S-3030, S-3260
AUTOMATED TEST SYSTEM

GENERAL-PURPOSE
PROCESSING DATA
SUBPROGRAMS

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

MEASUREMENT SYSTEMS DIVISION

062-3398-00

FIRST PRINTING JUNE 1976

SOFTWARE LICENSE

Software supplied by Tektronix, Inc., as a component of a system or as a separate item is furnished under a license for use on a single system and can be copied (with the inclusion of copyright notice) only for use on that single system.

First Printing, First Edition – June 1976
Second Printing, First Edition – March 1977
Third Printing, First Edition (Revised) – December 1977
Fourth Printing, First Edition – June 1978

PREFACE

This manual describes the general-purpose processing data subprograms. You may use these subprograms in device tests run in the foreground on an S-3260 or an S-3030. Also, these subprograms may be used in background programs (that is, programs run under control of the REDUCE program) on an S-3260, S-3030, and S-3455. The system displays the error codes (e.g., AC) mentioned in this manual on the test station control unit PROGRAM ERROR readout display.

This manual is divided into five sections and three appendices. Section One describes the subprograms in the TIME file that read the system date and time. Section Two documents the bit array subprograms in the BARRAY file. Section Three deals with the graphics subprograms in the files GRAPH1 and GRAPHV. Section Four describes the string handling subprograms in the files STRING and ADSTNG. Section Five describes the extended function set in the file ARITH3.

Appendix A is a summary of how to declare subprograms. Appendix B shows the decimal, octal, and Radix-50 equivalents of the ASCII character set. Appendix C gives a summary of all subprograms described in this manual.

This manual assumes the reader is familiar with the data reduction language.

CONTENTS

SECTION THREE: GRAPHICS

SECTION FOUR: STRING HANDLING SUBPROGRAMS

NOMENCLATURE CONVENTIONS

This manual uses a standard nomenclature to show the general form of each command and its parameters. The nomenclature conventions are:

- Parameters shown in upper case letters, special characters, and punctuation marks (including blanks) are *literal* parameters. When you use them with the command, you must type them exactly as shown in the general form.

- Parameters shown in lower case letters are *variable* parameters. When you use them with the command, you must supply a valid name or value in place of the variable name appearing in the general form. For example, the variable name *pinnum* indicates that you must specify a pin number.

- Parameters enclosed in square brackets ([ ]) are *optional* parameters. You may supply these parameters or not, depending on the way you wish to use the command. (Since the brackets are a nomenclature convention only, you must not type them when you use the command.)

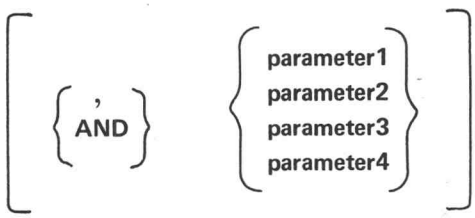
- A vertical list of parameters enclosed in braces ({ }) indicates that you must choose one line from the list when you use the command. Which parameter you choose depends on the function you wish the command to perform. (Since the braces are a nomenclature convention only, you must not type them when you use the command.)

- A vertical list of parameters enclosed in square brackets indicates that the parameter is optional. If you decide to use the parameter, you must select one line from the vertical list shown. Which parameter you choose depends on the function you wish the command to perform.

- Parameters not enclosed in square brackets or braces are *mandatory* parameters — you must supply the parameter when you use the command.

- When the general form shows the same parameter twice, separated by an ellipsis (i.e., parameter,...,parameter), you may enter the parameter once or repeat it as many times as desired.

- When parameters are nested within square brackets and braces, you interpret the brackets and braces by working from the outermost pair of brackets or braces to the innermost pair. For example,



In the above example, the outermost square brackets indicate that any parameters which are enclosed within the brackets are optional parameters. The inner braces indicate that if you decide to specify the optional parameters, you must select one line from each vertical list shown.

Throughout this manual the examples show user-typed information in **boldface**. Information the system prints at your terminal is shown in lightface.

In addition, this manual assumes that you type a carriage return after each line you type at your terminal. Whenever there is any doubt about the necessity of the carriage return, it is indicated by the symbol ↵. For example,

```
*↵
```

In the above example, the ↵ symbol indicates that the user must type a carriage return after the system prints the asterisk at the terminal.

The symbol □ indicates a blank.

LOGICAL UNIT NUMBERS

Some of the subprograms described in this manual are based on logical unit numbers (luns). That is, you call these subprograms without specifying specific input and output devices. The system associates the luns with peripheral devices.

The system assigns all foreground luns to the terminal at system bootup. When entering the REDUCE program, the system assigns all luns for that background to the terminal. To assign or change the device with which a lun is associated, use the LOG program or REDUCE program ASSIGN command.

For more information on logical unit numbers and the ASSIGN command, see the Processing Data manual.

SECTION ONE:

READING THE SYSTEM AND LOGGED DATES AND TIMES

This section describes the subprograms that read the system date and time, and the date and time recorded in a log file. The TIME subprograms perform the computation and code conversion to put the date-time value in a readable format.

The TIME file provides subprograms that:

1. Read the current date and time,
2. Read the date and time stored in a log file,
3. Return the date and time as a floating-point number,
4. Return the date and time as ASCII characters to a lun, and
5. Store the ASCII value of the date and time in a string array.

The chart below shows the relationships between the subprograms.

Action	Date or Time?	Subprograms for Current System Data	Subprograms for Log File Data
Returned as a Floating-Point Value	Date	CURDAY	FILDAY
	Time	CURSEC	FILSEC
Printed in ASCII Characters	Date	CRDATE	FLDATE
	Time	CRTIME	FLTIME
Stores in a String	Date and Time	MOVDAT	MOVFDA

In the following discussion, midnight is the start of a new day. That is, 0.0 seconds into the current day. Printing is the process of transferring ASCII characters to an output device or file assigned to a lun.

The form hh:mm:ss, used for time, indicates two hour digits (hh), two minute digits (mm), and two second digits (ss). For example, 13:00:53. The form dd□mmm□yy, used for dates, indicates two day digits (dd), the first three letters of the name of a month (mmm), and two year digits (yy) for the year 19yy. For example, 04 JUL 76.

SECTION ONE

System Date and Time

The system maintains the date-time as the number of seconds from 1 January 1900 to the date-time last entered into the system. The system requests the date and time when the system is booted. You may change the currently specified date and time with the Executive commands DATE and TIME (see the Command Language Reference Guide).

Logged Date and Time

A test program that logs data usually includes LOGMARKER statements to flag devices and groups of devices. The date-time resides in the marker records. When a LOGMARKER statement executes, the system date-time is recorded. In addition, whenever you close a log file LOG records the test date-time in the EOF record.

When the ASSIGN command assigns a lun to a log file, it sets the log date-time to 0.0. When a marker record is read, the log date-time is set to the value from that record. Therefore, when you read a log file date-time, the result is 0.0 if a marker record has not been read. Otherwise, the result is the value from the most recently read marker record.

The Real-Time Clock Option

With the real-time clock on the system, the system date-time contains the number of seconds from midnight 1 January 1900 to the date and time last entered by the operator, plus the time in seconds since that entry. Therefore, a marker record records the exact date and time the record was logged.

Summary of the Date and Time Subprograms

These subprograms are on the TIME file.

Function	Declaration	Purpose
CURDAY	CURDAY(0)	Returns the current date in days since 1 January 1900.
CURSEC	CURSEC(0)	Returns the current time in seconds since midnight.
FILDAY(e,ilun)	FILDAY(N,V)	Returns the date, logged in the specified log file, in days since 1 January 1900.
FILSEC(e,ilun)	FILSEC(N,V)	Returns the time, logged in the specified log file, in seconds since midnight.

Subroutine	Declaration	Purpose
CRDATE(olun)	CRDATE(V)	Prints the current date.
CRTIME(olun)	CRTIME(V)	Prints the current time.
FLDATE(e,ilun,olun)	FLDATE(N,V,V)	Prints the date logged in the specified log file.
FLTIME(e,ilun,olun)	FLTIME(N,V,V)	Prints the time logged in the specified log file.
MOVDAT(string,start,stop)	MOVDAT(I,V,V)	Stores the current date and time in string.
MOVFDA(e,ilun,string, start, stop)	MOVFDA(N,V,I,V,V)	Stores the date and time, logged in the specified file, in string.

Function Call: **CURDAY**

Declaration: **FUNCTION CURDAY(0):TIME**

Purpose: CURDAY returns the current date in days since 1 January 1900.

Function Call: **CURSEC**

Declaration: **FUNCTION CURSEC(0);TIME**

Purpose: CURSEC returns the current time in seconds since midnight of the current day.

Function Call: **FILDAY(e,ilun)**

Declaration: **FUNCTION FILDAY(N,V):TIME**

Purpose: FILDAY returns the date, logged in the log file assigned to **ilun**, in days since 1 January 1900.

Arguments: **ilun** must be assigned to an input log file.

e is an error indicator. If FILDAY successfully reads a date, **e** equals 0. If the lun is not the correct type or is unassigned, **e** equals -2.

Function Call: **FILSEC(e,ilun)**

Declaration: **FUNCTION FILSEC(N,V):TIME**

Purpose: FILSEC returns the time, logged in the specified log file, in seconds since midnight of the day the file was logged.

Arguments: **ilun** must be assigned to an input log file.

e is an error indicator. If FILSEC successfully reads a time, **e** equals 0. If the lun is not of the correct type or is unassigned, then **e** equals -2.

Subroutine Call: **CRDATE(olun)**

Declaration: **SUBROUTINE CRDATE(V):TIME**

Purpose: CRDATE prints the current date in the form dd☐mmm☐yy without any leading or trailing spaces.

Argument: **olun** is the output lun receiving the nine ASCII characters. If the lun is unassigned or not an output lun, then CRDATE acts as a no-op and does not print any output.

Subroutine Call: **CRTIME(olun)**

Declaration: **SUBROUTINE CRTIME(V):TIME**

Purpose: CRTIME prints the current time in the form hh:mm:ss without leading or trailing spaces.

Argument: **olun** is the output lun receiving the eight ASCII characters. If the lun is unassigned or not an output lun, then CRTIME acts as a no-op and does not print any output.

Subroutine Call: **FLDATE(e,ilun,olun)**

Declaration: **SUBROUTINE FLDATE(N,V,V):TIME**

Purpose: FLDATE prints the date logged in the selected log file in the form dd□mmm□yy without leading or trailing spaces.

Arguments: **olun** is the output lun receiving the nine ASCII characters. If **olun** is unassigned, FLDATE acts as a no-op and does not print any output.

ilun is the input lun. It must be assigned to an input log file.

e is an error indicator. If FLDATE successfully reads a date, **e** equals 0. If the input lun is not of the correct type or is unassigned, then **e** equals -2.

Subroutine Call: **FLTIME(e,ilun,olun)**

Declaration: **SUBROUTINE FLTIME(N,V,V):TIME**

Purpose: FLTIME prints the time, logged in the selected log file, in the form hh:mm:ss without leading or trailing spaces.

Arguments: **olun** is the output lun receiving the eight ASCII characters. If **olun** is unassigned, FLTIME acts as a no-op and does not print any output.

ilun is an input lun which must be assigned to an input log file.

e is an error indicator. If FLTIME successfully reads the time, **e** equals 0. If the input lun is not of the correct type or is unassigned, then **e** equals -2.

Subroutine Call: **MOVDAT(string,start,stop)**

Declaration: **SUBROUTINE MOVDAT(I,V,D):TIME**

Purpose: MOVDAT stores the current date-time in **string**. It transfers up to 19 characters in the form hh:mm:ss□□dd□mmm□yy and pads unused string space with spaces.

Arguments: **string** is the name of an integer array that receives the date-time characters. **start** specifies the character element in **string** at which MOVDAT starts storing characters. **stop** specifies the character element at which MOVDAT stops storing characters.

For an explanation of strings, see Section Four of this manual. Note that if provision for only eight characters is made, MOVDAT only stores time in the string.

If **start** or **stop** is outside the integer array subscript range, then the system displays the AC error and the test program halts.

Subroutine Call: **MOVFDA(e,ilun,string,start,stop)**

Declaration: **SUBROUTINE MOVFDA(N,V,I,V,V):TIME**

Purpose: MOVFDA stores the date-time, logged in the specified log file, in **string** in the form hh:mm:ss□□dd□mmm□yy. It transfers up to 19 characters and pads unused trailing string space with spaces.

Arguments: **string** is the name of an integer array that receives the ASCII date-time characters. **start** specifies the character element at which MOVFDA starts storing characters. **stop** specifies the character element at which MOVFDA stops storing characters.

ilun is an input lun which must be assigned to an input log file.

e is an error indicator. If MOVFDA successfully stores the date-time, **e** equals 0. If the input lun is not the correct type or is unassigned, then **e** equals -2.

SECTION TWO:

STORING BINARY INFORMATION IN INTEGER ARRAYS

The subprograms described in this section store and recover binary information (for example, pattern data) using the integer arrays. After you define a binary array, use these subprograms to modify and read individual bits, or groups of bits, in the binary array.

Summary of BARRAY Subprograms

The subprograms described in this section are on the file BARRAY.

Subroutine	Declaration	Purpose
BARRAY(iarray,xmax,ymax,zmax)	BARRAY(I,V,V,V)	Dimensions an existing integer array into a binary array.
SETBIT(state,barray,x,y,z)	SETBIT(V,I,V,V,V)	Modifies an individual bit of the binary array.
SETCEL(value,barray,x,y,z)	SETCEL(V,I,V,V,V)	Modifies a group of bits in the X direction.
SETCOL(value,barray,x,y,z)	SETCOL(V,I,V,V,V)	Modifies a group of bits in the Z direction.
SETROW(value,barray,x,y,z)	SETROW(V,I,V,V,V)	Modifies a group of bits in the Y direction.

Function	Declaration	Purpose
GETBIT(barray,x,y,z)	GETBIT(I,V,V,V)	Reads an individual bit of the binary array.
GETCEL(barray,x,y,z)	GETCEL(I,V,V,V)	Reads a groups of bits in the X direction.
GETCOL(barray,x,y,z)	GETCOL(I,V,V,V)	Reads a group of bits in the Z direction.
GETROW(barray,x,y,z)	GETROW(I,V,V,V)	Reads a group of bits in the Y direction.

Subroutine Call: **BARRAY(iarray,xmax,ymax,zmax)**

Declaration: **SUBROUTINE BARRAY(I,V,V,V):BARRAY**

Purpose: You can store binary data in single bits within an integer array. Each array element stores 16 binary bits. BARRAY dimensions the space in an existing **iarray** into a three-dimensional bit array. BARRAY must dimension an integer array before the other binary information subprograms can store or recover binary data.

Arguments: **iarray** specifies the name of the integer array in which the bit array is to be dimensioned.

xmax, **ymax**, and **zmax** specify the dimensions in bits of the bit array. If BARRAY attempts to exceed the space available in **iarray**, an AC error results.

Comments: BARRAY uses the first three **iarray** elements to store the dimensions of the bit array. Therefore, the space available for the binary data is:

number of bits = (n - 3) * 16, where n is the number of **iarray** elements.

Example: First, declare an integer array:

```
4.01 IARRAY IA(67)
```

Then, dimension this array into a bit array:

```
4.02 BARRAY(IA,8,8,16)
```

Therefore, IA is dimensioned to be 8 bits by 8 bits by 16 bits, for a total of 1024 bits. The total available space in IA is (67 - 3) * 16, or 1024 bits.

Subroutine Call: **SETBIT(state,barray,x,y,z)**

Declaration: **SUBROUTINE SETBIT(V,I,V,V,V):BARRAY**

Purpose: SETBIT modifies an individual bit within a bit array.

Arguments: **state** determines if the selected bit position is set or cleared. If **state** equals zero, the bit is cleared. Otherwise, it is set.

barray specifies the name of the array to be modified.

x, y, and **z** select the bit to be modified.

Example: 5.01 SETBIT(X,IA,5,5,10)

SETBIT sets the bit at (5,5,10) in the bit array IA if X is not zero, and clears the bit if X is zero.

Function Call: **GETBIT(barray,x,y,z)**

Declaration: **FUNCTION GETBIT(I,V,V,V):BARRAY**

Purpose: GETBIT reads an individual bit within a bit array.

Arguments: **barray** specifies the name of the array which is to be read.

x, y, and **z** specify the bit whose value is converted to a floating-point number and returned as the value of the function.

Example: 5.02 V = GETBIT(IA,5,5,10)

The result equals one if bit (5,5,10) is set. Otherwise, the result equals zero.

Subroutine Call: **SETCEL(value,barray,x,y,z)**

Declaration: **SUBROUTINE SETCEL(V,I,V,V,V):BARRAY**

Purpose: SETCEL modifies up to 16 bits within a bit array.

Arguments: SETCEL converts **value** to a 16-bit integer. Then, SETCEL uses the integer, one bit at a time, to modify up to 16 bits in the bit array.

barray specifies the bit array to be modified.

SETCEL modifies **barray** by storing the least significant bit of **value** in **(x,y,z)**. SETCEL increments **x** and modifies the next bit with the next bit of **value**, until it accesses 16 bits or reaches the maximum **x** value of the array.

Example: 5.03 SETCEL(V,IA,1,5,5)

SETCEL converts the value of V into an integer and uses V to modify the bits at (1,5,5), (2,5,5), (3,5,5), . . . until SETCEL accesses 16 bits or reaches the maximum X value of the array.

Function Call: **GETCEL(barray,x,y,z)**

Declaration: **FUNCTION GETCEL(I,V,V,V):BARRAY**

Purpose: GETCEL reads up to 16 bits in a bit array.

Arguments: **barray** specifies the bit array to be read.

Starting at bit address **(x,y,z)**, GETCEL reads **barray**, one bit at a time, incrementing **x** after each read. GETCEL continues until it reads 16 bits or reads the maximum **x** limit.

Result: The result of GETCEL determines, on a bit-by-bit basis, each bit of a 16-bit integer. GETCEL converts this integer to a floating-point value and returns it as the value of the function.

Example: 5.04 F = GETCEL(IA,1,5,5)

The least significant bit of the variable F equals one if the bit at (1,5,5) is set. Likewise, GETCEL reads the bits at (2,5,5), (3,5,5), . . . to determine the value of the corresponding bits of the variable F. As the coordinate increases, GETCEL modifies a more significant bit.

Subroutine Call: **SETROW(value,barray,x,y,z)**

Declaration: **SUBROUTINE SETROW(V,I,V,V,V):BARRAY**

Purpose: SETROW modifies up to 16 bits of a bit array.

Arguments: SETROW converts **value** to a 16-bit integer. Then, SETROW uses the integer, one bit at a time, to modify up to 16 bits in the bit array.

barray specifies the bit array to be modified.

SETROW modifies **barray** by storing the least significant bit of **value** in (**x**,**y**,**z**). SETROW increments **y** and modifies the next bit with the next bit of **value**, until it accesses 16 bits or reaches the maximum **y** value.

Example: 5.10 SETROW(V,IA,1,5,5)

SETROW converts the value of V into an integer and uses it to modify the bits (1,5,5), (1,6,5), (1,7,5),... until SETCOL accesses 16 bits or reaches the maximum Y value.

Function Call: **GETROW(barray,x,y,z)**

Declaration: **FUNCTION GETROW(I,V,V,V):BARRAY**

Purpose: GETROW reads up to 16 bits in a bit array.

