AT&T

305-506
Issue 1

AT&T 3B2 Computer
UNIX™ System V Release 2.0
Utilities – Volume 2

NOTICE

Replace this

page with the

GRAPHICS

tab separator.

AT&T

AT&T 3B2 Computer
UNIX™ System V Release 2.0
Graphics Utilities Guide

CONTENTS

Chapter 1

INTRODUCTION

Chapter 1

INTRODUCTION

GENERAL

This guide describes command formats (syntax) and use of the Graphics Utilities available with your AT&T 3B2 Computer. The numerical and graphical commands described in this guide are used to build and edit numerical data plots and hierarchical charts. This guide is designed for individuals experienced in using the **UNIX*** Operating System. Although these individuals are not expected to know UNIX System Shell Programming Language to use this guide, it would be helpful in understanding the examples at the end of Chapter 3.

* Trademark of AT&T

GUIDE ORGANIZATION

This guide is structured so you can easily find information without having to read the entire text. The remainder of this guide is organized as follows:

- Chapter 2, ''OVERVIEW,'' gives a general description of the basic concepts of the Graphics Utilities and how to get started using the Graphics Utilities.

- Chapter 3, ''STAT- A TOOL FOR ANALYZING DATA,'' describes routines that can be interconnected using the UNIX System shell to form numerical processing networks.

- Chapter 4, ''COMMAND DESCRIPTIONS,'' describes the formats (syntax) for each command in the Graphics Utilities. The descriptions include the purpose of the command, a discussion of the command syntax and options, and examples of using each command.

- Chapter 5, ''GRAPHICS EDITOR(**ged**),'' describes an interactive editor used to display, edit, and build drawings on a TEKTRONIX 4014* display terminal.

* Registered Trademark of Tektronix, Inc.

Chapter 2

OVERVIEW

PAGE

Chapter 2

OVERVIEW

INTRODUCTION

The **Graphics Utilities** is a collection of numerical and graphical commands used to build and edit numerical data plots and hierarchical charts. This chapter will help a user get started when using the **Graphics Utilities**. The best way to learn about graphics is to log onto the 3B2 Computer and use it. The examples below assume that the user is familiar with the UNIX System.

INTERFACING THE 5620 DMD TO THE 3B2 COMPUTER

To display drawings from the **Graphics Utilities**, you must use a graphics display terminal. The recommended graphics display terminal for the 3B2 Computer is the **TELETYPE*** 5620 Dot-Mapped Display (DMD) terminal. This terminal can emulate a TEKTRONIX 4014, a **HP**† 2621, and an **APS**‡ 5. Unless otherwise noted, capabilities in this guide pertains to a 5620 DMD connected to a 3B2 Computer.

To interface the 5620 DMD to the 3B2 Computer, the following steps should be completed.

1. Interconnect the 5620 DMD to the 3B2 Computer.

2. Install the 5620 DMD Core Utilities. Instructions on how to install this utilities can be found in the *5620 Dot-Mapped Display Administrator Guide.*

3. Log on the system and create a layer. Instructions on how to do this can be found in the *5620 Dot-Mapped Display User Guide.*

4. Load the TEKTRONIX 4014 program in the layer you just created. The strap options *GINcount -g -u* must be entered so that the graphics editor (**ged**) will operate on a four-stage position instead of a two-stage position. Instructions on how to do this can be found in the *5620 Dot-Mapped Display User Guide.*

* Trademark of AT&T

† Trademark of Hewlett-Packard, Inc

‡ Trademark of Autologic, Inc

HOW COMMANDS ARE DESCRIBED

A common format is used to describe each of the commands. This format is as follows:

- **General:** The purpose of the command is defined. Any uncommon or special information about the command is also provided.
- **Command Format:** The basic command line format (syntax) is defined and the various arguments and options discussed.
- **Sample Command Use:** Example command line entries and system responses are provided to show you how to use the command.

In the command format discussions, the following symbology and conventions are used to define the command syntax.

- The basic command is shown in bold type. For example: **command** is in bold type.
- Arguments that you must supply to the command are shown in a special type. For example: **command** *argument*
- Command options and arguments that do not have to be supplied are enclosed in brackets ([ ]). For example: **command** [*optional arguments*]
- The pipe symbol (¦) is used to separate arguments when one of several forms of an argument can be used for a given argument field. The pipe symbol can be thought of as an exclusive OR function in this context. For example: **command** [*argument1* ¦ *argument2*]

In the sample command discussions, user inputs and 3B2 Computer response examples are shown as follows:

```
This style of type is used to show system generated
responses displayed on your screen.
```

This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: <*CR*> carriage return, <*CTRL d*> control d, <*ESC g*> escape g, passwords, and tabs.

This style of italic type is used for notes that provide you with additional information.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

ACCESSING THE GRAPHICS UTILITIES

To access the **graphics** commands when logged in on the 3B2 Computer, type **graphics**. The shell variable *PATH* will be altered to include the **graphics** commands, and the shell primary prompt will be changed to^.

```
$graphics<CR>
^
```

Any command accessible before typing **graphics** will still be accessible; **graphics** only adds commands, it does not take any away. Once in **graphics**, a user can find out about any of the **graphics** commands using **whatis**. Typing **whatis** by itself on a command line will generate a list of all the commands in **graphics** along with instructions on how to find out more about any of them.

All the **graphics** commands accept the same command line format:

- A *command* is a *command-name* followed by *argument*(s).
- A *command-name* is the name of any of the **graphics** commands.
- An *argument* is a *file-name* or an *option-string*.
- A *file-name* is any file name not beginning with –, or a – by itself to reference the standard input.
- An *option-string* is a – followed by *option*(s).
- An *option* is a letter(s) followed by an optional value. Options may be separated by commas.

The **graphics** commands can be removed from the user's *PATH* by typing an end-of-file indication (<*CTRL-d*> control-d on most terminals) or by typing exit. This will put you in the UNIX System shell.

```
^exit<CR>
$
```

BASIC CONCEPTS

Note: Many of the basic concepts of the Graphics Utilities will be explained in this chapter by using some of the **graphics** commands. It is not necessary now to fully understand these commands. However, if you need a more detailed explanation of a command, refer to Chapter 4, " Command Descriptions."

The basic approach taken with **graphics** is to generate a drawing by describing it rather than by drafting it. Any drawing is seen as having two fundamental attributes: **its underlying logic** and **its visual layout**. The layout contains one representation of the logic. For example, consider the $y=x^2$ for the value of x being between 0 and 10:

- The logic of the plot is the description as just given, namely $y=x^2$, for the value of x being between 0 and 10.

- The layout consists of an x-y grid, axis labeled perhaps 0 to 10 and 0 to 100, and lines drawn connecting the x-y pairs 0,0 to 1,1 to 2,4 etc.

The way to generate a drawing in **graphics** is:

1. Gather Data
2. Transform the Data
3. Generate a Layout
4. Display the Layout.

The following is an example of generating a drawing of $y=x^2$, for the value of x being between 0 and 10 and displaying it on a TEKTRONIX 4014 display terminal.

- The **gas** command is used to *gather the data.* The **gas** command generates a sequence of numbers, in this case **s**tart at 0 and **t**erminating at 10.
- The **af** command is used to *transform the data.* The **af** command performs general arithmetic transformations.
- The **plot** command is used to *generate a layout.* The **plot** command builds x-y plots.
- The **td** command is used to *display the layout.* The **td** command displays drawings on TEKTRONIX 4014 display terminal.

The command line format to generate the drawing for $y=x^2$ would be:

```
^gas -s0,t10 | af " x^2" | plot | td<CR>
```

The results of the drawing is shown in Figure 2-1.

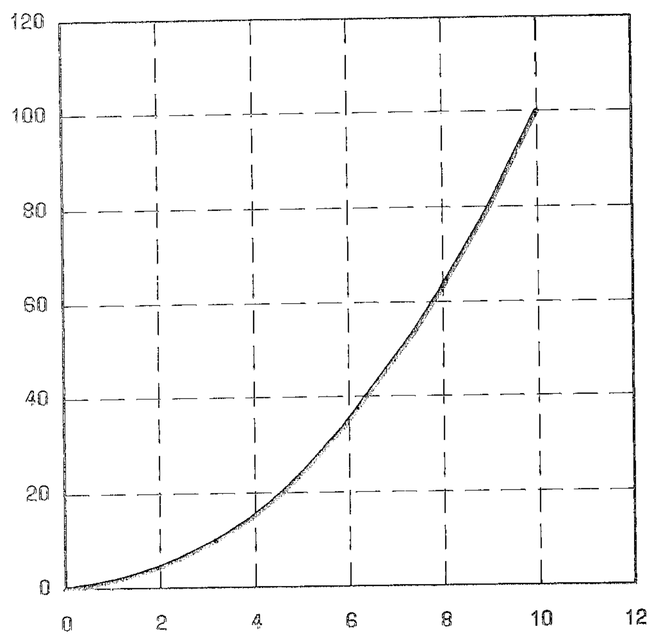


Figure 2-1. Plot of **gas -s0,t10 | af "x^2" | plot | td**

To clear the screen after displaying a drawing use the **erase** command.

```
^erase<CR>
```

The layout generated by a **graphics** program may not always be precisely what is wanted. There are two ways to influence the layout. The first way to influence the layout is to use the **plot** command options. For instance, in the previous example, it may be desired to have the x-axis labels show each of the numbers plotted and not have any y-axis labels at all. To achieve this the **plot** command would be changed to:

```
^gas -s0,t10 | af " x *^2"  | plot -xi1,ya | td<CR>
```

The results of the drawing is shown in Figure 2-2.

The second way to influence a layout is by editing it directly at a display terminal using the graphical editor, **ged**. To edit a drawing really means to edit the computer representation of the drawing. For **graphics**, the representation is called a graphical primitive string or GPS. All the drawing commands (for example **plot**) write GPS, and all the device filters (for example **td**) read GPS. **Ged** allows manipulation of GPS at a display terminal by interacting with the drawing that the GPS describes (see Chapter 5.)

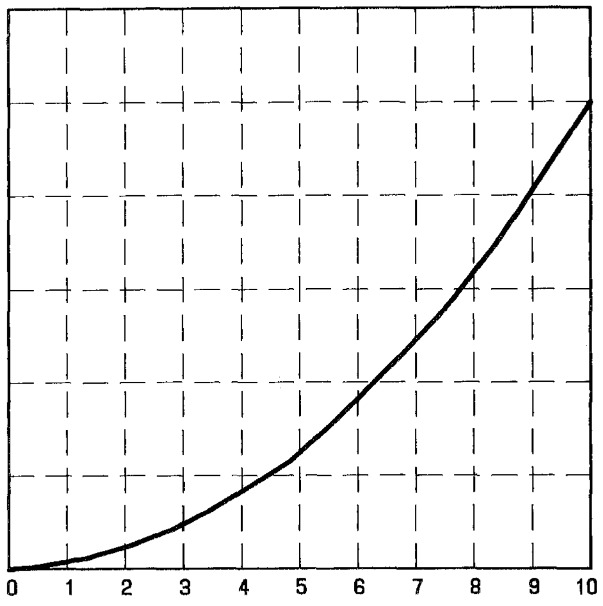


Figure 2-2. Plot of gas -s0,t10 | af " x^2" | plot -xi1,ya | td

The GPS describes graphical objects drawn within a Cartesian plane (Figure 2-3). The Cartesian plane has 65,534 units on each x and y axis. The plane, known as the *universe*, is partitioned into 25 equal-sized square regions. Multi-drawing displays can be produced by placing drawings into adjacent regions and then displaying each region. This will be shown in Chapter 5 (graphics editor).

