derived slopes.

Examples

```
100 FOR J=0 TO 1 STEP .01
110 CALL "WRITE",1,EXP(J)
120 NEXT J
130 CALL "COPY",1,2
140 CALL "MDIF3",1
```

A file of 101 data elements is created here and a copy made before the differentiation is performed on the original data.

MIN1 Finds the minimum value and its postion in a file.

Format

CALL "MIN1",fi,M,P[,i1[,i2]]

where:
fi = file identifier
M = minimum value (target variable)
P = position (target variable)
i1 = starting item position (optional)
i2 = ending item position (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
minimum value	must be a numeric variable that can be assigned the value of the resulting minimum number found.
position	must be a numeric variable that can be assigned the value of the corresponding item position of the minimum value found.
starting point	specifies the starting item position to be searched. May be expressed as a simple variable, a numeric expression, or a literal value.
ending point	specifies the last item position to be searched. May be expressed as a simple variable, a numeric expression, or a literal value.

Why Use It?

You may wish to locate the minimum value and its position from a large array of numbers very rapidly. This routine allows you to do so very easily with a single statement.

What It Does

The MIN1 routine finds the minimum value and its position in a file. The entire file can be searched or a selected range within the file.

How To Use It

The file specified must be a Random, numeric data file. Sequential files are not allowed. If the optional starting position is omitted from the argument field, the default starting position will be the first item and the ending position will be the last item in the file. If only the ending position is omitted, the starting point can still be specified, and the end point will default to the last item position.

The file may be designated either by its name or number. If numerous accesses are to be made to the file, however, the file number should be used instead of the file name in order to achieve a faster execution speed. If the file is remembered by its name rather than its number, then use the MOPEN command to find the number.

Examples

```
100 CALL "MIN1",3,M,P
```

File 3 is searched for the minimum value which is assigned to the variable M. The position in the file of this value is assigned to P.

```
100 CALL "MIN1",F,M,P,100,120
```

The file designated by F is searched from item 100 to item 120 for the minimum value which is then assigned to M and its position to P.

MINT Performs integration on the specified file.

Format

CALL "MINT",fi[,i1[,i2]]

where: fi = file identifier
i1 = starting position (optional)
i2 = ending position (optional)

Arguments

file identifier	a numeric expression or a string that specifies the number or name of the file that is to be integrated.
starting position	a numeric expression that specifies the starting position within the file where integration is to occur. The first item is the default if this argument is omitted.
ending position	a numeric expression that specifies the last position within the file through which integration is to occur. The last item is the default position if this argument is omitted.

Why Use It?

Use this routine to perform integration directly on data stored in a file. This is a function useful for signal processing applications. The direct file access is useful both in handling larger arrays than can be held in user memory and for processing data stored directly in auxiliary memory by data acquistion peripherals.

What It Does

The MINT routine performs a integration on the specified file with the original contents of the file being replaced with the results. The integration is calculated using the trapezoidal rule for approximating the definite integral as follows:

$$b(1) = 0$$
$$b(t) = b(t-1)+(a(t-1)+a(t))/2 \quad \text{for } t=2,3,...,n$$

where: a = original file data
b = resulting file data
n = number of elements in file (or specified range)

How To Use It

You must specify the file to be integrated. It must be a random numeric data file with 3 or more elements. The range of items to operate on may optionally be specified. The default will be the entire file.

If you wish to maintain a copy of the original file data, you must make a copy before performing the integration. You may do this most easily using the COPY command.

Examples

```
100 FOR J=0 TO 2*PI STEP 2*PI/100
110 CALL "WRITE",1,SIN(J)
120 NEXT J
130 CALL "COPY",1,2
140 CALL "MINT",1
```

A file of 101 data elements is created here and a copy made before the integration is performed on the original data.

MLINK Replaces current user program with a new program.

Format

CALL "MLINK",f,l

where: fi = file identifier
l = line number

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
line number	must be a valid BASIC line number in the program being loaded. It specifies where execution is to begin.

Why Use It?

The MLINK routine allows large programs to be broken into smaller segments that together with data will fit in the memory space available. MLINK will then load these program segments into executable memory as they are needed.

MLINK is convenient to use because it automatically deletes the resident program segment without disturbing the variable definitions. It then loads the next program overlay from the specified file and continues running. This is done very rapidly so that it appears as though the original program were entirely resident at once.

What It Does

The MLINK routine is used to bring the next segment of a large program into user memory and continue running where the previous program segment left off, with all variables still intact. This is done by first deleting all lines of program currently in memory but retaining all variables. The specified program file is then loaded into executable memory and started at the specified line number.

While the program loading is done quite fast, the actual time required to load a given program into user memory depends somewhat on the number of variables encountered, since they must be 'mapped' into the system variable table.

How To Use It

The MLINK routine requires a file identifier of either a file number or a file name. It also requires a starting line number which is used to indicate the first line of the program being loaded that is to be executed. In loading the specified program, the previous program in user memory is deleted, but the varibles are kept intact.

The program will automatically begin execution at the specified line number only if MLINK is CALLed from a BASIC program. Otherwise, the program must be started manually.

Example

```
100 CALL "MLINK',2,100
```

The program contained in file 2 of the Auxiliary Memory is loaded into user memory and begins running at line 100. All variables from the previous program remain unaffected.

```
CALL "MLINK","FFT",1
```

This line causes the program named FFT to be loaded into executable memory, replacing the former program but keeping the data. It may then be executed with the RUN statement.

MOLD Loads a program from an Auxiliary Memory file into user memory.

Format

CALL "MOLD",fi call format
OLD f$ key word format

where: fi = file identifier
f$ = file name

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
file name	must be a string of 1 to 28 characters expressed either as a variable or a literal.

Why Use It?

To transfer a binary program previously saved in the Auxiliary Memory into the main user memory. An implicit DELETE ALL is performed in the process.

What It Does

The MOLD (OLD) command loads a program stored in an Auxiliary Memory file into the 4050 system memory. If the MOLD command is entered as an immediate instruction from the keyboard, then the program must be run manually. If the MOLD command is incorporated into a program, then the designated program will run automatically as soon as it is loaded from the Auxiliary Memory file.

Before loading the specified program file, all current program lines and variables are deleted from the 4050 system. This is equivalent to a DELETE ALL command.

How To Use It

The MOLD (or OLD) command will only work on a file containing a program previously saved using the MSAVE (or SAVE) command. MOLD will accept either a file number or a file name. The OLD (key word format) command will only accept a file name. Either version may be invoked manually from the keyboard or imbedded in a program. A file loaded manually from the keyboard must be started manually with the RUN command. A file loaded under program control will automatically begin execution at the first line.

NOTE: In order to use the OLD command as a key word instead of CALL "MOLD" in a configuration where another file manager besides the Auxiliary Memory is present in the system, you must observe the unit selection protocol described by the MUNIT and DUNIT commands.

Example

```
100 CALL "MOLD",2
```

The program contained in file 2 is brought into main memory and execution begins at the first line in the program.

```
OLD "INTERPOLATE"
```

This loads the program named INTERPOLATE into the system memory ready to be run.

```
400 PRINT "Enter the name of the program you wish to run: ";
450 INPUT A$
500 CALL "MOLD",A$
```

This program prompts the user for a program name and then loads and runs the program file referenced by the name entered into A$.

MOPEN Finds and returns the file number for the named file.

Format

CALL "MOPEN",F,n$

where: F = target variable
n$ = name string

Arguments

target variable	must be a simple variable that will be used to pass back the value of the file number of the named file.
name string	a string variable or literal containing the name of the file whose number is to be found.

Why Use It?

The auxiliary memory file system has permanently assigned numbers for referencing files as they are created. There is also an optional name which may be assigned to the file. Since it is often times easier to remember a file by its name rather than its number, this routine allows the file number to found based on its name.

In most instances it is possible to reference a file directly either by its name or number. However, if a file is to be accessed numerous times, it is more efficient to use the number. In this case, the MOPEN routine should be used to look up the file number so that it may be used rather than the name for repeated accesses.

What It Does

The MOPEN routine returns the file number of the named file. It takes the name from the string literal or variable and compares it with each name in the directory starting from the beginning and proceeding until either the name is found, or the end of the directory is reached.

If the name is found, the number of that file is assigned to the simple variable provided as the target in MOPEN and passed back to the user or the program. If the end of the directory is reached before the file name is found, a value of zero will be assigned to the target variable.

How To Use It

You must supply a valid file name from 1 to 28 characters in length making sure that it is spelled correctly and that it has the same upper and lower case characters as in the original name. MOPEN will then scan the directory looking for this name. If there is more than one file with this same name, the first one only will be located. The number of that file will then be passed back in the target variable provided. If the name was not found, the value returned will be zero.

Example

```
100 PRINT "ENTER FILE NAME TO BE OPENED: ";
110 INPUT A$
120 CALL "MOPEN",F,F$
130 IF F=0 THEN 100
140 PRINT A$;" IS ASSIGNED TO FILE NUMBER";F
150 END
```

MPLOT Plots absolute format UDU data directly from a file.

Format

CALL "MPLOT",fi[,i0[,i1[,i2]]] format for x,y pairs in 1 file
CALL "MPLOT",f1,f2;i0[,i1[,i2]] format for x & y in 2 files

where:
fi = file identifer for x,y values
f1 = file number for x values
f2 = file number for y values
io = I/O device number (optional)
i1 = starting item position (optional)
i2 = ending item position (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal. This file contains x,y coordinate pairs.
file number for x	a numeric expression designating the file containing the x values to be plotted.
file number for y	a numeric expression designating the file containing the y values to be plotted.
device number	a numeric expression designating either the screen (device 32), or an external plotting device on the GPIB.
starting point	a numeric expression that specifies the starting item position to be plotted.
ending point	a numeric expression that specifies the last item position to be plotted.

Why Use It?

Use MPLOT to plot data expressed in user definable units (UDU's), subject to WINDOW and VIEWPORT limits, but in the positive domain only so that negative values, on either x or y values may be used to designate moves.

This format allows you to have embedded moves in an array of data so that entire plots can be output from a file with a single command. This format may also result in faster plotting speeds than if individual MOVE and DRAW statements are utilized.

What It Does

The MPLOT routine plots graphical information directly from the specified file(s) to the specified I/O device or by default to the screen.

How To Use It

Data to be plotted can be stored in one of two ways. A single file may contain a series of x,y coordinate pairs. In this case, the MPLOT routine requires a single file identifier, which may be name or number, and all other arguments are optional. If the I/O device number is omitted, then the plot, by default, will be made to the screen.

The second way is where the x and y coordinates are stored in separate files. In this case, the files must be referenced by number, where the first file contains x values and the second file contains y values, and the second file argument MUST be followed by a semicolon. This is necessary as it is the only way to ditinguish between single file format and dual file format. Also, since the CALL statement cannot end with a semicolon, the I/O argument which comes next is not optional in this case. Therefore, when a plot is targeted to the screen, the device address of 32 must be inserted following the y file's semicolon.

The optional range specifiers are the same for both file formats. If the optional starting position is omitted, the default starting position will be the first item, and the ending position will be the last item in the file. If only the ending position is omitted, the starting point can still be specified and the ending point will be the last item position.

Examples

```
100 CALL "MPLOT","CARTOON"
```

The contents of the file named CARTOON are plotted to the screen. Data in this file is stored in a series of x and y coordinate values.

```
100 DIM X(100),Y(100)
110 WINDOW -1,1,-1,1
120 VIEWPORT 15,115,0,100
130 I=0
140 FOR A=0 TO 2*PI STEP 2*PI/99
150 I=I+1
160 X(I)=COS(A)
170 Y(I)=SIN(A)
180 NEXT A
190 Y(1)=-Y(1)
200 CALL "WRITE",1,X
210 CALL "WRITE",2,Y
220 CALL "MPLOT",1,2;32
```

This program computes 100 coordinates on a unit circle and saves the x and y coordinates in separate files. Note that the first y coordinate is made negative before the data is written to the file. This will cause the first point to be a move. This circle is then plotted to the screen, centered and expanded by the WINDOW and VIEWPORT.

Note that if it is desired to MOVE to location (0,0), it is not possible to make either coordinate negative to flag the move since both are zero. In this case you may solve the problem by making one value a small negative number near zero, such as 1.0E-300.

```
CALL "MPLOT",F1,10,101,200
```

This statement will plot x,y data from the file specified by F1 to an external GPIB address of 10. The data plotted is taken from the 100 data values starting at location 101, making up 50 x,y pairs.

MSAVE Stores the current program in a memory file.

Format

CALL "MSAVE",fi[,f$][,I1[,I2]] call format
SAV[E] f$[;I1[,I2]] key word format

where: fi = file identifier
f$ = file name
I1 = beginning line number
I2 = ending line number

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
file name	must be a string of 1 to 28 characters expressed either as a variable or a literal.
beginning line	a numeric expression designating the first line of the current program to start from.
ending line	a numeric expression designating the last line of the current program to save.

Why Use It?

Use MSAVE to save the program currently in user memory to the specified file in the Auxiliary Memory. It may then be brought back into user memory when needed at a later time very rapidly.

What It Does

MSAVE sends a binary copy of the current user program to the specified file in the Auxiliary Memory. If the optional beginning line number is present, then only that line will be saved. If both the beginning and ending line numbers are supplied, then that range of program lines will be saved.

How To Use It

Using the MSAVE command, you can supply a file number, a file name, or both to designate the file. The key word version, SAVE, will only accept a file name. If a file name only is supplied, then the file manager will automatically assign the next lowest un-used file number to the file. If a name is not supplied, the file manager will assign a default name of "Noname" to the file.