Arguments: **barray** specifies the bit array to be read.

Starting at bit address (**x**,**y**,**z**), GETROW reads **barray**, one bit at a time, incrementing **y** after each read. GETROW continues until it reads 16 bits or reads the maximum **y** limit.

Result: The result of GETROW determines, on a bit by bit basis, each bit of a 16-bit integer. GETROW converts this integer to a floating-point value and returns it as the value of the function.

Example: 5.06 H = GETROW(IA,1,5,5,)

The least significant bit of the variable H equals one if the bit at (1,5,5) is set. Likewise, GETROW reads the bits at (1,6,5), (1,7,5),... to determine the value of the corresponding bits of the variable H. As the coordinate increases, GETROW modifies a more significant bit.

Subroutine Call: **SETCOL(value,barray,x,y,z)**

Declaration: **SUBROUTINE SETCOL(V,I,V,V,V):BARRAY**

Purpose: SETCOL modifies up to 16 bits of a bit array.

Arguments: SETCOL converts **value** to a 16-bit integer. Then, SETCOL uses the integer, one bit at a time, to modify up to 16 bits in the bit array.

barray specifies the bit array to be modified.

SETCOL modifies **barray** by storing the least significant bit of **value** in (**x**,**y**,**z**). SETCOL increments **z** and modifies the next bit with the next bit of **value**, until it accesses 16 bits or reaches the maximum **z** value.

Example: 5.03 SETCOL(V,IA,1,5,5)

SETCOL converts the value of V into an integer and uses V to modify the bits (1,5,5), (1,5,6), (1,5,7),... until SETCOL accesses 16 bits or reaches the maximum Z value.

Function Call: **GETCOL(barray,x,y,z)**

Declaration: **FUNCTION GETCOL(I,V,V,V):BARRAY**

Purpose: GETCOL reads up to 16 bits in a bit array.

Arguments: **barray** specifies the bit array to be read.

Starting at bit address (**x**,**y**,**z**), GETCOL reads **barray**, one bit at a time, incrementing **z** after each read. GETCOL continues until it reads 16 bits or reads the maximum **z** limit.

Result: The result of GETCOL determines, on a bit-by-bit basis, each bit of a 16-bit integer. GETCOL converts this integer to a floating-point value and returns it as the value of the function.

Example: 5.80 F = GETCOL(IA,1,5,5)

The least significant bit of the variable F equals one if the bit at (1,5,5) is set. Likewise, GETCOL reads the bits at (1,5,6), (1,5,7),... to determine the value of the corresponding bits of the variable F. As the coordinate increases, GETCOL modifies a more significant bit.

SECTION THREE:

GRAPHICS

Graphics software consists of two system files: GRAPH1 and GRAPHV. GRAPH1 contains the subroutines needed to produce direct graphic displays on the terminal screen. The subroutines in GRAPHV contain the subroutines needed to produce direct graphic displays and have the added capability of producing proportional graphic displays from data with a range equal to the range of a single-precision floating-point number.

Direct graphics work directly with the 781 x 1024 addressable points on the terminal screen. Proportional graphics allows you to define the range of the data; the subroutines translate the data to fit within a specified area on the terminal screen.

Graphics output from these subroutines always goes to logical unit number 12. The user can assign the logical unit number to a file or to any output device. Graphics input, such as that from the cross-hair positioning subroutines CRSHR and CRSHRV, is independent of the logical unit number and is always from the terminal.

If you desire the capability of direct graphics only, use the GRAPH1 file. This is because the GRAPH1 file requies less memory space than GRAPHV. Therefore, more space is available for test programs. To include the proportional graphics subroutines, only call the GRAPHV file.

The Terminal Screen

The terminal screen is a two-dimensional surface consisting of a discrete 1024 x 1024 matrix of addressable points; 1024 x 781 of these points lie in the viewable area* of the terminal screen (Figure 3-1). The origin of the screen lies at the extreme lower left corner.

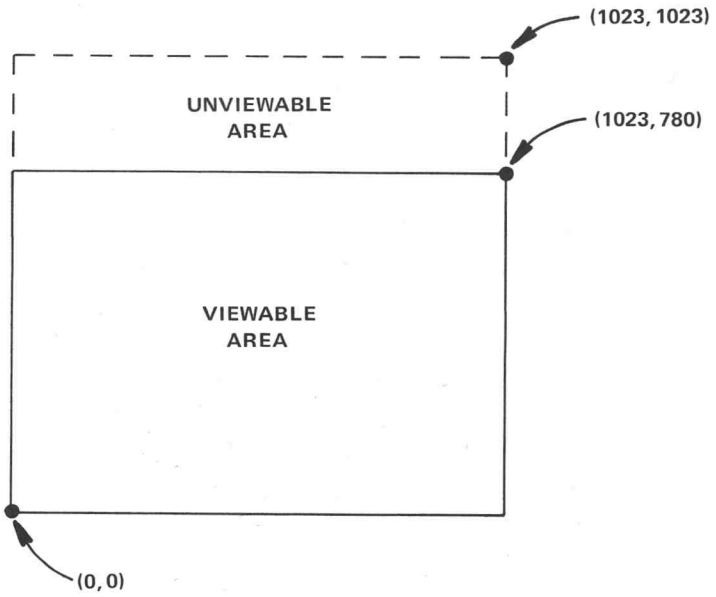


Figure 3-1. TERMINAL SCREEN
Bounded by 0 and 1023 on the X-Axis and by 0 and 1023 on the Y-Axis, but only 0 through 780 on the Y-Axis is in the viewable area.

*Vectors just above 780 on the Y-axis may be visible but marginal in quality. For the purpose of this manual, such vectors are considered part of the unviewable area.

Summary of the Graphic Subroutines

The subroutines listed below are in both the GRAPH1 and GRAPHV files.

Subroutine	Declaration	Purpose
ALFPOS(tstat,x,y)	ALFPOS(N,N,N)	Returns the coordinates of the bottom left corner of the alpha cursor and the terminal status.*
CRSHR(char,x,y)	CRSHR(N,N,N)	Turns on the terminal screen cross hairs, waits for you to press a key, then returns the cross-hair coordinates and the value of the character pressed.*
CURSOR(x,y)	CURSOR(V,V)	Moves the bottom left corner of the alpha cursor to the screen coordinates (x,y).
DRAW(x,y,mode)	DRAW(V,V,V)	Draws a vector from the current beam position to the screen coordinates (x,y).
LOCATE(x,y)	LOCATE(N,N)	Returns the cross-hair coordinates in (x,y) without user intervention.*
POINT(x,y,type)	POINT(V,V,V)	Plots a period (.), minus (-), or plus (+) at screen cordinates (x,y).
VECTRF(x1,y1,x2,y2)	VECTRF(V,V,V,V)	Draws a vector on the terminal screen between (x1,y1) and (x2,y2).

*For proper operation of these subroutines, the 4010 terminal graphic input terminators must be strapped for No CR, No EOT. See the 4010 display terminal manual (070-1225).

The following subroutines are only on the GRAPHV file.

Subroutine	Declaration	Purpose
ALFPSV(tstat,xv,yv)	ALFPSV(N,N,N)	Returns the user data-space coordinates at the bottom left corner of the alpha cursor and the terminal status.*
CRSHRV(char,xv,yv)	CRSHRV(N,N,N)	Turns on the terminal screen cross hairs, waits for you to press a key, then returns the cross-hair coordinates and the value of the character pressed.*
DCOORD(xvmin,xvmax,yvmin,yvmax)	DCOORD(V,V,V,V)	Defines the coordinates of the user data-space window.
DRAWV(xv,yv,mode)	DRAWV(V,V,V)	Draws a vector from the current imaginary beam position to (xv,yv) on the user data-space.
LOCATV(xv,yv)	LOCATV(N,N)	Returns the user data-space cross-hair coordinates in (xv,yv) without operator intervention.*
MAPOUT(xs,ys,map,xv,yv)	MAPOUT(N,N,N,V,V)	Returns the screen coordinates scaled and translated from user data-space coordinates.
POINTV(xv,yv,type)	POINTV(V,V,V)	Plots a period (.), minus (-) or plus (+) on the user data-space.
SCOORD(xsmin,xsmax,ysmin,ysmax)	SCOORD(V,V,V,V)	Defines the boundaries of the screen window.
VECTRV(xv1,yv1,xv2,yv2)	VECTRV(V,V,V,V)	Draws a vector from (xv1,yv1) to (xv2,yv2) on the user data-space.

*For proper operation of these subroutines, the 4010 terminal graphic input terminators must be strapped for No CR, No EOT. See the 4010 display terminal manual (070-1225).

Direct Graphics

Direct graphics relate directly with the terminal screen. Therefore, you work at a basic graphic level and avoid the overhead of the proportional graphics transformation routines. With direct graphics, you have the responsibility of keeping data points on the screen. That is, the ranges are: $0 \leqslant X \leqslant 1023$ and $0 \leqslant Y \leqslant 780$.

Mode entry and appropriate output handling is automatic. Direct graphics are primarily used with numeric output and for display layout. You may freely alternate between direct and proportional graphics.

Subroutine Call: **VECTRF(x1,y1,x2,y2)**

Declaration: **SUBROUTINE VECTRF(V,V,V,V):** {GRAPH1 / GRAPHV}

Purpose: VECTRF draws vectors on the terminal screen between (**x1,y1**) and (**x2,y2**).

Arguments:
x1 is the x coordinate of the first point.
y1 is the y coordinate of the first point.
x2 is the x coordinate of the second point.
y2 is the y coordinate of the second point.

Subroutine Call: **POINT(x,y,type)**

Declaration: **SUBROUTINE POINT (V,V,V):** {GRAPH1 / GRAPHV}

Purpose: POINT draws a point graph.

Arguments: **x** specifies the X coordinate of the point. **y** specifies the Y coordinate of the point.

You specify what is drawn at (**x,y**) with **type**.
If **type** = 1, draw a point* (.),
2, draw a minus (-),
3, draw a plus (+).

Any other value of **type** causes a point*.

*The + and - are alphanumeric characters. The point is created by intensifying the spot at the current beam position (i.e., the point is *not* an alphanumeric period). The point is brighter (more intense) than the period.

Subroutine Call: **CURSOR(x,y)**

Declaration: **SUBROUTINE CURSOR(V,V):** {GRAPH1 / GRAPHV}

Purpose: CURSOR moves the bottom left corner of the alpha cursor to the position on the screen specified by **x** and **y**.

Arguments: **x** and **y** are the screen coordinates of the desired position.

Comments: The axis of the graph can be drawn with the VECTRF subroutine and the graph labeled with the CURSOR subroutine and the PRINT statement.*

CURSOR spaces characters horizontally at 14 point-intervals and vertically at 22 point-intervals. This 14-point x 22-point area includes the blank space necessary for separating the characters one from another, and the blank space between lines of characters. The actual character size is 10 points x 16 points (on a TEKTRONIX 4010 terminal – the 4014/4015 terminals support other character sizes). The 10-point x 16-point character is justified in the lower left corner of the 14-point x 22-point area.

*See the *Data Reduction Language* manual for a discussion of the PRINT statement.

Subroutine Call: **DRAW(x,y,mode)**

Declaration: **SUBROUTINE DRAW(V,V,V):** {GRAPH1 / GRAPHV}

Purpose: This subroutine draws a vector from the current beam position to coordinates (**x,y**) on the terminal screen.

Arguments: **x** and **y** are the screen coordinates to which the vector is drawn.

mode defines the operation. If **mode** is:

- <0, DRAW moves the current beam position to (**x,y**) without drawing a visible vector and returns the terminal to alphanumeric mode.
- = 0, DRAW puts the terminal in the graphics mode and moves the current beam position to (**x,y**) without drawing a visible vector.
- >0, DRAW draws a visible vector.

A call must be made to DRAW with **mode** equal to 0 before visible vectors can be drawn with **mode** greater than 0.

A call to DRAW with **mode** <0, or a call to CURSOR, causes DRAW to forget the current beam position.

NOTE

DRAW produces the fastest graphics.

Subroutine Call: **CRSHR(char,x,y)**

Declaration: **SUBROUTINE CRSHR(N,N,N):** {GRAPH1 / GRAPHV}

Purpose: CRSHR turns on the cross-hair cursor, allows you to adjust its position with the thumbwheels, and waits for you to press a key. CRSHR then returns the cross-hair coordinates in (**x,y**) and the value of the character you pressed in **char**.

Arguments: **x** and **y** receive the screen coordinates of the cross-hair cursor.

char receives the floating-point value of the ASCII representation of the character you pressed.

Comments: The characters CTRL/C, CTRL/S, CTRL/T, CTRL/V, and ALTMODE retain their normal system functions when typed in response to CRSHR. For example, pressing CTRL/C aborts the program.

Subroutine Call: **ALFPOS(tstat,x,y)**

Declaration: **SUBROUTINE ALFPOS(N,N,N):** {GRAPH1 / GRAPHV}

Purpose: ALFPOS returns the screen coordinates of the bottom left corner of the alpha cursor in (**x**,**y**) and the terminal status in **tstat**.

Arguments: **x** and **y** are the screen coordinates of a character position extending 14 points to the right of **x** and 22 points above **y**.

tstat receives code specifying the terminal status.* Bit assignments for this word are:

Bit	Octal Value	Function if the Bit Is Set
0	1	Auxiliary device is not enabled or is nonexistent.
1	2	The cursor is at margin one.
2	4	The terminal is not in the graphics mode.
3	10	Linear interpolation is off.
4	20	The hard copy unit is not ready.
5	40	This bit is always set.

tstat normally equals 45_8 during alphanumeric I/O.†

Subroutine Call: **LOCATE(x,y)**

Declaration: **SUBROUTINE LOCATE(N,N):** {GRAPH1 / GRAPHV}

Purpose: LOCATE returns the current cross-hair coordinates without operator intervention.

Arguments: **x** and **y** are the screen coordinates of the cursor.

Comments: LOCATE is similar to CRSHR, but does not wait for you to press a key. Therefore, it does not return the value of a character.

*Refer to the display terminal manual (070-1255) for more information.

†Alphanumeric I/O displays characters. Graphics I/O displays lines.

User Data-Space

The user data-space is an imaginary two-dimensional surface with a range in both the X and Y directions equal to the range of a single precision floating-point number (Figure 3-2). Using the data-space, the user may construct drawings, pictures, and graphs of extreme complexity and detail.

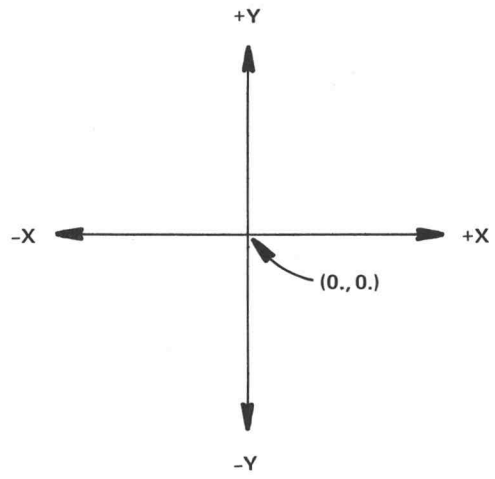


Figure 3-2. USER DATA-SPACE Bounded only by the floating-point range.

Since the unit of measurement of the user data-space is arbitrary, it may be assumed to be representative of any measurement unit from microns to light-years, with all measurements translated to the assumed unit for the given drawing. For example, the user decides that the basic unit of the data-space will represent inches. Then the coordinate (2., 0.5) represents a point two inches to the right of the origin on the X-axis and one-half inch up on the Y-axis. To indicate the point one mile (63,360 inches) to the left of the origin along the Y-axis, the coordinate (-63360.0, 0.0) would be used.

The user data-space is similar to normal displays and plotting devices in that there is a movable point which may be thought of as the writing cursor on the data-space. This point is called the imaginary beam.

Windowing

Any portion or all of the user data-space may be viewed through the technique of windowing. The portion of the data-space to be displayed is defined by a rectangular boundary. This rectangle is called the user data-space window, and only those vectors within the window are displayed.

It is not necessary to use the whole terminal screen for display of the user data-space. You may define a rectangular section of any size and location on the screen as the window area. This rectangle is called the screen window and, together with the user data-space window, defines the transformation between the data-space and the screen (Figure 3-3).

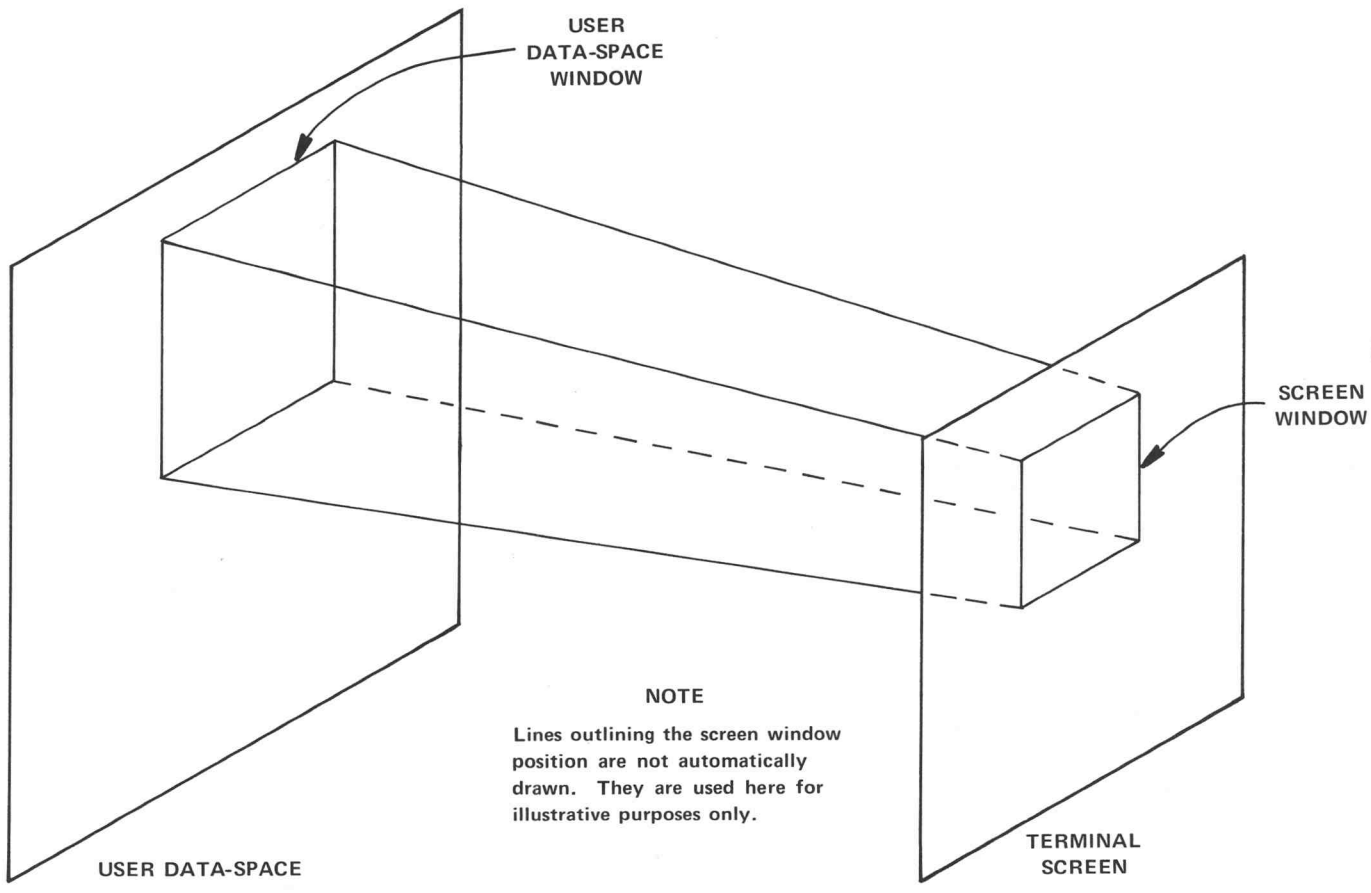


Figure 3-3.