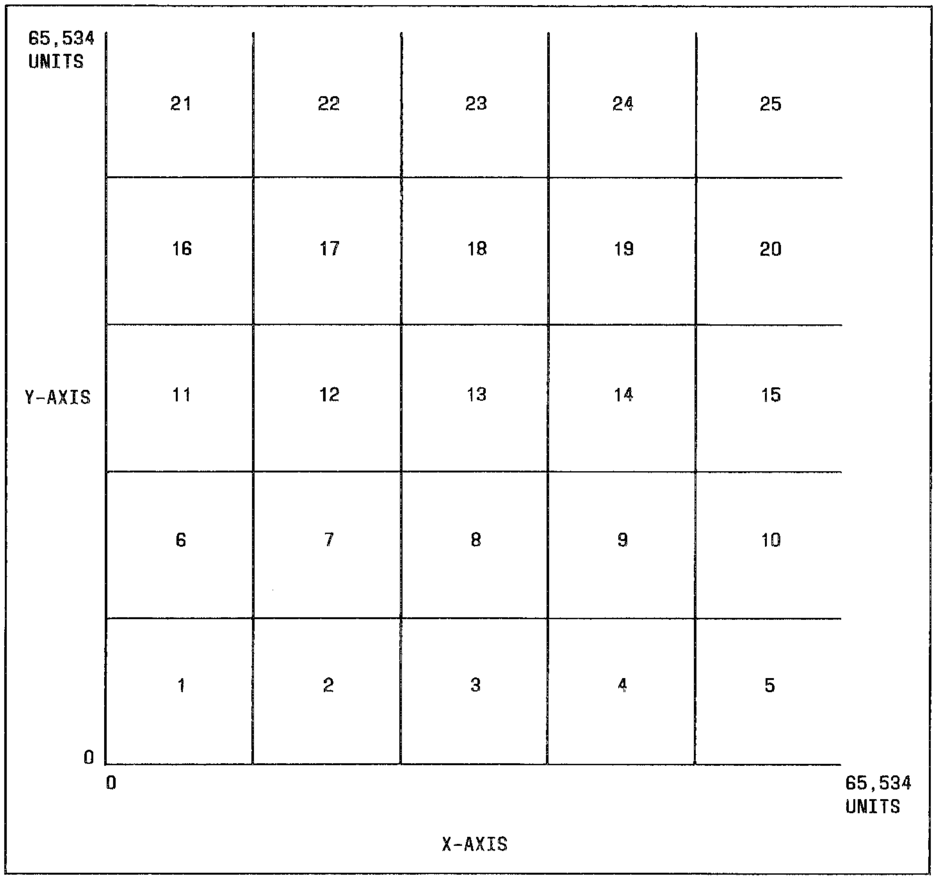


Figure 2-3. Cartesian plane

EXAMPLES OF WHAT YOU CAN DO

Numerical Manipulation and Plotting

Stat is a collection of numerical and plotting commands. All these commands operate on vectors. A vector is a text file that contains a sequence of numbers separated by a delimiter and where a delimiter is anything that is not a number.

For example:

1 2 3 4 5, and
hhh tty47 July 19 09:52

are both vectors. The vectors are:

First Vector = 1 2 3 4 5
Second Vector = 47 19 09 52

Here is an easy way to generate a Celsius-Fahrenheit conversion table using **gas** to generate the vector of Celsius values:

```
^gas –s0,t100,i10 | af " C,9/5*C+32" <CR>
```

The results are:

```
0               32
10              50
20              68
30              86
40              104
50              122
60              140
70              158
80              176
90              194
100             212
```

where:

- **gas –s0,t100,i10** generates a sequence that **s**tarts at 0, **t**erminates at 100, and the **i**ncrement between successive elements is 10.
- **af " C,9/5*C+32"** generates the table. Arguments to **af** are expressions. Operands in an expression are either constants or file names. If a file name is given that does not exist in the current directory, it is taken as the name for the standard input. In this example, **C** references the standard input.

Here is an example that illustrates the use of vector titles and makes a multi-line plot:

```
^gas | title –v" first ten integers" >N<CR>
^root N >RN<CR>
^root –r3 N >R3N<CR>
^root –r1.5 N >R1.5N<CR>
^plot –FN,g N R1.5N RN R3N | td<CR>
```

where:

- **title -v"** *name*" associates a *name* with a vector. Here, the **first ten integers** are associated with the vector output by **gas**. The vector is stored in file **N**.
- **root -r***n* outputs the *n*th root of each element on the input. If **-r***n* is not given, then the square root is output. Also, if the input is a titled vector, the title will be transformed to reflect the **root** function.
- **plot -F***X,g Y(s)* generates a multi-line plot with *Y(s)* plotted versus *X(s)*. The **g** option causes tick marks to appear instead of grid lines.

The results of the plot is shown in Figure 2-4.

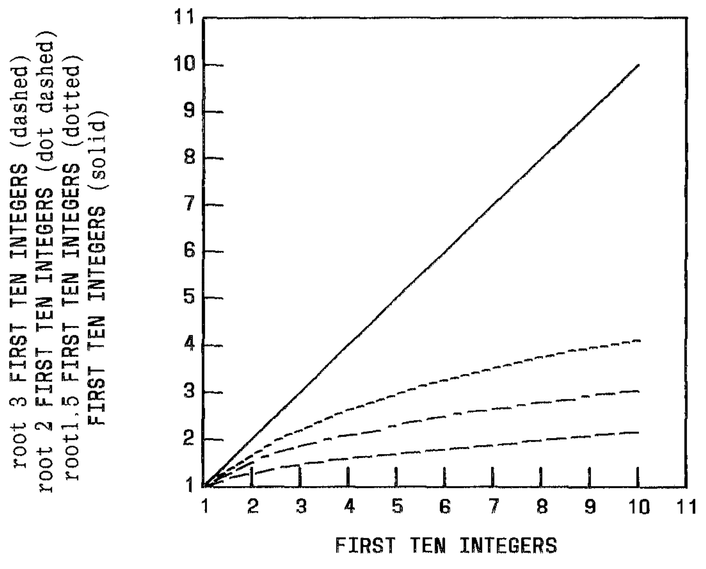


Figure 2-4. Some Roots of the First Ten Integers

The next example generates a histogram of random numbers:

```
^rand –n100 | title –v" 100 random numbers" | qsort |
   bucket | hist | td<CR>
```

where:

- **rand –n100** outputs random numbers using **rand**(3C). Here, 100 numbers are output in the range 0 to 1.
- **title -v" name"** associates a name with a vector. In this case, **100 random numbers** is associated with the vector output by gas.
- **qsort** sorts the elements of a vector in ascending order.
- **bucket** breaks the range of the elements in a vector into intervals and counts how many elements from the vector fall into each interval. The output is a vector with odd elements being the interval boundaries and even elements being the counts.
- **hist** builds a histogram based on interval boundaries and counts.
- **td** command displays drawings on TEKTRONIX 4014 display terminal.

The output is shown in Figure 2-5.

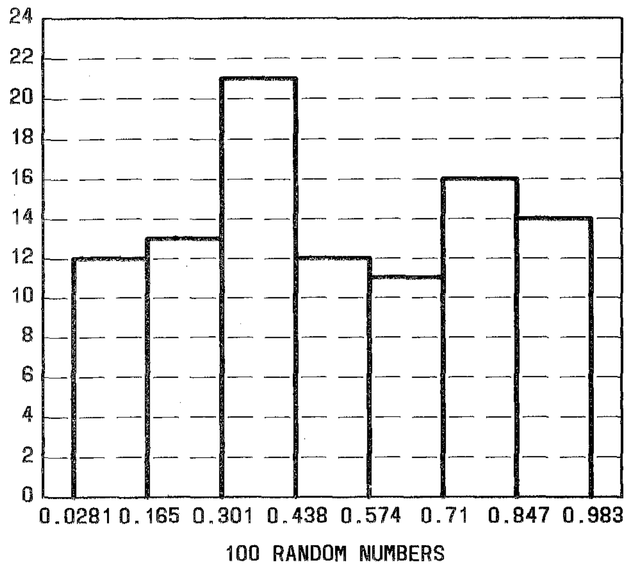


Figure 2-5. Histogram of 100 Random Numbers

Drawings Built From Boxes

There is a large class of drawings composed from boxes and text. Examples are structure charts, configuration drawings, and flow diagrams. In **graphics**, the general procedure to build such box drawings is the same as that for numerical plotting; namely, gather and transform the data, build and display the layout.

As an example, for hierarchical charts, the command line

```
^dtoc | vtoc | td<CR>
```

outputs drawings representing directory structures.

- The **dtoc** command outputs a table of contents that describes a directory structure (Figure 2-6). The fields from left to right are the level number, the directory name, and the number of ordinary readable files contained in the directory.
- The **vtoc** command reads a (textual) table of contents and outputs a visual table of contents, or hierarchical chart (Figure 2-7). Input to **vtoc** consists of a sequence of entries, each describing a box to be drawn. An entry consists of a level number, an optional style field, a text string to be placed in the box, and a mark field to appear above the top right-hand corner of the box.
- The **td** command displays the drawing on a TEKTRONIX terminal.

```
0.          "source"       2
1.          "glib.d"       1
1.1.        "gpl.d"        12
1.2.        "gsl.d"        14
2.          "gutil.d"      6
2.1.        "cvrtopt.d"    7
2.2.        "gtop.d"       8
2.3.        "ptog.d"       5
3.          "stat.d"       54
4.          "tek4000.d"    5
4.1         "ged.d"        37
4.4.        "td.d"         8
5.          "toc.d"        3
5.1.        "ttoc.d"       3
5.2.        "vtoc.d"       22
6.          "whatis.d"     108
```

Figure 2-6. Output of dtoc Command

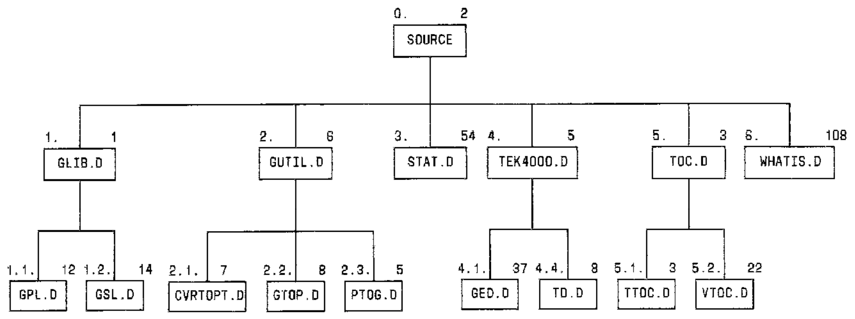


Figure 2-7. Output of vtoc Command

Chapter 3

STAT - A TOOL FOR ANALYZING DATA

Chapter 3

STAT - A TOOL FOR ANALYZING DATA

INTRODUCTION

Stat is a collection of command level functions (nodes) that can be interconnected using the UNIX System shell to form processing networks. Included within **stat** are programs to generate simple statistics and pictorial output.

This chapter introduces **stat** concepts by using a few commands through a collection of examples. A complete definition of all **stat** commands with examples is discussed in Chapter 4.

Much of the power for manipulating text in the UNIX System comes from well-defined, text-processing programs such as the DOCUMENTER'S WORKBENCH* software text processing utilities. These processing programs can be readily interfaced to one another. The general interface

* Trademark of AT&T

is an unformatted text string, and the interconnection mechanism is usually the UNIX System shell. The programs are independent of one another, so new functions can easily be added and old ones changed. Because the text editor operates on unformatted text, arbitrary text manipulation can always be performed even when the more specialized routines are insufficient.

Stat uses the same mechanisms to bring similar power to the manipulation of numbers. It consists of a collection of numerical processing routines that read and write unformatted text strings. It includes programs to build graphical files that can be manipulated using a graphical editor. And since **stat** programs process unformatted text, they can readily be connected with other UNIX System command-level (that is, callable from shell) routines.

It is useful to think of the shell as a tool to build processing networks in the sense of data-flow programming. Command-level routines are the nodes of the network, and pipes and tees are the links. Data flows from node-to-node in the network via data links.

NODE DESCRIPTIONS

Stat nodes are divided into four classes. These classes are:

- Transformer
- Summarizer
- Translator
- Generator.

All these nodes accept the same command-line format:

- A *command* is a *command-name* followed by zero or more *arguments*.
- A *command-name* is the name of any **stat** node.
- An *argument* is a *file-name* or an *option-string*.
- An *option-string* is a — followed by one or more *options*.
- An *option* is one or more letters followed by an optional value. Options may be separated by commas.
- A *file-name* is any name not beginning with a –, or a – by itself (to reference the standard input).

Each file argument to a node is taken as input to one occurrence of the node. That is, the node is executed from its initial state once per file. If no files are given, the standard input is used. All nodes, except *generators*, accept files as input.

Vector(s)

All numerical data in **stat** are stored in text files. These text files are vectors, where a vector is a sequence of numbers separated by delimiter and a delimiter is anything that is not a number. These vectors are processed by command-level routines called nodes.