In order to use the SAVE command as a key word instead of CALL "MSAVE" in a configuration where another file manager besides the Auxiliary Memory is present in the system, you must observe the unit selection protocol described by the MUNIT and DUNIT commands.

When a program is saved, an implicit KILL is performed on the old copy of the file before the new copy is saved, if the file specified was previously used as a program file. This frees up memory when a smaller program is saved as the new version.

To find out what file number is assigned a file where a name only is supplied, you can look at a directory listing or use the MOPEN command to return the file number from the file name.

The optional line numbers may be used to save a selected range of program lines. This may be useful for saving programs automatically where a control program is located at line numbers beyond the range of numbers to be saved.

Examples

```
100 CALL "MSAVE",1,100,500
```

Lines 100 through 500 of the current user program will be saved in the Auxiliary Memory in file 1. If file 1 previously existed as a program file and was named, then that name will be retained. If it is a new file number, then the name assigned will be "Noname", since no name was supplied.

```
SAVE "MYPROG";1,9999
```

This command creates a file named MYPROG, and saves any lines in the current user program that are between 1 and 9999. If MYPROG already existed as a program file, then the same file number will be used for the new copy and the old copy will be deleted and over-written. If MYPROG is a new file name, then a new file number will be selected by the file manager. This number may be found by the MOPEN command.

```
CALL "MSAVE",10,"FILEPLOTTER"
```

This command saves all lines of the current program to file 10 and is named "FILEPLOTTER".

MSPACE Gives the number of free bytes of memory.

Format

CALL "MSPACE"[,S]

where: S = target variable (optional)

Arguments

target variable	must be a scalar variable that can be assigned the value of the number of free bytes currently available.

Why Use It?

MSPACE can be used to determine how much memory is available for use. This can be done either directly by the user to see reported on the screen the amount of memory available, or under program control the memory space can be directed into a variable for use by the program in file allocation and creation decisions.

What It Does

MSPACE returns the number of free bytes available in the Auxiliary Memory. The Auxiliary Memory is mapped out into blocks of 256 bytes each. The total number of free bytes indicated by the MSPACE routine is really the number of free blocks available multiplied by 256.

This means that although there may not be any more free blocks available for allocation to a file, there can still be partial blocks within other files that can only be used by those particular files. Whenever a file fills one entire block, it then requests another. Thus, when a memory full condition occurs in the auxiliary memory, there can still be memory available to existing files with partially filled blocks.

How To Use It

MSPACE may be used with or without a target variable. If the target variable is omitted, the number of free bytes will be printed to the screen. If a target variable is present, the value of the number of free bytes will be assigned to it.

Examples

```
100 CALL "MSPACE",M
```

The total number of free bytes (free blocks expressed in bytes) currently available in the auxiliary memory is assigned to M.

```
CALL "MSPACE"
```

The current byte count is printed to the screen.

MTEST Performs a set of diagnostic test routines.

Format

CALL "MTEST",1,c1,c2,c3	address counter test
CALL "MTEST",2,S0,S1	status register test
CALL "MTEST",3	read test
CALL "MTEST",4,d0,d1	write test
CALL "MTEST",5,a,n	read test II
CALL "MTEST",6,d0,d1,a,n	write test II
CALL "MTEST",7	read decoded locations
CALL "MTEST",8,a,k	memory card check
CALL "MTEST",9,r,v	register write (poke)
CALL "MTEST",10,r,V	register read (peek)

where:

c1 = high order address register (65K)
c2 = mid order address register (256)
c3 = low order address register (1)
S0 = target for status (should return zeros)
S1 = target for status (should return ones)
d0 = even data byte to write
d1 = odd data byte to write
a = address to start read/write
n = number to read/write before repeating
k = number of cards to check
r = register address
v = value to write (poke)
V = target for byte read

Arguments

address registers	numeric expressions that specify the address in memory where read/write operations are to occur.
status targets	simple variables to receive the status bytes.

MTEST Arguments (continued)

even/odd data	numeric expressions with values from 0–255 that are to be written at even and odd address locations alternately.
address	a numeric expression that specifies a physical memory address for data read/writes.
number of bytes	a numeric expression that specifies the number of data bytes to read or write before repeating.
number of cards	a numeric expression that specifies how many memory cards are installed and are to be checked.
register	a numeric expression that specifies the address of the register to read or write.
value to write	a numeric expression that is to be written to a specified address.
target for value	a simple variable that will receive the value of a byte from a register location.

Why Use It?

Use this series of routines to perform tests or diagnostic checks of the hardware. This may help you isolate problems and facilitate coordination with the factory for repair parts or ideas.

What It Does

Test number 1 continuously loads the address registers with the specified values. This should be used in conjunction with a scope probe to verify register contents.

Test 2 loads the status register with zeros and then ones reading after each write and returning the results in two target variables.

Test 3 reads sequentially from the memory starting at address 0 and proceding through 1M locations and then repeats continuously. The address counters may be checked during this pattern.

Test 4 writes the same series of locations as test 3 using the data bytes supplied, one for even address locations, and one for odd.

Test 5 is a read test similar to test 3 but allow you specify the starting address and the number of bytes to read before repeating.

Test 6 is a write test similar to test 4 but like test 5 allows you to specify the starting address and number of bytes to write.

Test 7 continously reads a series of decoded locations that may be probed.

Test 8 writes and reads two locations (even/odd) on a series of memory cards. You specify the location within each card and the number of cards to check. The results are printed in binary to the screen and should consist of 16 ones and then 16 zeros.

Test 9 allows any value to be written to any register location.

Test 10 allows any register location to be read.

How To Use It

Some of the tests, as explained above, will print results to the screen or return them in variables. For some tests that require probing, you may need the use of an extender board to give you access to the locations on the ROM interface board. In these situations you may want to refer service to the factory.

If your facility is equipped to handle service you may want to purchase or obtain a loaner extender board to perform these tests along with a schematic and list of probe locations.

MUNIT/DUNIT

MUNIT/DUNIT Selects the Memory or Disk as current File Manager.

Format

CALL "MUNIT"	Select Memory File Manager
CALL "DUNIT"	Select Disk File Manager

Arguments

none.

Why Use It?

You may want to use some of the BASIC key words available for the memory file manager when a disk file manager is also present. These commands let you switch back and forth between two file manager ROM packs.

What It Does

MUNIT writes the bank or slot address of the Auxiliary Memory ROM pack file manager into a special 4050 system location for the currently active file manager. This enables control of any file manager key word commands to be routed to the Auxiliary Memory for processing.

Before writing its address into this location, the Auxiliary Memory file manager reads the address of the other file manager presently in the system and stores it in a reserved location in its own memory so that it can be restored at a later time by DUNIT.

DUNIT restores the address of the disk file manager ROM pack that was active at the time MUNIT was last called. Therefore, MUNIT must always be the first command executed.

How To Use It

When switching between 2 file managers, the MUNIT command must be called prior to each access or series of accesses to the auxiliary memory. The other resident file manager(s) can be selected by a combination of the CALL "DUNIT" and UNIT commands. DUNIT must be CALLed only after having first CALLed MUNIT which saves the Disk unit's file manager address in its reserved memory.

NOTE: MUNIT and DUNIT do not need to be used if the Auxiliary Memory is the only file manager present!

Examples

```
CALL "MUNIT"
DIR
```

Selects the Auxiliary Memory as the current file manager until the next DUNIT command is issued. The directory for the auxiliary memory is then printed on the screen.

```
CALL "DUNIT"
UNIT 13
OLD "@CALC"
```

Activates the disk file manager as the current file manager and loads in a program from the disk unit 13.

NXTFIL Returns the next available file number.

Format

CALL "NXTFIL",F

where: F = target variable

Arguments

target variable	a simple variable used to pass back the value of the next lowest un-used file number.

Why Use It?

The Auxiliary Memory file structure pre-allocates enough directory space for 248 files. If a file number greater than this is used, an additional block of memory must be allocated and every reference to that file must be made by linking through the block pointers to it. This means that more memory is used and access to high file numbers is slightly slower. It is therefore desireable to use the lowest file numbers available when creating new files in order to maximize efficiency.

The NXTFIL routine will automatically locate the next lowest unused file number for you. While file numbers may generally be chosen in any order and need not be consecutive, this routine provides the capability to easily locate the next best number to use.

What It Does

The NXTFIL routine searches for the next lowest un-used file number that may be used to create a new file. This number is returned in the target variable provided.

How To Use It

To use the NXTFIL routine, simply provide a target variable for the value of the file number to be returned in.

Examples

```
100 CALL "NXTFIL",F
110 CALL "WRITE",F,X1,X2,X3
```

In these two lines of program, a new file number is found by the NXTFIL routine, and a new file is created using that number with the WRITE routine.

```
100 PRINT "ENTER A NAME FOR THIS NEW SEQUENTIAL DATA FILE: ";
110 INPUT N$
120 CALL "NXTFIL",N
130 CALL "MCREATE",N,N$,2
```

This program has the user enter a file name and then locates a new file number for that name. A sequential file is then created using the requested name and the newly found file number.

ON EOF(0) Traps end of file conditions.

Format

ON EOF(0) THEN l

where: l = line number of service routine

Arguments

line number	a literal numeric integer that specifies the line number in the current BASIC program where control is passed if the end of a data file is reached.

Why Use It?

The ON EOF statement allows you to trap end of file conditions under program control and avoid errors that abort program flow.

What It Does

The ON EOF statement allows you to specify a line number of the BASIC program where execution will continue when an end of file is detected during a READ of a memory file. Transfer to this line is equivalent to a gosub. This allows the program to return to the line following the error after handling the end of file condition.

How To Use It

This statement requires a file number or unit number as the argument in paranthesis. This must be a zero, which is the same unit as is used for the internal tape files. This means that after this statement is executed, an end of file on either mag-tape file or an auxiliary memory file will cause the trap to occur. If it is desired to distinguish between the two, then two statements must be present in the program, one before each read operation to the two devices.

Examples

```
100 CALL "SETIP",11,1
110 S=0
120 ON EOF(0) THEN 160
130 CALL "READ",11,X
140 S=S+X
150 GOTO 130
160 PRINT S
```

This program sets the item pointer in file 11 to 1 and then reads numbers until the end of file is reached. Each number read is summed and the total is printed in line 160 after control is transferred there after being trapped by the condition set up in line 120.

PROT/UPROT Protects and unprotects file from modification.

Format

CALL "PROT",fi
CALL "UNPROT",fi

where: fi = file identifier

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.

Why Use It?

Use the PROT command to protect a file from inadvertant modification. This will prevent another user from writing to or deleting your file. It might also prevent you from accidently writing to a file that you would like to keep un-changed.

Use UNPROT to clear the write protect flag from a file you have previously write protected to enable you to add to, modify, or delete it.

What It Does

The PROT command protects the specified file from modification by setting a write protect bit in the file header. This bit is checked by the file manager prior to any WRITE or KILL operation. If this bit is on, than an error message will be printed to the screen indicating that an attempt was made to modify a write-protected file.

UNPROT clears the write protect flag set by the PROT command allowing the specified file to be KILLed, written to, or otherwise modified.

The directory listing will also note any write-protected files by using a 'wp' designation following the file.

Note that the file is not protected against the CALL "INIT" statement which deletes all files, nor the MCHECK statement which over-writes all of memory. It is also possible to over-write a write protected file with the WBYTES routine.

How To Use It

Any file can be write protected. Program files, Random data files, and Sequential data files are all write protectable. To protect a file, simply execute CALL "PROT" with the file name or file number of the file you wish to protect. CALL "UNPROT" to remove the protection.

Examples

```
CALL "PROT","DATATABLE"
```

This statement sets the write protect flag in file header of the file named DATATABLE.

```
100 CALL "UNPROT",10
110 CALL "LSTIP",10,I
120 CALL "WRITE",10,I;A$
130 CALL "PROT",10
```

This program un-protects file 10, locates the last item position and adds a new item to the end of the file and then re-assigns the write protect status.

```
CALL "UNPROT","DATAFILE1"
KILL "DATAFILE1"
```

The write protection assigned to DATAFILE1 is removed and the file is then deleted from the memory.

RBYTES Reads data bytes into a string from a physical address.

Format

CALL "RBYTES",A$,a,n

where: A$ = target variable
a = address
n = number to read

Arguments

target variable	the string variable used as a target will be automatically dimensioned to N bytes if it is not previously dimensioned. An error will result if the current dimension is not large enough.
address	a numeric expression representing an absolute address from 0 to the highest address of installed memory.
number to read	a numeric expression that specifies the number of data bytes to read into the target string. Must be a value from 1 upto the dimensioned length of the string.

Why Use It?

You can use the RBYTES routine to read data from the auxiliary memory at any physical address. This may be necessary either to retrieve data stored by a DMA peripheral such as the A/D converter ROM pack, or other external device using Option 11 or 12.

RBYTES is also used to read data stored by the WBYTES routine. It may be used in conjuction with the GETIA and GETADD commands to read data from the file structure for special applications.

What It Does

The RBYTES routine reads a specified number of data bytes from the auxiliary memory starting at the specified address. This gives the user complete access to every available memory location.

How To Use It

RBYTES reads data into a target string which must be dimensioned large enough to receive the number of bytes requested. The string can be dimensioned as large as desired up to the maximum amount of user memory available. Any physical address address may be specified as first address to read from. The addressing starts at zero and goes to the highest address available for the amount of installed memory.