The graphics routines automatically eliminate vectors and portions of vectors which lie outside the user data-space window, as well as scale and convert the vectors that are contained in or pass through the user data-space window.

The initial window definition is set so that the portion of the user data-space with coordinates equivalent to the screen are displayed.

User Data-Space Window Initial Values:

X minimum – 0., Xmaximum – 1023
Y minimum – 0., Ymaximum – 780

Screen Window Initial Values:

X minimum – 0, X maximum – 1023
Y minimum – 0, Ymaximum – 780

The data-space is used by first defining the window, then constructing a graph with the use of the graphic routines. The user may display several portions of the data-space at one time by redefining the window and reprocessing the data-space for each (Figure 3-4) or may superimpose data from several data-spaces by using a common screen window (Figure 3-5). All transformations between the data-space and the screen are based upon the latest window definitions.

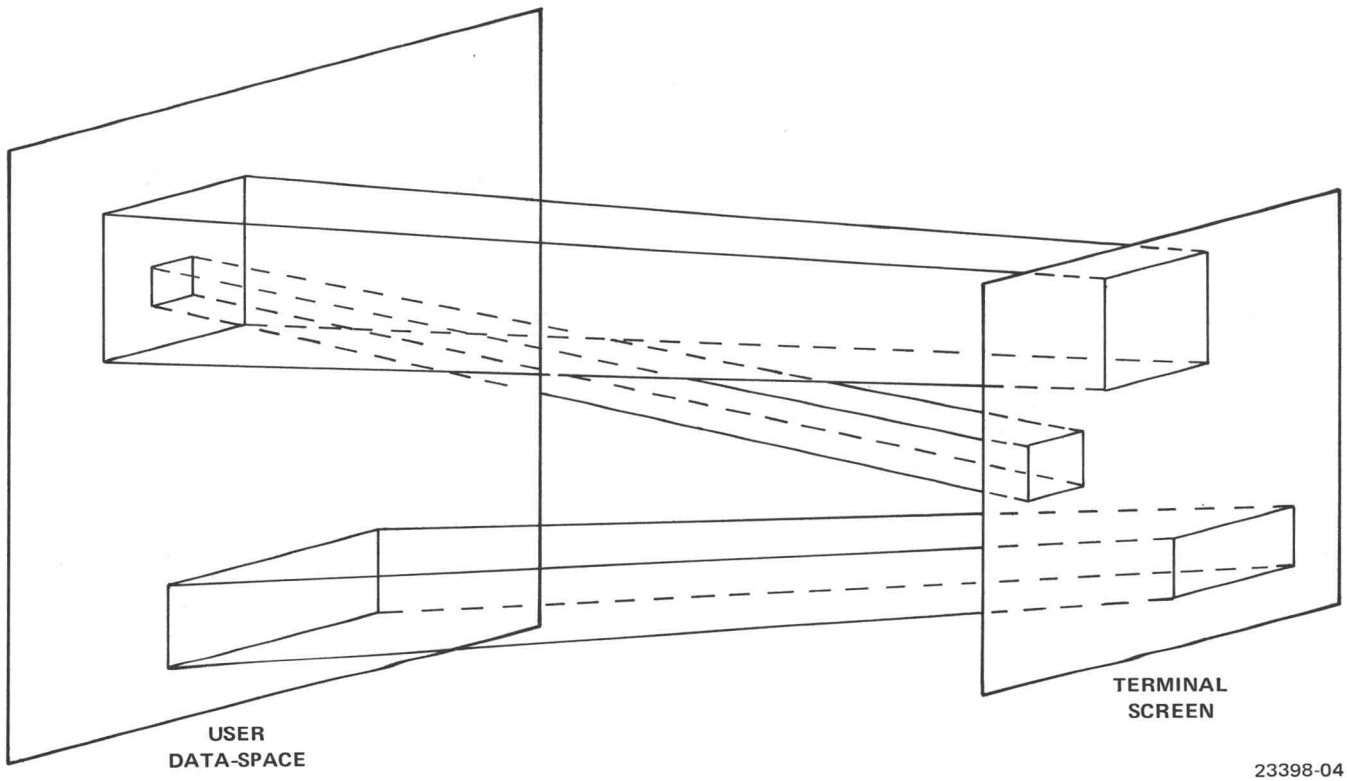


Figure 3-4. Use of Several Windows.

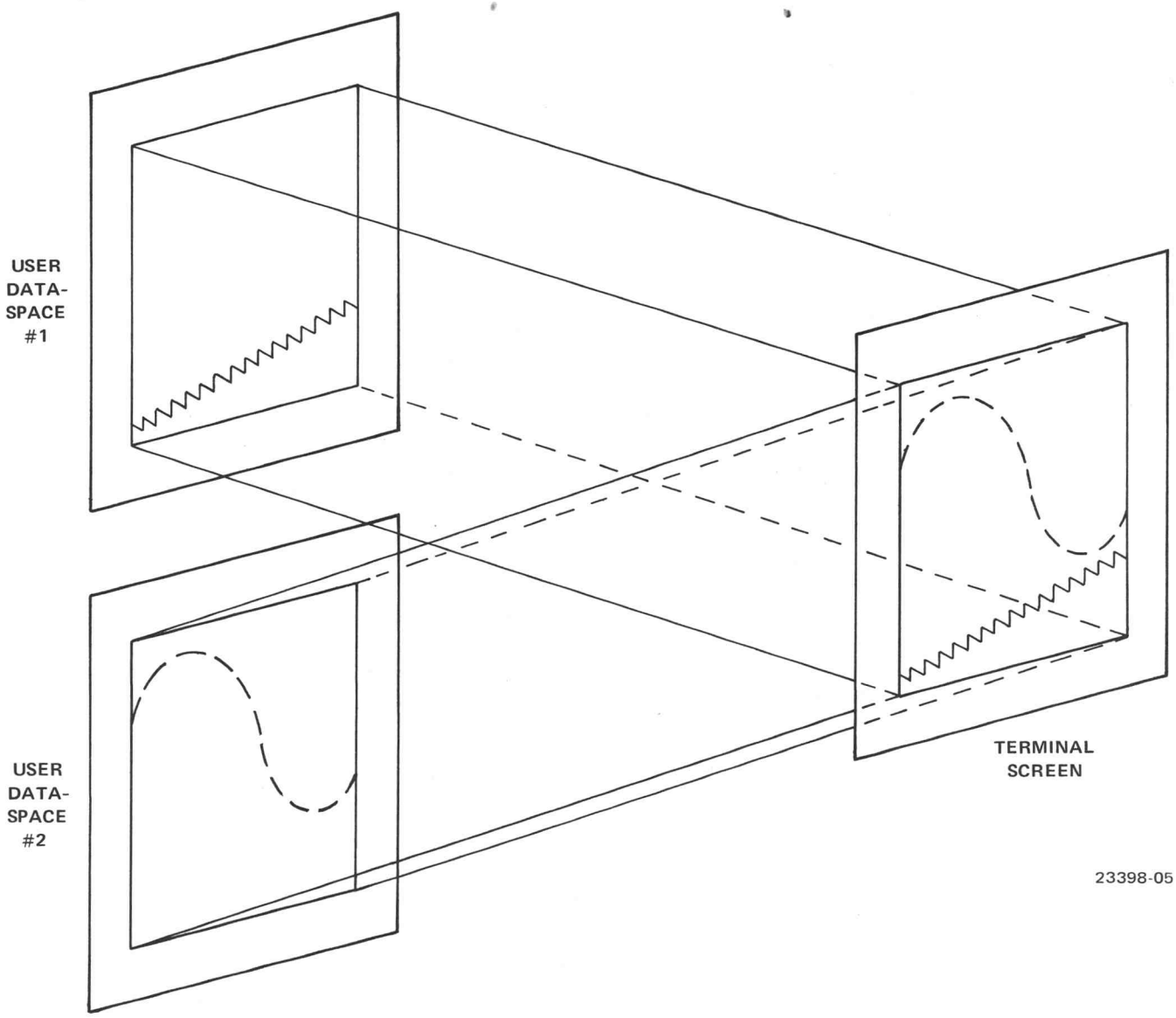
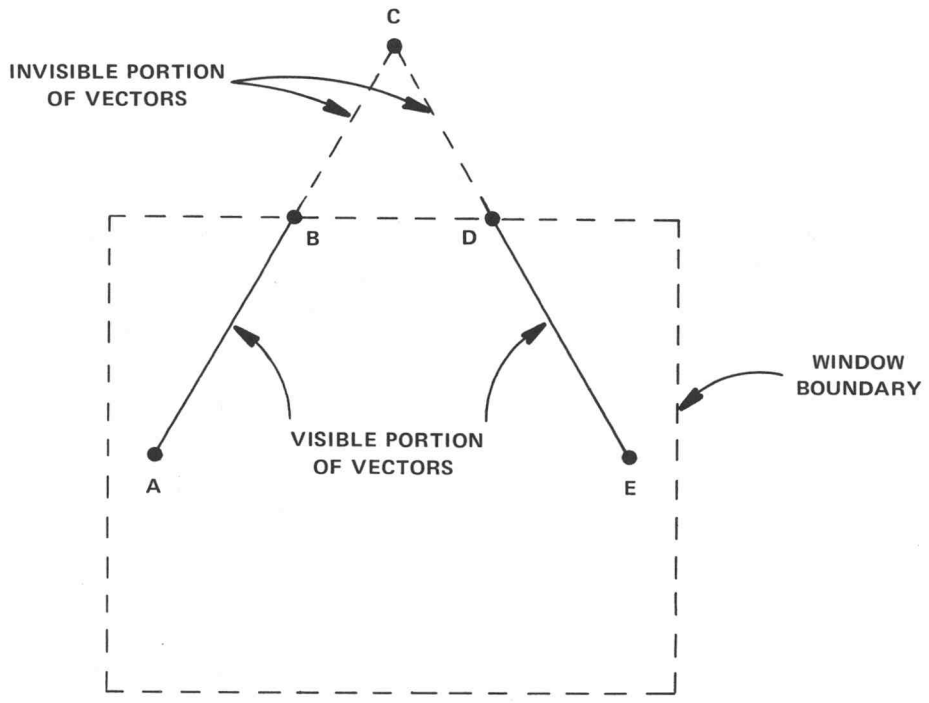


Figure 3-5. Screen window common to several data-spaces.

Since only the portions of vectors and the points which lie within the current window are displayed, the imaginary beam position does not always represent the actual storage beam position. The actual beam is represented on the user data-space by the real beam, which is updated to reflect the actual output to the terminal. Figure 3-6 illustrates the differences between the imaginary beam and the real beam.



ACTION	IMAGINARY BEAM	REAL BEAM
1) Vector drawn from A to C.	Moved from vector start point, A, to vector end point, C.	Moved from vector start point, A, to vector intercept with window boundary, B.
2) Vector drawn from C to E.	Moved from vector start point, C, to vector end point, E.	Moved from B to vector intercept, D, then move with drawing of vector to end point, E.

23398-06

Figure 3-6. Imaginary and Real Beams.

NOTE

When using a proportional graphic routine after use of direct graphics or alphanumeric output, the imaginary beam is positioned at the user data-space coordinate that is equivalent to the screen coordinate of the beam position under the current window transformation.

Cursor

It is often useful to be able to indicate a point on the user data-space with the graphic cursor. The routine CRSHRV allows you to do this by enabling the graphic cursor. After you position the graphic cursor with the thumbwheels, the screen coordinates are transmitted by pressing a keyboard character. CRSHRV constructs the data-space cursor by transforming the graphic cursor into data-space coordinates according to the current window definition (Figure 3-7). The data-space cursor does not affect the imaginary or real beam position.

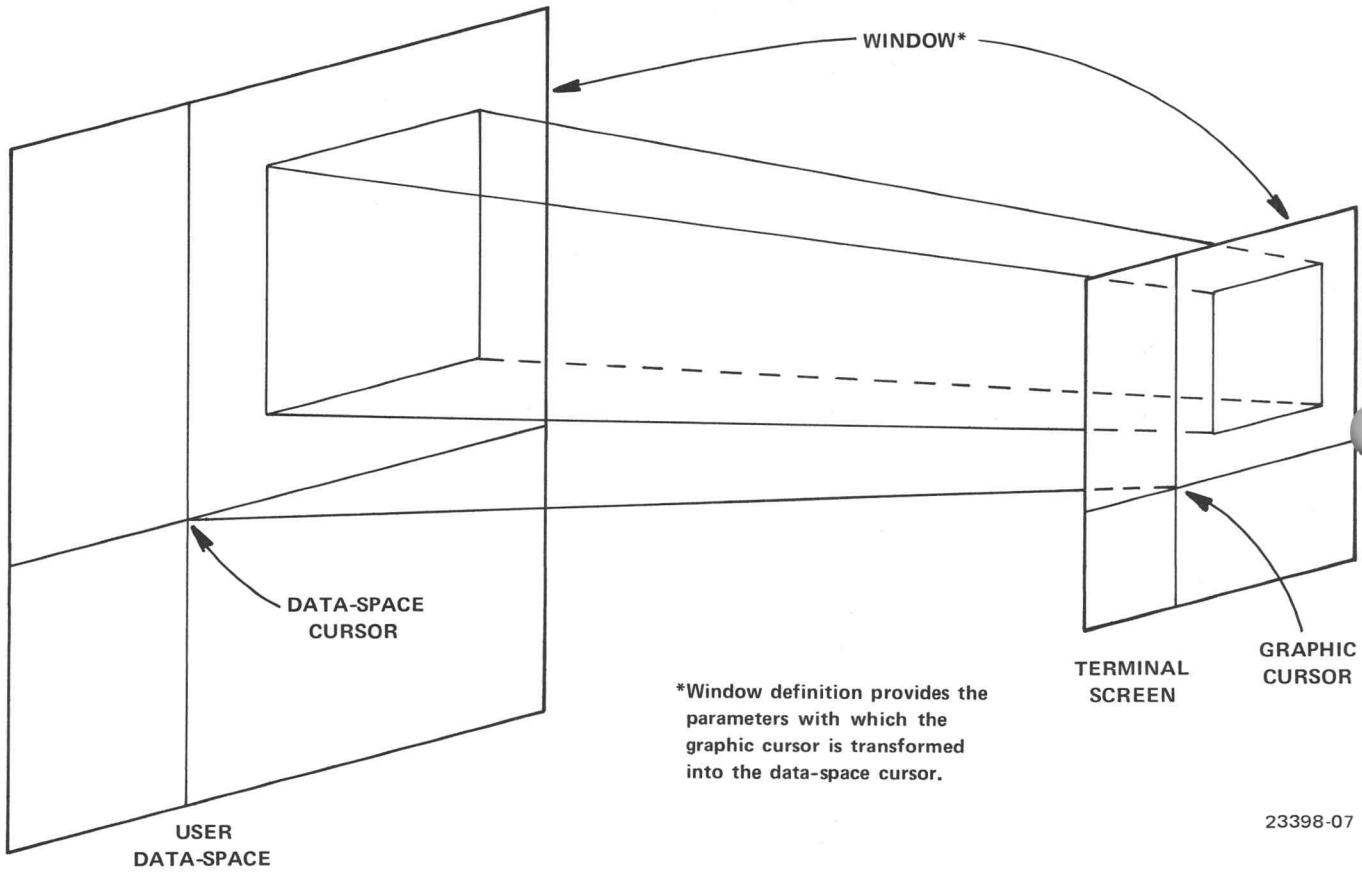


Figure 3-7. The User Data-Space Cursor.

The transformation assumes that the screen area outside the window is a continuation of the user data-space with the scale implied by the current window. This allows the user to receive valid data-space coordinate data even if the graphic cursor is positioned outside the current window.

The keyboard character which triggers input of the graphic cursor's position, is also returned as an argument. This character may be used for command purposes or data identification.

Subroutine Call: **DCOORD(xvmin,xvmax,yvmin,yvmax)**

Declaration: **SUBROUTINE DCOORD(V,V,V,V):GRAPHV**

Purpose: DCOORD defines the boundaries of the user data-space window. It confines the data plotted. The horizontal space begins at **xvmin** and extends to **xvmax**. The vertical space begins at **yvmin** and extends to **yvmax**.

Arguments: **xvmin** and **yvmin** are the coordinates of the origin.

xvmax and **yvmax** are the coordinates of the maximum point.

The coordinates (and the data plotted) must be in the range from $-1.7*10^{38}$ to $1.7*10^{38}$. Either coordinate pair can be the larger pair. For example, **xvmin** can be greater than **xvmax**, with the result that the display is a mirror image of the data.

Subroutine Call: **SCOORD(xsmin,xsmax,ysmin,ysmax)**

Declaration: **SUBROUTINE SCOORD(V,V,V,V):GRAPHV**

Purpose: SCOORD defines the screen window boundaries. Data in the user data-space window are projected into this area.

Arguments: **xsmin** defines the left side of the screen window.

xsmax defines the right side of the screen window.

ysmin defines the bottom of the window.

ysmax defines the top of the window.

The coordinates of the screen window must be kept in the ranges:

$$0 \leqslant \textbf{xsmin} < \textbf{xsmax} \leqslant 1023$$

$$0 \leqslant \textbf{ysmin} < \textbf{ysmax} \leqslant 780$$

If the coordinates are outside these ranges, the system displays an integer overflow error (AE).

Subroutine Call: **MAPOUT(xs,ys,map,xv,yv)**

Declaration: **SUBROUTINE MAPOUT(N,N,N,V,V):GRAPHV**

Purpose: Given user data-space coordinates **xv** and **yv** (defined with DCOORD), MAPOUT returns screen coordinates **xs** and **ys** and indicates if the point (**xs**,**ys**) is within the screen window (defined with SCOORD).

Arguments: **xv** and **yv** are the user data-space coordinates to be scaled and translated. **xs** and **ys** are the screen coordinates derived from **xv** and **yv**.

map indicates the screen-coordinates position relative to the screen window. The bit assignments for **map** are:

Bit	Octal Value	Position if Bit Is Set
0	1	Above screen window.
1	2	Below screen window.
2	4	Right of the screen window.
3	10	Left of the screen window.

If **map** is 0, then the coordinate is within the screen window.

For example, the value 5 indicates that the coordinate is above and to the right of the screen window.

Comments: MAPOUT attempts to convert any user data-space coordinate between $-1.7*10^{38}$ and $1.7*10^{38}$ to a screen coordinate. Screen coordinates outside the range -32768 to 32767 are given default values of -32768 or 32767.