Transformer Node

A *transformer* is a node that reads an input element, operates on it, and outputs the resulting value. For example, suppose vector **A** contains

```
1 2 3 4 5
```

then the command:

```
^root A<CR>
```

produces the square root of each input elements.

```
1      1.41421      1.73205      2      2.23607
```

Af, for arithmetic function, is a particularly versatile *transformer*. Its argument is an expression that is evaluated once for each complete set of input values. A simple example is:

```
^af " 2*A^2" <CR>
```

that produces

```
2          8          18          32          50
```

twice the square of each element from **A**. Expression arguments to **af** are surrounded by quotes since some of the operator symbols have special meaning to the shell.

The following is a list of all *transformer* commands that are discussed in detail in Chapter 4:

- **abs**, absolute value
- **af**, arithmetic function
- **ceil**, ceiling function
- **cusum**, cumulative sum function
- **exp**, exponential function
- **floor**, floor function
- **gamma**, gamma function
- **list**, list vector
- **log**, logarithm function
- **mod**, modulus function
- **pair**, pair element group
- **power**, power function
- **root**, root function
- **round**, rounded value
- **siline**, generate a line given slope and intercept
- **sin**, sin function
- **subset**, generate a subset.

Parameters

Most nodes accept parameters to direct their operation. Parameters are specified as command-line options. **Root**, for example, is more general than just square root, any root may be specified using the **r** option. For example:

```
^root -r3 A<CR>
```

produces

```
1     1.25992     1.44225     1.5874     1.70998
```

the cube root of each element from **A**.

Summarizers Node

A *summarizer* is a node that calculates a statistic for a vector. Typically, *summarizers* read in all the input values; then, calculates and outputs the statistic. For example, using the vector **A** from the previous example,

A = 1 2 3 4 5

```
^mean A<CR>
```

produces

```
3
```

The following is a list of all *summarizer* commands that are discussed in detail in Chapter 4:

- **bucket**, generates buckets and counts
- **cor**, ordinary correlation coefficient
- **hilo**, finds high and low values
- **lreg**, linear regression
- **mean**, mean function
- **point**, empirical cumulative density function point
- **prod**, product function
- **qsort**, quick sort
- **rank**, rank of vectors
- **total**, sum total
- **var**, variance function.

Building Networks

Nodes are interconnected using the standard UNIX System shell concepts and syntax. A pipe is a linear connector that attaches the output of one node to the input of another. As an example, to find the mean of the cube roots of vector **A** is:

```
^root –r3 A | mean<CR>
```

that produces

```
1.39991
```

Often the required network is not so simple. Tees and sequence can be used to build nonlinear networks. The tee is a pipe fitting that transcribes the standard input to the standard output and makes a copy in a file. To find the mean and median of the transformed vector **A** is:

```
^root –r3 A | tee B | mean; total B<CR>
```

that produces

```
1.39991
6.99955
```

Beware of the distinction between the sequence operator (;), and the linear connector, the pipe (|). The pipe (|) takes the output from one command and inputs it to the other command. Each command is run as a separate process; the shell waits for the last command to end. The sequence operator semicoln (;) allows you to put more than one command on a command line. The output is not directed unless otherwise specified.

Command Substitution

There is a special case of nonlinear networks where the result of one node is used as command-line input for another. Command substitution makes this easy. For example, to generate residuals from the mean of **A** is simply

A = 1 2 3 4 5
mean A = 3

```
^af " A-■mean A■ <CR>
```

that results in

```
-2        -1        0        1        2
```

This example shows that command substitution does the operation in grave accents (``) first, then substitutes that value for the expression in the grave accents. Here it takes the mean of **A** that is 3. It then substitutes the value 3 for **mean A** in the grave accents. Then, the arithmetic expression in quotation marks is completed.

Generator Node

Thus far, vectors have been used but not created. One way to create a vector is by using a *generator*. A *generator* is a node that accepts no input, and outputs a vector based on definable parameters. **Gas** is a *generator* that produces additive sequences. One parameter to **gas** is the **n**umber of elements in the generated vector. As an example, to create the vector **A** that we have been using is:

```
^gas –n5<CR>
```

that produces

```
1          2          3          4          5
```

To name the vector **A**, you can direct the output to **A**.

```
^gas -n5 > A<CR>
```

Vectors are, however, merely text files. Hence, the text editor can be used to create and change the same vector.

```
$vi A<CR>
<a> 1<CR>
2<CR>
3<CR>
4<CR>
5<CR>
<ESC>
<ZZ>
```

A useful property of vectors is that they consist of a sequence of numbers surrounded by delimiters, where a delimiter is anything that is not a number. Numbers are constructed in the usual way

[sign](digits)(.digits)[**e**[sign]digits]

where fields are surrounded by brackets and parentheses. All fields are optional, but at least one field surrounded by parentheses must be present.

An example of entering the number $2.7x10^6$ in a text file **T** would be:

```
^vi T<CR>
<a> +2.7e+06<CR>
<ESC>
<ZZ>
```

Thus, vector **B** could also be created by building the file **B** in the text editor as

```
$vi B<CR>
<a> 1partridge,2tdoves,3frhens,4cbirds,5gldnrings<CR>
<ESC>
<ZZ>
```

Note: Remember that a vector is separated by a delimiter. A delimiter is anything that is not a number.

that, when read by

```
^list B<CR>
```

produces

```
1         2         3         4         5
```

The following is a list of all *generator* commands that are discussed in detail in Chapter 4.

- **gas**, generate additive sequence
- **prime**, generate prime numbers

- **rand**, generate random sequence.

A Simple Example: Interacting with a Data Base

When used with the UNIX System tools for manipulating text, **stat** provides an effective means for exploring a numerical data base. Suppose, for example, there is a subdirectory called **data** containing data files that include the lines:

path length = *nn* (*nn* is any number)
node count = *nn*

To access the value for **node count** from each file, sort the values into ascending order, store the resulting vector in file **C**, and get a copy on the terminal by typing

```
^grep " node count"  data/* | qsort | tee C<CR>
17     19     22     32     39
50     68     78     125    139
```

The slash (/) in the above example was used because we were in our home directory when this command was entered. This will scan all files in the subdirectory **data**.

If some of the data files have numbers in their name, we must protect those numbers from being considered data. Using **cat**, this is easy:

```
^cat data/* | grep " node count"  | qsort | tee C<CR>
```

To get a feel for the distribution of node counts, shell iteration can be used to an advantage. In this example, we will generate the lower hinge, the median, and the upper hinge of the sorted vector **A**.

```
for i in .25 .5 .75<CR>
do point -p$i A<CR>
done<CR>
24.5
44.5
75.5
```

Translators Node

Translators are used to view data pictorially. A *translator* is a node that produces a stream of a different structure from what it consumes. Graphical *translators* consume vectors and produce pictures in a language called GPS, for graphical primitive string. A GPS is a format for storing a picture. A picture is defined in a Cartesian plane of 64K points on each axis. The plane, or universe, is divided into 25 square regions numbered 1 to 25 from the lower left to the upper right (see Figure 2-3.) Various commands exist that can display and edit a GPS.

The following is a list of all *translator* commands that are discussed in detail in Chapter 4.

- **bar**, build a bar chart
- **hist**, build a histogram
- **pie**, build a pie chart
- **plot**, plots a x-y plot.

For example:

Hist is a *translator* that produces a GPS that describes a histogram from input consisting of interval limits and counts. The *summarizer* **bucket** produces limits and counts, thus:

```
^bucket A | hist | td<CR>
```

generates a histogram of the data of vector **A** and displays it on a display terminal (Figure 3-1). **Td** translates the GPS into machine code for TEKTRONIX 4014 display terminals.

A wide range of X-Y plots can be constructed using the *translator* **plot**. For example, to build a scatter plot of **path length** with **node count** (Figure 3-2) is:

```
^grep " path length"  data/* | title -v" path length"  >A<CR>
^grep " node count"  data/* | title -v" node count"
    | plot -FA,dg | td<CR>
```

A vector may be given a title using **title**. When a titled vector is plotted, the appropriate axis is labeled with the vector title. The **plot -FA,dg** uses the **A** vector for the x-axis and standard input for the y-axis.

When a titled vector is passed through a *transformer*, the title is altered to reflect the transformation. Thus, in a graph of log **node count** versus the cube root of **path length**, such as

```
^grep " node count" | title -v" node count" | log >B<CR>
^root -r3 A | plot -F-,dg B | td<CR>
```

the axis labels automatically agree with the vectors plotted (Figure 3-3). The **plot -F-,dg B** uses the **B** vector as the y-axis and standard input for the x-axis.