Example

```
100 CALL "RBYTES",A$,0,1000
```

This statement causes 1000 bytes to be read starting at the first possible address location. The data is stored in the target variable A$.

RDELET Deletes record from a file and compresses remaining items.

Format

CALL "RDELET",fi,i
where: fi = file identifier
i = item number to be deleted

Arguments

file identifier	a numeric expression that specifies the file that is to be compressed by deleting a record.
item number	a numeric expression that specifies the position of the record to be deleted.

Why Use It?

Use the DELETE routine to delete a single record from a random data file and compress the remaining items. This will enable you to remove a data item from a list keeping the remaining items in order.

What It Does

The DELETE routine starts from the record following the record to be deleted and moves it back one, overwriting the specified record. This operation then proceeds forward copying each record back one position until the last item has been moved back one position.

Since the last item is copied back one position, it will occupy the last two positions. The DELETE routine does not alter the last item position. The same number of items will exist after DELETE as before, only the order will be changed.

How To Use It

The DELETE routine may only be used on random data files. It requires a file specifier and a record number for the record position to be deleted. The record is deleted by moving the rest of the data list down one position writing over the top of the record to be deleted.

Example

```
100 PRINT "ENTER THE NAME TO BE DELETED: ";
110 INPUT A$
120 CALL "SETIP",1,1
130 CALL "LSTIP",1,L
140 FOR J=1 TO L
150 CALL "READ",1,B$
160 IF A$=B$ THEN 200
170 NEXT J
180 PRINT "NAME NOT FOUND!"
190 GOTO 100
200 CALL "DELETE",1,J
```

This program searches file 1 for certain name and deletes that record if it is found.

READ

READ Reads numeric and string data from memory files.

Format

CALL "READ",fi,[r;]D1[,D2...]

where: fi = file identifier
r = record number
D_n = input target variable list

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
record number	specifies record number to begin reading at
target variable(s)	any number of input variables: string or numeric, scalar or array.

Why Use It?

To read data items from the Auxiliary Memory file into the target variables specified.

What It Does

The READ routine is used to read data into target variables from auxiliary memory files. You can read data into array variables even if the data was not written to the file as an array. This is because each data element occupies one item position of the memory file. The dimensioned size of the array will determine how many numbers are read.

String and numeric data may be accessed in any order. However, the data type is not saved with each data item, so the user must know in what order the data was saved in order to read it back.

If the data item length is longer than the space available in the target variable, the possible, the remaining bytes will be discarded, and the item pointer will be advanced to the beginning of the next item.

How To Use It

The READ routine accepts any number of input variables, which can be strings or numerics in any combination. Reading takes place from the current position of the item pointer. The item pointer is updated after each item is read. The current item pointer can be re-positioned to any item with either the SETIP command, or the optional item pointer parameter in the READ routine itself.

The file can be referenced either by number or by name. However, for repeated accesses, it is faster to access the file by number. If the file is remembered by name, then use the MOPEN command to find the file number from the name.

Examples

```
100 PRINT "ENTER THE FILE YOU WISH TO READ FROM: ";
110 INPUT F$
120 CALL "MOPEN",F,F$
130 IF F=0 THEN 100
140 CALL "SETIP",F,1
150 CALL "READ",F,I1,I2,A$
```

The variables I1, I2, and A$ are read from file F starting with item 1.

```
100 DIM A(1000),B(1000)
110 FOR I=1 TO 100
120 CALL "READ",I,10;A
130 B=B+A
140 NEXT I
```

This program reads 1000 elements from 100 files starting in the 10th item position of each file. Each array read is then summed in a separate array. Note the semicolon following the second argument is required to indicate that 10 is the record to read.

RMPLOT Plots relatively positioned absolute format UDU data.

Format

CALL "RMPLOT",fi[,i0[,i1[,i2]]] format for x,y pairs in 1 file
CALL "RMPLOT",f1,f2;i0[,i1[,i2]] format for x & y in 2 files

where:
fi = file identifer for x,y values
f1 = file number for x values
f2 = file number for y values
io = I/O device number (optional)
i1 = starting item position (optional)
i2 = ending item position (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal. This file contains x,y coordinate pairs.
file number for x	a numeric expression designating the file containing the x values to be plotted.
file number for y	a numeric expression designating the file containing the y values to be plotted.
device number	a numeric expression designating either the screen (device 32), or an external plotting device on the GPIB.
starting position	a numeric expression that specifies the starting item position to be plotted.
ending position	a numeric expression that specifies the last item position to be plotted.

Why Use It?

Use RMPLOT to make relative plots from absolute data origined about (0,0). This can be positive domain UDU data with negative values interpreted as moves (as in MPLOT). The last graphic position will be used as the relative position for the plot.

This format will enable you to construct graphic images or symbols with imbedded moves and described by easier to use absolute coordinates but capable of being relatively positioned and rotated within any defined window and viewport.

What It Does

The RMPLOT routine plots graphical information directly from the specified file(s) to the specified I/O device or by default to the screen.

The data is in the form of positive domain, User Definable Units, absolutely referenced from an origin of (0,0). The RMPLOT routine takes these coordinates, and based on the position of the last coordinate plotted, will convert the data to relative format. They are then rotated, scaled, and offsett according to the currently defined window, viewport, and rotation angle. Any negative values encountered will be interpreted as moves.

How To Use It

Data to be plotted can be stored in one of two ways. A single file may contain a series of x,y coordinate pairs. In this case, the RMPLOT routine requires a single file identifier, which may be name or number, and all other arguments are optional. If the I/O device number is omitted, then the plot, by default, will be made to the screen.

The second way is where the x and y coordinates are stored in separate files. In this case, the files must be referenced by number, where the first file contains x values and the second file contains y values, and the second file argument MUST be followed by a semicolon. This is necessary as it is the only way to ditinguish between single file format and dual file format. Also, since the CALL statement cannot end with a

semicolon, the I/O argument which comes next is not optional in this case. Therefore, when a plot is targeted to the screen, the device address of 32 must be inserted following the y file's semicolon.

The optional range specifiers are the same for both file formats. If the optional starting position is omitted, the default starting position will be the first item, and the ending position will be the last item in the file. If only the ending position is omitted, the starting point can still be specified and the ending point will be the last item position.

Examples

```
100 SET DEGREES
110 ROTATE 90
120 MOVE 50,50
130 CALL "RMPLOT","G-SYMBOL"
```

The contents of the file named G-SYMBOL are plotted to the screen after setting the rotation angle to 90 degrees and positioning the plot to a starting location of (50,50).

```
500 CALL "SCALE",1,3,2
510 FOR D=0 TO 2*PI STEP 2*PI/5
520 MOVE X,Y
530 ROTATE D
530 CALL "RMPLOT",1,2;32
540 NEXT D
```

This program first scales the x data by muliplying each element by 2. It then performs a series of rotations, plotting the contents of files 1 and 2 repeatedly at different rotations, each referenced to the location defined by the values in X and Y.

Note that if it is desired to MOVE to location (0,0), it is not possible to make either coordinate negative to flag the move since both are zero. In this case you may solve the problem by making one value a small negative number near zero, such as 1.0E-300.

```
100 CALL "SETIP",1,1
100 FOR I=1 TO 10
110 CALL "READ",1,X,Y,S,N
130 MOVE X,Y
140 CALL "RMPLOT",2,2,S,N
150 NEXT I
```

This program reads from a data file the x,y starting postions of 10 symbols and their starting and ending positions within the file. All the symbols are contained in file 2 and are plotted to a plotter on the GPIB with an address of 2.

RPLOT Plots relative format UDU data directly from a file.

Format

CALL "RPLOT",fi[,[-]i0[,i1[,i2]]] format for x,y pairs in 1 file
CALL "RPLOT",f1,f2;[-]i0[,i1[,i2]] format for x & y in 2 files

where: fi = file identifer for x,y values
f1 = file number for x values
f2 = file number for y values
[-]io = I/O device number (optional)
i1 = starting item position (optional)
i2 = ending item position (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal. This file contains x,y coordinate pairs.
file number for x	a numeric expression designating the file containing the x values to be plotted.
file number for y	a numeric expression designating the file containing the y values to be plotted.
device number	a numeric expression designating either the screen (device 32), or an external plotting device on the GPIB.
starting position	a numeric expression that specifies the starting item position to be plotted.
ending position	a numeric expression that specifies the last item position to be plotted.

Why Use It?

Use RPLOT to plot relative format data expressed in user definable units that will conform to the currently defined WINDOW and VIEWPORT.

This format allows objects or symbols to be relatively described, positioned, and rotated. Plotting can then be performed directly from a file without reading the data into user memory.

What It Does

The RPLOT routine plots graphical information expressed in relative UDU format directly from the specified file(s) to the specified I/O device or by default to the screen.

How To Use It

Data to be plotted can be stored in one of two ways. A single file may contain a series of x,y coordinate pairs. In this case, the RPLOT routine requires a single file identifier, which may be name or number, and all other arguments are optional. If the I/O device number is omitted, then the plot, by default, will be made to the screen.

The second way is where the x and y coordinates are stored in separate files. In this case, the files must be referenced by number, where the first file contains x values and the second file contains y values, and the second file argument MUST be followed by a semicolon. This is necessary as it is the only way to ditinguish between single file format and dual file format. Also, since the CALL statement cannot end with a semicolon, the I/O argument which comes next is not optional in this case. Therefore, when a plot is targeted to the screen, the device address of 32 must be inserted following the y file's semicolon.

The optional range specifiers are the same for both file formats. If the optional starting position is omitted, the default starting position will be the first item, and the ending position will be the last item in the file. If only the ending position is omitted, the starting point can still be specified and the ending point will be the last item position.

Examples

```
100 SET DEGREES
110 ROTATE A
120 MOVE X1,Y1
130 CALL "RPLOT","DESK",-32
```

This program plots the contents of the file named DESK to the screen starting from a relative position defined by X1,Y1 and rotated to an angle in degrees specified by the value in A. The minus sign on the device number for the screen instructs the first coordinate be a move.

```
200 POINTER X,Y,Z$
210 IF Z$="S" THEN 250
220 MOVE X,Y
230 CALL "RPLOT",11,12;-32,1171,2194
240 GOTO 200
250 END
```

This program uses the cross hairs to place a symbol at various locations on the screen. The x data is contained in file 11, and the y data is contained in file 12. The device number 32 selects the screen, and the minus sign causes the first coordinate plotted to be a move. The data from the two files is taken from locations 1171 through 2194, for a total of 1023 coordinates.

Note that if it is desired to MOVE to location (0,0), it is not possible to make either coordinate negative to flag the move since both are zero. In this case you may solve the problem by making one value a small negative number near zero, such as 1.0E-300.

SCALE Performs direct file add, subtract, multiply, and divide operations.

Format

CALL "SCALE",fi,s,o,[i1,[i2]]

where:
fi = file identifier
s = scalar operator
o = operation code
i1 = starting point (optional)
i2 = ending point (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
scalar operator	the numeric value to be used in scaling the specified file. Can be a simple variable, a numeric expression, or a literal value.
operation code	must be a value from 1–4 where 1=Add, 2=Subtract, 3=Multiply, and 4=Divide. Can be a simple variable, a numeric expression, or a literal value.
starting point	a numeric expression that specifies the starting item position within the file to be scaled.
ending point	a numeric expression that specifies the last item position to be plotted within the file that is to be scaled.

Why Use It?

Use the SCALE routine to perform arithmetic array like operations on data directly in auxiliary memory files. This has the benefit of being very fast, similar to math operations on array variables, but can operate on much larger data arrays than possible in user memory, plus it does not require any user memory data or program space.

It can be used to scale or offset graphical data prior to plotting directly from the file without ever moving any data to main memory for processing.

What It Does

The SCALE routine performs addition, subtraction, mulitplication, or division of scalar values over entire files or a selected range of data elements within a file.

How To Use It

The SCALE routine requires three arguments: the file identifier, the scalar operand value, and the op-code. The file identifier may be the file's name or number. If many accesses are made to the file, it is recommended that the file number be used for faster access.

The operand may be any value or numeric expression. The opcode must be an integer value of 1, 2, 3, or 4. This code specifies add, subtract, multiply, or divide operations respectively.

The order of operands for the Add and Multiply operations is the following: f(i)=f(i)+S, and f(i)=f(i)*S. Note that subtraction of the scalar operand from the file element can be achieved by negating it, and division of the file element by the scalar operator can be achieved by inverting the scalar operand.

For this reason the opposite operand order was chosen for the Subtract and Divide operations which are as follows: f(i)=S-f(i), and f(i)=S/f(i). This permits the greatest flexibility in achieving the arithmetic operation desired.

Examples

```
100 CALL "SCALE",2,1.5,1
```

The scalar value of 1.5 is added to all elements of file 2.

```
100 CALL "SCALE","CURVE DATA",1,4,100,150
```

This line causes each of the elements from 100 to 500 in the file named "CURVE DATA" to be inverted with the operand chosen being 1, and the operation chosen being divide (so, f(i)=1/f(i)).

```
CALL "SCALE",1,-5,1
```

A value of -5 is added to all elements of file 1.

```
200 CALL "SCALE",F,1/N,2,100
```

This line causes each element beginning with item 100 in the file specified by F, to be effectively divided by the value in N. Multiplication is the operation selected, but N is inverted before the multiply which gives the effect of dividing by N.

Note that if divide was selected and N was not inverted, the result would be to divide N by each element.

SEARCH Performs a conditional search for strings or numerics.