Example: The coordinate position relative to the screen window can be determined with the MAP variable and the AND function.

```
13.01   SUBROUTINE MAPOUT(N,N,N,V,V):GRAPHV
13.05   MAPOUT(XS,YS,MAP,XV,YV)
13.11   IF(MAP)  13.13
13.12   PRINT "WITHIN"
13.13   IF(AND(MAP,#1))  13.14, 13.16
13.14   PRINT "ABOVE"
13.16   IF(AND(MAP,#2))  13.17, 13.18
13.17   PRINT "BELOW"
13.18   IF(AND(MAP,#4))  13.19, 13.20
13.19   PRINT " RIGHT"
13.20   IF(AND(MAP,#10))  13.21, 13.22
13.21   PRINT " LEFT"
13.22   PRINT CR
```

Subroutine Call: **POINTV(xv,yv,type)**

Declaration: **SUBROUTINE POINTV(V,V,V):GRAPHV**

Purpose: POINTV plots the point (**xv,yv**) on the user data-space. If the data is outside the user data-space window, no point is plotted.

Arguments: **xv** and **yv** specify the coordinates of the point and must be in the range $-1.7*10^{38}$ to $1.7*10^{38}$.

type specifies the display at (**xv,yv**). If **type** equals 1, draw a point* (.),
2, draw a minus (-),
3, draw a plus (+).

Any other value of **type** causes a point.*

*The + and - are alphanumeric characters. The point is created by intensifying the spot at the current beam position (i.e., the point is *not* an alphanumeric period). The point is brighter (more intense) than the period.

Subroutine Call: **VECTRV(xv1,yv1,xv2,yv2)**

Declaration: **SUBROUTINE VECTRV(V,V,V,V):GRAPHV**

Purpose: VECTRV draws a vector from (**xv1,yv1**) to (**xv2,yv2**) on the user data-space.

Arguments: **xv1** and **yv1** are the coordinates of the initial point of the vector.

xv2 and **yv2** are the coordinates of the second point of the vector.

The coordinates must be in the range from $-1.7*10^{38}$ to $1.7*10^{38}$.

Subroutine Call: **DRAWV(xv,yv,mode)**

Declaration: **SUBROUTINE DRAWV(V,V,V):GRAPHV**

Purpose: This subroutine draws a vector from the current imaginary beam position to **(xv,yv)** on the user data-space.

Arguments: **xv** and **yv** are the coordinates of the point.

mode defines the operation. If **mode** is:

- <0, DRAWV moves the imaginary beam position to **(xv,yv)** without drawing a visible vector and returns the terminal to the alphanumeric mode.
- = 0, DRAWV puts the terminal in the graphics mode and moves the imaginary beam position to **(xv,yv)** without drawing a visible vector.
- >0, DRAWV produces a visible vector.

Before a visible vector can be drawn, an initial call to DRAWV with **mode** equal to 0 must be made to switch the terminal to its graphic mode.

Comments: Since DRAWV internally maintains the screen coordinates of the current beam position to optimize the plot to the next point, a call to DRAWV with **mode** less than 0 or a call to CURSOR is essential when resetting the terminal to the alphanumeric mode. This tells DRAWV that it no longer knows the beam position.

Subroutine Call: **CRSHRV(char,xv,yv)**

Declaration: **SUBROUTINE CRSHRV(N,N,N):GRAPHV**

Purpose: CRSHRV turns on the cross hairs, allows you to position them with the thumbwheels, and waits for you to press a key. CRSHRV then returns the user data-space cross-hair coordinates in **(xv,yv)** and the value of the character you pressed in **char**.

Arguments: **xv** and **yv** receive the user data-space coordinates of the cross-hair cursor.

char receives the floating-point representation of the ASCII code for the character you pressed.

The characters CTRL/C, CTRL/S, CTRL/T, CTRL/V, and ALTMODE retain their normal system functions when typed in response to CRSHRV. For example, pressing CTRL/C causes the program to abort.

Subroutine Call: **ALFPSV(tstat,xv,yv)**

Declaration: **SUBROUTINE ALFPSV(N,N,N):GRAPHV**

Purpose: ALFPSV returns the user data-space coordinates of the bottom left corner of the alpha cursor in (**xv,yv**) and the terminal status in **tstat**.

Arguments: **xv** and **yv** receive the coordinates of the cursor.

tstat indicates the terminal status.* Bit assignments for **tstat** are:

Bit	Octal Value	Function if Bit Is Set
0	1	Auxiliary device is not enabled or is nonexistent.
1	2	The cursor is at margin one.
2	4	The terminal is not in the graphics mode.
3	10	Linear interpolation is off.
4	20	The hard copy unit is not ready.
5	40	This bit is always set.

tstat is normally 45_8 during alphanumeric I/O.

Example:

```
10.12  ALFPSV(TSTAT,XV,YV)
10.13  IF(AND(TSTAT,#20))  21.14
```

The program branches to statement 21.14 if the hard copy unit is not ready or if it is busy.

Subroutine Call: **LOCATV(xv,yv)**

Declaration: **SUBROUTINE LOCATV(N,N):GRAPHV**

Purpose: This subroutine returns the user data-space cross-hair coordinates in (**xv,yv**) without operator intervention.

Arguments: **xv** and **yv** are the cross-hair coordinates.

*Refer to the display terminal manual (070-1225) for more information.

Examples

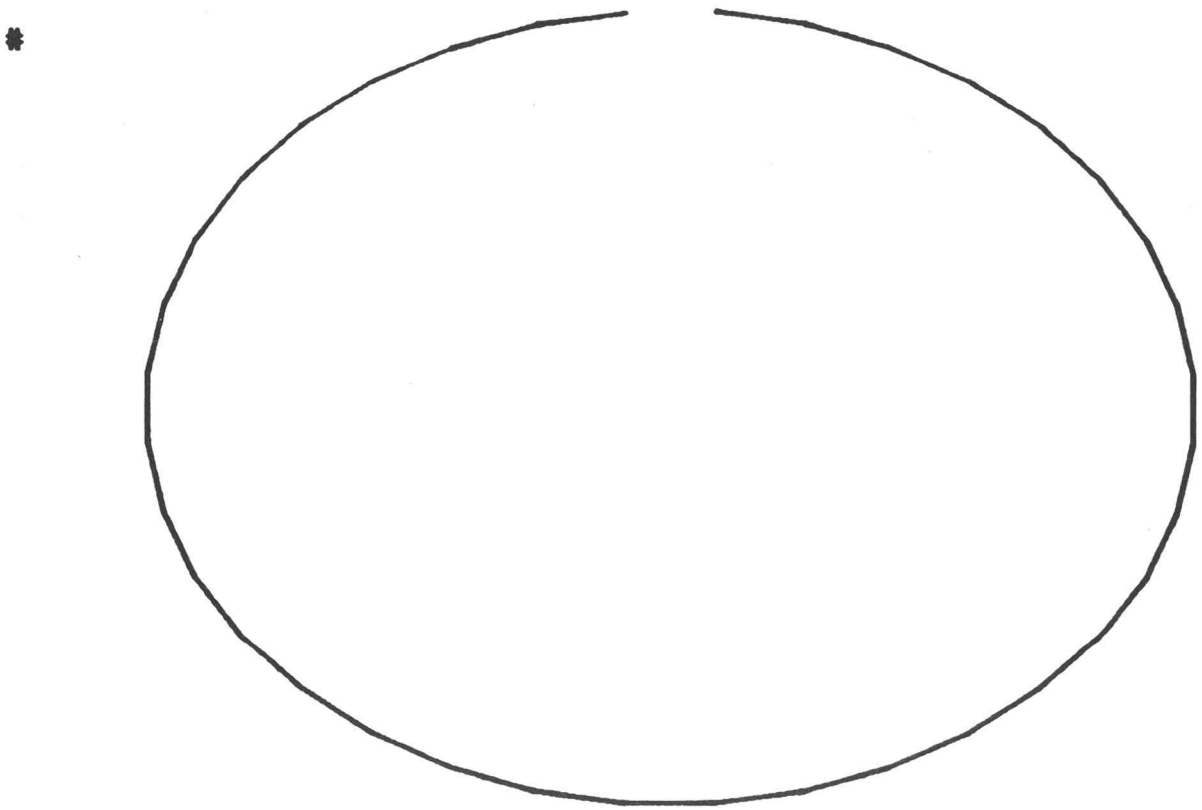
To draw an ellipse on the terminal screen, first write the program in EDIT.

```
1.0000 *  CURVE IS A PLOTTING ROUTINE TO DRAW EITHER
1.0100 *  CIRCLES OR ELIPSES OR PART CURVES ON SCREEN
1.0200 *  ALL DIMENSION REFERENCES ARE IN SCREEN POINTS
1.0300    SUBROUTINE VECTRF(V,V,V,V):GRAPH1

4.1000    ACCEPT "WHERE IS THE PLOT CENTER IN X?",X,CR
4.2000    ACCEPT "WHERE IS THE PLOT CENTER IN Y?",Y,CR
4.3000    ACCEPT "X RADIUS =  ",DX,CR
4.4000    ACCEPT "Y RADIUS (SAME AS X FOR CIRCLES) = ",DY,CR
4.5000    ACCEPT "WHAT IS THE PLOT START ANGLE IN DEGREES?",DEG,CR
4.6000    ACCEPT "WHAT IS THE PLOT STOP ANGLE IN DEGREES?",REE,CR
4.7000    ACCEPT "WHAT IS THE PLOT RESOLUTION IN DEGREES?",NUM,CR
4.8000    ACCEPT "WHAT DIRECTION TO ROTATE? 1=CCW, 0=CW " ,ROTATE,CR
4.9000    PRINT ERASE

10.0000   Z=0
10.1000   LOOP 10.5 ANGLE = DEG,REE,NUM
10.1300   ANGL = ANGLE * 3.1415926 / 180
10.1350   IF(ROTATE EQ 1) 10.18
10.1500   X2 = DX * SIN(ANGL) + X
10.1700   Y2 = DY * COS(ANGL) + Y
10.1750   GOTO 10.2
10.1800   Y2 = DY * SIN(ANGL) + X
10.1900   X2 = DX * COS(ANGL) + Y
10.2000   IF(Z NE 0) 10.25
10.2100   Z = 1
10.2200   X1 = X2
10.2300   Y1 = Y2
10.2500   VECTRF(X1,Y1,X2,Y2)
10.2600   X1 = X2
10.2700   Y1 = Y2
10.5000   CONTINUE
```

After translating the program and saving it under the name CURVE in TRAN, then run it from REDUCE.

```
$TRAN
PROGRAM NAME (EDT):  CURVE
PIN ASSIGNMENT TABLE (PIN):
TEST PROGRAM NAME (TST):  CURVE
TEST PROGRAM SIZE    945. WORDS

$REDUCE
#CURVE
WHERE IS THE PLOT CENTER IN X?500

WHERE IS THE PLOT CENTER IN Y?300

X RADIUS =  400

Y RADIUS (SAME AS X FOR CIRCLES) = 300

WHAT IS THE PLOT START ANGLE IN DEGREES?5

WHAT IS THE PLOT STOP ANGLE IN DEGREES?355

WHAT IS THE PLOT RESOLUTION IN DEGREES?10

WHAT DIRECTION TO ROTATE? 1=CCW, 0=CW 0
```

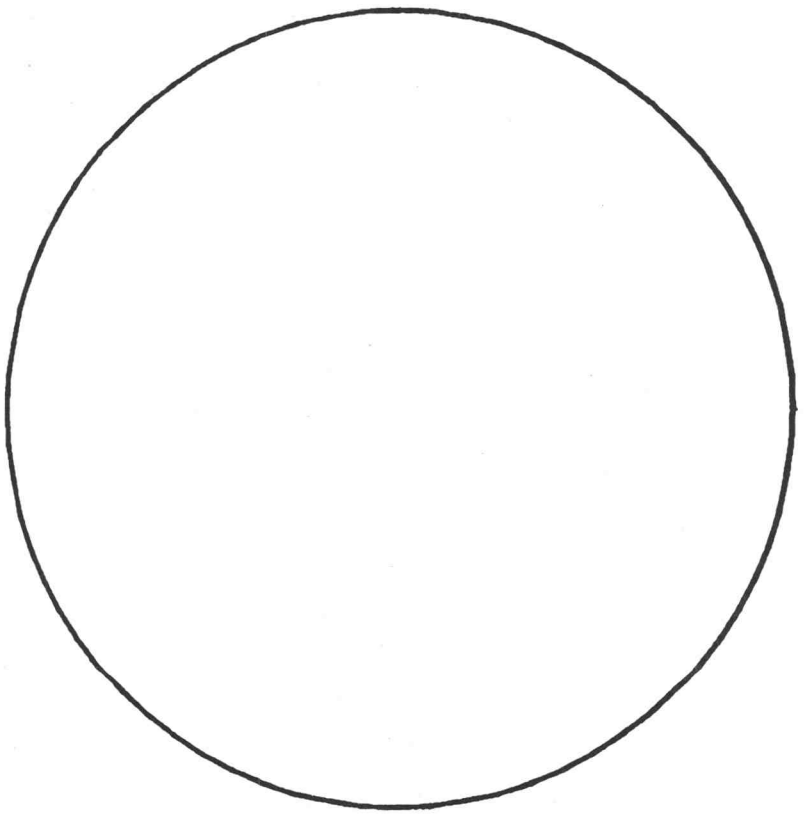
The display is:



Run the program again.

```
RUN CURVE
WHERE IS THE PLOT CENTER IN X?400

WHERE IS THE PLOT CENTER IN Y?600

X RADIUS =  300

Y RADIUS (SAME AS X FOR CIRCLES) = 300

WHAT IS THE PLOT START ANGLE IN DEGREES?0

WHAT IS THE PLOT STOP ANGLE IN DEGREES?360

WHAT IS THE PLOT RESOLUTION IN DEGREES?5

WHAT DIRECTION TO ROTATE? 1=CCW, 0=CW 1
```

The display is:



#

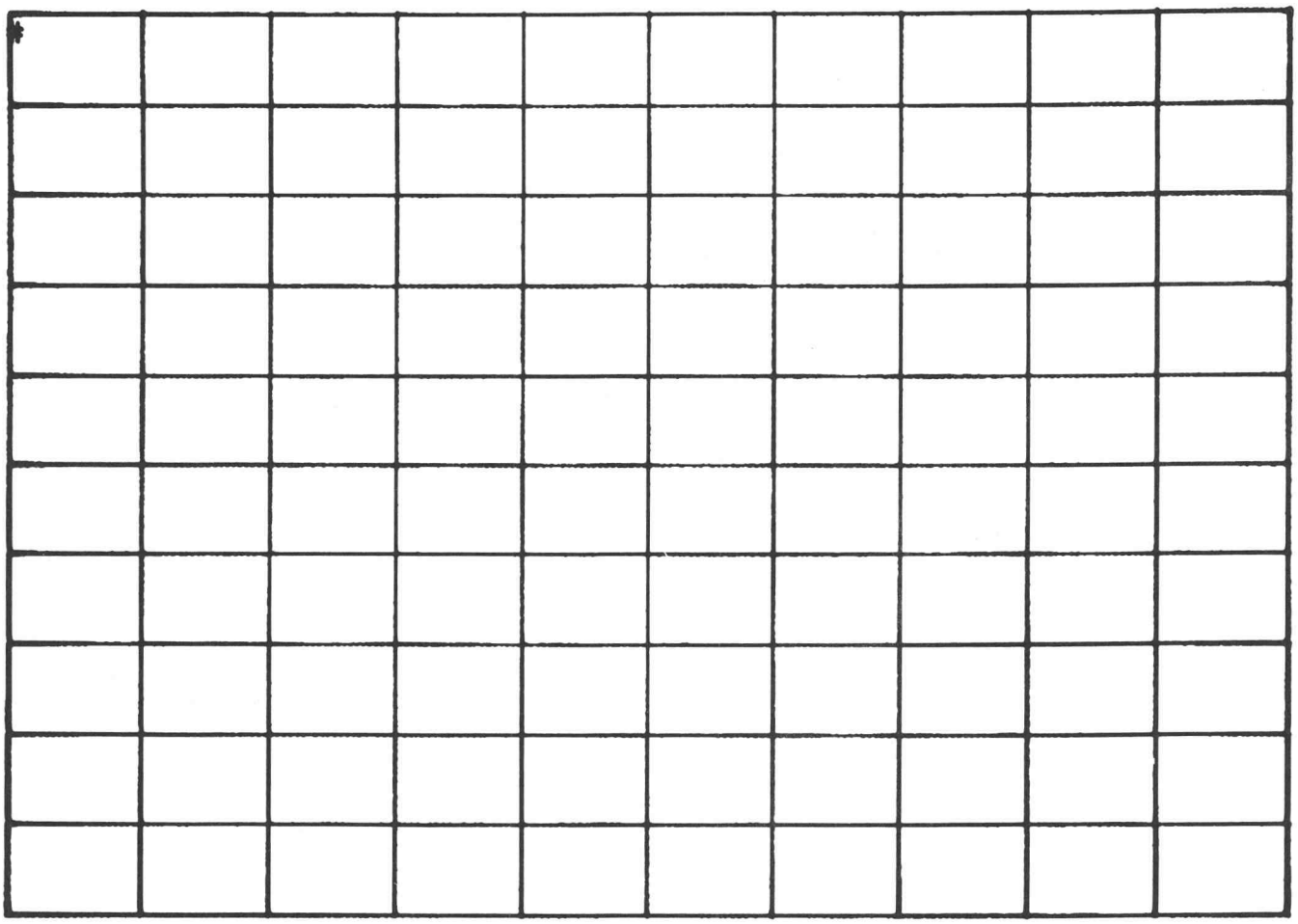
2. To draw a grid on the terminal screen, first write the program in EDIT.

```
1.0000 *  LINPLT ALLOWS THE USER TO DRAW A GRID ON SCREEN
1.0010 *  DEFINED AS NEEDED IN SIZE AND NUMBER OF DIVISIONS
1.0100    SUBROUTINE VECTRF(V,V,V,V):GRAPH1
1.0330    ACCEPT "WHAT IS THE X ORIGIN ON THE SCREEN?",XORG,CR
1.0340    ACCEPT "WHAT IS THE Y ORIGIN ON THE SCREEN?",YORG,CR
1.0350    ACCEPT "WHAT IS THE MAX X DIMENSION?",XFIN,CR
1.0360    ACCEPT "WHAT IS THE MAX Y DIMENSION?",YFIN,CR
1.0400    ACCEPT "ENTER NUMBER OF HORIZONTAL DIVISIONS",CR,NUM
1.0500    ACCEPT "ENTER NUMBER OF VERTICAL DIVISIONS",CR,YNUM
1.0800    XINC=(XFIN-XORG)/NUM
1.0900    YINC=(YFIN-YORG)/YNUM

2.0200    PRINT ERASE

3.0000 *  VERTICAL LINES
3.0100    LOOP 3.9 A=XORG,XFIN,XINC
3.0200    VECTRF(A,YORG,A,YFIN)
3.9000    CONTINUE

4.0000 *  HORIZONTAL LINES
4.1000    LOOP 4.9 B=YORG,YFIN,YINC
4.2000    VECTRF(XORG,B,XFIN,B)
4.9000    CONTINUE
```

After translating the program and saving it under the name LINPLT in TRAN, run it from REDUCE.

```
$REDUCE
#RUN LINPLT
WHAT IS THE X ORIGIN ON THE SCREEN?1

WHAT IS THE Y ORIGIN ON THE SCREEN?1

WHAT IS THE MAX X DIMENSION?1001

WHAT IS THE MAX Y DIMENSION?700

ENTER NUMBER OF HORIZONTAL DIVISIONS
10
ENTER NUMBER OF VERTICAL DIVISIONS
10
```