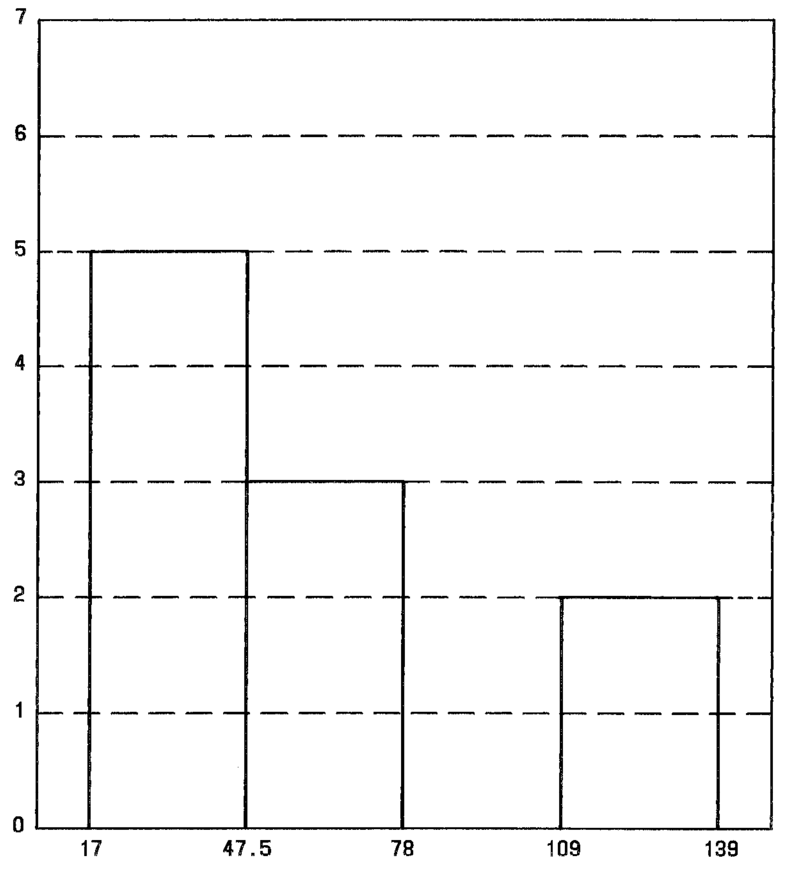


Figure 3-1. bucket A | hist | td

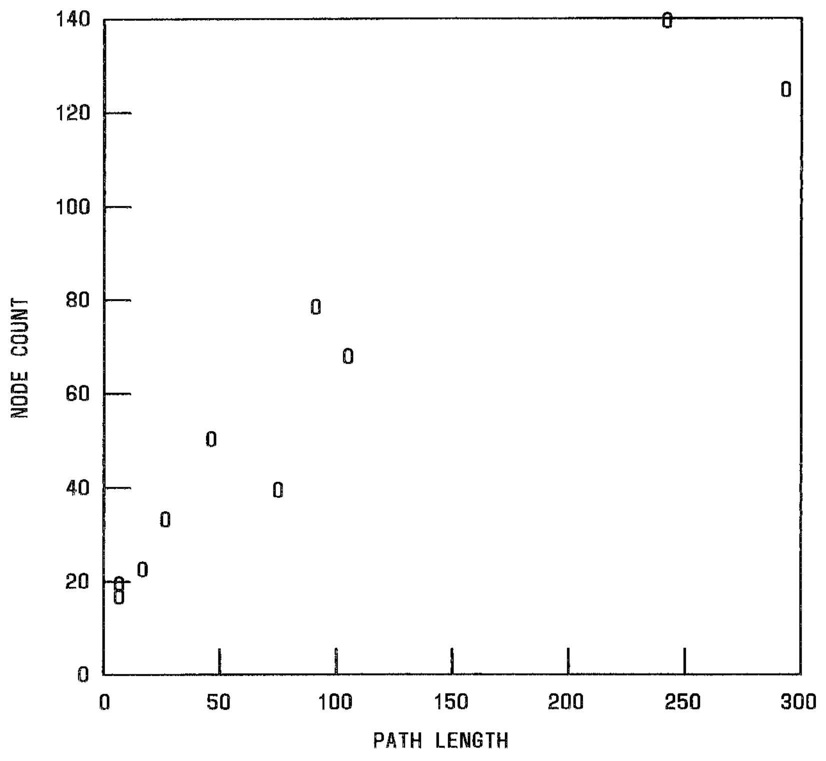


Figure 3-2. Scatter Plot

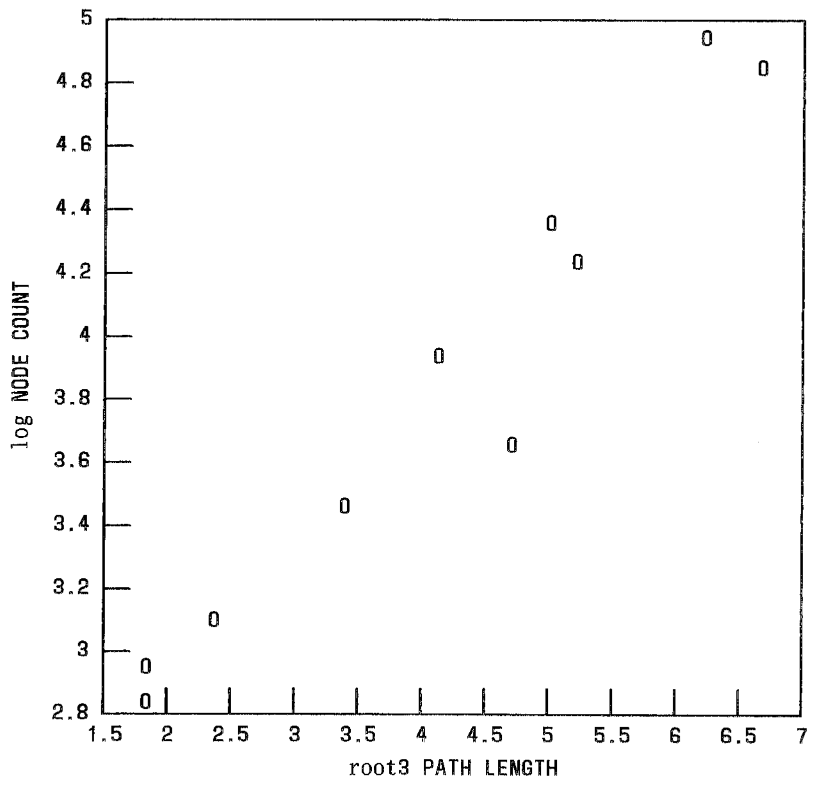


Figure 3-3. Transformed Scatter Plot

EXAMPLES

Example 1

Calculate the total value of an investment held for many years at an interest rate compounded annually.

SOLUTION

```
^Principal=1000<CR>
^echo Total return on $Principal units compounded annually<CR>
^echo " rates:\t\t\c" ; gas -s.05,t.15,i.03 | tee rate<CR>
^for Years in 1 3 5 8<CR>
^do echo " $Years year(s):\t\c" ; af " $Principal*(1+rate)^$Years" <CR>
^done<CR>

Total return on 1000 units compounded annually rates:
        0.05     0.08     0.11     0.14

1 year(s):  1050      1080      1110      1140
3 year(s):  1157.62   1259.71   1367.63   1481.54
5 year(s):  1276.28   1469.33   1685.06   1925.41
8 year(s):  1477.46   1850.93   2304.54   2852.59
```

Note:

Notice the distinction between vectors and constants as operands in the expression to **af**. The shell variables *$Principal* and *$Years* are constants to **af**, while the file *rate* is a vector. **Af** executes the expression once per element in *rate*.

Example 2

PROBLEM

Three ordered vectors (**A**, **B**, and **C**) of scores from many tests are given. Each vector is from one test-taker, each element in a vector is the score on one test. There are missing scores in each vector shown by the value −1. Generate three new vectors containing scores only for those tests where no data is missing.

SOLUTION

```
^echo Before:<CR>
^gas –n■ank A■| tee N | af " label,A,B,C" <CR>

^for i in N B C A<CR>
^do subset –FA,l–1 $i >s$i; done<CR>
^for i in N A C B<CR>
^do subset –FsB,l–1 s$i | yoo s$i; done<CR>
^for i in N A B C<CR>
^do subset –FsC,l–1 s$i | yoo s$i; done<CR>
^echo " \nAfter:" <CR>
^af " sN,sA,sB,sC" <CR>

Before:
1       5       6       –1
2       7       10      10
3       –1      10      9
4       10      –1      8
5       6       5       –1
6       5       7       5
7       –1      7       8
8       –1      –1      8
9       3       –1      8
10      6       10      10
11      7       5       7

After:
2       7       10      10
6       5       7       5
10      6       10      10
11      7       5       7
```

Notes:

1. The approach is to eliminate those elements in all vectors that correspond to –1 in the base vector. Each of the three vectors takes a turn at being the base. It is important to subset the base last. The command **yoo** (see **gutil** in the *AT&T 3B2 Computer User Reference Manual*) takes the output of a pipeline and copies it into a file used in the pipeline. This cannot be done by redirecting the output of the pipeline as this would cause a concurrent read and write on the same file.

2. The printing of the "Before" matrix illustrates a useful property of **af**. The first name in an expression that does not match any name in the present working directory is a reference to the standard input. In this example, **label** references the input coming through the pipe.

Example 3

PROBLEM

Generate a bar chart of the percent of execution time consumed by each routine in a program.

SOLUTION

```
^prof | cut –c1–15 | sed –e 1d –e " / 0.0/d" –e " s/^  *//" >P<CR>
^echo These are the execution percentages; cat P<CR>
^title P –v" execution time in percent"  | bar –xa –yl0,
  yh100 | label –br–45,FP | td<CR>

These are the execution percentage:

_fork  32.9
_creat  14.3
_sbrk  14.3
_read  14.3
_open  14.3
_prime  9.9
```

Note:

Prof is a UNIX System command that generates a listing of execution times for a program (see **prof**(1)). **Cut** and **sed** are used to eliminate extraneous text from the output of **prof**. (Because verbiage can get in the way, **stat** nodes say little.) Notice that **P** is a vector to **title**, while it is a text file to **cat** and **label**.

Figure 3-4 shows the output of these commands.

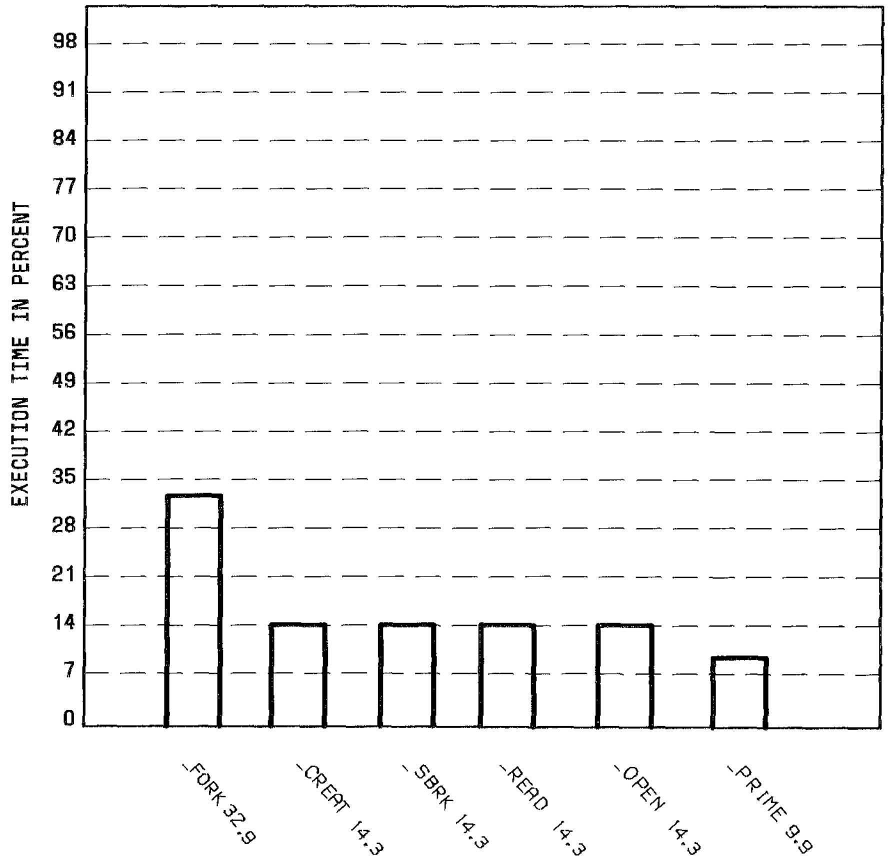


Figure 3-4. Bar Chart Showing Execution Profile

Example 4

PROBLEM

Plot the relationship between the execution time of a program and the number of processes in the process table.

SOLUTION

The first program generates the performance data

```
^for i in `gas -n12`<CR>
^do<CR>
^ps -ae | wc -l >>Procs&<CR>
 time prime -n1000 >/dev/null 2>>Times<CR>
 sleep 300<CR>
^done<CR>
```

The second program analyzes and plots the data

```
^for i in real user sys<CR>
^do<CR>
^grep $i Times | sed " s/$i//"  |
    awk –F: " { if(NF==2) print \$1*60+\$2; else
   print }"  | title –v" $i time in seconds"  >$i<CR>
^siline –[illegible]reg –o,FProcs $i [illegible]Procs >$i.fit<CR>
^done<CR>
^title –v" number of processes"  Procs | yoo Procs<CR>
^plot –dg,FProcs real –r12 >R12<CR>
^plot –ag,FProcs real.fit –r12 >>R12<CR>
^plot –dg,FProcs sys –r13 >R13<CR>
^plot –ag,FProcs sys.fit –r13 >>R13<CR>
^plot –dg,FProcs user –r8 >R8<CR>
^plot –ag,FProcs user.fit –r8 >>R8<CR>
^ged R12 R13 R8<CR>
```

Notes:

1. The performance data is the execution time, as reported by the UNIX System **time** command, to generate the first 1000 prime numbers. **Times** outputs three times for each run:

   - The time in system routines
   - The time in user routines
   - Total real time.

2. The output of the **time** command is saved in the file **Times**. Each of these types of time is treated separately by the analysis program.

3. In the file **Procs** are the number of processes running on the system during each execution of **prime**. The short **awk** program converts "minutes:seconds" format to "seconds." **Lreg** does a linear regression of the time vectors on the size of the process table. **Siline** generates a line based on the parameters from the regression. One plot is generated for each type of time. Each plot is put into a different region so that they can be displayed and manipulated simultaneously in **ged**.

4. Figure 3-5 shows the output of these commands.

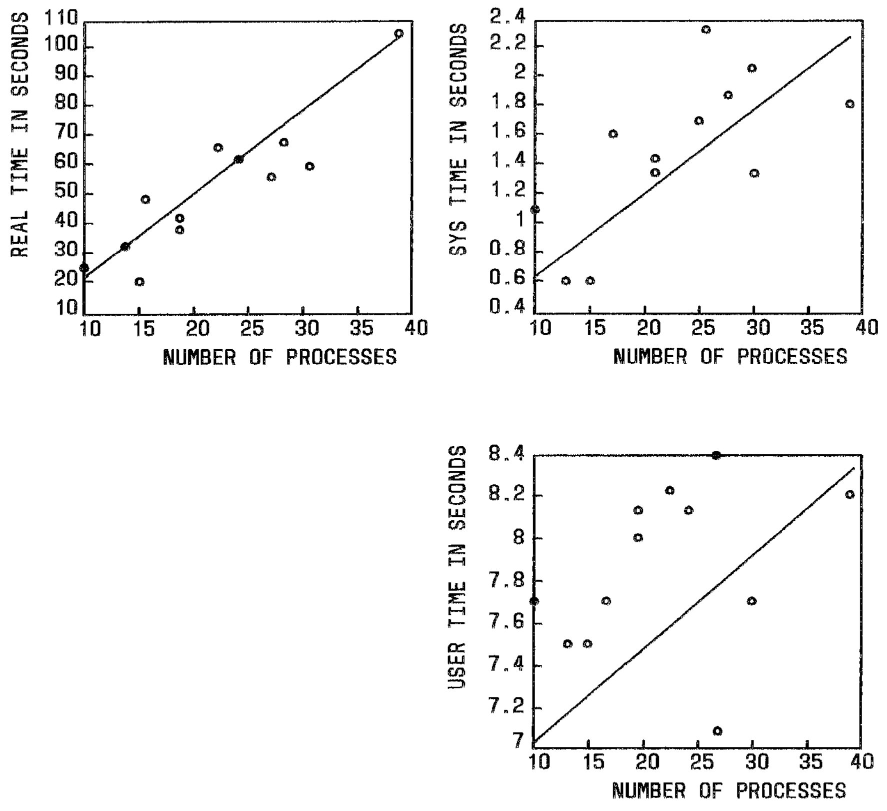


Figure 3-5. Relationship Between Execution Time and Number of Processes

Chapter 4

COMMAND DESCRIPTIONS

Chapter 4

COMMAND DESCRIPTIONS

COMMAND SUMMARY

The Graphics Utilities provide 60 UNIX System commands. A summary of these commands are provided in Figure 4-1.