Format

CALL "SEARCH",f,P,s$[,c$][,s[,i1[,i2]]] format for strings
CALL "SEARCH",f,P,n[,c$][,s[,i1[,i2]]] format for numerics

where: p = target variable for record number
s$ = string key to search for
n = numeric key to search for
c$ = search code string
s = starting character position
i1 = starting item number
i2 = ending item number

Arguments

target variable	a simple variable that will receive the record number that contains the item matching the search requirements.
string key	a string lieteral or variable containing characters that are to be matched.
numeric key	a numeric expression that is to be matched by the conditions of the search code string.
search code	a string literal or variable that contains relational characters and symbols specifying how the search is to take place.
starting position	a numeric expression that specifies the starting character position in each record.
starting item	a numeric expression that specifies the the first item in the file to search from.
ending item	a numeric expression that specifies the last item in the file to be searched.

Why Use It?

The SEARCH function enables you to locate numeric or string data items based on certain rules of comparison. You may want to do this to locate related data by search keys or to maintain sorted lists. It is also useful for locating strings when only part of the string is known by wild cards. The position of numberic data that falls within a certain range may also be lodated.

What It Does

This routine provides a great deal of flexibility in that it allows searching within a range of items, and within a specified field of the items. It also has several comparison rules that include greater than, less than, equal to, wild cards, and case symbols that may be used in any logical combination.

How To Use It

The type of search performed is first determined by the search key provided whether it is string or numeric. Numeric searches use floating point comparisons. String searches use character by character comparisons.

The search code string may be composed of any logical combination of any of the following characters:

'<' less than
'=' equals
'>' greater than
'*' relative starting position in record
'@' exact match required (to end of record or item)
'C' Case of characters must match (no case is the default)
'?' Allows '?' as wild card character in string matches

The starting character position refers to string searches and determines the field or starting position within a data item where a search is to begin. If this argument is omitted, the search will begin from the first character of each item.

The starting and ending item positions to search on are also optional. If these arguments are omitted, then all items in the file will be searched. If only the last item position is omitted, then the search will go from the specified starting position to the end of the file.

The position of the record containing the data item that matches the search requirements will be assigned to the target variable and passed back to the user. If no items satisfy the search requirements, a value of zero will be assigned to the target.

Examples

```
CALL "SEARCH",1,P,"JOHN"
```

This searches file 1 for a string that is exactly equal to JOHN. If this string is located, its position will be assigned to P. If it is not found, P will be set equal to zero.

```
CALL "SEARCH",F,N,10,">"
```

This finds the first item whose value is greater than 10 in file F and returns its position in the target variable N.

```
CALL "SEARCH","OBJECTS",P0,A$,"C?*=",2,1,100
```

This searches the first 100 items in the file named OBJECTS for a string contained in A$ that must match its case (upper and lower) but will look for the occurrence of A$ at any position with in the item starting at or after the 2nd character position. Also, any '?' characters in A$ will automatically match corresponding character positions in the data item. The position of the first matching item will be assigned to P0.

```
CALL "SEARCH",10,I,PI,100,200
```

This searches items 100 through 200 in file 10 for a value of PI. The position will be returned in I. Note that the search code is omitted and the defualt is an exact 'equal-to' match. Also the starting character position is not used for numeric searches.

SETIP Sets the current item pointer for the specified file.

Format

CALL "SETIP",F,I

where: fi = file identifier
i = new item pointer position

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
item number	must be an integer value between 1 and the the value of the Last Item Pointer plus one.

Why Use It?

To re-position the current Item Pointer preparatory to a READ or a WRITE. Allows random access to items within a file. The file may be either a random or sequential file type.

What It Does

SETIP allows the current item pointer position to be moved to any valid item in the specified file. Any subsequent READ or WRITE commands will start from this position.

How To Use It

This routine may be used with either Random or Sequential files to randomly re-position the current Item Pointer. Although a Random file is faster than a Sequential file, the SETIP routine is used the same for both. A separate Item Pointer is maintained for each file. The current Item Pointer can be set to the next available item position following the Last Item but not beyond. A WRITE can be performed from this position but not a READ.

Examples

```
100 CALL "SETIP",1,10
```

Sets the current Item Pointer in file 1 to 10.

```
CALL "SETIP",F3,1
```

Resets the current Item Pointer in file F3 to the first item.

SETLST Sets the last item pointer in a newly created file.

Format

CALL "SETLST",fi,n

where: fi = file identifier
n = new position for last item pointer

Arguments

file identifier	a numeric expression that specifies the file to be pre-extended by forcing its last item pointer to some position.
new position	a numeric expression that specifies the new last item position in the NEW file.

Why Use It?

Use the SETLST routine to reserve a section of contiguous memory that can be used by primitive commands without diturbing the file structure.

Since an artificial number of items is specified, the file manager routines for random data files may be used to access the data written to this section of memory from the DMA, data acquistion sequence, or primitive WBYTES operation.

What It Does

The SETLST routine allows the last item pointer in a random data file to be artificially set to some arbitrary position beyond its current position forcing allocation of a certain amount of contiguous memory. The memory allocated in this operation is first zeroed.

How To Use It

SETLST can only be used with random data files. The amount of memory allocated will be the number of 256 byte blocks necessary to accomodate the number of records indicated by the last item position specified. This is the product of the record size times the last item position.

It is intended that SETLST be used only after creating a new random file with some desired record length. This file must be created while memory is still contiguous. This is only guaranteed after the INIT statement has been executed. Several files may be created and pre-extended by SETLST in sequence so long as no files are deleted or dynamcially expanded during this sequence causing allocation to become non-contiguous.

If the space reserved in the file by SETLST is not to be used as a random data file, it may be easiest to create the file with a record length of 1. SETLST may then specify the last item pointer as the actual number of bytes to be reserved.

The GETADD command must be used to get the starting address in the file that is needed to access the reserved memory by the primitive routines.

Examples

```
100 CALL "INIT"
110 CALL "MCREATE",1,1,1
120 CALL "SETLST",1,1000
```

This program first initializes the memory. This makes all memory blocks contiguous. Line 110 then creates a random data file with a record length of one byte. Line 120 then sets the last item pointer to 1000. The file header of 44 bytes plus the 1000 one byte items requires a minimum of 5 blocks (1280 bytes) since the 1044 bytes will not fit in 4 blocks (1024 bytes).

```
200 CALL "INIT"
210 FOR F=10 TO 20
220 CALL "MCREATE",F,1,2
230 CALL "SETLST",F,5000
240 NEXT F
250 CALL "GETADD",10,A
260 CALL "WBYTES",A$,A
270 CALL "SETIP",10,24
280 CALL "READ",10,X
290 PRINT X
```

This program initializes the memory and then creates 11 random data files numbered from 10 to 20, each with 5,000 records of 2 bytes each. Line 250 then gets the starting physical memory address of file 10 which is used by WBYTES to write a data string from A$. SETIP is then used to select the 24th 2 byte record position in file 10. Line 280 then reads this data item into X and prints the value to the screen.

SORT Performs an alphabetical sort on string data files.

Format

CALL "SORT",F,[S,[N,[I1,[I2]]]]

where:
- fi = file identifier
- s = starting character position (optional)
- n = no. to compare (optional)
- i1 = starting point (optional)
- i2 = ending point (optional)

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
starting character	specifies the first character position within the data item to be compared. Can be a simple variable, a numeric expression, or a literal value. Defaults to the first characer if omitted.
no. to compare	specifies the maximum number of characters to compare against from the starting character position. Can be a simple variable, a numeric expression, or a literal value. Default is all characters.
starting point	a numeric expression that specifies the starting item position within the file to be sorted.
ending point	a numeric expression that specifies the last item position within the file that is to be sorted.

Why Use It?

You may wish to sort a list of names, products, items, or any string data. You can specify the field to sort on, so the list can be sorted a variety of ways for various purposes. You can also sort numeric data as long as it is positive signed.

What It Does

The SORT routine performs an alphabetic sort on the specified Random data file. The algorithm used is a classic bubble sort, which requires a minimum of memory overhead to sort in place. The comparisons are made on a character by character basis from a specified starting character position upto a specified number of characters. You may also specify the starting and ending item within the file to sort.

Since numerics are stored in binary format, they will not always be sorted correctly. This is because the sign bit causes negative numbers to be considered greater than positive numbers.

How To Use It

The file specified must be a Random file with a record length not greater than 100 bytes. By specifying the starting character position within the data item, a list can be alphabetized by different fields within the data item.

Examples

```
100 CALL "SORT",1
```

All items in file 1 are sorted in alphabetical order.

```
200 CALL "SORT",F,10,5,15,50
```

Items 15 through 50 are sorted according to the 5 characters starting in the 10th field position of each item in the file specified by F.

TBACK Writes contents of auxiliary memory to tape.

Format

CALL "TBACK"[,p]

where: p = memory page to back up (optional)

Arguments

memory page	a numeric expression that specifies which page of memory is to be backed up.

Why Use It?

TBACK allows you to back-up all or selected sections of memory with a single command. This is useful if you have data in the memory that is not in the file structure. It is also useful if you want to back-up all files in the memory without saving them one-by-one.

What It Does

The TBACK routine writes the contents of the auxiliary memory to a tape file so that it may later be restored. This does not a allow selective file back-up or restoration. It is rather like taking a snap shot of memory that can be restored at a later time. See the description of the TRESTORE command for more details on restoring memory data backed-up with TBACK.

How To Use It

The TBACK command may be called without an argument, in which case it will attempt to back-up all of memory. This means you must first mark a file large enough to hold an equivalent number of bytes. For example, if

your memory has one card of 128K bytes, then the tape file must be marked at least this large. You must FIND the file before calling TBACK.

Since for larger memory sizes it may not be possible to fit the entire memory contents on one tape file, an optional argument for TBACK is provided. This allows you to place pages of memory in separate tape files which are defined as 65,536 bytes each. This means there are exactly 2 pages per memory card. A 256K byte memory (a 2 card system) will then have 4 pages of memory which are numbered 1, 2, 3, and 4.

Also, if you know that your data is only contained within certain pages of memory, then you may selectively back-up those pages.

Examples

```
FIND 1
MARK 1,130000
FIND 1
CALL "TBACK"
```

These statements first mark a file large enough to hold all of a 1 card memory system and then the data from the entire memory is written to tape.

```
100 FIND 1
110 MARK 4,66000
120 FOR F=1 TO 4
130 FIND F
140 CALL "TBACK",F
150 NEXT F
```

This program starts by marking 4 files just larger than 65,536 bytes which holds one memory page. It then goes through a loop to save 4 pages of memory to each of those 4 tape files.

TLOAD Restores memory files from tape back to the auxiliary memory.

Format

CALL "TLOAD"[,f]

where: f = new file number (optional)

Arguments

file number	a numeric expression that specifies the new file number of the file being restored in the auxiliary memory.

Why Use It?

You must use this command to restore files saved to tape from the auxiliary memory with the TSAVE command. Since it is difficult to save programs on the tape and then load them into the auxiliary memory under program control, these commands make this operation easy and automatic.

What It Does

Program or data files saved to tape with the TSAVE command may be restored or re-loaded to the auxiliary memory with the TLOAD command. If the optional new file number argument is omitted when this routine is called, then the file will be restored at its original file number.

How To Use It

You must first FIND the tape file where the auxiliary memory file was saved with the TSAVE command. It may then be loaded back into the auxiliary memory by calling the TLOAD command. If you are restoring it to the same file location as before, then no argument is needed.

If you want to assign a new file number to the file when re-loading it, then you must supply the new file number as the only argument to TLOAD when calling it.

Examples

```
100 FOR F=1 TO 20
110 FIND F+10
120 CALL "TLOAD",F
130 NEXT F
```

The auxiliary memory files stored on tape files 11 through 30 are restored to the auxiliary memory at files 1 to 20.

```
FIND 3
CALL "TLOAD"
```

This finds tape file 3 and re-loads the file saved there to its original auxiliary memory file number.

TRESTORE Restores auxiliary memory from data saved on a tape file.

Format

CALL "TRESTORE"

Arguments

none.

Why Use It?

You must use this command to restore data to the auxiliary memory from a tape file that was saved by the TBACK command. This commands allow you to back-up and restore all program and data files or free formatted data very easily.

You may want to do this routinely each day in starting up from a power down condition, or periodically for archiving data sets or protecting against power failures.

What It Does

TRESTORE reads data from a tape file that was stored by the TBACK command and writes it back into the auxiliary memory in exactly the same locations as it was originally taken from.

How To Use It

This command requires no arguments. It will only restore data in exactly the same locations it was taken from. This information about where the data came from and how much was stored is all saved by the TBACK command. TRESTORE reads this from the header of the tape file. This makes restoration of memory completely automatic.

You must first FIND the tape file before calling TRESTORE. Only tape files with headers created by the TBACK command may be accessed by TRESTORE.

Examples

```
100 FIND 1
110 CALL "TRESTORE"
```

The contents of memory are restored from tape file 1.

```
100 FOR T=1 TO 2
110 PRINT "INSERT TAPE";T;" AND PRESS RETURN"
120 INPUT A$
130 FOR I=1 TO 4
140 FIND I
150 CALL "TRESTORE"
160 NEXT I
170 NEXT T
180 END
```

This program restores the memory from 8 separate pages of 65,536 bytes each. Each page was stored in a separate file with 4 files per tape on two tapes. The tapes may be inserted in any order since the file headers contain the information for TRESTORE to properly address the memory.

TSAVE Writes to tape the specified memory file.

Format

CALL "TSAVE",fi

where: fi = file identifier

Arguments

file identifier	a numeric expression that specifies the file that is to be saved to the current mag-tape file.