The display is:

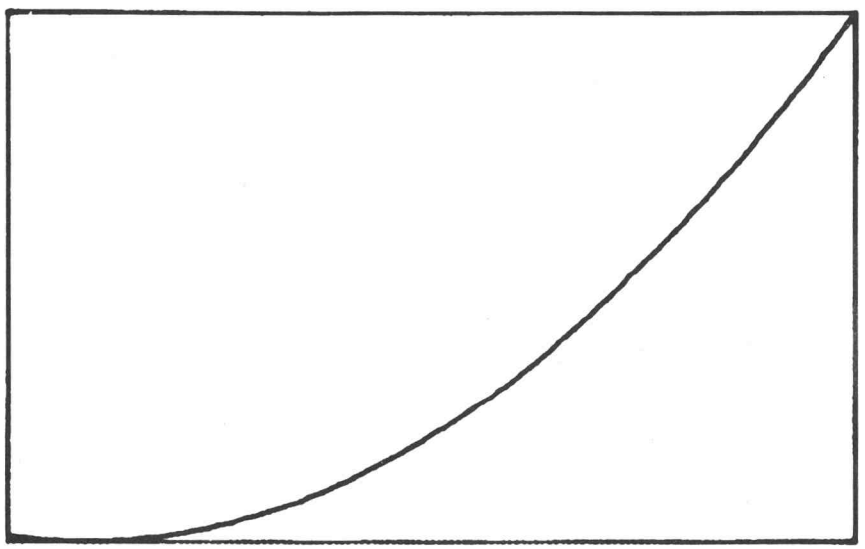


3. To draw a parabola, first write the program in EDIT. (This example uses the proportional graphic routines.)

```
1.0100    SUBROUTINE VECTRF(V,V,V,V),DRAWV(V,V,V),CURSOR(V,V):GRAPHV
1.0200    SUBROUTINE SCOORD(V,V,V,V),DCOORD(V,V,V,V)

2.0100    N=13E-13
2.0120 *   SETTHE SCREEN WINDOW BOUNDARIES
2.0130    SCOORD(200,847,200,600)
2.0200 *   SET THE DATA-SPACE WINDOW FOR PART OF THE N**2 PARABOLA
2.0300    DCOORD(-0.1*N,N,0,N**2)

3.0000 *   DRAW A FRAME AROUND THE SCREEN WINDOW
3.0100    VECTRF(200,200,847,200)
3.0200    VECTRF(847,200,847,600)
3.0300    VECTRF(847,600,200,600)
3.0400    VECTRF(200,600,200,200)

4.0000 *    ATTEMPT TO DRAW THE ENTIRE PARABOLA
4.0100    DRAWV(-N,-N**2,0)
4.0200    LOOP 4.04 X=-N,N,.01*N
4.0300    Y=X**2
4.0400    DRAWV(X,Y,1)

5.0100    CURSOR(300,167)
5.0200    PRINT "GRAPH OF Y=X**2"
5.0300    CURSOR(0,767)
```

After translating the program and saving it under the name EXPYX in TRAN, run the program from REDUCE.

```
RUN EXPYX
#
```



GRAPH OF Y=X**2

SECTION FOUR:

STRING HANDLING SUBPROGRAMS

The string handling subprograms enable your program to perform these operations:

1. Move characters between and within arrays.
2. Send characters to output devices.
3. Receive characters from input devices.
4. Test for relationships between strings.

The string subprograms are in two files: STRING and ADSTNG.

The ADSTNG file contains the same subprograms as the STRING file. In addition, ADSTNG has code conversion subprograms (not available in STRING) which allow you to perform these operations:

1. Convert ASCII strings to Radix-50 code and floating-point numbers.
2. Convert Radix-50 code and floating-point numbers to ASCII strings.

The STRING file uses about half the memory required by the ADSTNG file. If you desire the capability to manipulate ASCII character strings, but do not need the code conversion facilities, use the STRING file. To include the code conversion subprograms, call the string handling subprograms from the ADSTNG file, excluding the STRING file. The subprograms are assembled in this manner because they use common internal subroutines which cannot be linked externally by the Translator.

Storage of ASCII Character Strings

ASCII character strings can reside in integer arrays. Each array element contains two characters. For example, the first array element contains the first and second string characters. To individually address the characters stored in an array, the string handling subprogram arguments are based on character numbers rather than array element numbers. Arguments such as **start**, **stop**, and **position** refer to a character number, *not* an array element. For example, character number 13 is in array element 7. Incrementing a character-number-based variable causes it to reference the next character in the string.

An array element range of M_1 to M_2 has a character number (n) range of $2M_1 - 1 \leqslant n \leqslant 2M_2$, where $M_2 \geqslant M_1 > 0$. (Zero is an illegal character number.) Declare integer arrays with a size of one-half the number of characters in the largest string expected. (Round up to the next integer, if necessary.)

The number of characters in a string range from **start** to **stop** is calculated as (**stop** - **start** + 1). For example, if start is 5 and stop is 15, then the number of characters is 11. If **start** and **stop** have the same value, then one character is in the range.

The string handling subprograms return a subscript error message (AC) for any of the following reasons.

1. A character number is less than one.
2. A character number exceeds twice the array size.
3. The ending character number in a range is less than the starting character number of that range.

Inputting Strings from the Keyboard

Special control characters used by the system (CTRL/C) by the REDUCE program (CTRL/S, CTRL/T, CTRL/V), and by Terminal Control Mode (S-3260/S-3030) (CTRL/N, CTRL/O, CTRL/P, CTRL/Q, CTRL/R, CTRL/S, CTRL/T) are not placed in the terminal input queue. Therefore, these control characters are not input by STRNGI and CHARI, nor can KBSTAT sense their presence.

Summary of Subprograms

These subprograms are in both the STRING and ADSTNG files.

Subroutine	Declaration	Purpose
CHARO(olun,char)	CHARO(V,V)	Sends the specified character to the output device.
CLRKB	CLRKB(0)	Clears the keyboard input queue.
JUSTFY(side,tally,string, start,stop)	JUSTFY(V,N,I,V,V)	Removes imbedded spaces and justifies the string.
SCON(dststr,start,stop, "stringconstant")	SCON(I,V,V,C)	Stores a string constant into the destination string.
SMOV(dststr,start1,stop1, srcstr,start2,stop2)	SMOV(I,V,V,I,V,V)	Moves the source string into the destination string.
STRNGI(ilun,count,iarray, start,stop)	STRNGI(V,N,I,V,V)	Inputs a string from the input device into iarray.
STRNGO(olun,iarray,start, stop)	STRNGO(V,I,V,V)	Sends the specified characters to the output device.
STRNGS(char,iarray,position)	STRNGS(V,I,V)	Stores the ASCII value of the character char.

Function	Declaration	Purpose
CHARI(ilun)	CHARI(V)	Returns the floating-point value of the input ASCII character.
CMPCON(string,start,stop, "stringconstant")	CMPCON(I,V,V,C)	Compares a string with a string constant.
KBSTAT	KBSTAT(0)	Returns the input queue status.
SCMP(string1,start1,stop1, string2,start2,stop2)	SCMP(I,V,V,I,V,V)	Compares two strings.
STRNGF(iarray,position)	STRNGF(I,V)	Returns the floating-point value of an ASCII character.

These subprograms are in the ADSTNG file only.

Subroutines	Declaration	Purpose
DFLTYP("typ")	DFLTYP(C)	Sets the default file type for the file descriptor in PAKFIL calls.
DFLUID("uid")	DFLUID(C)	Sets the default user identification code for the file descriptor in PAKFIL calls.
NUMOUT(value,code,tally, string,start,stop)	NUMOUT(V,V,N,I,V,V)	In accordance with the selected format, converts a floating-point value into ASCII characters.
PAKFIL(file,delim,string, start,stop)	PAKFIL(F,N,I,N,V)	Packs a string into a Radix-50 four-word file-descriptor.
PAKSYM(symbol,delim, string,start,stop)	PAKSYM(S,N,I,N,V)	Packs a string into a two-word Radix-50 symbol.
RADPAK(value,delim,string, start,stop)	RADPAK(N,N,I,N,V)	Packs a string into a two-word Radix-50 variable.
RADUP(value,string,start,stop)	RADUP(V,I,V,V)	Unpacks two Radix-50 words into a six-character ASCII string.
UNPFIL(file,string,start,stop)	UNPFIL(F,I,V,V)	Unpacks a four-word Radix-50 file-descriptor into a 14-character ASCII string.
UNPSYM(symbol,string, start,stop)	UNPSYM(S,I,V,V)	Unpacks a two-word Radix-50 symbol into a six-character ASCII string.

Function	Declaration	Purpose
FMTNUM(delim,string, start,stop)	FMTNUM(N,I,N,V)	Converts an ASCII string into a floating-point number.
SPFMT(delim,string,start, stop,dtabl)	SPFMT(N,I,N,V,I)	Converts an ASCII string into a floating-point number using the supplied delimiter table.

Function Call: **STRNGF(iarray,position)**

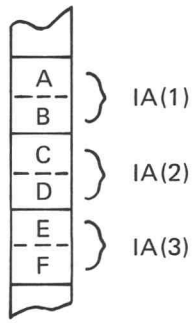
Declaration: **FUNCTION STRNGF(I,V):** {STRING / ADSTNG}

Purpose: This is the string fetch function. STRNGF returns the floating-point value of the character at **position** in **iarray**.

Arguments: **iarray** is an integer array name. **position** is a character number within the array range.

Result: The result is a value (n) in the range $0 \leqslant n \leqslant 255$. This value represents an ASCII character.

Example: E = STRNGF (IA,5)



E would equal 69, which is the ASCII value of E.

Subroutine Call: **STRNGS(char,iarray,position)**

Declaration: **SUBROUTINE STRNGS(V,I,V):** {STRING / ADSTNG}

Purpose: This is the string storing subroutine. STRNGS stores the ASCII value of **char** at **position** in **iarray**.

Arguments: The value of **char** is the floating-point value of an ASCII character. It must be in the range $0 \leqslant$ **char** $\leqslant 255$.

iarray is an integer array name. **position** is a character number within the array range.

Subroutine Call: **STRNGO(olun,iarray,start,stop)**

Declaration: **SUBROUTINE STRNGO(V,I,V,V):** {STRING / ADSTNG}

Purpose: This is the string output subroutine. STRNGO sends to the selected output device the characters from **iarray** starting at **start** and ending at **stop**.

Arguments: **olun** must be assigned to an output device or file.

iarray is an integer array name. **start** is the character number of the first character to transfer. **stop** is the character number of the last character to transfer.

Example:

```
1.0100  IARRAY OUT(10)
1.0200  SUBROUTINE STRNGO(V,I,V,V):STRING

2.0200  OUT(1)=72
2.0300  OUT(2)=69
2.0400  OUT(3)=76
2.0500  OUT(4)=76
2.0600  OUT(5)=79

3.0000  PRINT ERASE
3.0100  STRNGO(0,OUT,1,10)

4.0100  OUT(1)=17736
4.0200  OUT(2)=19532
4.0300  OUT(3)=00079
4.0400  OUT(4)=0
4.0500  OUT(5)=0

5.0500  PRINT CR,CR,CR
5.0600  STRNGO(0,OUT,1,10)
```

If you run this program under the control of the REDUCE program with lun 0 assigned to the terminal, the display produced is:

```
HELLO

HELLO
```

The terminal ignores all nulls.

Subroutine Call: **STRNGI(ilun,count,iarray,start,stop)**

Declaration: **SUBROUTINE STRNGI(V,N,I,V,V):** {STRING / ADSTNG}

Purpose: This is the string input subroutine. STRNGI reads a string, terminated with CR and LF, from the selected input device and stores it in **iarray** from **start** to **stop**. The number of characters in the string is returned in **count**.

Arguments: **ilun** must be assigned to an input device or file.

STRNGI stores the number of characters stored in **iarray** as a result of this call in **count**.

iarray is an integer array name. **start** selects the character number in **iarray** where the first character in the input string is stored. **stop** is the character number at which the transfer stops.

Comments: A string that is shorter than the allotted space terminates with a CR and LF, and the remaining space is null filled. STRNGI truncates at **stop** a string that is longer than the allotted space and does not store CR and LF.

Subroutine Call: **CHARO(olun,char)**

Declaration: **SUBROUTINE CHARO(V,V):** {STRING / ADSTNG}

Purpose: This is the character output subroutine. CHARO sends the character **char** to the selected output device.

Arguments: **olun** must be assigned to an output device or file.

The value of **char** is the floating-point value of an ASCII character. It must be in the range $0 \leqslant$ **char** $\leqslant 255$.

char can have a value of 0 and parity ASCII only if the output device is the paper tape punch.

CHARO does not store parity ASCII and nulls on mass storage devices (disk and magnetic tape). As a result, **char** is a seven-bit value ($1 \leqslant$ **char** $\leqslant 127$) on mass storage devices.

Function Call: **CHARI(ilun)**

Declaration: **FUNCTION CHARI(V):** {STRING / ADSTNG}

Purpose: This is the character input function. CHARI returns the floating-point value of the ASCII character received from the input device.

Arguments: **ilun** must be assigned to an input device or file.

Result: The result is a floating-point number (n) in the range $1 \leqslant n \leqslant 127$. It is the code for an ASCII character. CHARI trims parity bits; it does not check parity.

Comments: With mass storage devices, CHARI interprets the first null character encounted as the end-of-file. If the input device is the paper tape reader, you may input nulls. However, CHARI ignores them. This allows you to start a paper tape on the leader preceding the ASCII code.

Subroutine Call: **SMOV(dststr,start2,stop2,srcstr,start1,stop1)**

Declaration: **SUBROUTINE SMOV(I,V,V,I,V,V):** {STRING / ADSTNG}

Purpose: This is the move-string subroutine. SMOV transfers characters from the source string array **srcstr** into locations in the destination string **dststr**.

Arguments: **dststr** is an integer array name. **start2** specifies the starting character number in **dststr**. **stop2** specifies the ending character number in **dststr**.

srcstr is an integer array name. **start1** specifies the starting character number in **srcstr**. **stop1** specifies the ending character number in **srcstr**.

Comments: If the destination string-space is shorter than the source string-length, SMOV truncates the source string. If the destination string-space is longer than the source string-length, SMOV fills the trailing character locations in the destination string-space with ASCII spaces.

Subroutine Call: **SCON(dststr,start,stop,"stringconstant")**

Declaration: **SUBROUTINE SCON(I,V,V,C):** { **STRING** / **ADSTNG** }

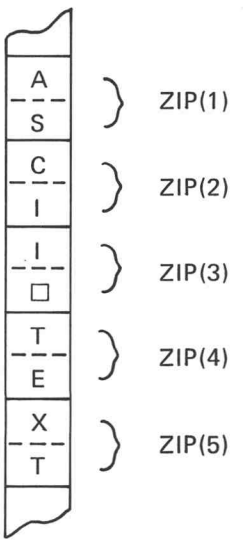
Purpose: SCON moves the string **stringconstant** into locations in the destination string **dststr.**

Arguments: **dststr** is an integer array name. **start** specifies the starting character number in **dststr. stop** specifies the ending character number in **dststr.**

stringconstant is an ASCII character string enclosed in paired delimiters. " is an arbitrarily chosen delimiter. Any character may be used as the delimiter, as long as it does not appear in **stringconstant.**

Comments: If the destination string-space is shorter than the string constant, SCON truncates the string. If the destination string-space is longer than the string constant, SCON fills the trailing character locations in the destination string space with ASCII spaces.

Examples: SCON(ZIP,1,10,"ASCII TEST")



Function Call: **SCMP(string1,start1,stop1,string2,start2,stop2)**

Declaration: **FUNCTION SCMP(I,V,V,I,V,V):** {STRING / ADSTNG}

Purpose: This is the string comparison function. SCMP compares **string1** with **string2** one character at a time by their ASCII representations. The first character difference determines the value returned. The value returned is:

-1 if the value of the **string1** character is less than the **string2** character value.

0 if **string1** is identical to **string2**.

+1 if the value of the **string1** character is greater than the **string2** character value.

If one string ends before a difference is found, SCMP considers the shorter string to be the one of less value.

Arguments: **string1** is the name of an integer array that contains an ASCII string. **start1** specifies the starting character number in **string1**. **stop1** specifies the ending character number in **string1**.

string2 is the name of an integer array that contains an ASCII string. **start2** specifies the starting character number in **string2**. **stop2** specifies the ending character number in **string2**.

Function Call: **CMPCON(string,start,stop,"stringconstant")**

Declaration: **FUNCTION CMPCON(I,V,V,C):** {STRING / ADSTNG}

Purpose: CMPCON compares **string** with **stringconstant** one character at a time by their ASCII representations. The first character difference determines the value returned. The value returned is:

-1 if the value of the **string** character is less than the **stringconstant** character value.

0 if **string** is identical to the **stringconstant**.

+1 if the value of the **string** character is greater than the **stringconstant** character value.

If one string ends before a difference is found, CMPCON considers the shorter string to be the one of less value.

Arguments: **string** is the name of an integer array that contains an ASCII string. **start** specifies the beginning character number in **string.** **stop** specifies the ending character number in **string.**

stringconstant is an ASCII character string enclosed in paired delimiters. " is an arbitrarily chosen delimiter. Any character may be used as a delimiter, as long as it does not appear in the character string.

Subroutine Call: **CLRKB**

Declaration: **SUBROUTINE CLRKB(0):** {STRING / ADSTNG}

Purpose: This is the keyboard clear subroutine. CLRKB deletes characters which have been typed at the terminal but not yet requested by a program.

Function Call: **KBSTAT**

Declaration: **FUNCTION KBSTAT(0):** {STRING / ADSTNG}

Purpose: This is the keyboard status function. The result of KBSTAT is:

1 if the queue is empty and the terminal is local.

2 if the queue is not empty and the terminal is local.

-1 if the queue is empty and the terminal is remote.

-2 if the queue is not empty and the terminal is remote.

Subroutine Call: **JUSTFY(side,tally,string,start,stop)**

Declaration: **SUBROUTINE JUSTFY(V,N,I,V,V):** {**STRING** / **ADSTNG**}

Purpose: This is the string-justify subroutine. JUSTFY packs the text between **start** and **stop** in **string**. It removes any imbedded spaces and returns in **tally** the count of all the non-space characters. When JUSTFY finishes, all non-space characters are packed at one end of **string**. JUSTFY fills the remainder of **string** with spaces.

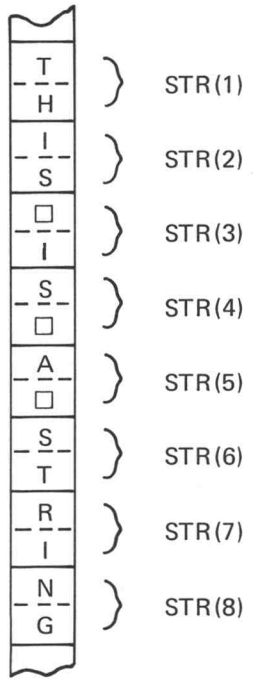
Arguments: The value of **side** tells JUSTFY to pack in a specific direction: pack **string** starting at **start** if side equals 0, or pack **string** toward **stop** if side is not equal to 0.

tally is the variable name that receives the count of non-space characters in the selected range of **string.**

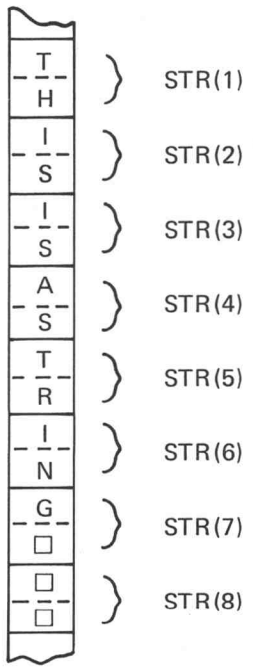
string is an integer array name. **start** specifies the starting character number. **stop** specifies the ending character number.