COMMAND	DESCRIPTION
abs	**Absolute value**, its output is the absolute value for each element of the input vector(s).
af	**Arithmetic function**, its argument is an expression that is evaluated once for each complete set of input values.

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 1 of 8)

COMMAND	DESCRIPTION
bar	**Bar chart**, its output is a GPS that describes a bar chart display.
bel	**Bel**, causes most terminals to sound an audible tone, a useful nonvisual signal.
bucket	**Bucket**, breaks the range of a vector into intervals and counts how many elements from the vector fall into each interval. The output is a vector with odd elements being the interval boundaries and even elements being the counts.
ceil	**Ceiling function**, its output is a vector with each element being the smallest integer greater than the corresponding element from the input vector(s) (rounds up to the next integer).
cor	**Ordinary correlation coefficient**, its output is the ordinary correlation coefficient between a base vector and another vector.
cusum	**Cumulative sum**, its output is a vector that calculates the sum of all the elements found in the input vector.

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 2 of 8)

COMMAND	DESCRIPTION
cvrtopt	**Cvrtopt**, it reformats the arguments (usually the command line arguments of a calling shell procedure) to improve processing by shell procedures.
dtoc	**Directory table of contents**, its output is a list of all readable subdirectories beginning at a specified directory or the current directory.
erase	**Erases the screen of the TEKTRONIX 4014 display terminal.**
exp	**Exponential function**, its output is a vector with elements e raised to the *x* power, where e is about 2.71828, and *x* are the elements from the input vector(s).
floor	**Floor function**, its output is a vector with each element being the largest integer less than the corresponding element from the input vector(s) (rounds down to next integer).
gamma	**Gamma function**, its output is the gamma value for each element of the input vector(s).
gas	**Generate additive sequence**, its output is a vector of number elements determined by the parameters **s** (start), **t** (terminate), and **i** (interval).
gd	**Gd**, prints a human readable listing of GPS.

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 3 of 8)

COMMAND	DESCRIPTION
ged	**Graphical editor**, allows displaying and editing of GPS.
graph	**Graph**, it takes pairs of numbers from the standard input as abscissas and ordinates of a graph. Then, it draws a straight line connecting successive points.
graphics	**Graphics**, puts the 3B2 Computer into the graphics mode.
gtop	**Gtop**, it transforms a GPS into plot(5) format displayable by the **tplot** command.
hardcopy	When issued from a TEKTRONIX 4014 display terminal with a hard copy unit (printer), it generates a screen copy on the unit.
hilo	**Hilo**, its output is the high and low values found across all the input vector(s).
hist	**Hist**, its output is a GPS that describes a histogram display.
hpd	Display GPS on a HP 7221A Graphics Plotter, its output is scope coded for a HP 7221A Plotter.
label	**Label**, it appends the axis labels from a label file to a GPS of a data plot (like that produced by hist, bar, and plot).
list	**List**, its output is a listing of the elements of the input vector(s).

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 4 of 8)

COMMAND	DESCRIPTION
log	**Logarithm**, its output is the logarithm for each element of the input vector(s).
lreg	**Linear regression**, its output is the slope and intercept from the least squares linear regression of each vector on a base vector.
mean	**Mean**, its output is the mean of the elements in the input vector(s).
mod	**Modulo function**, its output is a vector with each element being the remainder of dividing the corresponding element from the input vector(s).
pair	**Pair element groups**, its output is a vector with elements taken alternately from a base vector and from a vector.
pd	**Pd**, prints a human readable listing of plot(5) format.
pie	**Pie**, its output is a GPS that describes a pie chart.
plot	**Plot**, its output is a GPS that describes an x-y graph.
point	**Empirical cumulative density function point**, its output is a linearly interpolated value from the empirical cumulative density function (e.c.d.f) for the input vector.

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 5 of 8)

COMMAND	DESCRIPTION
power	**Power function**, its output is a vector with each element being a power of the corresponding element from the input vector(s).
prime	**Generate prime numbers**, its output is a vector of number elements determined by the parameters low and high.
prod	**Product**, its output is the product of the elements in the input vector(s).
ptog	**Ptog**, transforms plot(5) format into a GPS.
qsort	**Quick sort**, its output is a vector of the elements from the input vector in ascending order.
quit	**Quit**, ends the current terminal session.
rand	**Generate random sequence**, its output is a vector of number elements determined by the parameters low, high, multiplier, and seed.
rank	**Rank**, its output is the number of elements in each input vector.
remcom	**Remove comments**, the input is copied to its output, with the comments removed.

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 6 of 8)

COMMAND	DESCRIPTION
root	**Root function**, its output is a vector with each element being the root of the corresponding element from the input vector(s).
round	**Rounded value**, its output is the rounded value for each element of the input vector(s).
siline	**Siline**, it generates a line given slope and intercept.
sin	**Sine**, its output is the sine for each element of the input vector(s).
spline	**Interpolate smooth curve**, it takes pairs of numbers from the standard input as abscissas and ordinates of a function.
subset	**Generates a subset**, its output is elements selected from the input based on a key and option(s).
td	**Td**, it displays a GPS on a TEKTRONIX 4014, its output is scope coded for a TEKTRONIX 4014 terminal.
tekset	**Tekset**, clears the display screen, sets the display mode to alpha, and the characters to the smallest font.

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 7 of 8)

COMMAND	DESCRIPTION
title	**Title**, it appends a title to a vector or it appends a title to a GPS.
total	**Total**, its output is the sum total of the elements in the input vector(s).
tplot	**Tplot**, it reads plotting instructions from the standard input for a particular terminal.
ttoc	**Ttoc**, its output is the textual table of contents generated by the .H macro of the nroff or troff raw data of the Document Workbench Utilities.
var	**Var**, it finds the difference between the slope point and the outer point.
vtoc	**Visual table of contents**, its output is a GPS that describes a Visual table of contents (vtoc or hierarchical chart) of the Textual Table of Contents (TTOC) entries from the input.
whatis	Brief online documentation, prints a brief description of each command given. If no command is given, then the current list of description commands are printed.
yoo	**Yoo**, is a piping primitive that deposits the output of a pipeline into a file used in the pipeline.

Figure 4-1. Graphics Utilities—COMMAND SUMMARY (Sheet 8 of 8)

COMMAND DESCRIPTIONS

abs — Absolute Value

General

The absolute value (**abs**) is a transformer node that is used to find the absolute value of each element of the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
abs [-option] [vector(s)]
```

Option:

c*n*	*n* is the number of output elements per line. By default, c = 5.

Command Example

The following example will output the absolute value of each element of the vector **A**, two per line.

A = 23 -44 55 -66 70

```
^abs -c2 A<CR>

23  44
55  66
70
```

af — Arithmetic Function

General

The arithmetic function (**af**) is a particularly versatile transformer node. Its argument is an expression that is evaluated once for each complete set of input values. The input values comes from vectors specified by an expression. The expression consists of an operand and an operator.

An operand is either a vector, function, or constant.

Expression operands are:

Vectors:	Filenames with the restriction that they must begin with a letter and be composed only from letters, digits, '_', and '.'. The first unknown filename (one not in the current directory) references the standard input. A warning will appear if a file cannot be read.
Functions:	The name of a command, followed by the command arguments in parentheses. Arguments are written in command-line format.
Constants:	Floating point or integer (but not E notation) number.

The operators are listed below in order of their decreasing precedence.

The x_i (y_i) represents the start element from X (Y) for the expression.

- $'Y$ – reference y_{i+1}. y_{i+1} is consumed; the next value from Y is y_{i+2}. Y is a vector.

- $X\hat{}-Y$ – x_i raised to the y_i power, negation of y_i. Association is left to right. X and Y are expressions.

- $X*Y$ X/Y $X\%Y$ – x_i multiplied by, divided by, modulo y_i. Association is left to right. *X* and *Y* are expressions.

- $X+Y$ $X-Y$ – x_i plus, minus y_i. Association is left to right. *X* and *Y* are expressions.

- X,Y – yields x_i, y_i. Association is left to right. *X* and *Y* are expressions.

Note: Parentheses may be used to alter precedence. Because many of the operator characters are special to the shell, it is good practice to surround *expressions* in quotes.

Command Format

```
af [-option(s)] expression(s)
```

Options:

c*n*	*n* elements per line in the output
t	output is titled from the vector on the standard input
v	verbose mode, function expansions are echoed.

Command Example

The following example will solve the expression $y=3(A)^2$ for each element of vector **A**, two per line.

A = 1 2 3 4 5

```
^af -c2 " 3*A^2" <CR>

3   12
27  48
75
```

bar — Build a Bar Chart

General

The **bar** command is a translator node thats output is a GPS that describes a bar chart. The input is a vector of counts that describes the y-axis. By default, x-axis will be labeled with integers beginning at 1; for other labels, see label. If no vector is given, then the standard input is assumed.

Command Format

```
bar [-option(s)] [vector(s)]
```

Options:

a	Suppress axis
b	Plot bar chart with bold weight lines; otherwise, use medium.
f	Do not build a frame around plot area.
g	Suppress background grid.
r*n*	Put the bar chart in GPS region *n*, where *n* is between 1 and 25, inclusively. The default is 13.
w*n*	*n* is the ratio of the bar width to center-to-center spacing expressed as a percentage. Default is 50, giving equal bar width and bar space.
x*n* **(y***n***)**	Position the bar chart in the GPS universe with x-origin (y-origin) at *n*.

xa (ya)	Do not label x-axis (y-axis).
yl*n*	*n* is the y-axis low-tick value.
yh*n*	*n* is the y-axis high-tick value.

Command Example

The following example will show how to create a bar chart of the vector **C**.

C = 1 2 3 6 7 8 9

```
^bar C | td<CR>
```

where

- **bar C** output is a GPS describing a bar chart.
- **td** command displays the drawing on a TEKTRONIX 4014 display terminal.

See Figure 4-2 for a result of the drawing.

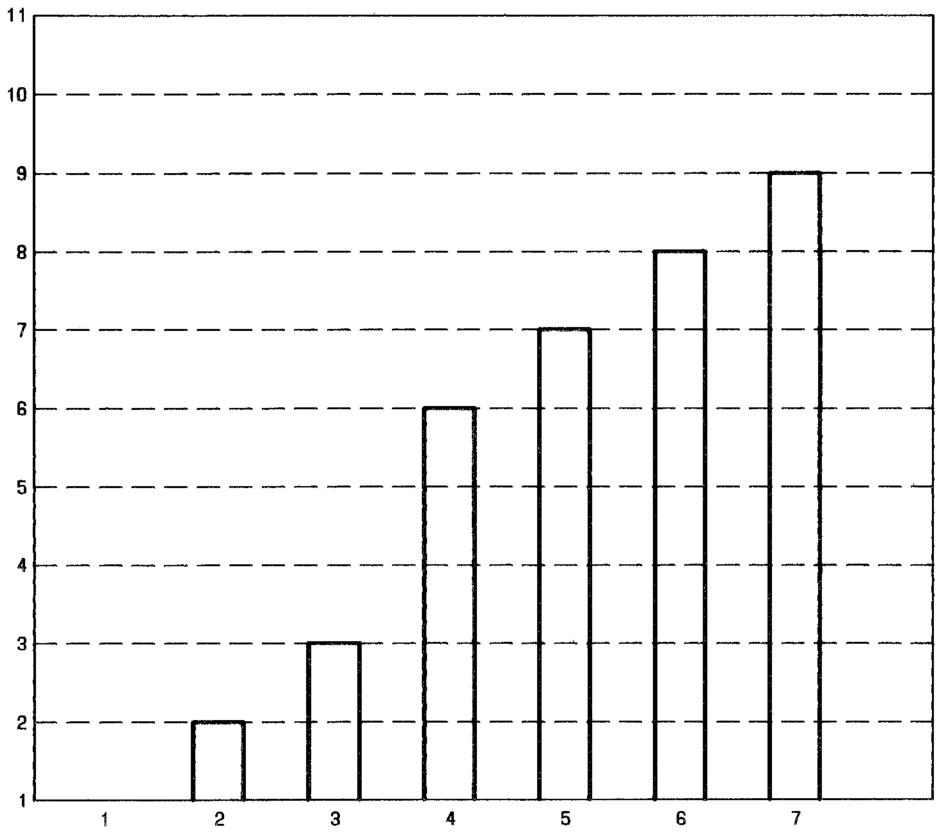


Figure 4-2. Plot of bar C ¦ td

bel — Bell Character

General

The **bel** command causes most terminals to sound an audible tone, which is a useful nonvisual signal.

bucket — Generate Buckets and Counts

General

The **bucket** command is a summarizer node that breaks the range of a vector into intervals and counts. The output is a vector with odd values being bucket limits (in parentheses) and even values being the number of elements from the input within the limits. Input is assumed to be sorted. If no input vector(s) are given, the standard input is assumed.

Command Format