Why Use It?

The TSAVE routine allows you to save program or data files to the internal mag-tape unit for back-up purposes or for routine storage so they may be easily restored later under program control. This makes it very convenient save and restore programs to the tape when power failures or planned power-downs are contemplated.

What It Does

This routine saves a specified file in the auxiliary memory to the last found tape file. The tape file header is marked as a memory back-up file. The file number is stored as well so that it may optionally be restored automatically to the same file if desired.

How To Use It

You must first MARK a tape file large enough to store the program and then position the tape to the start of that file using the FIND statement. You then simply call the TSAVE routine specifying the file from the auxiliary memory to be saved.

If an existing tape file is to be used, it must either be NEW or previously used by the TSAVE routine. Otherwise the file must first be KILLed.

Examples

```
FIND 8
MARK 1,10000
FIND 8
CALL "TSAVE","MAINPROG"
```

This series of statements prepares a tape file and then saves the auxiliary memory file named MAINPROG to the tape file number 8.

```
100 FIND 10
110 CALL "TSAVE",113
```

This program saves file number 113 from the memory to file 10 on the tape.

WBYTES Writes data bytes from a string to a physical address.

Format

CALL "WBYTES",a$,a

where: a$ = data string
a = address

Arguments

data string	must be a string variable previously defined with the data that is to be written to memory.
address	must be an absolute address starting from 1 to 65,536 (for a 64K memory) This parameter can be a variable, a numeric expression, or a literal value.

Why Use It?

The WBYTES routine allows you to write data bytes to the memory at any physical memory address, independant of the file structure. You may wish to access the memory in this manner for special applications such as data acquisition or other purposes requiring a more primitive and straight forward method of accessing the auxiliary memory.

This is also the fastest method of access with the least amount of overhead which may be critical to some time sensitive applications.

What It Does

The WBYTES routine writes a series of data bytes from a string to the auxiliary memory starting at a specified absolute address. This gives the user complete access to every available memory location, including those normally used by the file manager for the directory and allocation map.

How To Use It

The WBYTES routine requires only two arguments. The first is the string of data bytes to be written, and the second is the starting address in memory to begin writing.

If this routine is used in conjunction with the file structure, then care must be taken not to damage any files or system data. The only way this can be done is by using the GETADD and SETLST routines. The GETADD routine enables you to find the beginning physical address of a given file. The SETLST routine enables you to pre-extend a file with a certain number of contiguous memory blocks. See pages 3-20 and 3-102 for more details on these routines.

The user assumes all responsability with this routine to keep track of how many bytes are written and where.

Example

```
100 CALL "WBYTES",A$,100
```

The string of bytes in A$ will be written to the memory starting at the 100th memory location. The next available address would be 100 plus the length of A$.

WRITE Writes numeric and/or string data to Auxiliary Memory files.

Format

CALL "WRITE",fi,[i;]d1[,d2,...]

where: fi = file identifier
i = item number
d_n = data list

Arguments

file identifier	can be either a file number represented by a numeric expression, or a file name expressed as a string variable or literal.
item number	a numeric expression that specifies the item position for both random and sequential files. This optional parameter, if used, MUST be followed by a semicolon to flag its presence.
data list	can be any number of data items: numeric expressions, array variables, string literals, and variables. There must be at least one data item in the list.

Why Use It?

Use the WRITE routine to create data files for numeric and/or string data. The default random data file is analagous to a subscritped array, but it accepts both numeric and string data types and, of course, can accept much larger array sizes than possible in the 4050 system memory.

The optional sequential file type may also be referenced like an array, with the file's item pointer corresponding to the subscript of an array element. The sequential file will be more memory efficient for string, or mixed string/numeric data, but is slower to access randomly.

What It Does

It is used to write data items to a specified file starting at the position of the current-item-pointer, or the optional user supplied starting item position. Numeric arrays are written one element per record, or item position, and the current item pointer is updated once for each item in the array.

How To Use It

If the file referenced has not been previously created, it will be automatcally created as a random file with a record length of 8 bytes when the first data item in the list is numeric, or 72 bytes if the first data item is a string. A file name is not required, but if one is desired, it must be assigned with the MCREATE command. Alternate record sizes from 1-255 may also be assigned with MCREATE.

Record sizes smaller than 8 bytes cause numeric values to be saved in a reduced precision format. This is useful when the normal 14 digit (8 byte) accuracy is not needed. Data files may contain both numeric and string data. However, the data type is not stored with the data item and the user must keep track of the order of the data types.

You may optionally create a sequential file prior to the first WRITE. This must also be done with the MCREATE command.

A unique item pointer is maintained for each data file. Each time a data item is written to a file, the current item pointer for that file is automatically advanced by 1. The last item position is updated whenever the old last item position is passed. However, a new last item position (end of file) is created every time a write is performed to a Sequential file. Any data items after this point from previous writes will be lost.

The file can be referenced either by number or by name. However, for repeated accesses, it is faster to access the file by number. If the file is remembered by name, then use the MOPEN command to find the file number from the name.

Examples

```
100 CALL "WRITE",5,A,X,B$
```

Three data items: A, X, and B$ are written to file 5. The default file type is Random and the default Record Size is 8 bytes.

```
100 CALL "WRITE",F2,"LITERAL STRING"
```

The string in quotes is written to file F2 and a default record size of 72 is established.

764MEM Prints the current firmware revision message.

Format

CALL "764MEM"

Arguments

none.

Why Use It?

Use this routine to find the firmware version number and code for your file manager. This may be useful when reporting problems to the factory or in deciding if an upgrade is available and warranted.

The firmware version number is also referenced at times in this manual in the firmware release notes. This may help you decide what features you have and what the known problems are for you firmware version.

What It Does

This routine prints to the screen the firmware version number for your Auxiliary Memory file manager ROM pack.

How To Use It

There are no arguments for this routine, you simply CALL "764MEM", and the firmware version number will be printed to the screen.

Section 4

Direct Memory Access Card Options

Introduction

This section explains the use of the DMA (Direct Memory Access) options. These are plug-in cards that allow data from external devices or host computers to transfer data to and from the Auxiliary Memory at very high speeds (upto 750K bytes/second). Option 11 accepts 8 bit parallel data from IEEE-488 (GPIB) format compatible interfaces. Option 12 accepts 8 or 16 bit parallel data from a general purpose interface using 4 control lines.

The routines used to operate the DMA functions will be described first, and then the two options will be discussed separately. Several examples are also included.

Refer to Section 1 for installation instructions of the DMA options.

DMA8/DMA16 Set the Auxiliary Memory in the DMA mode.

Format

CALL "DMA8",a
CALL "DMA16",a

where: a = beginning address

Arguments

beginning address	a numeric expression that specifies the beginning physical address in the auxiliary memory where data is to be stored or retrieved.

Why Use It

Use DMA8 to set up the memory to receive 8 bit data bytes from an external device in a direct memory transfer. Use DMA16 to set up the memory to receive 16 bit data words from an external device in a direct memory transfer.

What It Does

These commands activate the DMA for standard 8-bit (byte), or 16 bit (word) data transfers. The argument required indicates the starting address in the Memory unit where the data being transferred is to be stored or retrieved in sequential order.

How To Use It

If you are using the Auxiliary Memory strictly for data acquisition and do not need the file structure used by the file manager, you may select any physical address in the memory as the beginning data address.

You may then use the RBYTES and WBYTES routines described in Section 3 to access the DMA data.

If you wish to perform DMA transfers and preserve the file structure, you must use the GETADD routine to give you the starting address of the file you are using for the DMA data. This may also be used in conjunction with the RBYTES and WBYTES routines. You will also want to understand how to reserve some contiguous data space data space using the SETLST routine to pre-extend a file. Refer to Section 3 for details on all these routines before attempting to perform DMA transfers within the file structure.

Examples

```
100 CALL "DMA8",500
```

The DMA is activated to store data from the external device sequentially in the memory starting at location 500, or to retrieve data sequentially from the memory unit beginning at location 500.

```
CALL "DMA16",0
```

This sets the beginning memory address to zero and enables the interface to initiate a 16 bit direct memory transfer.

```
100 CALL "GETADD",1,A
110 CALL "DMA8",A
```

These two lines enable 8 bit DMA to occur starting at the first data address in file 1 as returned by the GETADD routine.

DMACLR Clears the DMA mode and enables 4050 to access the memory.

Format

CALL "DMACLR"

Arguments

none.

Why Use It

Use DMACLR to end a DMA transfer and return control of the Auxiliary Memory to the File Manager. While in the DMA mode, it is not possible to access the memory with the file manager commands.

What It Does

DMACLR clears the line which gives control of the memory to the DMA interface.

How To Use It

Simply CALL this routine to return control of the memory to the 4050 after having initiated a DMA sequence with either DMA8 or DMA16. No arguments are required.

Example

```
100 CALL "DMACLR"
```

This clears the DMA mode and allows the 4050 to read or write to the memory with any of the file manager commands.

DMASTA Returns DMA status information.

Format

CALL "DMASTA",x

where: x = target variable

Arguments

target variable a simple numeric variable that will receive the value representing the status of the DMA interface.

Why Use It

Use this routine to see if DMA mode is currently active or not.

What It Does

DMASTA enables the HOST to check the status of the DMA to see if it is active or not. If X=1, the DMA is active. X=0 indicates that the DMA is not active or that a transfer has been terminated.

How To Use It

This routine requires a simple scalar variable as its only argument that can receive the value of the DMA status. The value returned will be 0 or 1, indicating not active or active. You may wish to use this in a loop after activating the DMA to see when you are allowed to continue with the part of the program that is to access the memory from the 4050.

Examples

```
100 CALL "DMA8",100
110 CALL "DMASTA",X
120 IF X=1 THEN 110
130 PRINT "DMA TRANSFER COMPLETED"
```

The DMA has been activated and a loop is created to check to see if the DMA transfer has finished. When X=0, signifying that the DMA is inactive, a message is printed to the 4050 screen.

```
100 CALL "GETADD",6,X
110 CALL "DMA16",X
120 CALL "DMASTA",S
130 IF S<>0 THEN 120
140 CALL "RBYTES",A$,X,100
```

The beginning absolute address of file 6 in the Memory unit is obtained, and then used as the starting address for a DMA transfer. Line 120 checks the status and waits for completion of the DMA transfer. Line 140 then reads 100 bytes from the same beginning address.

Option 11 - IEEE-488 8 Bit Parallel Direct Memory Access Card

The 8 bit IEEE-488 compatible DMA (Direct Memory Access) option for the TransEra Memory module allows data transfers to or from the memory over the industry standard IEEE-4888 (GPIB) bus. Data may be stored or retrieved by the user in an unformatted mode that either honors or ignores the memory's file structure. This is useful for certain data aquisition applications or for special diagnostics.

The General Purpose Interface Bus (GPIB) allows the DMA to communicate with any external device which has input/output (I/O) compatibility with the standards of the IEEE Standard #488-1975 document. This standard describes a byte-serial, bit parallel interface system. For further information on how the GPIB is utilized by the Tektronix 4050 series computers, please refer to Appendix C (Interfacing Information) in the Tektronix 4050 Series Reference Manual.

In the following descriptions and explanations, the term DMA USER refers to the device/person reading or writing data to the DMA over the GPIB bus. The term HOST COMPUTER, or just HOST, refers to the Tektronix 4051/4052/4054 to which the Memory unit with the DMA Option 11 is connected.

All data transactions with the DMA Option 11 occur over the GPIB and are similar to those done with any other peripheral device on this bus. The DMA has been designed to respond to the General Purpose Interface Addresses 9 and 10, and therefore all other devices on the bus should have different addresses.

With the DMA configured as a device on the GPIB, it can be interrogated by the DMA USER by using the PRINT@, INPUT@, WBYTE, and RBYTE commands. Detailed information on these commands can be obtained from the Tektronix 4050 Series Reference Manual under Section 7, Input/Output Operations. Reading from device 9 will return the status of the DMA (whether or not it has been activated from the HOST COMPUTER). Writing a "0" or a "1" at device 9 indicates, from the USER, whether or not a valid data transfer is to occur. Writing to device 10 stores data to the memory through the DMA and reading from this device will retrieve data from the memory.

These operations, performed by the USER, along with the corresponding WBYTE and RBYTE commands are outlined in the following table.

DEVICE #9: STATUS/DATA VALID

Command	Meaning
PRINT @9:"0"; (WBYTE @41:0;)	Terminate Data transactions. Subsequent data on the GPIB bus is no longer valid for the DMA.
PRINT @9:"1"; (WBYTE @41:1;)	Enables the Option 11 and indicates that subsequent data on the GPIB is valid.
INPUT @9: X$ (WBYTE @73: RBYTE X)	This will give the binary value of 1 or 0 indicating whether Option 11 has been activated from the HOST. A "1" indicates that the DMA has been activated and transfers to or from the memory are possible. A "0" indicates that the memory is not available for DMA transactions.

DEVICE #10: DATA READ/WRITE

Command	Meaning
PRINT @10:A$; (WBYTE @42:X)	Stores A$ to the memory through the DMA.
INPUT @10: A$ (WBYTE @74: RBYTE A)	Retrieves data from the memory and stores the result in A$.

These commands will be further illustrated later in several examples.

The command PRINT @9:"0"; from the USER terminates any DMA transaction, and subsequent transfers must be preceeded by reactivating the DMA by the call statement CALL "DMA8", at the desired address.