Comments: Since JUSTFY removes all spaces, use the comma rather than the space to set apart command parameters if the command string will be saved and packed.

Example: If the integer array STR contains:



Then a call to JUSTFY(0,TALLY,STR,1,16) results in TALLY equal to 13 and STR containing:



Function Call: **FMTNUM(delim, string, start, stop)**

Declaration: **FUNCTION FMTNUM(N,I,N,V):ADSTNG**

Purpose: FMTNUM converts an ASCII string to a line number, Radix-50 value or a floating-point number, depending on the string format. FMTNUM scans **string** from **start** until it encounters a delimiter or until it reads the character at **stop**.

FMTNUM returns the value of the delimiter ending the scan in **delim** and returns the string character number of the position following the last character read in **start.**

Arguments: **delim** receives the floating-point value of the ASCII code for the delimiting character. If FMTNUM ends at **stop**, it returns 255 in **delim**. The delimiters recognized by FMTNUM are: space, carriage return, semicolon, and comma.

string is an integer array name. The ASCII strings stored in **string** must be in the format:

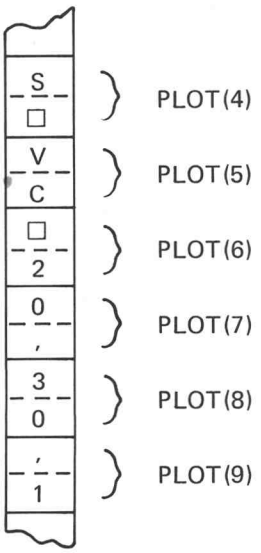
number,
L"linenumber",
$linenumber,
'asciicharacter,
C"asciicharacter", or
S"rad50symbol".

start is a variable name that contains the starting character number. It also receives the character number of the position following the last character read. **start** is now ready for another scan of **string.**

stop contains the ending character number.

Comments: If the ASCII string read does not have the correct format, FMTNUM returns a negative zero and stores a -1 in **delim.**

Example: If PLOT contains:



and, START equals 12. Then, the statement:

```
VAL = FMTNUM(DELIM,PLOT,START,100)
```

returns with VAL equal to 20, DELIM equal to 44 (the ASCII value of ,), and START equal to 15.

Function Call: **SPFMT(delim,string,start,stop,dtabl)**

Declaration: **FUNCTION SPFMT(N,I,N,V,I):ADSTNG**

Purpose: SPFMT converts an ASCII string to a line number, Radix-50 value, or a floating-point number, depending on the string format. SPFMT scans **string** from **start** until it encounters a delimiter or until it reads the character at **stop**.

SPFMT returns the value of the delimiter ending the scan in **delim** and returns the string character number of the position following the last character read in **start.**

Arguments: **delim** receives the floating-point value of the ASCII code for the delimiting character. If SPFMT ends at **stop**, it returns 255 in **delim**. The delimiters recognized are defined in **dtabl**.

string is an integer array name. The ASCII strings stored in string must be of the following formats:

number,
L"linenumber",
$linenumber,
'asciicharacter,
C"asciicharacter", or
S"rad50symbol".

start is a variable name that contains the starting character number. It also receives the character number of the position following the last character read. **start** is now ready for another scan of string.

stop contains the ending character number.

dtabl is the name of an integer array that contains a list of delimiters. The last entry in the array must be a null character (0). The following characters cannot be used as delimiters:

" # $ ' + - . 0 1 2 3 4 5 6 7 8 9 A C E F H K L M N P S U V

Comments: If the ASCII string read does not have the correct format, SPFMT returns a -0 and stores a -1 in **delim.**

Subroutine Call: **NUMOUT(value,code,tally,string,start,stop)**

Declaration: **SUBROUTINE NUMOUT(V,V,N,I,V,V):ADSTNG**

Purpose: NUMOUT converts the floating-point number in **value** into ASCII characters and stores them in **string. code** specifies the conversion mode. After conversion, **tally** contains the count of the characters deposited in **string.**

Arguments: **value** contains the number to be converted.

code specifies the format code of the conversion mode. The format code can be represented as a three-digit octal number. Each digit controls one aspect of the conversion. Denoting the digits as $d_2d_1d_0$, the digits have the following meanings:

d_0 specifies the output format.

d_1 is the digit count.

d_2 is the compact flag.

The digit count (d_1) specifies the number of significant digits for floating-point, exponential, integer, and octal notation. For real notation, it specifies the number of digits to the right of the decimal point in a constant nine-character field. In line-number format, the digit count must be zero. For the special output, d_1 selects the character output type.

Output Format	d_0	Digit Count (d_1)	Example of Format
Floating point	0	$3 \leqslant d_1 \leqslant 7$	7.025M
Exponential	1	$2 \leqslant d_1 \leqslant 7$	4.957E+13
Real	2	$0 \leqslant d_1 \leqslant 7$	.003
Integer	3	$0 \leqslant d_1 \leqslant 7$	9425
Octal	4	$1 \leqslant d_1 \leqslant 7$	001042
Line number	5	$d_1 = 0$	12.0500
Special	6	$d_1 = 1$ for expanded Radix-50 symbol $d_1 = 5$ for ASCII character	

If the compact digit (d_2) is 0, then NUMOUT stores one leading and one trailing space with the output. If the compact digit is 1, NUMOUT omits these spaces. NUMOUT fills unused character positions in the string range with spaces.

Format codes not defined above cause an error condition, and NUMOUT stores a -1 in **tally**. The system does not give an error message.

tally is a variable name that receives the number of characters NUMOUT stored in **string.**

string is an integer array name. **start** selects the starting character number. **stop** specifies the ending character number.

PRINT Statement Format Code	NUMOUT Format Codes NUMOUT code Value		Remarks
	Octal	Decimal	
F	040	32	Floating point notation. Default to four digits if the digit field is blank. Print suffixes P, N, U, M, K where appropriate. Use E format if X<1E-15 or X≥1E+6.
F3	030	24	
F4	040	32	
F5	050	40	
F6	060	48	
F7	070	56	
E	051	41	Exponential notation (e.g., 7.23E+09). Default to five digits if the digit field is blank.
E2	021	17	
E3	031	25	
E4	041	33	
E5	051	41	
E6	061	49	
E7	071	57	
R0	002	2	Real number notation (e.g., R2→ -76543.21).
R1	012	10	
R2	022	18	
R3	032	26	
R4	042	34	
R5	052	42	
R6	062	50	
R7	072	58	
I	053	43	Integer notation. Default to five digits if the digit field is blank.
I0	003	3	
I1	013	11	
I2	023	19	
I3	033	27	
I4	043	35	
I5	053	43	
I6	063	51	
I7	073	59	
O	064	52	Octal notation (0076). Print leading zeroes. Default to six digits if the digit field is blank.
O1	014	12	
O2	024	20	
O3	034	28	
O4	044	36	
O5	054	44	
O6	064	52	
O7	074	60	
L	005	5	Line number notation.
S	016	14	Radix-50 symbol output.
C	056	46	Character output.

Subroutine Call: **PAKSYM(symbol,delim,string,start,stop)**

Declaration: **SUBROUTINE PAKSYM(S,N,I,N,V):ADSTNG**

Purpose: PAKSYM packs ASCII chracters into the two-word **symbol**. The characters are from the integer array **string** starting at **start** and ending at **stop**, the sixth character, or the first non-Radix character. PAKSYM stores the first non-Radix character in **delim** and the character number of the position following the last character read in **start.**

Arguments: **symbol** is a Radix-50 symbol that receives the result.

delim is a variable name that receives the ASCII value of the delimiter. If PAKSYM ends at **stop**, it returns the value 255 in **delim.**

string is the name of an integer array that contains ASCII characters. **start** is the name of a variable that contains the starting character number. It receives the character number of the position following the last delimiter read. **stop** specifies the ending character number.

Comments: PAKSYM ignores leading spaces and tabs, then packs up to six characters in **symbol**. If less than six characters are found before the next space, period, or any non-Radix-50 character, the routine pads **symbol** on the right with spaces.

If the range of **string** is longer than six alphanumeric characters, PAKSYM packs the first six, then scans for the delimiter at the end of the alphanumeric characters. If only spaces follow the alphanumeric characters, then the delimiter is a space, unless **stop** was encountered, and **start** indexes to the next alphanumeric character. If a delimiter imbedded in spaces follows the alphanumeric characters, then the delimiter goes into **delim** and **start** indexes to the following space.

If the first character in **string**, after leading spaces, is not a valid Radix-50 character, PAKSYM stores a 0 in **symbol** and a -1 in **delim** as an error indicator. The system does not give an error message.

Example: After execution of PAKSYM, all references to **symbol** reflect its new value. Thus, if SM2 is called by SM2(SCOPEA,MAT1) and PAKSYM(MAT1,D,A,SR,10) is executed, SM2 connects SCOPEA to the specified range in string A.

NOTE

PAKSYM changes only the names of symbols.

Subroutine Call: **RADPAK(value,delim,string,start,stop)**

Declaration: **SUBROUTINE RADPAK(N,N,I,N,V):ADSTNG**

Purpose: RADPAK packs the variable **value** with the Radix-50 equivalent of ASCII characters. The characters are from the integer array **string** starting at **start** and ending at **stop**, the sixth character, or a non-alphanumeric character. RADPAK then stores the first non-alphanumeric character in **delim** and the character number of the position following the last character read in **start.**

Arguments: **value** receives the result.

delim receives the ASCII value of the delimiter. If RADPAK ends at **stop**, it returns the value 255 in **delim.**

string is the name of an integer array that contains the ASCII characters. **start** is a variable name that contains the starting character number. It receives the character number of the position following the last character read. **stop** specifies the ending character number.

Comments: See the comments for PAKSYM for more information.

Subroutine Call: **PAKFIL(file,delim,string,start,stop)**

Declaration: **SUBROUTINE PAKFIL(F,N,I,N,V):ADSTNG**

Purpose: PAKFIL packs ASCII characters into the four-word Radix-50 file descriptor **file**. The characters are from the integer array **string**, starting at **start** and ending at **stop**, the 14th character, or a delimiter. PAKFIL then stores the delimiter in **delim** and the character number of the position following the last character read in **start.**

Arguments:

file is a four-word file descriptor receiving the result.

delim is a variable name that receives the delimiter. If PAKFIL ends at **stop**, 255 is stored in **delim.**

string is an integer array name that contains ASCII characters.

start is a variable name that contains the starting character number. It receives the character number of the position following the last delimiter read.

stop specifies the ending character number.

Comments: PAKFIL packs up to 12 characters from **string**, in standard file descriptor format (filnam.typ:uid), into four Radix-50 words. PAKFIL packs the first one to six alphanumeric characters into the first two words of the file descriptor. If a period is the next delimiter, it packs the next zero to three characters into the third word. If a colon is the next delimiter, it packs the next zero to three characters in the fourth word.

In every case, PAKFIL ignores leading spaces and tabs, converts from zero to three (or six) characters, then scans for the next delimiter, ignoring trailing spaces and tabs. If only spaces or tabs precede the next alphanumeric character, then the space or tab ahead of that alphanumeric character is the delimiter for that file name. If less than three Radix-50 characters exist in a word, PAKFIL pads with spaces on the right.

Several file descriptors can be stored in **string** and each processed with a call to PAKFIL. **start** contains the character number for scanning the next file descriptor in the string.

If the first character after the leading spaces is not a valid Radix-50 character, the subroutine stores a zero in the first word of the file descriptor and a -1 in **delim** as an error indicator. The system does not issue any error messages.

Example:

All references to **file** reflect the new name packed by PAKFIL. For example, after the sequence:

```
SR=1
SCON(A,SR,8,"ZAPO:SYS")
PAKFIL(ZORCH:MIN,DEL,A,SR,8)
ASTORE(DATARY,ZORCH:MIN,1,11.02)
```

the data from DATARY is stored in the file ZAPO.ARY:SYS rather than the file ZORCH.ARY:MIN as originally specified.

This is because the PAKFIL subroutine has altered the value of the file descriptor constant ZORCH.ARY:MIN to ZAPO.ARY:SYS.

NOTE

PAKFIL cannot change the name of a program executed by the RUN statement. (See the Data Reduction Language manual.)

Subroutine Call: **DFLTYP("typ")**

Declaration: **SUBROUTINE DFLTYP(C):ADSTNG**

Purpose: DFLTYP sets the default file type for the file descriptor in PAKFIL calls. DFLTYP converts up to three characters into Radix-50 code. Subsequent calls to PAKFIL with no file type specified causes the default file type to be stored in the third word of the file descriptor.

Argument: **typ** is an ASCII character string enclosed in paired delimiters. " is an arbitrarily chosen delimiter. Any character may be used as a delimiter as long as it does not appear in the character string.

If only the delimiters are present with no **typ** (for example, DFLTYP(" ")), then a call to PAKFIL with no file type specified causes 0 to be stored in the third word of the file descriptor.

The default file type in a program is initially 0.

Subroutine Call: **DFLUID("uid")**

Declaration: **SUBROUTINE DFLUID(C):ADSTNG**

Purpose: DFLUID sets the default user identification code for the file descriptor in PAKFIL calls. DFLUID converts up to three characters to Radix-50 code. Subsequent calls to PAKFIL with no **uid** specified causes the default code to be stored in the fourth word of the file descriptor.

Argument: **uid** is an ASCII character string enclosed in paired delimiters. " is an arbitrarily chosen delimiter. Any character may be used as a delimiter as long as it does not appear in the character string.

If only the delimiters are present with no **uid** (for example, DFLUID(" ")), then a call to PAKFIL without a **uid** specified stores the currently specified user identification code in the fourth word of the file descriptor.

The default user identification code in a program is initially 0.

Comments: Three conditions exist for a call to PAKFIL with no user identification code specified.

1. You do not use DFLUID. Therefore, PAKFIL stores a zero in the user identification code word.
2. You use DFLUID with the desired default user identification code. PAKFIL uses the default code.
3. You use DFLUID without specifying a **uid**. PAKFIL uses the currently specified user identification code.

Subroutine Call: **UNPSYM(symbol,string,start,stop)**

Declaration: **SUBROUTINE UNPSYM(S,I,V,V):ADSTNG**

Purpose: This is the symbol unpacking subroutine. UNPSYM converts the two Radix-50 words in **symbol** into six ASCII characters. It stores the characters in the integer array **string** starting at **start**. UNPSYM truncates excess characters or fills any unused locations between **start** and **stop** with ASCII spaces.

Arguments: **symbol** contains the two-word Radix-50 code to be converted.

string is an integer array name. **start** specifies the starting character number. **stop** specifies the ending character number.

Comments: If **symbol** contains zero (two cleared words), UNPSYM deposits all spaces in **string** from **start** to **stop**.

Subroutine Call: **RADUP(value,string,start,stop)**

Declaration: **SUBROUTINE RADUP(V,I,V,V):ADSTNG**

Purpose: This subroutine unpacks Radix-50 symbols from a variable. RADUP converts the Radix-50 data in **value** into six ASCII characters. RADUP stores the characters in the integer array **string** starting at **start**. RADUP truncates excess characters or fills any unused **string** positions between **start** and **stop** with ASCII spaces.

Arguments: **value** specifies the data to be converted.

string is an integer array name. **start** selects the starting character number. **stop** selects the ending character number.

Comments: If **value** contains zero (two cleared words), RADUP deposits spaces in **string** from **start** to **stop**. This is useful as an initialization method.

Subroutine Call: **UNPFIL(file,string,start,stop)**

Declaration: **SUBROUTINE UNPFIL(F,I,V,V):ADSTNG**

Purpose: UNPFIL is the file descriptor unpacking subroutine. UNPFIL converts the four-word file-descriptor **file** into 14 ASCII characters. UNPFIL stores the characters in the integer array **string** starting at **start**. It then fills any unused **string** positions in the specified range with ASCII spaces or it truncates any excess characters if less than 14 positions are reserved.

Arguments: **file** is a four-word Radix-50 file descriptor (filnam[.typ][:uid]).

string is an integer array name. **start** specifies the starting character number. **stop** specifies the ending character number.

Comments: UNPFIL formats the string with:

a six-character file name,

followed by a period and a three-character file type,

followed by a colon and a three-character user identification code.

Fewer than the allotted number of characters in a given word of the file descriptor causes the corresponding field in **string** to be right-filled with spaces. A file type or user identification code containing zero causes UNPFIL to replace the associated delimiter (period or colon) with a space and to fill the rest of the field with spaces.

Examples:

1. This routine allows you to decide from the terminal whether or not to continue the program. Use this routine as part of a larger program.

   Write the program in EDIT.

```
1.0100 * THIS ROUTINE ALLOWS YOU TO DECIDE FROM THE TERMINAL
1.0200 * KEYBOARD WHETHER OR NOT TO CONTINUE
1.0300 * RESPONSE MUST BE YES OR NO
1.1000 * LUN 0 IS ASSIGNED AS KB AT RUN TIME
1.2000   SUBROUTINE STRNGI(V,N,I,V,V):STRING
1.3000   FUNCTION CMPCON(I,V,V,C)
1.4000   IARRAY GOT(2)

2.6000   PRINT <0> "DO YOU WISH TO CONTINUE?"
2.6050 * PROGRAM WAITS FOR KEYBOARD RESPONSE
2.6100   STRNGI(0,COUNT,GOT,1,3)
2.6200   RESULT = CMPCON(GOT,1,3,"YES")
2.6300   IF(RESULT EQ 0) 2.64,2.7
2.6400   PRINT <0> "THIS LINE WOULD BE A GOTO WHERE EVER YOUR YES",
         CR
2.6500   PRINT <0> RESPONSE SHOULD TAKE YOU",CR
2.6600   GOTO 2.75
2.7000   RESULT = CMPCON(GOT,1,2,"NO")
2.7100   IF(RESULT EQ 0) 2.74,2.72
2.7200   PRINT <0> "YOUR ANSWER HAS TO BE YES OR NO",CR
2.7300   GOTO 2.6
2.7400   PRINT <0> "BAIL OUT SINCE RESPONSE WAS NO",CR
2.7500 STOP
```

After translating the program and saving it under the name EXPDUM in TRAN, run it from REDUCE.

```
REDUCE
#RUN EXPDUM
DO YOU WISH TO CONTINUE? YES
THIS LINE WOULD BE A GOTO WHERE EVER YOUR YES
RESPONSE SHOULD TAKE YOU

#RUN EXPDUM
DO YOU WISH TO CONTINUE? NO
BAIL OUT SINCE RESPONSE WAS NO

#RUN EXPDUM
DO YOU WISH TO CONTINUE? YUP
YOUR ANSWER HAS TO BE YES OR NO
DO YOU WISH TO CONTINUE? YES
THIS LINE WOULD BE A GOTO WHERE EVER YOUR YES
RESPONSE SHOULD TAKE YOU

#
```

2. This is a sample program that processes commands. The four commands are:

ONE, TWO, THR, and EXI.