```
bucket [-option(s)] [sorted vector(s)]
```

Options:

a*n*	Choose limits such that *n* is the average count per bucket.
c*n*	*n* elements per line in the output.
F*vector*	Take limit values from *vector*.
h*n*	*n* is the highest limit.
i*n*	*n* is the interval between limits.
l*n*	*n* is the lowest limit.
n*n*	*n* is the number of buckets.

Command Example

The following example will determine the intervals and counts of the vector **D**, by using the **bucket** command.

D = 10 20 30 40 50

```
^bucket D<CR>

(10) 2 (23.333) 1 (36.6667) 2 (50)
```

where

- The first bucket interval is between 10 and 23.3. The bucket count is 2, composed of elements 10 and 20.
- The second bucket interval is between 23.3 and 36.6. The bucket count is 1, composed of element 30.
- The third bucket interval is between 36.6 and 50. The bucket count is 2, composed of elements 40 and 50.

ceil — Ceiling Function

General

The ceiling function (**ceil**) is a transformer node. The output is a vector with each element being the smallest integer greater than or equal to the corresponding element from the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
ceil [-option] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line.

Command Example

The following example will find the ceiling function of each element of the input vector **E**.

E = 10.5 -20.5 30 40.6 50

```
^ceil E<CR>

11 -20 30 41 50
```

cor — Ordinary Correlation Coefficient

General

The ordinary correlation coefficient (**cor**) is a summarizer node that determines if a base vector and another vector are related. The base vector is specified by using the **F** option. If the base or vector is not given, it is assumed to come from the standard input. Each vector is compared to the base. Both base and vector must be of the same length.

Command Format

```
cor [-option] [vector(s)]
```

Option:

F*vector*	*vector* is the base.

Command Example

The following is an example of finding the correlation coefficient between the base vector **A** and another vector **B**.

A = 10 20 30 40 50
B = 20 30 40 50 60

```
^cor -FA B<CR>

1
```

where

- 0 = vector(s) are independent.
- 1 = vector(s) are related.
- -1 = vector(s) are inverse related.

Note: The ordinary correlation coefficient does not have to be only 0,1,-1. It could be .9, .3, ..etc.

where

.9	shows a strong relationship between vectors.
.3	shows a weak relationship between vectors.

cusum — Cumulative Sum

General

The cumulative sum (**cusum**) is a transformer node that calculates the running sum of all the elements found in the input vector. If no input vector is given, then the **cusum** implements a running accumulator. The data is then entered by the keyboard until an end-of-file command is given. On most terminals, the end-of-file command is control-d (*<CTRL-d>*).

Command Format

```
cusum [-option] [vector(s)]
```

Option:

c*n*	*n* is the number of output elements per line.

Command Example

The following example finds the cumulative sum of all the elements of vector **F**, two per line.

F = 20 30 40 50 60

```
^cusum -c2 F<CR>

20  50
90  140
200
```

The following example uses a running accumulator for the standard input in finding the cumulative sum of the values.

```
^cusum<CR>
20<CR>
20  30<CR>
50  40<CR>
90  50<CR>
140 60<CR>
200
<control-d>
```

cvrtopt — Options Converter

General

The **cvrtopt** command reformats arguments (usually the command line arguments of a calling shell procedure) to improve processing by shell procedures. An argument is either a filename (a string not beginning with a ′-′, or a ′-′ by itself) or an option string (a string of options beginning with a ′-′). Output is of the form:

```
-option -option . . . filename(s)
```

All options appear singularly and preceding any filenames. Option names that take values (e.g., -r1.1) or are two letters long must be described through options to **cvrtopt**. Output is to the standard output.

Cvrtopt is usually used with set, in the following way, as the first line of a shell procedure:

```
set - ‘cvrtopt [-option(s)] $@‘
```

Set will reset the command argument string ($1,$2,...) to the output of **cvrtopt**. The minus option to set turns off all flags so that the options produced by **cvrtopt** are not interpreted as options to set.

Command Format

```
cvrtopt [-option(s)] arg(s)
```

s	String accepts string values.
f	String accepts floating point numbers as values.
i	String accepts integers as values.
t	String is a two letter option name that takes no value.

Note: String is a one or two letter option name.

Command Example

The following example shows how the **cvrtopt** command breaks up the option string (-lds).

```
^cvrtopt -lds<CR>

-l -d -s
```

dtoc — Directory Table of Contents

General

The **dtoc** command outputs a table of contents that describes a directory structure. It lists all readable subdirectories beginning at directory. If no directory is given, the list begins at the current directory. The output is as a textual table of contents (TTOC) readable by **vtoc**. The number of nondirectory files in each directory is shown in the marked field of the table of contents. The fields from left to right are level number, directory name, and the number of ordinary readable files contained in the directory.

Command Format

```
dtoc [directory]
```

Command Example

The following is an example of using the **dtoc** command to describe the directory **mtp**.

```
^dtoc<CR>

0. "mtp"  13
```

where

- **0.** is the level number.
- **mtp** is the directory name.
- **13** is the number of files in that directory.

erase — Erase Character

General

The **erase** command erases the screen of the TEKTRONIX 4014 display terminal.

exp — Exponential Function

General

The exponential function (**exp**) is a transformer node. The output is a vector with elements e raised to the _x power, where e is about 2.71828, and _x are the elements from the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
exp [-option] [vector(s)]
```

Option:

c*n*	*n* is the number of output elements per line.

Command Example

The following is an example of finding the exponential function of each element of vector **G**, the values are printed out two per line.

G = 10 20 30 40 50

```
^exp -c2 G<CR>

22026.5      4.85165e+08
1.06865e+13  2.35385e+17
5.18471e+21
```

floor — Floor Function

General

The floor function (**floor**) is a transformer node. The output is a vector with each element being the largest integer less than the corresponding element from the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
floor [-option] [vector(s)]
```

Option:

c*n*	*n* is the number of output elements per line.

Command Example

The following example will find the floor function of each element of the input vector **E**.

E = 10.5 -20.5 30 40.6 50

```
^floor E<CR>

10 -21 30 40 50
```

gamma — Gamma Function

General

The gamma function (**gamma**) is a transformer node. The output is the gamma value for each element of the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
gamma [-option] [vector(s)]
```

Option:

c*n*	*n* is the number of output elements per line.

Command Example

The following example will find the gamma function of each element of the input vector **A**. The output will be placed three per line.

A = 10 20 30 40 50

```
^gamma -c3 A<CR>

367880      1.21645e+17      8.84176e+30
2.o3979e+46 6.08282e+62
```

gas — Generate Additive Sequence

General

The **Gas** command is a generator node that produces additive sequences. A generator is a node that accepts no input, and outputs a vector based on definable parameters. These parameters are **s** (start), **t** (terminate), and **i** (interval). These parameters are set by command options.

Command Format

```
gas [-option(s)]
```

Options:

c*n*	*n* elements per output line.
i*n*	*n* defines interval. If not given, interval = 1.
n*n*	*n* = number. If not given, number = 10, unless terminate is given, then number = (terminate - start) divided by the interval.
s*n*	*n* = start. If not given, start = 1.
t*n*	*n* = terminate. If not given, terminate = positive infinity. The default value of number usually terminates generation before positive infinity is reached.

Command Example

The following example will generate a vector that has values starting at 0, incremented by 10, and terminated at 100.

```
^gas -s0,t100,i10<CR>

10 20 30 40 50 60 70 80 90 100
```

gd — GPS Dump

General

The **Gd** command prints a human readable listing of GPS. If no file is given, the standard input is used.

Command Format

```
gd [GPS file(s)]
```

Command Example

The following is an example of creating a GPS of the expression $y=x^2$ and printing the GPS in readable form.

```
^gas | af " x^2"  | plot >A<CR>
^gd A<CR>

comment    37777700002   37777771037   37777771037
lines    col0 wt1 st0    -3553 -3553    5554 -3553
text     col  0 fon 16 tsz200 tro  0  -3553 -3833 0
text     col  0 fon 16 tsz200 tro  0  -2642 -3833 1
text     col  0 fon 16 tsz200 tro  0  -1732 -3833 2
text     col  0 fon 16 tsz200 tro  0  -821 -3833 3
text     col  0 fon 16 tsz200 tro  0  90 -3833 4
text     col  0 fon 16 tsz200 tro  0  1001 -3833 5
text     col  0 fon 16 tsz200 tro  0  1911 -3833 6

(All data not shown.)
```

ged — Graphical Editor

General

The graphical editor (**ged**) allows displaying and editing of the GPS. The graphics editor will not be discussed in detail at this point. Chapter 5 has been devoted to the graphics editor.

R	Invoke the editor in a restricted shell environment.
e	Do not erase screen before initial display.
r*n*	Window on GPS region *n*, *n* between 1 and 25, inclusively.
u	Window on the entire GPS universe.

Command Example

The following is an example of how to enter the graphics editor.

```
^ged<CR>
*
```

graph — Draw a Graph

General

The **graph** command, with no options, takes pairs of numbers from the standard input as abscissas and ordinates of a graph. Successive points are connected by straight lines. The graph is encoded on the standard output for display by the **tplot** command.

If the coordinates of a point are followed by a non-numeric string, that string is printed as a label beginning on the point. Labels may be surrounded with quotes ("), in which case they may be empty or contain blanks and numbers; labels never contain new-lines.

Command Format

```
graph [-options]
```

Options:

a	Supply abscissas automatically (they are missing from the input); spacing is given by the next argument (default 1). A second optional argument is the starting point for automatic abscissas (default 0 or lower limit given by -x).
b	Break (disconnect) the graph after each label in the input.
c	Character string given by next argument is default label for each point.

g	Next argument is grid style, 0 no grid, 1 frame with ticks, 2 full grid (default).
l	Next argument is labeled for graph.
m	Next argument is mode (style) of connecting lines: 0 disconnected, 1 connected (default). Some devices give distinguishable line styles for other small integers (such as the TEKTRONIX 4014: 2=dotted, 3=dash-dot, 4=short-dash, 5=long-dash).
s	Save screen, do not erase before plotting.
x [**l**]	If l is resent, x axis is logarithmic. Next 1 (or 2) arguments are lower (and upper) x limits. Third argument, if present, is grid spacing on x axis. Normally these quantities are determined automatically.
y [**l**]	Similarly for y.
h	Next argument is fraction of space for height.
w	Similarly for width.
r	Next argument is fraction of space to move right before plotting.
u	Similarly to move up before plotting.
t	Transpose horizontal and vertical axis. (Option -x now applies to the vertical axis.)

Note: A legend indicating grid range is produced with a grid unless the -s option is present. If a specified lower limit exceeds the upper limit, the axis is reversed.

Command Example

The following is an example of plotting an x-y graph using the input vector **A**.

```
A = 0 0
    1 2
    3 4
    5 6
    7 9
   10 11
```

```
^graph A | tplot<CR>
```

Note: The output of **graph A** is a plot(5) format that requires the **tplot** command to draw on a display terminal.

The results of the drawing is shown in Figure 4-3.