Data Transfer Examples

Example 1: WRITING A STRING TO THE DMA

HOST	DMA USER
100 CALL "DMA8", 100	100 INP @9:X$
110 CALL "STATUS",A	110 IF ASC(X$)=0 THEN 100
120 IF A=1 THEN 110	120 PRINT @9:"1";
130 PRINT "TRANSFER DONE"	130 PRINT @10: A$;
	140 PRINT @9:"0";

In this example, the HOST COMPUTER activates the DMA to initiate transfers at memory location 100. The HOST's program then checks the DMA's status to see if the transfer initiated by the DMA USER has been completed. The DMA USER first checks to see if the Memory has been set up to receive a DMA transfer. This status is received as a binary 1 or 0, and must be converted to an ASCII character to be checked.

When it has been verified that the DMA has been activated, the USER then indicates that a valid transfer is to take place by printing a "1" to device 9. It should be noted that this must be a string character followed by a semicolon. The semicolon suppresses the carriage return from being sent to the DMA, which would then be stored in the memory unit. In line 130, the DMA USER then writes an ASCII string to the memory.

Any number of strings can be written without re-activating the DMA if line 140 is not executed. All subsequent strings stored to the DMA will be written in sequential order in the Memory unit. As mentioned, the semicolon following the command in line 130 inhibits a <CR> (carriage return) from being sent. If a carriage return at the end of the string is desired, then the semicolon should be omitted.

Example 2: READING A STRING FROM THE DMA

HOST	DMA USER
100 CALL "DMA8", 100	100 INPUT @9:X$
110 CALL "STATUS",A	110 IF ASC(X$)=0 THEN 100
120 IF A=1 THEN 110	120 PRINT @9:"1";
130 PRINT "DONE"	130 INPUT @10:B$
	140 PRINT @9:"0";

This example is identical to example 1 except for line 130 of the DMA USER program. INPUT @10:B$ will read ASCII data from the Memory unit through the DMA to the target string variable B$. Only one string can be read from the DMA at a time without re-initialization, therefore the target string variable must be dimensioned large enough to accept the entire data transfer. A single transfer will terminate when either a <CR> (carriage return) is read or until the target string variable is full.

Example 3: WRITING DATA TO THE DMA

HOST	DMA USER
100 CALL "DMA8", 100	100 INP @9:X$
110 CALL "STATUS",A	110 IF ASC(X$)=0 THEN 100
120 IF A=1 THEN 110	120 PRINT @9:"1";
130 PRINT "TRANSFER DONE"	130 PRINT @10: A
	140 PRINT @9:"0";

This example is identical to Example 1 with the exception that a numeric variable (A) is being transferred instead of a string variable (A$). This can be a scalar or an array variable. Any number of variables (scalars or arrays) can be written without re-activating the DMA if line 140 is not executed. All subsequent data stored to the DMA will be written in sequential order in the Memory unit.

Example 4: READING DATA FROM THE DMA

HOST	DMA USER
100 CALL "DMA8", 100	100 INPUT @9:X$
110 CALL "STATUS",A	110 IF ASC(X$)=0 THEN 100
120 IF A=1 THEN 110	120 PRINT @9:"1";
130 PRINT "DONE"	130 INPUT @10: B
	140 PRINT @9:"0";

This example in identical to Example 2 with the exception that a numeric variable (B) is being read instead of a string variable (B$). As opposed to reading a string variable, several consecutive reads are possible. Each array variable will be filled to its dimensioned value, and each additional read will retrieve data from the memory in the same sequential order that it was written.

For example, assume that the values from 1-100 were stored in the Memory beginning at location 100 and that the command CALL "DMA8",100 has been executed from the HOST COMPUTER to set up the memory for a DMA transfer. We then execute the following program:

```
100 DIM A(10),B(10),C(79)
110 PRINT @9:"1";
120 INPUT @10: A,B,C,D,E
130 PRINT @9:"0";
140 END
```

The values from 1-10 would be read into array A, 11-20 into B, 21-99 into C, 100 into D and extraneous data would be read into E (since there were only one hundred valid numbers).

Option 12 - 8/16 Bit General Purpose DMA

The Option 12 General Purpose Parallel DMA Interface uses four hand-shake/control lines are that necessary for operation of the DMA. They are: Clock (CLK), Directon (DIR), Data Valid (DAV), and DMA Available (DMAAVL). The CLK, DIR, and DAV lines must be supplied by the user whereas the DMAAVL is output from the DMA device. The following is a description of these lines:

CLOCK (CLK)	Clocks the data (8 or 16 bits per clock pulse) on the DMA bus into the memory or initiates a read from the memory on the rising edge of the signal.
DIRECTION (DIR)	Indicates which direction the transfer is to occur: 1 = Retrieve from Memory, 0 = Store to Memory.
DATA VALID (DAV)	Indicates whether there is valid data on the DMA bus. A low level (0) enables the option 12 and indicates that subsequent data on the DMA bus is valid. A high level (1) disables the device. A low-to-high transition on this line will terminate all DMA transactions. Subsequent transfers can be made without re-initiating the DMA when this control line is held low.
DMA AVAILABLE (DMAAVL)	This line, output from the option 12, tells whether the option 12 has been activated. A high level (1) indicates that the DMA is disabled and that the memory is not available for DMA transactions. A low level (0) indicates that the DMA is active and transfers to or from the memory are possible.

The DIR and DAV lines must be set and stable when the DMA is initiated from the 4050. Please refer to the timing diagrams at the end of these instructions.

Support Routines

Information that has been transferred to the memory can be read by the 4050 using the CALL "RBYTES" routine. Information to be transferred out from the memory can be first stored by the CALL "WBYTES" routine. These two routines read and write data sequentially in the memory.

When data is to be accessed in the Auxiliary Memory within the file structure such that no files are disturbed, then you must make use of the GETADD, and SETLST routines. All these routines are explained in Section 3.

Example 1:

```
100 A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
110 CALL "WBYTES",A$,500
120 CALL "DMA8",500
```

The DIR line has been set to retrieve data from the memory. A$ is given a value and stored in the memory beginning at location 500. The Option 12 DMA is then activated to sequentially read this information from the Memory unit.

Example 2:

```
100 CALL "DMA16",500
110 CALL "DMASTA",S
120 IF S<>0 THEN 110
200 CALL "RBYTES",A$,500,50
```

The DIR line has been set to store data into the memory. The Option 12 is activated to store the in-coming data starting at location 500. After the desired data is transferred, the user terminates the transfer and the data (50 bytes in this example) is read from the memory into the target string variable A$.

Signal Terminations And Loading

The standard termination for all lines except CLK is 3K ohms to +5VDC and 6.2K ohms to GND. The CLK line termination is 220 ohms to +5VDC and 330 ohms to GND. All signals to the DMA are loaded by no more than one standard TTL load.

Connector

The Option 12 DMA uses a D-Type, 24 pin, female connector which has been configured to allow both 8 and 16 bit parallel transfers. See Fig. 1.

When the routine CALL "DMA8" is initiated from the 4050, all 8-bit (byte) transfers are made over data lines 0 through 7. During these transfers, data lines 8 through 15 are three-stated.

Section 5 - Memory Architecture

Introduction

This section discusses the organization of the memory for those who wish to access the memory using the primitive read/write commands without disturbing the file structure. The space reserved for System Memory is outlined, the procedure for reserving contiguous file space is discussed, and some methods for doing data acquisition are presented.

Memory Organization

The memory is divided into Bytes, Blocks, and Pages. A Byte is the smallest addressable unit and contains 8 bits (binary digits). A Block contains 256 bytes, and a Page contains 256 blocks (65,536 bytes). The File Manager allocates memory a block at a time.

Upon power-up for the first time, or whenever CALL "INIT" is executed, the file manager will go through a process of counting the number of memory cards installed to determine how much memory is available. It then pre-allocates a certain number of blocks for use by the directory, the link map, and some System data bytes.

The System data bytes always occupy the first 16 bytes of memory. The directory immediately follows these bytes and uses the rest of the first block plus the following block for a total reserved space of 496 bytes. A link map immediately follows the directory.

The Auxiliary Memory File Manager employs a Link Map for allocating blocks of memory. The Link Map is a series of 2-byte pointers, each one corresponding to a block of memory in the system. These pointers are initially set to zero, indicating all blocks of memory are free and available to be allocated. Since each block in the link map contains 128 pointers that reference 128 blocks, four of these link map blocks are required for each 128K byte memory card installed.

When a block is reserved or allocated to a file, the file manager scans the link map for a zero pointer. If the 10th pointer is free, this means the 10th block is free. It is allocated by setting the most

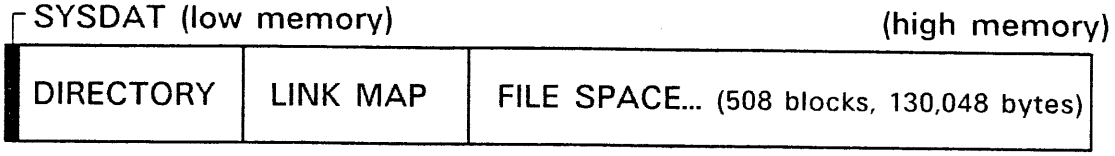
significant bit of the link pointer high. If a block was previously assigned to that file, its link pointer is then updated to point to the new block assigned. This forms a chain with the link pointer of the last block in the chain always having its MSB set high.

Directory Structure

As stated, the directory has 496 reserved bytes. At 2 bytes per entry, this will accomodate 248 file entries. Each time a file is created, an entry for it is made in the directory. This entry consists of a pointer to the first block of memory assigned to that file. The first 44 bytes of that block is the file header which contains the remainder of the information about the file used by the directory.

The number assigned to a file determines its position in the directory. When file 5 is created, it will occupy the 5th directory entry. File numbers greater than 248 will cause dynamic allocation of 1 or more additional memory blocks to the directory. Each additional block allocated will accomodate 128 entries. The first extra block will be for file numbers 249-376. Unused file numbers have null directory entries. These structures are diagramed below:

Memory Map (128K Memory)

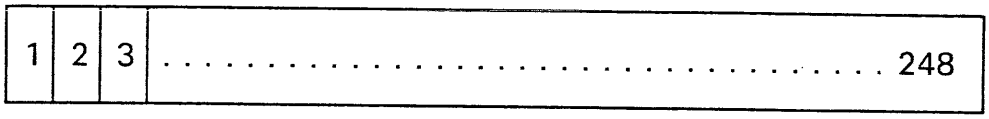


File Manager System Data Bytes (first 16 bytes of memory)

NOF(2)	PAGES(2)	NFB(2)	PWRID(2)	UNIT(2)	Free(6)

NOF	Number of Files
PAGES	Number of Pages of installed memory
NFB	Number of Free Blocks
PWRID	Power-up ID
UNIT	Address of the currently active file manager unit
Free	Un-used system data bytes

Directory Entries (file entries 1-248 reserved at addresses 16-511)



Link Map (4 blocks, 1024 bytes reserved at addresses 512-1535)

Block 1 1-128	Block 2 129-256	Block 3 257-383	Block 4 385-512

File Header (44 bytes)

CIA(3)	CIN(3)	LIN(3)	FSIZE(3)	FFLAGS(1)	RECSIZ(2)	NAME(28)+1

CIA	Current Item Address
CIN	Current Item Number
LIN	Last Item Number
FSIZE	File Size
FFLAGS	File Flags (Attribute Bits)
RECSIZ	Record Size
NAME	File Name

File Flags (8 bits)

OPEN	RAND	SEQU	PROG	PACK	SGND	*****	PROT

OPEN	Valid open file flag
RAND	Random file type
SEQU	Sequential file type
PROG	Program file type
PACK	Packed integer data format
SGND	Signed integer data format
*****	not used
PROT	Write Protect status bit

Contiguous File Space

The file manager allocates 1 block at a time as needed by a file. If you create 2 files in succession, each will be one block long. If you then write data into the first file causing it to dynamically expand, its next allocated block will come after the second file's first block. This means that normally you cannot expect a given file to have more contiguous file space than 1 block (256 bytes). Unless you understand the file system and take specific measures to force or deliberately arrange for contiguous file space, it probably won't be.

Of course, under normal conditions you will have no need for a large segment of contiguous file space. The file manager takes care of all file pointers and chains together series of non-contiguous blocks so that they appear contiguous. However, for data acquisition, or when using the WBYTES routine to transfer long strings of data into memory, you will need to guarantee actual contiguous file space. This will enable you to use the file system at the same time as your direct access Memory space.

Knowing that after CALL "INIT" is executed all blocks are free for allocation, you can then create a single file and write data into it to force it to some maximum size with a contiguous chain of blocks. Another way to accomplish the same thing is to use the SETLST routine after creating the file to artificially pre-extend the file to some desired length. This is faster and easier than using a series of data writes to fill the file. SETLST will fill the file with zero or null data. This routine is described in greater detail in Section 3.

Direct File Access

The GETADD routine described in Section 3 can be used to conveniently return the starting physical address of any file. The address returned is the first data location after the file header. If you have need to directly access the header information, then subtract the length of the file header from the address returned by GETADD to point at the beginning of the header or the first block assigned to the file. The length of the header is 44 bytes in firmware version 5. You can also compute the beginning of the header knowing that each file header address begins on an even block boundary divisible by 256.

Using the starting data address of a file obtained from GETADD, you can perform direct file access with the primitive I/O commands RBYTES and WBYTES, or the DMA routines DMA8 and DMA16, or with other ROM packs programed to access the Memory directly such as the TransEra A/D converters.