The command prompter is ≡. After you give a command in response to the prompter, a message occurs.

```
 2.0500 *          SUBROUTINE AND FUNCTION DECLARATIONS
 2.1000    SUBROUTINE SCONK(I,V,V,C),STRNGI(V,N,I,V,V):STRING
 2.2000    FUNCTION SCMP(I,V,V,I,V,V),CMPCONK(I,V,V,C):STRING

 5.0500 *          INTEGER ARRAYS FOR COMMAND AND INPUT STRINGS
 5.1000    IARRAY LIST(6),COMAND(37)
 5.1500 *          PUT THE COMMAND LIST IN A STRING ARRAY
 5.2000    SCONK(LIST,1,12,"ONETWOTHREXI")
 5.3000    ILUN = 14

10.0030 *          COMMAND INTERPRETER
10.0040 *
10.0050 *          PRINT A PROMPTER
10.0100    PRINT "=^H_"
10.0150 *          ACCEPT THE COMMAND INPUT
10.0200    STRNGI(ILUN,TALLY,COMAND,1,74)
10.0250 *          IGNORE LEADING SPACES
10.0300    LOOP 10.05 START = 1, TALLY
10.0400              IF (CMPCONK(COMAND,START,START," ")) 10.06
10.0500    CONTINUE
10.0550 *          IF JUST CR,LF IN STRING, LOOP BACK
10.0600    IF (TALLY LE START+1) 10.01
10.0650 *          COMPARE THE COMMAND WITH THE LIST
10.0700    LOOP 10.09 JMP = 0, 3
10.0800              IF (SCMP(COMAND,START,START+2,LIST,JMP*3+1,JMP*3+3
           )) 10.09,10.12
10.0900    CONTINUE
10.0950 *          IF COMMAND NOT IN LIST, COMPLAIN
10.1000    PRINT "INVALID COMMAND^G",CR
10.1100    GOTO 10.01
10.1150 *          ACKNOWLEDGE THE COMMAND REQUEST
10.1200    GOTO(JMP+1) 20.1, 30.1, 40.1, 50.1

20.1000    PRINT "ONE FOR THE MONEY",CR
20.2000    GOTO 10.01

30.1000    PRINT "TWO FOR THE SHOW",CR
30.2000    GOTO 10.01

40.1000    PRINT "THREE TO GET READY",CR
40.2000    GOTO 10.01

50.1000    PRINT "GOODBY!",CR
50.2000    STOP
```

After translating the program and saving it under the name COMMAND in TRAN, run it from REDUCE.

```
$REDUCE
#RUN COMMAND
=WHEE
INVALID COMMAND
=ONE
ONE FOR THE MONEY
= TWO
TWO FOR THE SHOW
=   THREE
THREE TO GET READY
=EXIT
GOODBY!

#
```

SECTION FIVE:

EXTENDED FUNCTION SET

This section describes an extension to the standard functions discussed in the Data Reduction Language manual. These extended functions include:

Computing numbers to a specified modulus,

Generating pseudo-random numbers, and

Finding the minimum and maximum of a group of numeric values.

Refer to *Fundamental Algorithms: The Art of Computer Programming*, Vol. 1, by Donald E. Knuth, for further information about the functions described in this section.

Remember, an ordinary array is an array declared with an ARRAY statement in your program. An integer array is an array declared with an IARRAY statement.

Summary of ARITH3 Functions

The extended function set functions are in the ARITH3 file.

Function	Declaration	Purpose
AMAX(a)	AMAX(A)	Returns the maximum value from an ordinary array.
AMIN(a)	AMIN(A)	Returns the minimum value from an ordinary array.
AMOD(x,y)	AMOD(V,V)	Computes real x modulo y.
ENT(x)	ENT(V)	Returns the greatest integer less than or equal to x.
IMAX(ia)	IMAX(I)	Returns the maximum value stored in an integer array.
IMIN(ia)	IMIN(I)	Returns the minimum value stored in an integer array.
MOD(x,y)	MOD(V,V)	Computes integer x modulo y.
POS(x)	POS(V)	Returns unsigned twos complement integer from a signed integer argument.
RAN(s)	RAN(N)	Generates pseudo-random numbers.
RMAX(x,y)	RMAX(V,V)	Returns the maximum of two values.
RMIN(x,y)	RMIN(V,V)	Returns the minimum of two values.
ROUND(x)	ROUND(V)	Rounds the argument.
SIGN(x,y)	SIGN(V,V)	Returns the absolute value of x with the sign of y.

Function Call: **AMAX(array)**

Declaration: **FUNCTION AMAX(A):ARITH3**

Purpose: AMAX returns the maximum value stored in the ordinary **array.**

Argument: **array** is an ordinary array name.

Function Call: **AMIN(array)**

Declaration: **FUNCTION AMIN(A):ARITH3**

Purpose: AMIN returns the minimum value stored in the ordinary **array.**

Argument: **array** is the name of an ordinary array.

Example: AMIN and AMAX can perform column searches on two-dimensional arrays.

```
1.01  ARRAY MAT(4,4),A(4),B(4),C(4),D(4)
1.02  EQUIVALENCE MAT WITH A,B,C,D
  .
  .
  .
2.01  C1 = AMAX(A)
2.02  C2 = AMAX(B)
2.03  C3 = AMIN(C)
2.04  C4 = AMIN(D)
```

C1 and C2 contain the maximum values stored in two of the columns of MAT. C3 and C4 contain the minimum values of the other two columns of MAT.

Function Call: **IMAX(iarray)**

Declaration: **FUNCTION IMAX(I):ARITH3**

Purpose: IMAX returns the maximum value stored in the integer array. The value is returned as a floating-point number.

Argument: **iarray** is an integer array name. IMAX assumes that signed one-word integer array elements in the range from -32768 to 32767 are stored in **iarray**.

Function Call: **IMIN(iarray)**

Declaration: **FUNCTION IMIN(I):ARITH3**

Purpose: IMIN returns the minimum value stored in the integer array. The value is returned as a floating-point number.

Argument: **iarray** is an integer array name. IMIN assumes that signed one-word integer array elements in the range from -32768 to 32767 are stored in **iarray**.

Example:

```
1.00   FUNCTION IMAX(I),IMIN(I):ARITH3
1.01   IARRAY FIND(8)
2.01   FIND(1) = 1.1
2.02   FIND(2) = 2.2
  .
  .
  .
2.08   FIND(8) = 8.8
3.01   FMAX = IMAX(FIND)
3.02   FMIN = IMIN(FIND)
3.03   PRINT "FMAX = ",FMAX,"FMIN = ",FMIN,CR
```

The terminal screen display is:

```
FMAX = 9.000  FMIN = 1.000
```

Function Call: **RMAX(x,y)**

Declaration: **FUNCTION RMAX(V,V):ARITH3**

Purpose: RMAX returns the maximum of two values.

Arguments: **x** and **y** are any legal expressions.

Result: RMAX returns the value of **x** if **x** is greater than **y**. Otherwise, it returns the value of **y**.

Function Call: **RMIN(x,y)**

Declaration: **FUNCTION RMIN(V,V):ARITH3**

Purpose: RMIN returns the minimum of two values.

Arguments: **x** and **y** are any legal expressions.

Result: RMIN returns the value of **x** if **x** is less than **y**. Otherwise, it returns the value of **y**.

Function Call: **SIGN(x,y)**

Declaration: **FUNCTION SIGN(V,V):ARITH3**

Purpose: SIGN returns the absolute of **x** with the sign of **y**. That is, the result is:

$$|x| \frac{|y|}{y} \quad \text{if } y \neq 0$$

$$|x| \quad \text{if } y = 0$$

Arguments: **x** is any expression. The value of **x** determines the absolute of the result.

y is any expression. **y** determines the sign of the result.

Example: S = SIGN(-25.0,Y)

For a positive (or zero) Y, S equals 25.0. For a negative Y, S equals -25.0.

Function Call: **ENT(x)**

Declaration: **FUNCTION ENT(V):ARITH3**

Purpose: ENT returns the greatest integer less than or equal to **x**.

Arguments: **x** is any legal expression. The table below shows the results of ENT.

Value of x	Result
$-1.67(10^7) \leq x \leq 1.67(10^7)$	$\lfloor x \rfloor$ That is, the greatest integer less than or equal to **x** (the floor of **x**).
$-1.7(10^{38}) \leq x < -1.67(10^7)$	**x**
$1.67(10^7) < x \leq 1.7(10^{38})$	**x**

Examples:

1. If $0 \leq x < 1$, the result is 0.
2. If $-1 \leq x < 0$, the result is -1.
3. Note that ENT(-1.5) equals -2 but INT(-1.5) equals -1.

Comments: To obtain the ceiling of **x** ($\lceil x \rceil$) program:

-ENT (-x)

The ceiling of **x** is the least integer greater than or equal to **x**.

Function Call: **ROUND(x)**

Declaration: **FUNCTION ROUND(V):ARITH3**

Purpose: ROUND rounds the value of the argument and returns an integer result.

Argument: **x** is any legal expression whose value is in the range $-1.7(10^{38}) \leq x \leq 1.7(10^{38})$.

Result: The result is:

$$\lfloor |x| + .5 \rfloor \frac{|x|}{x} \qquad \text{if } x \neq 0$$

$$0 \qquad \text{if } x = 0$$

$\lfloor n \rfloor$ specifies the greatest integer less than or equal to n.

The result, in terms of the standard functions, is:

```
INT(ABS(x) + .5) * ABS(x) / x
```

The result, in terms of the extended functions, is:

```
SIGN(ENT(ABS(x) + .5),x)
```

Examples:

x	Result
1.3	1
1.6	2
-1.2	-1
-1.7	-2
0.49	0
0.50	1

Function Call: **AMOD(x,y)**

Declaration: **FUNCTION AMOD(V,V):ARITH3**

Purpose: This is a real modulo function. It returns the real remainder of **x/y** adjusted between 0 and **y**. AMOD performs the modulo operation defined by:

$$x \bmod y = x - y\lfloor x/y \rfloor \qquad \text{if } y \neq 0$$

$$x \bmod 0 = x$$

$\lfloor n \rfloor$ specifies the greatest integer less than or equal to n.

If $y > 0$, then $0 \leq x \bmod y < y$.

If $y < 0$, then $0 \geq x \bmod y > y$.

Arguments: **x** is any legal expression.

y is the modulus. It is any legal expression.

AMOD produces valid results for all argument values in the range from $-1.67(10^7)$ to $1.67(10^7)$. It also produces valid results for argument values greater than $1.67(10^7)$ if **x** and **y** are near the same magnitude. For example: **x** equals $1.7(10^{13})$ and **y** equals $2.9(10^{12})$.

Examples:

1. 10.00 PI = 3.141593
   10.01 R = AMOD(X,2*PI)

   R is a value between 0 and 2*PI.

2. 11.03 F = AMOD(X,1)

   For a positive X, this statement returns the fractional part of X. For a negative X, it returns the fractional part added to 1. If X equals 14.732, then F equals 0.732. If X equals -14.732, then F equals 0.268.

3. The following example corrects round-off errors for quantities near integer values.

   13.01 IF (.0001<AMOD(X,1)<.9999) 13.03
   13.02 X = ROUND(X)
   13.03 N = ENT(X)

Function Call: **MOD(x,y)**

Declaration: **FUNCTION MOD(V,V)ARITH3**

Purpose: This is an integer modulo function. It returns the signed integer remainder of **x/y** adjusted between 0 and **y**. MOD performs the operation defined by:

$$x \bmod y = x - y\lfloor x/y \rfloor \qquad \text{if } y \neq 0$$

$$x \bmod 0 = x$$

where $\lfloor n \rfloor$ equals the greatest integer less than or equal to n.

If **y** is greater than 0, then $0 \leqslant x \bmod y < y$.

If **y** is less than 0, then $0 \geqslant x \bmod y > y$.

Arguments: **x** must be in the range $-32767 \leqslant x \leqslant 32767$.

y is the modulus. **y** must be in the range $-32767 \leqslant y \leqslant 32767$.

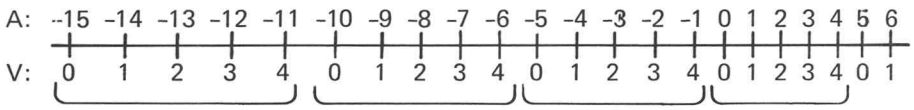
Examples:

1. The table below gives the results of the MOD function for selected **x** and **y** arguments.

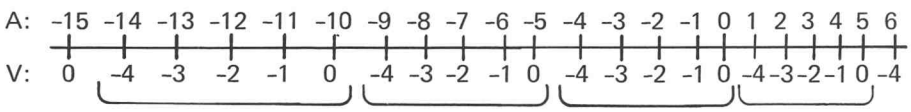
x	y	Result
13	4	1
15	4	3
-13	4	3
-15	4	1
13	-4	-3
15	-4	-1
14.7	4	3
15	3.7	3
-14.7	0	-15

2. The illustration below shows how MOD works.

If V=MOD(A,5)



If V=MOD(A,-5)



3. In this example, line 5.03 determines the ceiling of a value. Line 5.04 uses the MOD function to produce a number between 1 and 5.

```
2.1000    FUNCTION ENT(V),MOD(V,V):ARITH3

3.3000    ARRAY A(150)
3.3500 *          FILL THE ARRAY WITH A NUMERICAL SEQUENCE
3.4000    LOOP 3.5 I=1,150
3.5000              A(I) = I

5.0050 *          LIST DATA ORDERED BY COLUMNS
5.0100    ACCEPT(14> "COUNT: ",N
5.0150    IF (1 < N < 150) 5.02, 5.01
5.0155 *          SET UP FOR FIVE COLUMNS OF LISTING
5.0200    COL = 5
5.0250 *          COMPUTE GREATEST NUMBER OF ROWS
5.0300    R = -ENT(-N/COL)
5.0350 *          COMPUTE NUMBER OF COLUMNS IN LAST ROW
5.0400    C = MOD(N,-COL) + COL
5.0500 *          LIST THE DATA
5.0700    LOOP 5.12 K = 1, R
5.0800              LOOP 5.1 J = 0, (COL-1)-(K EQ R)*(COL-C)
5.0900                    PRINT(13> A(K+J*R-(J GT C)*(J-C)):I3
5.1000              CONTINUE
5.1100              PRINT(13> CR
5.1200    CONTINUE

7.0100    GOTO 5.01
```

After writing the program in EDIT, translate it and save it under the name LISTER with TRAN, then run it under control of the REDUCE program.

```
$REDUCE
#RUN LISTER
COUNT: 21
    1     6    10    14    18
    2     7    11    15    19
    3     8    12    16    20
    4     9    13    17    21
    5
COUNT: 22
    1     6    11    15    19
    2     7    12    16    20
    3     8    13    17    21
    4     9    14    18    22
    5    10
COUNT: 23
    1     6    11    16    20
    2     7    12    17    21
    3     8    13    18    22
    4     9    14    19    23
    5    10    15
```

Function Call: **POS(x)**

Declaration: **FUNCTION POS(V):ARITH3**

Purpose: POS adjusts the number range $-32768 \leqslant x \leqslant 32767$ to the range $0 \leqslant n \leqslant 65535$.

Arguments: **x** is any legal expression whose value is an integer in the range from -32768 to 32767.

Result: The result is an integer in the range from 0 to 65535. POS returns (**x** + 65536) if **x** is less than 0, and returns **x** if **x** is greater than or equal to 0.

Comments: Integer arrays normally store signed numbers between -32768 and 32767. POS allows integer array elements to be used as twos complement integers in bit comparisons and logical operations. Use POS whenever dealing with octal numbers stored in integer arrays.

Example: P = POS(-1)

POS returns the value 65535 (177777_8) in P.

Function Call: **RAN(s)**

Declaration: **FUNCTION RAN(N):ARITH3**

Purpose: This function is a pseudo-random number-generator. RAN generates numbers uniformly distributed over the range $0 \leqslant n < 1$, with a period of 65536. That is, after 65536 executions, RAN repeats its results.

Argument: **s** is a variable whose value provides two integers: s1 and s2. The sign of **s** is always positive when RAN is generating a number.

s1 equals the high order word of **s**, modulo 2^{15}. s2 equals the low order word of **s**, modulo 2^{16}.

Result: The pseudo-random number is produced with the algorithm:

$n = [(a1*s1 + c1) \bmod 2^{15} + (a2*s2 + c2) \bmod 2^{16}] \bmod 2^{16}$

$a1 = 5^{13} \bmod 2^{16} = 29589 = (8 * 3698) + 5$

$a2 = 5^{15} \bmod 2^{16} = 18829 = (8 * 2353) + 5$

$c1 = (1/2 - 1/6\ \sqrt{3}\ )\ 2^{15} = 6925$

$c2 = (1/2 - 1/6\ \sqrt{3}\ )\ 2^{16} = 13849$

The value $(a1*s1 + c1) \bmod 2^{15}$ is stored in s1. The value $(a2*s2 + c2) \bmod 2^{16}$ is stored in s2. These new values of s1 and s2 specify the value of **s** for the next call to RAN.

Comments: To initialize the random number sequence:

1. Store a positive number (or zero) in **s**, or
2. Store a negative number in **s**, which causes the generator to initialize itself with the current date - time.

If your program is to run several times and you desire a different random sequence each time, then keep **s** in COMMON.

To obtain random integers between 0 and k, take the integer part of the product of (k+1) and the random number:

```
X = INT((k + 1) * RAN(s))
```

The lesser significant digits are less random.

APPENDIX A:

SUMMARY OF HOW TO DECLARE SUBPROGRAMS

Summary of Function and Subroutine Declarations and Calls*

A subprogram must be declared in a program before it is called. The general form of the subprogram declaration is:

$$\left\{\begin{matrix}\text{FUNCTION}\\ \text{SUBROUTINE}\end{matrix}\right\}\ \text{name}\ \left\{\begin{matrix}(0)\\ (\text{list})\end{matrix}\right\}\ [:\text{filnam}],\ldots,\text{name}\ \left\{\begin{matrix}(0)\\ (\text{list}\end{matrix}\right\}\ [:\text{filnam}]$$

where

name is the name of a subprogram.

0 specifies that the subprogram does not have any arguments. **list** specifies the number, type, and sequence of arguments. The list letter codes are given below.

Letter Code	Description of Legal Arguments
A	Only the name of an ordinary array is a legal argument.
C	A string constant consisting of an ASCII string enclosed in paired delimiters must be specified.
D	A file descriptor, filnam.typ[:uid], must be specified. If uid is not specified, the current identification code is used. The file must already exist. The Translator searches the directory for the file and includes the contents of it in the Test Program File.
F	A disk file descriptor, filnam[.typ][:uid], must be specified.
I	Only the name of an integer array is a legal argument.
L	Any statement line number from the program is a legal argument.
N	Only simple variables or ordinary array elements are legal arguments.
P	Only the name of a pinlist is a legal argument.
S	Any symbol which starts with an alphabetic character and contains six or less alphanumeric characters is a legal argument.
T	A pinlist name, a singly or doubly indexed pinlist, or a pin name is a legal argument.
T1	A pin name or a singly indexed pinlist is a legal argument.
V	Any expression that gives a numeric result is a legal argument.

- **filnam** is a file name with the file type FCN that contains the load module of the subprogram. First the system searches the currently specified user identification code for **filnam**. If any specified file is not found, then the system searches SYS for **filnam**.