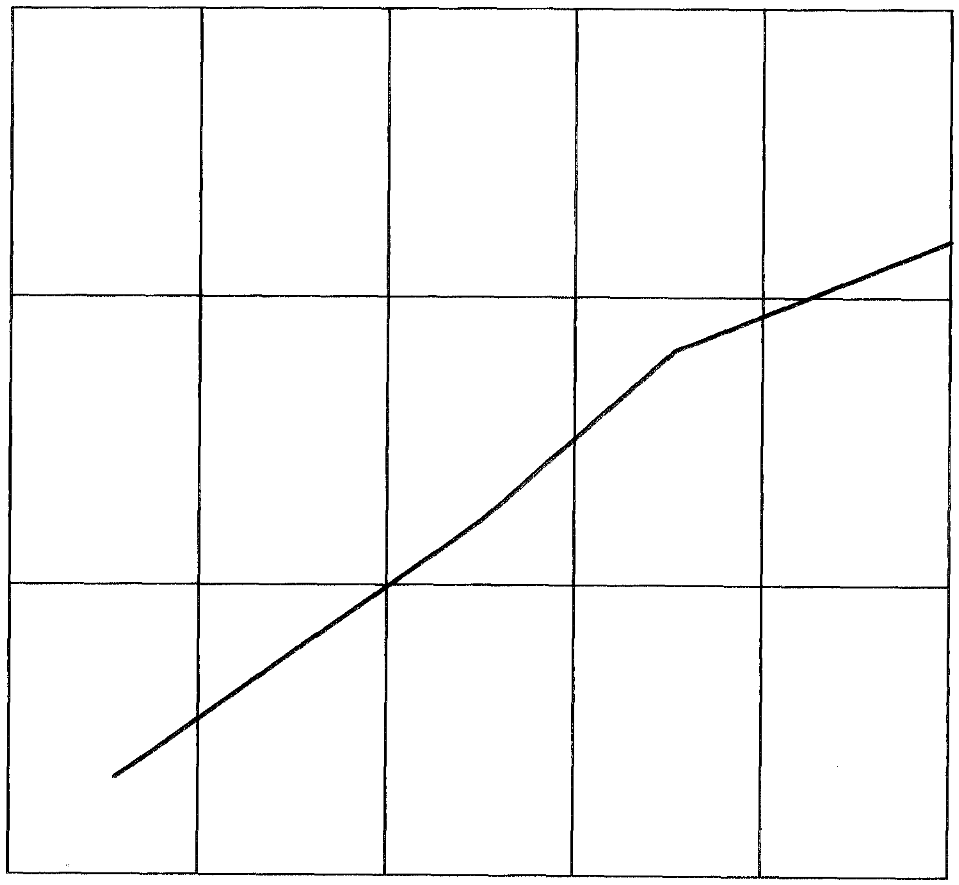


Figure 4-3. Plot of graph A | tplot

graphics — Access Graphical and Numerical Commands

General

The **graphics** command prefixes the path name /usr/bin/graf to the current $PATH value, changes the primary shell prompt to ^, and executes a new shell. The directory /usr/bin/graf contains all the graphics subsystem commands. To restore the environment that existed before issuing the graphics command, type EOT (*<CTRL-d>* control-d on most terminals). To log off from the graphics environment, type **quit**. If the -r option is given, access to the graphical commands is created in a restricted environment; that is, $PATH is set to :/usr/bin/graf:/rbin:/usr/rbin and the restricted shell, rsh, is invoked.

Command Format

```
$graphics<CR>
^
```

gtop — GPS to Plot(5) Format

General

The **gtop** command transforms a GPS into plot(5) format. The plot(5) format can be displayed on the 5620 DMD display terminal by using the **tplot** command. Input is taken from a file, if given; otherwise from the standard input. GPS objects are translated if they fall within the window that circumscribes the first file unless an option is given. Output is to the standard output.

Command Format

```
gtop [-option(s)] [GPS file(s)]
```

Options:

r*n*	Translate objects in GPS region *n*.
u	Translate all objects in the GPS universe.

Command Example

The following is an example showing the differences between displaying a drawing with a GPS and displaying a drawing with a plot(5) format.

With a GPS the command line is:

```
^gas | af " x^2"  | plot | td<CR>
```

With a plot(5) format the command line is:

```
^gas | af " x^2"  | plot | gtop | tplot<CR>
```

where

- The output of **plot** is a GPS.
- **td** command displays a GPS on a TEKTRONIX 4014.
- **gtop** transforms the GPS into plot(5) format.
- **tplot** command can display plot(5) format on such devices as a DASI 300, DASI 300s, DASI 450, TEKTRONIX 4014, Versatec* D12200A.

* Registered Trademark of Tektronix, Inc.

hardcopy — Sends Make Copy Character

General

When issued from a TEKTRONIX 4014 display terminal with a hard copy unit (printer), hardcopy generates a screen copy on the printer.

Command Format

```
^hardcopy
```

hilo — High and Low Values

General

The **hilo** command is a summarizer node. The output is the high and low values across all the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
hilo [-option(s)] [vector(s)]
```

Options:

h	Only output high value.
l	Only output low value.
o	Output high, low values in " option" form (see plot).
ox	Output high, low values with x prepended.
oy	Output high, low values with y prepended.

Command Example

The following is an example of finding the high and low values of vector **A**.

A = 10 20 30 40 50

```
^hilo A<CR>

low = 10    high = 50
```

hist — Build a Histogram

General

The **hist** command is a translator node that generates a GPS that describes a histogram. The input vector for the **hist** command must be made up of intervals and counts. If the input is not as intervals and counts, you must use the **bucket** command to put the input vector in that form. If no vector is given, then the standard input is assumed.

Command Format

```
hist [-option(s)] [vector(s)]
```

Options:

a	Suppress axes.
b	Plot histogram with bold weight lines, otherwise use medium.
f	Do not build a frame around the plot area.
g	Suppress background grid.
r*n*	Put the histogram in GPS region *n*, where *n* is between 1 and 25, inclusively.
x*n***(y***n***)**	Position the histogram in the GPS universe with x-orgin (y-orgin) at *n*.
xa (ya)	Do not label x-axis (y-axis).

yl*n*	*n* is the y-axis low-tick value.
yh*n*	*n* is the y-axis high-tick value.

Command Example

The following example will produce a histogram of the input vector **F**.

F = 1 1.2 1.3 2 3 7 9 22 64 70 70.4

```
^qsort F | bucket | hist | td<CR>
```

- **qsort F**, the vector F must be sorted for the bucket command.
- **bucket**, breaks the vector F into intervals and counts.
- The output of **hist** is a GPS that describes a histogram.
- **td** command displays drawings on TEKTRONIX 4014 terminal.

The results of the drawing is shown in Figure 4-4.

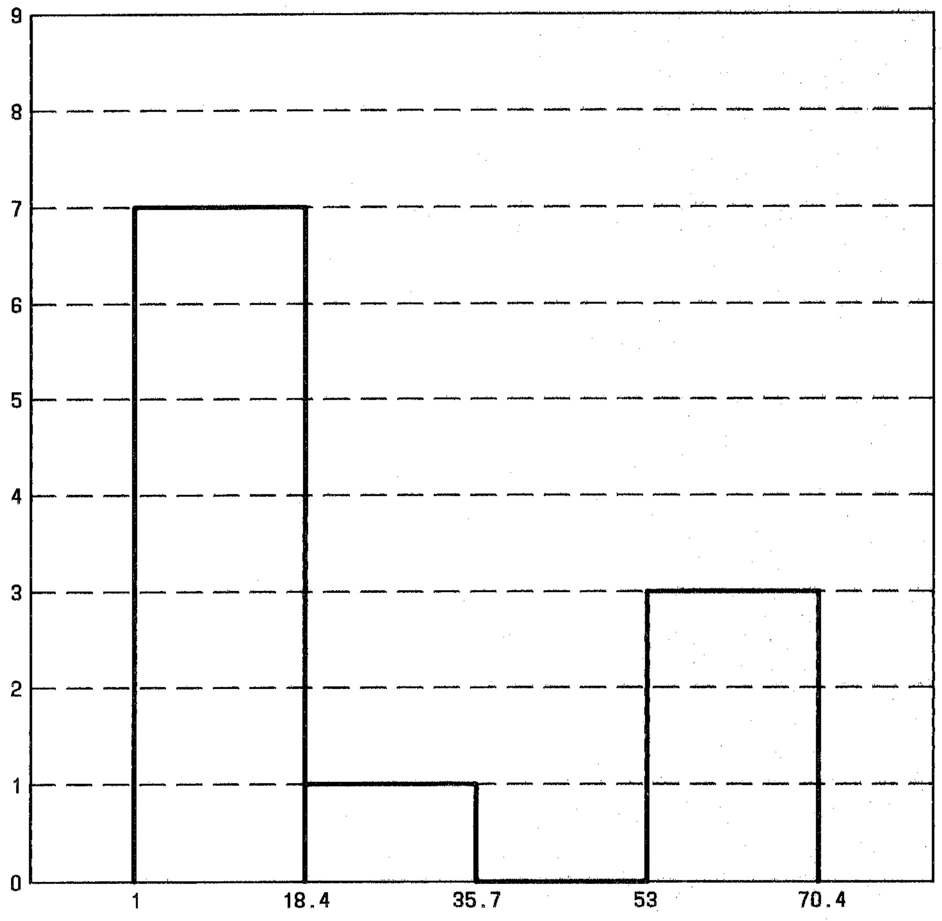


Figure 4-4. Plot of qsort F ¦ bucket ¦ hist ¦ td

hpd — Display GPS on a HP 7221A Graphics Plotter

General

The output is scope coded for a HEWLETT PACKARD 7221A Plotter. A viewing window is computed from the maximum and minimum points in the GPS file(s) unless the *r* or *u* option is provided. If no file is given, then the standard input is assumed.

Command Format

```
hpd [-option(s)] [GPSfile(s)]
```

Options:

c*n*	Select character set *n*, *n* between 0 and 5 (see the *HEWLETT PACKARD 7221A Plotter Operating and Programming Manual* and the *AT&T 3B2 Computer User Reference Manual*).
p*n*	Select pen numbered*n*, *n* between 1 and 4, inclusively.
r*n*	Window on GPS region *n*, *n* between 1 and 25, inclusively.
s*n*	Slant characters *n* degrees counterclockwise from the vertical.
u	Window on the entire GPS universe.
xd*n*	Set x displacement of the viewports lower left corner to *n* inches.

xv*n*	Set width of viewport to *n* inches.
yd*n*	Set y displacement of the viewports lower left corner to *n* inches.
yv*n*	Set height of viewport to *n* inches.

Command Example

The following is an example of displaying a drawing on a HEWLETT PACKARD 7221A Graphics Plotter.

```
^gas ¦ af " x^2" ¦ plot ¦ hpd<CR>
```

where the output of the **plot** command is a GPS that the **hpd** command displays that GPS on a HP 7221A Graphics Plotter.

label — Label the Axis of a Data Plot

General

The **label** command is a translator node that labels the axis of a data plot. The **label** command attaches the axis with a label by appending a label file to a GPS of a data plot (like that produced by **hist**, **bar**, and **plot**). Each line of the label file is taken as one label. Once the label has been appended to the GPS drawing, the label may not be in the appropriate position. The graphics editor (ged) can be used to position the label correctly. For plot labels, be sure to include *x i1* on the **plot** command line (see **plot**). Blank lines yield null labels. Either the GPS or the label file, but not both, may come from the standard input.

Command Format

label [-option(s)] *GPS file(s)*

Options:

b	Assume the input is a bar chart.
c	Retain lower case letters in labels; otherwise, all letters are upper case.
F*file*	*file* is the label file.
h	Assume the input is a histogram.
p	Assume the input is an x-y plot. This is the default.
r*n*	Labels are rotated *n* degrees. The pivot point is the first character.

x	Label the x-axis. This is the default.
xu	Label the upper x-axis (example, the top of the plot).
y	Label the y-axis.
yr	Label the right y-axis (example, the right side of the plot).

Command Example

The following is an example of appending a label to the y-axis of a GPS drawing:

1. Create your label file by using a text editor.

```
^vi lab<CR>
<a>Frequency of Random Numbers
<ESC>
<ZZ>
```

2. Create a GPS drawing and direct it to a file.

```
^rand -n1000 | qsort | bucket | hist >Randplot<CR>
```

3. Use the **label** command to append the label file to the GPS and display it on a TEKTRONIX 4014.