Another routine which will give you direct file access to selected data items within a file is the GETIA routine. This returns the physical item address of a given item position. You may then use any of the methods mentioned above to access the Memory using this address.

Examples

```
100 REM - EXTERNAL DMA ROUTINE
110 CALL "INIT"
120 CALL "MCREATE",1,"DIRECT ACCESS FILE",1,1
130 CALL "SETLST",1,10000
140 CALL "GETADD",1,A
150 CALL "DMA8",A
160 CALL "DMASTA",S
170 IF S<>0 THEN 150
180 DIM A$(10000)
190 CALL "RBYTES",A$,A,10000
200 PRINT A$
```

This routine first initializes the Memory to guarantee that contiguous space is available. It then creates a random data file with a record length of 1. The SETLST routine then specifies a last item position of 10000 which gives exactly 10000 bytes (since the record length is 1). GETADD then returns the starting data address of this file so that it can be used by DMA8. The DMA hardware then takes over and line 150, DMASTA, monitors the status, passing control to RBYTES in line 180 when the transfer is complete. RBYTES also uses the address obtained from GETADD and reads 10000 bytes which are then printed on the screen.

```
100 REM A/D DATA ACQUISTION WITH TRANSERA 752-ADC
110 CALL "INIT"
120 CALL "MCREATE",1,"A/D DATA",1,2
130 CALL "SETLST",1,5000
140 CALL "GETADD",1,A
150 CALL "BURST",A,5000,.001,1,0,1
160 CALL "MAXI",1,M,P
170 PRINT M,P
```

This program creates a random file with a record length of 2. SETLST then sets the last item position at 5000. At 2 bytes per record this again gives 10000 bytes of contiguous data space. The starting file address used by GETADD is used by the A/D routine BURST. After taking 5000 two byte (12 bit) samples, the MAXI routine is then called to return the maximum value and its position of the data acquired. This demonstrates the value of using the file structure for data acquisition.

```
100 REM - DIRECT ITEM ACCESS
110 CALL "INIT"
120 CALL "WRITE",10,"ITEM NUMBER ONE","ITEM NUMBER TWO"
130 CALL "GETIA",10,2,X
140 CALL "RBYTES",A$,X+12,3
150 PRINT A$

RUN

TWO
```

This program shows how to directly access a particular item within a file. After writing two string items to file 10, the item address of the second item is returned by GETIA in line 130. Rbytes is then used to read that item starting with its 12th character and reading the last 3. As you can see, this prints only the word "TWO" on the screen when the program is run.

Appendix A - Command Summary

CALL "APLOT",fi[,[-]i0[,i1[,i2]]] ...plots UDU format data to screen
CALL "APLOT",f1,f2;[-]i0[,i1[,i2]]

fi = file identifier (number or name) of file with x,y pairs
f1 = file of x values
f2 = file of y values
i0 = I/O device address
i1 = starting data record to plot
i2 = last data record to plot

APPEND n$,l[,i] ...file manager key word, functions same as CALL "MAPPEND"

n$ = name of program
l = target line to append to
i = optional line increment

CALL "CONCAT",f1,f2 ...combines 2 files into 1

f1 = file to be added to
f2 = file to be added

CALL "COPY",f1,f2 ...makes another copy a file creating a new one

f1 = file to copy
f2 = new file

DIR [i0] ...file manager key word, functions same as CALL "DIR"

[i0] = optional device address for output of directory

CALL "DUNIT" ...selects the alternate file manager

Note: MUNIT and DUNIT are not used if the Auxiliary Memory is the only file manager present. If another file manager is in the system, MUNIT must be called before each series of commands to the Auxiliary Memory, and DUNIT must be called before each series of commands to the other file manager.

CALL "EXCLUD",I ...sets exclusion level for remarks in programs loaded
I = exclusion level

CALL "FILHDR",f,H$ ...returns directory listing for specified file
f = file number
H$ = target string for file information

CALL "FILTYP",fi,t ...get the file type code
fi = file identifier, either file number or name
t = target variable for file type

CALL "FPLOT",fi[,i1[,i2]] ...plots GDU format data to screen
CALL "FPLOT",f1,f2;i1[,i2]
fi = file identifier (number or name) of file with x,y pairs
f1 = file of x values
f2 = file of y values
i1 = starting data record to plot
i2 = last data record to plot
Note: negative valued GDU data flags moves

CALL "GETADD",fi,a ...get the starting physical address of a file
fi = file identifier, either file number or name
a = target variable for pshysical address of first data byte in file (just after header)

CALL "GETIA",fi,a ...get the current physical item address
fi = file identifier, either file number or name
a = target variable for current phsysical item address

CALL "GETIP",fi[,I] ...get the current item pointer
fi = file identifier, either file number or name
I = target variable for item pointer

CALL "INIT" ...initialize the Auxiliary Memory, erase all files

CALL "INSERT",fi,i ...expands file at specified record position
fi = file identifier, either file number or name
i = item number to be inserted

KILL f$ ...delete a file referenced by name
f$ = name of file to delete

CALL "KILL",fi[,f2[,f3[,...]]] ...delete a file or list of file numbers
fi = file identifier, either file number or name
f2,f3,... = list of file numbers to be deleted

CALL "LSTIP",fi,[I] ...get the last item position of a file
fi = file identifier, either file number or name
I = target variable for last item pointer

CALL "MAPPEND",fi,I[,i] ...append a program from Memory to existing program
fi = file identifier, either file number or name
I = target line to append to
i = optional line increment

CALL "MAXI",fi,M,P[,i1[,i2]] ...find maximum value and its position
fi = file identifier, either file number or name
M = target variable for maximum value
P = target variable for position of maximum value
i1 = optional starting position in file
i2 = optional ending position in file

CALL "MCHECK" ...performs a diagnostic check of the memory

CALL "MCREATE",f[,n$],t,r ...create a file with specified name and attributes
f = file number
n$ = optional file name
t = file type (1=random,2=sequential,3=program)
r = record size (normally 0 for program and sequential file types)

Note: MCREATE replaces the former OPEN command. Besides the new name for the command, it differs only in that it allows an optional file name. This command can also be used to re-name a file that is already created. It also has the affect of setting the current item pointer in an existing file to one.

Integer Data Types
Integer data types are specified by the record number being 1 or 2 for 8 or 16 bit integers respectively. Signed integers are specified by making the record value -1, or -2.

CALL "MCREATE",f[,n$],1,-1 (signed integer data type range -128,127)
CALL "MCREATE",f[,n$],1,-2 (signed integer data type range -32768,32767)

CALL "MCROSS",fi,v,l[,n] ...locates values in a file that 'cross' a threshold
fi = file identifier, either file number or name
v = threshold value
l = target for location of cross
n = optional number of cross desired (default=1)

CALL "MCSUM",S,p ...performs checksum on memory
S = target for chekcsum remainder
p = page count (number of pages to sum)
may be useful for verifying memory before and after backup and restore operations from tape

CALL "MDIF2",fi ...performs 2 point differentiation on file specified
fi = file identifier, either file number or name
b(t)=a(t+1)-a(t) for t=1,2,...,n-1
b(n)=b(n-1)
where a=original file data, b=resulting file data

CALL "MDIF3",fi ...performs 3 point differentiation on file specified
fi = file identifier, either file number or name
b(1)=(-3*a(1)+a(2)-a(3))/2
b(t)=(a(t+1)-a(t-1))/2 for t=2,3,...,n-1
b(n)=(a(n-2)-4*a(n-1)+3*a(n))/2
where a=original file data, b=resulting file data

CALL "MINI",fi,M,P[,i1[,i2]] ...find minimum value and its position
fi = file identifier, either file number or name
M = target variable for minimum value
P = target variable for position of minimum value
i1 = optional starting position in file
i2 = optional ending position in file

CALL "MINT",fi ...performs integration on the specified file
fi = file identifier, either file number or name
b(1)=0
b(t)=b(t-1)+(a(t-1)+a(t))/2 for t=2,3,...,n
where a=original file data, b=resulting file data

CALL "MLINK",fi,l ...load a program from Memory but keep all variables
fi = file identifier, either file number or name
l = program line number to begin execution at

CALL "MOLD",fi ...load a program from Memory
fi = file identifier, either file number or name

CALL "MOPEN",F,a$ ...finds and returns the file number for the named file
F = target variable for file number
a$= string literal or variable containig file name whose file number is to be found

CALL "MPLOT",fi[,[-]i0,[i1,[i2]]] ...plots absolute UDU data
CALL "MPLOT",fi,f2;[-]i0[,i1[,i2]] ...plots absolute UDU data
fi = file identifier (number or name) of file with x,y pairs
f1 = file of x values
f2 = file of y values
[-]i0 = I/O device address (minus sign signals move on first coordinates)
i1 = starting data record to plot
i2 = last data record to plot

CALL "MSAVE",f[,n$] ... save current program in file number specified
f = file number
n$ = optional file name

CALL "MSAVE",n$ ...save the program with specified name
n$ = file name (file number will be next lowest unused number)

CALL "MTEST",n,... ...diagnostic test routines
n=1, address counter test, CALL "MTEST",1,c1,c2,c3
n=2, status register test, CALL "MTEST",2,t0,t2
n=3,

CALL "MUNIT" ...select the Auxiliary Memory as the current file manager

CALL "NXTFIL",F ...select the next unused file number
F = target for next available file number

OLD n$ ...file manager key word, functions same as CALL "MOLD"
n$ = name of program

ON EOF(0) ...traps end of file conditions
Shares the same device address as the internal mag tape. If both memory and tape files are to be trapped for end of file, then two ON EOF statements are required, one preceding each input.

CALL "PROT",fi ...assign write protect status to file
fi = file identifier, either file number or name

CALL "RBYTES",A$,a,n ...reads a string of bytes from Memory
A$ = target string variable for data to be read
a = starting physical address in memory to begin read from
n = number of bytes to read

CALL "RDELET",fi,i ...deletes record and compresses remainder of file
fi = file identifier, either file number or name
i = item number to be deleted

CALL "READ",fi,[r;]v1[,v2,...] ...reads data items from Memory
fi = file identifier, either file number or name
r = record number to read from

CALL "RMPLOT",fi[,i0[,i1[,i2]]] ...random plot from absolute data
CALL "RMPLOT",f1,f2;i0[,i1[,i2]]
fi = file identifier (number or name) of file with x,y pairs
f1 = file of x values
f2 = file of y values
i0 = I/O device address
i1 = starting data record to plot
i2 = last data record to plot

CALL "RPLOT",fi[,[-]i0[,i1[,i2]]] ...plots relative UDU data
CALL "RPLOT",fi,f2;[-]i0[,i1[,i2]]
fi = file identifier (number or name) of file with x,y pairs
f1 = file of x values
f2 = file of y values
[-]i0 = I/O device address (minus sign signals move on first coordinates)
i1 = starting data record to plot
i2 = last data record to plot

SAVE N$ ...file manager key word, functions same as CALL "MSAVE"
n$ = name of program

CALL "SCALE",fi,s,o,[i1,[i2]] ...add, subtract, multiply, or divide a value over a file

CALL "SEARCH",f,P,{s$,n}[,c$][,s[,i1[,i2]]] ...match numbers or strings
p = target variable for record number of key item found
{s$,n} = string or numeric key to search for
c$ = search code string composed of any logical combination of the following list of characters;
'<' less than
'=' equals
'>' greater than
'*' relative starting position in record
'@' exact match required (to end of record or item)
'C' Case of characters must match
'?' Allows '?' as wild card character in string matches
s = starting character position in each record
i1 = starting item number
i2 = ending item number

CALL "SETIP",fi,i ...set the current item pointer
fi = file identifier, either file number or name
i = position to set the item pointer to in file

CALL "SETLST",fi,n ...set last item pointer to a new position
fi = file identifier, either file number or name
n = new position for last item pointer

CALL "SORT",fi,[s,[n,[i1,[i2]]]] ...sort a file in alphabetic order
fi = file identifier, either file number or name
s = starting character (field) position
n = number (field length) to sort on
i1 = starting item position
i2 = ending item position

CALL "TBACK"[,p] ...writes contents of memory to tape
[p] = optional page number to back up (65536 bytes)
entire memory is backed up to tape if 'p' is omitted

CALL "TLOAD"[,f] ...reads memory file from tape and stores in back memory
f = new file number for file to be loaded from current tape file

CALL "TRESTORE" ...restores memory from current tape file
restores exactly the amount saved by TBACK

CALL "TSAVE",f ...writes to tape the specified memory file
fi = file identifier, either file number or name

CALL "UNPROT",fi ...remove write protect status from a file
fi = file identifier, either file number or name

CALL "WBYTES",a$,a ...write a string of bytes to a file
a$ = data string to write to file
a = physical address in the Memory to begin write at

CALL "WRITE",fi,[r;]v1[,v2,...] ...write data items to a file
fi = file identifier, either file number or name
r = record number to write to

CALL "764MEM" ...firmware revision message printer

Appendix B - Error Messages, Causes and Corrections

Error Message	Cause	Correction
Aux Memory Error #0 Invalid File Parameter	The first argument in each call list must be a numeric expression that represents a legal file number or a string that represents a file name.	Make sure that the first argument in the CALL list is a defined variable, literal, or expression that represents a valid file number or name.
Aux Memory Error #1 No Such File	A file has been referenced that does not exist such as requesting a plot at a file number that has not previously been used.	The file specified in the statement producing the error may be the wrong file. If it is the correct file, then you will have to rewrite the file according to what is required for that statement.
Aux Memory Error #2 File is Write Protected	An attempt has been made to write to or modify a file whose write protect flag has been set by the PROT command.	You must first remove the write protect status from the file using the UNPROT command if you really intend to modify the file.
Aux Memory Error #3 Aux Memory Full	All blocks of memory have been alloctated to existing files and there are no more available.	You must delete some file that is not needed to make more room for dynamic file expansion.