*For more information, see the Data Reduction Language manual.

APPENDIX B:

NUMERIC CHARACTER VALUES FOR ASCII CHARACTERS

Decimal and Octal Values

Decimal Value	Octal Value	Character Name and Remarks	
0	000	NUL	Null, tape feed, CONTROL/SHIFT/P.
1	001	SOH	Start of heading; also SOM, start of message, CONTROL/A.
2	002	STX	Start of text; also EOA, end of address, CONTROL/B.
3	003	ETX	End of text; also EOM, end of message, CONTROL/C.
4	004	EOT	End of transmission (END); shuts off TWX machines, CONTROL/D.
5	005	ENQ	Enquiry (ENQRY); also WRU, CONTROL/E.
6	006	ACK	Acknowledge; also RU, CONTROL/F.
7	007	BEL	Rings the bell. CONTROL/G.
8	010	BS	Backspace, also FEO, format effector. Backspaces some machines, CONTROL/H.
9	011	HT	Horizontal tab, CONTROL/I.
10	012	LF	Line feed or line space (new line); advances paper to next line, duplicated by CONTROL/J.
11	013	VT	Vertical tab (VTAB), CONTROL/K.
12	014	FF	Form Feed to top of next page (PAGE), CONTROL/L.
13	015	CR	Carriage return to beginning of line, duplicated by CONTROL/M.
14	016	SO	Shift out; changes ribbon color to red. CONTROL/N.
15	017	SI	Shift in; changes ribbon color to black. CONTROL/O.
16	020	DLE	Data link escape, CONTROL/P (DC0).
17	021	DC1	Device control 1, turns transmitter (READER) on, CONTROL/Q (X ON).
18	022	DC2	Device control 2, turns punch or auxiliary on, CONTROL/R (TAPE, AUX ON).
19	023	DC3	Device control 3, turns transmitter (READER) off, CONTROL/S (X OFF).

Decimal Value	Octal Value	Character Name and Remarks	
20	024	DC4	Device control 4, turns punch or auxiliary off, CONTROL/T (AUX OFF)
21	025	NAK	Negative acknowledge; also ERR, ERROR, CONTROL/U.
22	026	SYN	Synchronous file (SYNC), CONTROL/V.
23	027	ETB	End of transmission block; also LEM, logical end of medium, CONTROL/W.
24	030	CAN	Cancel (CANCL), CONTROL/X.
25	031	EM	End of medium, CONTROL/Y.
26	032	SUB	Substitute, CONTROL/Z.
27	033	ESC	Escape, CONTROL/SHIFT/K.
28	034	FS	File separator, CONTROL/SHIFT/L.
29	035	GS	Group separator, CONTROL/SHIFT/M.
30	036	RS	Record separator, CONTROL/SHIFT/N.
31	037	US	Unit separator, CONTROL/SHIFT/O.
32	040	SP	Space. Blank.
33	041	!	
34	042	"	
35	043	#	
36	044	$	
37	045	%	
38	046	&	
39	047	'	Apostrophe.
40	050	(	
41	051	)	
42	052	*	
43	053	+	
44	054	,	Comma.
45	055	–	Dash.
46	056	.	
47	057	/	
48	060	0	
49	061	1	
50	062	2	
51	063	3	
52	064	4	
53	065	5	

Decimal Value	Octal Value	Character Name and Remarks
54	066	6
55	067	7
56	070	8
57	071	9
58	072	:
59	073	;
60	074	<
61	075	=
62	076	>
63	077	?
64	100	@
65	101	A
66	102	B
67	103	C
68	104	D
69	105	E
70	106	F
71	107	G
72	110	H
73	111	I
74	112	J
75	113	K
76	114	L
77	115	M
78	116	N
79	117	O
80	120	P
81	121	Q
82	122	R
83	123	S
84	124	T
85	125	U
86	126	V
87	127	W
88	130	X
89	131	Y

Decimal Value	Octal Value		Character Name and Remarks
90	132	Z	
91	133	[	SHIFT/K.
92	134	\	SHIFT/L.
93	135	]	SHIFT/M.
94	136	Λ	Λ appears as ↑ on some terminals.
95	137	_	Underscore. This appears as ← on some terminals.
96	140	'	
97	141	a	
98	142	b	
99	143	c	
100	144	d	
101	145	e	
102	146	f	
103	147	g	
104	150	h	
105	151	i	
106	152	j	
107	153	k	
108	154	l	
109	155	m	
110	156	n	
111	157	o	
112	160	p	
113	161	q	
114	162	r	
115	163	s	
116	164	t	
117	165	u	
118	166	v	
119	167	w	
120	170	x	
121	171	y	
122	172	z	
123	173	{	
124	174	|	
125	175	}	This code generated by ALTMODE.
126	176	~	This code generated by PREFIX key (if present).
127	177	DEL	Delete, Rubout.

Not available on the 4010. Only available on the 4014.

Radix-50 Values

Character	ASCII Octal Equivalent	Radix-50 Octal Equivalent
space	40	0
A-Z	101-132	1-32
$	44	33
.	56	34
		(35 is not used)
0-9	60-71	36-47

The system computes a Radix-50 value for the three characters stored in a word. The Radix-50 octal value is:

$$a * (50_8)^2 + b * 50_8 + c$$

where a is the Radix-50 octal value of the first character,
b is the Radix-50 octal value of the second character, and
c is the Radix-50 octal value of the third character.

For example, assume the three characters are X2B.

Character	Radix-50 Octal Value
X	30
2	40
B	2

Using the above formula, the Radix-50 value of X2B is:

$$30 * 50^2 + 40 * 50 + 2 = 115402$$

The table below provides a convenient means of translating the ASCII character set into its Radix-50 equivalents. Using the table, the Radix-50 value of X2B is:

```
    X = 113000      (First character)
    2 = 002400      (Second character)
+   B = 000002      (Third character)
-------------
  X2B = 115402
```

Single Character or First Character		Second Character		Third Character	
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
unused	132500	unused	002210	unused	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

Presetting Integer Arrays with String Constants

The TEKTEST statement PRESET cannot be used to preset integer arrays with string constants. However, you can preset the arrays by storing the octal values of the individual characters in the string constants.

Recall that each word of an integer array stores two characters — one character in the even byte (bits 0–7) and one character in the odd byte (bits 8–15). The table below shows the octal values you should store in the odd and even bytes in order to preset arrays. When setting both bytes of a word, use octal addition to add the two values. An example follows the table.

Presetting the integer arrays can make the execution of your program more efficient and, because you can omit from your program the statements necessary to initialize the arrays during program execution, your program uses less core.

Character	Odd Byte	Even Byte	Character	Odd Byte	Even Byte
SP (Space, Blank)	040	20000	@	100	40000
!	041	20400	A	101	40400
"	042	21000	B	102	41000
#	043	21400	C	103	41400
$	044	22000	D	104	42000
%	045	22400	E	105	42400
&	046	23000	F	106	43000
' (Apostrophe)	047	23400	G	107	43400
(	050	24000	H	110	44000
)	051	24400	I	111	44400
*	052	25000	J	112	45000
+	053	25400	K	113	45400
, (Comma)	054	26000	L	114	46000
– (Dash)	055	26400	M	115	46400
.	056	27000	N	116	47000
/	057	27400	O	117	47400
0	060	30000	P	120	50000
1	061	30400	Q	121	50400
2	062	31000	R	122	51000
3	063	31400	S	123	51400
4	064	32000	T	124	52000
5	065	32400	U	125	52400
6	066	33000	V	126	53000
7	067	33400	W	127	53400
8	070	34000	X	130	54000
9	071	34400	Y	131	54400
:	072	35000	Z	132	55000
;	073	35400	[(SHIFT/K)	133	55400
<	074	36000	\ (SHIFT/L)	134	56000
=	075	36400	] (SHIFT/M)	135	56400
>	076	37000	Λ (or ↑)	136	57000
?	077	37400	_ (Underscore or ←)	137	57400

Example:

```
                                        .
                                        .
                                        .
4.4000      IARRAY LIST(6), WKDAY(11)
4.4500  *
4.5000  *
4.5500  *           SUN MON TUE WED THU FRI SAT
4.6000      PRESET WKDAY=#52523,#46516,#47117,#52524,#53505,#42105,#44124,#43125
            ,#44522,#40523,#124
4.6500  *       COMMANDS A D E G I L O P R S U
4.7000      PRESET LIST = #42101, #43505, #46111, #50117, #51522, #00125
                                        .
                                        .
                                        .
```

In the above example, the user first declared the integer arrays LIST and WKDAY. He then used PRESET and octal character values to preset the arrays. For example, #52523 sets the first word of WKDAY to SU (#123 + #52400), #46516 sets the second word to NM (#116 + #46400), and so on. LIST was preset in the same manner.

APPENDIX C:

SUMMARY OF GENERAL-PURPOSE PROCESSING DATA SUBPROGRAMS

General Form	Purpose	Function	Subroutine	Declaration	File Name
ALFPOS(tstat,x,y)	Returns the coordinates of the bottom left corner of the alpha cursor and the terminal status.		X	ALFPOS(N,N,N)	GRAPH1 and GRAPHV
ALFPSV(tstat,xv,yv)	Returns the user data-space coordinates of the bottom left corner of the alpha cursor and the terminal status.		X	ALFPSV(N,N,N)	GRAPHV
AMAX(a)	Returns the maximum value from a real array.	X		AMAX(A)	ARITH3
AMIN(a)	Returns the minimum value from a real array.	X		AMIN(A)	ARITH3
AMOD(x,y)	Computes real x modulo y.	X		AMOD(V,V)	ARITH3
BARRAY(iarray,xmax,ymax,zmax)	Dimensions an existing integer array into a binary array.		X	BARRAY(I,V,V,V)	BARRAY
CHARI(ilun)	Returns the floating-point value of the input ASCII character.	X		CHARI(V)	STRING and ADSTNG
CHARO(olun,char)	Sends the specified character to the output device.		X	CHARO(V,V)	STRING and ADSTNG
CLRKB	Clears the keyboard input queue.		X	CLRKB(0)	STRING and ADSTNG
CMPCON(string,start,stop, "stringconstant")	Compares a string with a string constant.	X		CMPCON(I,V,V,C)	STRING and ADSTNG
CRDATE(olun)	Prints the current date.		X	CRDATE(V)	TIME

General Form	Purpose	Function	Subroutine	Declaration	File Name
DRAW(x,y,mode)	Draws a vector from the current beam position to screen coordinates (x,y).		X	DRAW(V,V,V)	GRAPH1 and GRAPHV
DRAWV(xv,yv,mode)	Draws a vector from the last point to (xv,yv) on the user data-space.		X	DRAWV(V,V,V)	GRAPHV
ENT(x)	Returns the greatest integer less than or equal to x.	X		ENT(V)	ARITH3
FILDAY(e,ilun)	Returns the data, logged in the specified log file, in days since 1 January 1900.	X		FILDAY(N,V)	TIME
FILSEC(e,ilun)	Returns the time, logged in the specified log file, in seconds since midnight.	X		FILSEC(N,V)	TIME
FLDATE(e,ilun,olun)	Prints the date logged in the specified file.		X	FLDATE(N,V,V)	TIME
FLTIME(e,ilun,olun)	Prints the time logged in the specified file.		X	FLTIME(N,V,V)	TIME
FMTNUM(delim,string,start,stop)	Converts an ASCII string into a floating-point number.	X		FMTNUM(N,I,N,V)	ADSTNG
GETBIT(barray,x,y,z)	Reads an individual bit of a bit array.	X		GETBIT(I,V,V,V)	BARRAY
GETCEL(barray,x,y,z)	Reads a group of bits in the X direction of a bit array.	X		GETCEL(I,V,V,V)	BARRAY
GETCOL(barray,x,y,z)	Reads a group of bits in the Z direction of a bit array.	X		GETCOL(I,V,V,V)	BARRAY

General Form	Purpose	Function	Subroutine	Declaration	File Name
CRSHR(char,x,y)	Turns on the terminal screen cross hairs, waits for you to press a key, then returns the cross-hair coordinates and the value of the character pressed.		X	CRSHR(N,N,N)	GRAPH1 and GRAPHV
CRSHRV(char,xv,yv)	Turns on the terminal screen cross hairs, waits for you to press a key, then returns the cross-hair coordinates and the value of the character pressed.		X	CRSHRV(N,N,N)	GRAPHV
CRTIME(olun)	Prints the current time.		X	CRTIME(V)	TIME
CURDAY	Returns the current date in days since 1 January 1900.	X		CURDAY(0)	TIME
CURSEC	Returns the current time in seconds since midnight.	X		CURSEC(0)	TIME
CURSOR(x,y)	Moves the bottom left corner of the alpha cursor to screen coordinates (x,y).	X		CURSOR(V,V)	GRAPH1 and GRAPHV
DCOORD(xvmin,xvmax,yvmin, yvmax)	Defines the coordinates of the user data-space window.		X	DCOORD(V,V,V,V)	GRAPHV
DFLTYP("typ")	Sets the default file type for the file descriptor in PAKFIL calls.		X	DFLTYP(C)	ADSTNG
DFLUID("uid")	Sets the default user identification code for the file descriptor in PAKFIL calls.		X	DFLUID(C)	ADSTNG

General Form	Purpose	Function	Subroutine	Declaration	File Name
GETROW(barray,x,y,z)	Reads a group of bits in the Y direction of a bit array.	X		GETROW(I,V,V,V)	BARRAY
IMAX(ia)	Returns the maximum value stored in an integer array.	X		IMAX(I)	ARITH3
IMIN(ia)	Returns the minimum value stored in an integer array.	X		IMIN(I)	ARITH3
JUSTFY(side,tally,string,start,stop)	Removes imbedded spaces and justifies the string.		X	JUSTFY(V,N,I,V,V)	STRING and ADSTNG
KBSTAT	Returns the input queue status.	X		KBSTAT(0)	STRING and ADSTNG
LOCATE(x,y)	Returns the cross-hair coordinates in (x,y) without user intervention.		X	LOCATE(N,N)	GRAPH1 and GRAPHV
LOCATV(xv,yv)	Returns the user data-space cross-hair coordinates in (xv,yv) without user intervention.		X	LOCATV(N,N)	GRAPHV
MAPOUT(xs,ys,map,xv,yv)	Returns screen coordinates scaled and translated from user data-space coordinates.		X	MAPOUT(N,N,N,V,V)	GRAPHV
MOD(x,y)	Computes integer x modulo y.	X		MOD(V,V)	ARITH3
MOVDAT(string,start,stop)	Stores the current date and time in string.		X	MOVDAT(I,V,V)	TIME
MOVFDA(e,ilun,string,start,stop)	Stores the date and time, logged in the specified file, in string.		X	MOVFDA(N,V,I,V,V)	TIME

General Form	Purpose	Function	Subroutine	Declaration	File Name
NUMOUT(value,code,tally, string,start,stop)	In accordance with the selected format, converts a floating-point value into ASCII characters.		X	NUMOUT(V,V,N,I,V,V)	ADSTNG
PAKFIL(file,delim,string, start,stop)	Packs a string into a Radix-50 four-word file descriptor.		X	PAKFIL(F,N,I,N,V)	ADSTNG
PAKSYM(symbol,delim,string, start,stop)	Packs a string into a two-word Radix-50 symbol.		X	PAKSYM(S,N,I,N,V)	ADSTNG
POINT(x,y,type)	Plots a period (.), minus (-), or plus (+) at screen coordinates (x,y).		X	POINT(V,V,V)	GRAPH1 and GRAPHV
POINTV(xv,yv,type)	Plots a period (.), minus (-), or plus (+) on the user data-space.		X	POINTV(V,V,V)	GRAPHV
POS(x)	Returns an unsigned twos complement integer from a signed integer argument.	X		POS(V)	ARITH3
RADPAK(value,delim,string, start,stop)	Packs a string into a two-word Radix-50 variable.		X	RADPAK(N,N,I,N,V)	ADSTNG
RADUP(value,string,start,stop)	Unpacks two Radix-50 words into a six-character ASCII string.		X	RADUP(V,I,V,V)	ADSTNG
RAN(s)	Generates pseudo-random numbers.	X		RAN(N)	ARITH3
RMAX(x,y)	Returns the maximum of the two values.	X		RMAX(V,V)	ARITH3
RMIN(x,y)	Returns the minimum of the two values.	X		RMIN(V,V)	ARITH3

General Form	Purpose	Function	Subroutine	Declaration	File Name
ROUND(x)	Rounds the argument.	X		ROUND(V)	ARITH3
SCMP(string1,start1,stop1,string2, start2,stop2)	Compares two strings.	X		SCMP(I,V,V,I,V,V)	STRING and ADSTNG
SCON(dststr,start,stop, "stringconstant")	Stores a string constant into the destination string.		X	SCON(I,V,V,C)	STRING and ADSTNG
SCOORD(xsmin,xsmax,ysmin, ysmax)	Defines the boundaries of the screen window.		X	SCOORD(V,V,V,V)	GRAPHV
SETBIT(state,barray,x,y,z)	Modifies an individual bit of the bit array.		X	SETBIT(V,I,V,V,V)	BARRAY
SETCEL(value,barray,x,y,z)	Modifies a group of bits in barray in the X direction.		X	SETCEL(V,I,V,V,V)	BARRAY
SETCOL(value,barray,x,y,z)	Modifies a group of bits in barray in the Z direction.		X	SETCOL(V,I,V,V,V)	BARRAY
SETROW(value,barray,x,y,z)	Modifies a group of bits in barray in the Y direction.		X	SETROW(V,I,V,V,V)	BARRAY
SIGN(x,y)	Returns the absolute value of x with the sign of y.	X		SIGN(V,V)	ARITH3
SMOV(dststr,start1,stop1, srcstr,start2,stop2)	Moves the source string into the destination string.		X	SMOV(I,V,V,I,V,V)	STRING and ADSTNG
SPFMT(delim,string,start, stop,dtabl)	Converts an ASCII string into a floating-point number using the supplied delimiter table.	X		SPFMT(N,I,N,V,I)	ADSTNG

General Form	Purpose	Function	Subroutine	Declaration	File Name
STRNGF(iarray,position)	Returns the floating-point value of a character.	X		STRNGF(I,V)	STRING and ADSTNG
STRNGI(ilun,count,iarray, start,stop)	Inputs a string from the input device into iarray.		X	STRNGI(V,N,I,V,V)	STRING and ADSTNG
STRNGO(olun,iarray,start,stop)	Sends the specified characters to the output device.		X	STRNGO(V,I,V,V)	STRING and ADSTNG
STRNGS(char,iarray,position)	Stores the ASCII value of the character char.		X	STRNGS(V,I,V)	STRING and ADSTNG
UNPFIL(file,string,start,stop)	Unpacks a four-word Radix-50 file descriptor into a 14-character ASCII string.		X	UNPFIL(F,I,V,V)	ADSTNG
UNPSYM(symbol,string,start,stop)	Unpacks a two-word Radix-50 symbol into a six-character ASCII string.		X	UNPSYM(S,I,V,V)	ADSTNG
VECTRF(x1,y1,x2,y2)	Draws a vector on the terminal screen between (x1,y1) and (x2,y2).		X	VECTRF(V,V,V,V)	GRAPH1 and GRAPHV
VECTRV(xv1,yv1,xv2,yv2)	Draws a vector from (xv1,yv1) to (xv2,yv2) on the user data-space.		X	VECTRV(V,V,V,V)	GRAPHV

INDEX