Error Message	Cause	Correction
Aux Memory Error #4 Illegal File Type	The attempted operation has referenced an illegal file type such as a SORT on a sequential or program file type.	Check the file number or name you have used to reference the particular file. You may want to check the directory list to see the locate the file in question.
Aux Memory Error #5 End of File Encountered	This error usually results from an attempt to read beyond the end of the data file. Writes will dynamically expand a file, but reads are only good to last valid item in a file.	The ON EOF command can be used to trap end of file conditions. You can also use the LSTIP routine to tell you how many items are in a file so the program will know when to stop reading.
Aux Memory Error #7 No Program Found	This message will result from an attempt to OLD, APPEND, or LINK a file that does not contain a valid program.	Check the file you have referenced to see if it is the file you previously used to store the program. Check the length of the file to see if it seems reasonable.
Aux Memory Error #8 File Marked Too Small	The attempted tape operation requires a larger tape file to store the specified file or memory section.	You must choose a larger tape file, re-mark the existing tape file, or mark a new tape file large enough to hold the memory data. Use the file size plus an extra block or two for determining tape file size to mark.

Error Message	Cause	Correction
Aux Memory Error #9 File Already Used	The file specified is the wrong type for the attempted operation which requires either a new file or a certain type.	The file must be a new file or a certain type as required by the routine. Refer to the description of the routine in question for more details.
Aux Memory Error #10 Invalid Memory Data	This message will usually occur when a routine is expecting certain data such as during directory listing where file headers must contain certain data to be valid.	Since this error message indicates a possible corrupt memory you may want to reinitialize the memory, or save whatever files that can be accessed first, avoiding whatever files or commands produce the error until the memory can be re-initialized or otherwise corrected.
Aux Memory Error #36 Undefined Variable	A variable in the argument list is undefined that the routine requires to be defined.	Check the argument list for the variable that is un-defined. You may want to refer to the description of that routine to better understand what is wanted.
Aux Memory Error #12 Invalid Command Argument	An Argument in the CALL list is the wrong type, such as a string where a numeric was expected.	Re-enter the command with the correct argument list or type of arguments. Refer to the detailed description of the particular routine for more information.

Appendix C - ROM Pack Slot Priority

The position of the 6400 File Manager ROM Pack in the slot priority scheme may depend on what other ROM packs are present. In order to determine which slot to use, you must first understand the slot priority order. This may be somewhat confusing because the slot numbers do not represent the priority order. Therefore, refer to the table below to select the appropriate ROM Pack placement.

If a 4909 File Manager ROM Pack is present, the 6400 File Manager should be located **after** (at a lower priority than) the 4909 File Manager. This is to enable certain 4909 commands that are duplicated in the 6400 to be seen first by the 4909. In this case, the alternate CALL names in the 6400 may be used.

Otherwise, the preferred placement of the 6400 File Manager is in a back-pack slot rather than a ROM Expander slot. This is because the 6400 File Manager uses the ROM Pack slot as the data interface, and the back-pack slot provides a more direct and cleaner data connection than the ROM Expander does.

If this causes the 6400 File Manager to be placed at a higher priority than other file managers, this should present no problem (except for the above stated 4909 conflict) even though it may be stated that they must be placed at the highest priority.

Priority	Backpack Slot	ROM Bank
1 **Highest**	61	0
2	71	8
3	61-68 in 4050E01	16-23
4	71-78 in 4050E01	24-31
5	41	32
6	51	40
7	41-48 in 4050E01	48-55
8 **Lowest**	51-58 in 4050E01	56-63

The highest priority slot in a four slot backpack is number **61** and number **41** in a two slot backpack.

ROM Pack Slot Priority

If you use the 4050E01 ROM expander the **highest priority** slot is as follows:

Two Slot back Pack

41	with the ROM expander in slot 51
51	with the ROM expander in slot 41

Four Slot back Pack

71	with the ROM expander in slot 61

Appendix D - Hardware Specifications

Transfer Rate:	50K bytes/second
Storage Capacity:	128K to 1024K bytes
Expansion Card Size:	128K bytes
External DMA Transfer Rate:	750K bytes/second maximum
Power Requirements:	120 volts at 60 hertz 240 volts at 50 hertz
Power Consumption:	20 watts maximum (1024K bytes)
Batter Pack Life:	5 Minutes Minimum (1024K bytes)
Charge Time:	3-6 hours minimum 16 hours maximum
Physical Charactersitics:	
Memory Module	Length 30.5 cm (12 in) Width 24.6 cm (9.7 in) Height 10.2 cm (4 in) Weight 4 kg (9 lb)
ROM Pack	Length 6.6 cm (2.6 in) Width 13.3 cm (5.5 in) Height 2.3 cm (0.9 in) Weight 115 g (4 oz)
Operating Temperature:	10-40°C

Appendix E - Glossary

ASCII	American Standard Code for Information Interchange.
ARGUMENT	A data variable or constant used for input or output to a command, function, or routine.
BASIC	Beginners All-Purpose Symbolic Instruction Code. BASIC is the programming language used by a Tektronix 4050 series computer graphic system.
BINARY	Base 2 number representation.
BIT	A binary digit, a 1 or a 0.
BLOCK	A section of memory exactly 256 bytes long. May be referred to as the next least significant address part.
BYTE	A group of 8 binary bits. Also the smallest data item that can be addressed. May be referred to as the least significant address part.
COMMAND	A line of instructions containing a keyword(s) address(es) and/or an argument.
CONSTANT	Any entry in a field requiring a literal number rather than a variable.
CONTIGUOUS FILE	A file that is not scattered over different locations in the memory but is contained within one continuous area.
CURRENT DEVICE	The device specified by UNIT, CALL "MUNIT", or CALL "DUNIT" command.
DATA FILE	A collection of data stored in the memory.
DEFAULT	Value used when none is specified.

DELIMITER	The character(s) or space(s) used to separate one field from another.
DEVICE	The Auxiliary Memory, a disk drive, or any similar unit controlled by a file manager.
DEVICE ADDRESS	The Device Address is an integer number used to reference an external device.
DIMENSIONING	Enlarging or reducing of a string variable space to accomodate more or less than the 72 character default length.
DIRECTORY	A collection or of files that are listed in order of the numbers assinged to each for reference purposes.
ENTRY	The number or character or string placed in a field or the variable or string variable representing it. Entries are made through program operation or keyboard.
ERROR	An abnormal condition.
F.I.	File Identifier. Refers to the name or number of a program or data file by which it is referenced.
FIELD	One data item in a record or a command parameter.
FILE	A reserved block of storage for programs or data referred to by a name.
FILE IDENTIFIER	See F.I.
FILE NAME	The name of a program or data file.
FILE NUMBER	An integer value assigned to each file by which the file can be referenced.
FILE POINTER	An internal marker that specifies where an I/O operation is to begin.

HEADER	The area that begins each file which immediately preceeds the data space for that file.
HOST	Normally the 4050 graphic system, or any other computer that is in control.
INTERFACE	The ROM pack that provides the control and connection of the auxiliary memory to the computer system.
ITEM POINTER	The number of the item in a given file which is currently addressed or pointed to by the file manager. It is stored in the header of each file.
KEYWORD	The entry in the first field of all system commands. Certain file manager commands are keywords while others are routines implemented through the CALL keyword.
LAST ITEM POINTER	The position of the last item stored in a given file. It is stored in the header of each file created.
LINK MAP	The space reserved in the auxiliary memory for pointers that connect the series a blocks assigned to each file.
MEMORY	May refer to the Auxiliary Memory Unit or to 4050 system memory depending on context.
NUMERIC EXPRESSION	A combination of constants and/or variables with arithmetic operators or functions.
PAGE	A section of memory exactly 65,536 bytes long.
PARAMETER	A data variable or constant used for input or output to a command, function, or routine.
RANDOM ACCESS	The method of storing or locating data by specifying a record number and directly accessing that record only. Opposite of sequential access.

RANDOM FILE	A type of file with two or more records of identical length.
RECORD	A defined storage space of from 1 to 255 characters that holds exactly one data item.
SEQUENTIAL ACCESS	The method of storing or locating data by placing the file pointer at the beginning of the designated file and reading or writing towards the end of the file until the information is entered or encountered.
SEQUENTIAL FILE	A type of file with no divisions or records.
STATEMENT	A command preceded by a line number for program use.
STORAGE STRUCTURE	A file or collection of files in a directory.
STRING VARIABLE	A variable name ending in a '$' that is used to assign a character or characters to (usually ASCII).
SYSTEM	The "system" referred to in this manual includes the Auxiliary Memory and the host graphic system.

Appendix F – Firmware Release Notes

This appendix includes the release notes for firmware levels 5.0 and above for the 6400 Auxiliary Memory File Manager ROM Pack. The release notes document the firmware changes made and the known problems discovered since the last release of the reference manual. This information will be included in the next release of the memory manual.

Product: 6400 Auxiliary Memory and 764-MEM File Manager ROM Pack
For: 4052/54 and 4052A/54A
Firmware Level: 5.0
Release Date: 01-JUL-83

Change	Description
1.	MIN/MAX bug fixed that intermittantly returned bad values.
2.	COPY/CONCATINATE fixed to work correctly on files larger than 1 block.
3.	SECRET implemented so that programs may be saved to memory and will have secret protection when OLDed back in.
4.	KILL was modified to not produce an error message when attempts are made to delete non-existant files.
5.	DIR will now properly output to the tape drive.
6.	APPEND and LINK have been fixed to prevent problems that occurred under certain conditions.
7.	SETIP was fixed to prevent a periodic problem in addressing sequential data files when the item length attribute overlapped a block boundary of a non-contiguous block.
8.	MSPACE was adjusted to correctly report memory used by the directory in extending itself for high file numbers beyond the default size of 2 blocks
9.	EXCLUDE had a bug that was fixed.
10.	The firmware was made compatible with 4050 'A' version.
11.	File name facilities were added allowing names up to 28 characters.

12. Selected system file manager keywords for disk are now available for auxiliary memory access.

13. Signed integer data types were added.

14. Direct record addressing of item position is now allowed in READ and WRITE.

15. Memory to tape back-up routines have been added for files or memory sections. These are called TSAVE, TLOAD, TBACK, and TRESTORE.

16. Signal processing routines have been added that are similar to the Tektronix RO7 ROM Pack that allow direct file access.

17. New plotting routines have been added that include relative format data files and optional single or dual file format for storage of x,y data.

18. New, more descriptive error reporting formats have been added.

19. ON EOF error traping now available for end of file conditions.

20. Alternate CALL names have been added to avoid conflict with 4909 calls.

21. SAVE with beginning and ending line numbers no longer requires exact matching line numbers.

22. SETIP now rounds instead of truncates non-integer pointer values.

New or Modified Routines in version 5.0

764MEM	MCREATE	OLD
APPEND	MCROSS	ON EOF(0)
DIR	MCSUM	RDELETE
DUNIT	MDIF2	READ
EXCLUD	MDIF3	RPLOT
FILHDR	MINT	SAVE
FILTYP	MOLD	SEARCH
FPLOT	MOPEN	SETLST
GETADD	MPLOT	TBACK
GETIA	MSAVE	TLOAD
INSERT	MTEST	TRESTORE
KILL	MUNIT	TSAVE
MAPPEND	NXTFIL	WRITE

Product: 6400-764 Memory File Manager ROM Pack
For: 4052/54 and 4052A/54A
Firmware Level: 5.1
Release Date: 15-SEP-83

Change	Description

1. Tape commands TSAVE and TLOAD were fixed.

2. A problem in reading numerics from sequential files when a record size greater than 8 was specified has been fixed by no longer allowing record sizes for sequential files to be specifed for puposes of defining accuracy. The record size has no meaning for sequntial file and is listed as '-' in directory listings. Numerics are now stored with full precsion by default in sequential files.

3. A bug was fixed that allowed the file size reported in directory listings to be altered by SETIP followed by READ or WRITE.

4. A check was added to permit a wrap-around problem present on older style memory units to be detected during the power-up sequence that determines the memory size. This occurred on older 512K memories causing MSPACE to report a larger memory size.

5. The DMA routine CALL "STATUS" was changed to CALL "DMASTA" in order to avoid a name conflict with 4909 file manager.

6. CALL "OPEN" was added back in by demand for compatibility with exisiting software after it was replaced by CALL "MCREATE" which was necessary avoid conflicts with the 4909 OPEN command. Systems with 4909 file managers must now arrange the ROM pack slot priority so that it will be seen first and then use the MCREATE command to avoid this conflict. If a 4909 is not present, either OPEN or MCREATE may be used.

7. CALL "764MEM" was fixed to print revsion number. A bug in version 5.0 caused it to print nothing.

8. A 3 byte integer format has been added that replaces the 3 byte floating format. It may be signed or unsigned as with 1 and 2 byte integers.

9. CALL "RMPLOT" was added to allow relative plots from absolute data. This format allows imbedded moves flaged by negative data.

10. CALL "APLOT" was added to allow plotting of UDU's from all quadrants. MPLOT was modified in 5.0 to have this feature but APLOT has now been added to allow MPLOT to assume its former function of plotting positive domain UDU data with negative values that designate moves.

11. DIR was modified to print '-' instead of '0' for the record size in directory listings for sequential and program files types.