```
^label -Flab,h,r90,y Randplot ¦ td<CR>
```

See Figure 4-5 for a result of the drawing.

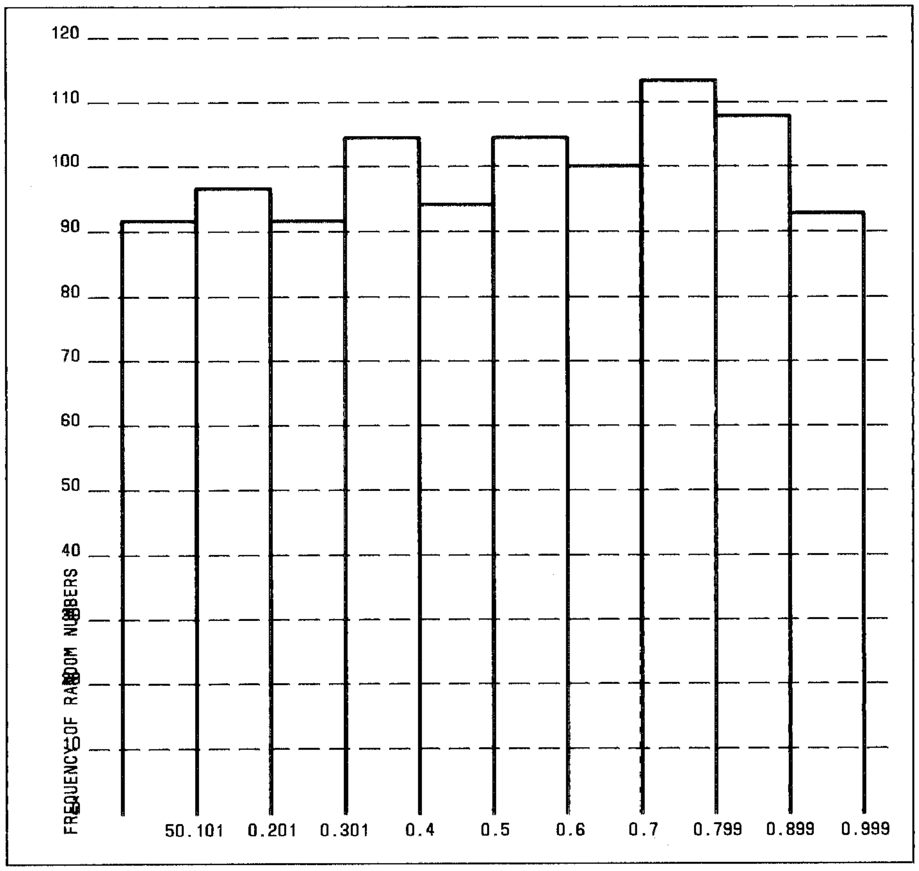


Figure 4-5. Plot of label -Flab,h,r90,y Randplot ¦ td

List — List Vector

General

The **list** command is a transformer node that list vectors. The output is a listing of the elements of the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
list [-option(s)] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line. Five is the default value.
d*string*	The characters in *string* serve as delimiters. Only elements that are delimited by these characters will be listed. The white space, characters space, tab, and newline are always delimiters.

Note: If **d** is not specified, then any character that is not part of a number is a delimiter. If **d** is specified, then the white space characters (space, tab, and new-line) plus the character(s) of string are delimiters. Only numbers surrounded by delimiters are listed.

Command Example

The following is an example of using the **list** command to display the **A** vector.

A = 10 20 30 40 50

```
^list A<CR>

10 20 30 40 50
```

log — Logarithm

General

The **log** command is a transformer node that takes the logarithm for each element of the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
log [-option(s)] [vector(s)]
```

Options:

b*n*	*n* is the logarithm base. If not given, 2.71828... is used.
c*n*	*n* is the number of output elements per line.

Command Example

An example of finding the logarithm function of the vector **A** follows:

A = 10 20 30 40 50

```
^log A<CR>
2.302  2.99  3.4  3.688  3.912
```

lreg — Linear Regression

General

The linear regression (**lreg**) is a summarizer node. The output is the slope and intercept from a least squares linear regression of each vector on a base vector. The base vector is specified using the **F** option. If the base is not given, it is assumed to be ascending positive integers from zero.

Command Format

```
lreg [-option(s)] [vector(s)]
```

F*vector*	*vector* is the base.
i	Only output the intercept.
o	Output the slope and intercept in " option" form (see siline).
s	Only output the slope.

Command Example

The following is an example of finding the linear regression of the vectors **A** and **C**; **A** is the base vector (x-axis) and **C** is the y-axis.

```
A = 10 20 30 40 50
C = 30 40 50 60 70
```

```
^lreg -FA,C<CR>

intercept=20   slope=1
```

mean — Mean

General

The **mean** command is a summarizer node. The output is the mean of the elements in the input vector(s). The input may optionally be trimmed. If no vector is given, then the standard input is assumed.

Command Format

```
mean [-option(s)] [vector(s)]
```

Options:

f*n*	Trim (1/n)*r elements from each end, where r is the rank of the input vector.
p*n*	Trim n*r elements from each end, n is between 0 and 0.5.
n*n*	Trim *n* elements from each end.

Command Example

The following is an example of finding the mean of the vector **A**.

A = 10 20 30 40 50

```
^mean A<CR>

30
```

mod — Modulo Function

General

The modulo function (**mod**)is a transformer node. The output is a vector with each element being the remainder of dividing the corresponding element from the input vector(s) by the modulus. If no vector is given, then the standard input is assumed.

Command Format

```
mod [-option(s)] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line.
m*n*	*n* is the modulus. If not given, 2 is used.

Command Example

The following is an example of finding the modulo function of vector **G**.

G = 1 2 3 4 5 6 7 8 9

```
^mod -m3 G<CR>

1 2 0 1 2 0 1 2 0
```

pair — Pair Element Group

General

The **pair** command is a transformer node. The output is a vector with elements taken alternately from a base vector and a vector. The base vector is specified either with the **F** option, or else it comes from the standard input. Vector(s) are specified either on the command line or else one may come from the standard input. If both the base and vector come from the standard input, base precedes vector.

Command Format

```
pair [-option(s)] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line.
F*vector*	*vector* is the base.
x*n*	*n* is the number of elements taken from the base for each one element taken from vector.

Command Example

The following is an example of finding the pair element group of vectors **A** and **C**.

```
A = 10 20 30 40 50
C = 30 40 50 60 70
```

```
^pair -FA,C<CR>

10 30 20 40 30
50 40 60 50 70
```

pd — Plot(5) Format Dump

General

The **pd** command prints a human readable listing of plot(5) format. If no file is given, then the standard input is assumed.

Command Format

```
pd [plot(5) file(s)]
```

Command Example

The following is an example of printing a human readable listing of a plot(5) format.

1. Create a plot(5) format and direct it to a file.

```
^gas ! af " x^2" ! plot ! gtop >Y
```

2. Print a readable listing of plot(5) format.

```
^pd Y<CR>
erase
space   13461  13351  19765  19655
jump    13461  13351
text                 ;
jump    14607  14607
cont    19160  14607
cont    19160  18660
cont    14607  18660
cont    14607  14607
jump    14607  14467
text                 0
jump    15062  14467
text                 1
jump    15517  14467

(All data not shown.)
```

pie — Build a Pie Chart

General

The **pie** command is a translator node. Its output is a GPS that describes a pie chart. The input is a text file that has the data form:

```
[<control>] value [label]
```

The control field specifies the way that slice should be handled. Legal values are:

i	The slice will not be drawn, although a space will be left for it.
e	The slice is " exploded," or removed from the pie.
f	The slice is filled. The angle of fill lines depends on the color of the slice.
c*color*	The slice is drawn in *color* rather than the default black. Legal values for *color* are 'b' for black, 'r' for red, 'g' for green, and 'u' for blue.

The pie is drawn with the value of each slice printed inside and the label printed outside. If no file is specified, the standard input is assumed.

Command Format

```
pie [-option(s)] [file(s)]
```

Options:

b	Draw pie chart in bold weight lines; otherwise, use medium
p	Output value as a percentage of the total pie
pp*n*	Only draw *n* percent of a pie
pn*n*	Output value as percentage, but total of percentages equals *n* rather than 100. pn100 is equivalent to p
v	Do not output the values
o	Output values around the outside of the pie
r*n*	Put the pie chart in region *n*, where *n* is between 1 and 25, inclusively
x*n* **(y***n***)**	Position the pie chart in the GPS universe with x-origin (y-origin) at *n*.

Command Example

The following is an example of creating a pie chart.

1. Create the input text file.

> ***Note:*** The following sample commands show the *i*, *e*, and *f* enclosed in brackets. These brackets and the corresponding control field must be entered by the user. This is an exception to the rule on "HOW COMMANDS ARE DESCRIBED"in Chapter 2.

```
^vi piedata<CR>
<a><i> 1 a<CR>
<e> 2 b<CR>
<f> 3 c<CR>
4<CR>
<ESC>
<ZZ>
```

2. Create the pie drawing and display it on a TEKTRONIX 4014.

```
^pie -p piedata | td<CR>
```

See Figure 4-6 for a result of the drawing.

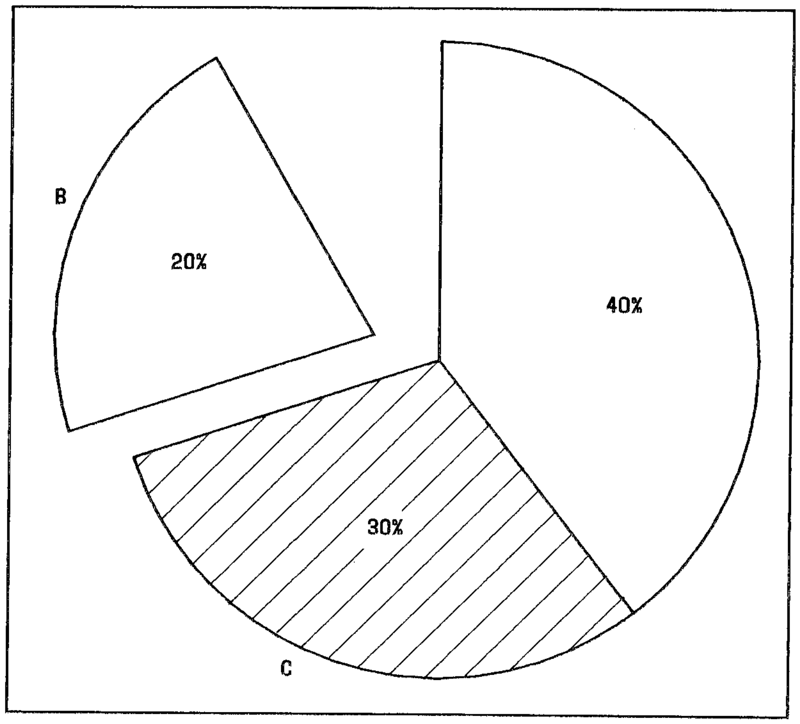


Figure 4-6. Plot of pie -p piedata | td

plot — Plot an X-Y Graph

General

The **plot** command is a translator node that describes an x-y graph. Input is one or more vector(s). Y-axis values come from vector(s), x-axis values from the **F** option. Axis scales are determined from the first vector(s) plotted. If no vector is given, then the standard input is assumed.

Command Format

```
plot [-option(s)] [vector(s)]
```

Options:

a	Suppress axis.
b	Plot graph with bold weight lines; otherwise, use medium.
c*char(s)*	Use *char(s)* for plotting characters, implies option m. The first character of *char(s)* is used to mark the first graph, the second is used to mark the second graph, etc.
d	Do not connect plotted points, implies option m.
F*vector*	Use *vector* for x-values, otherwise the positive integers are used.
g	Suppress background grid.

m	Mark the plotted points.
r*n*	Put the graph in GPS region *n*, where *n* is between 1 and 25, inclusively.
x*n* (**y***n*)	Position the graph in the GPS universe with x-origin (y-origin) at *n*.
xa (ya)	Omit x-axis (y-axis) labels.
xi*n* (**yi***n*)	*n* is the x-axis (y-axis) tick increment.
xl*n* (**yl***n*)	*n* is the x-axis (y-axis) low-tick value.
xh*n* (**yh***n*)	*n* is the x-axis (y-axis) high-tick value.
xn*n* (**yn***n*)	*n* is the approximate ticks on the x-axis (y-axis).
xt (yt)	Omit x-axis (y-axis) title.

Command Example

The following is an example of creating an x-y graph by using the **plot** command.

B = 1 2 3 4 5 6 7 8 9 10

```
^gas | af " x^2" | plot -F-,dg B | td<CR>
```

Note: The **plot -F-,dg B** uses the **B** vector as the y-axis and uses the standard input for the x-axis.

See Figure 4-7 for a result of the drawing.

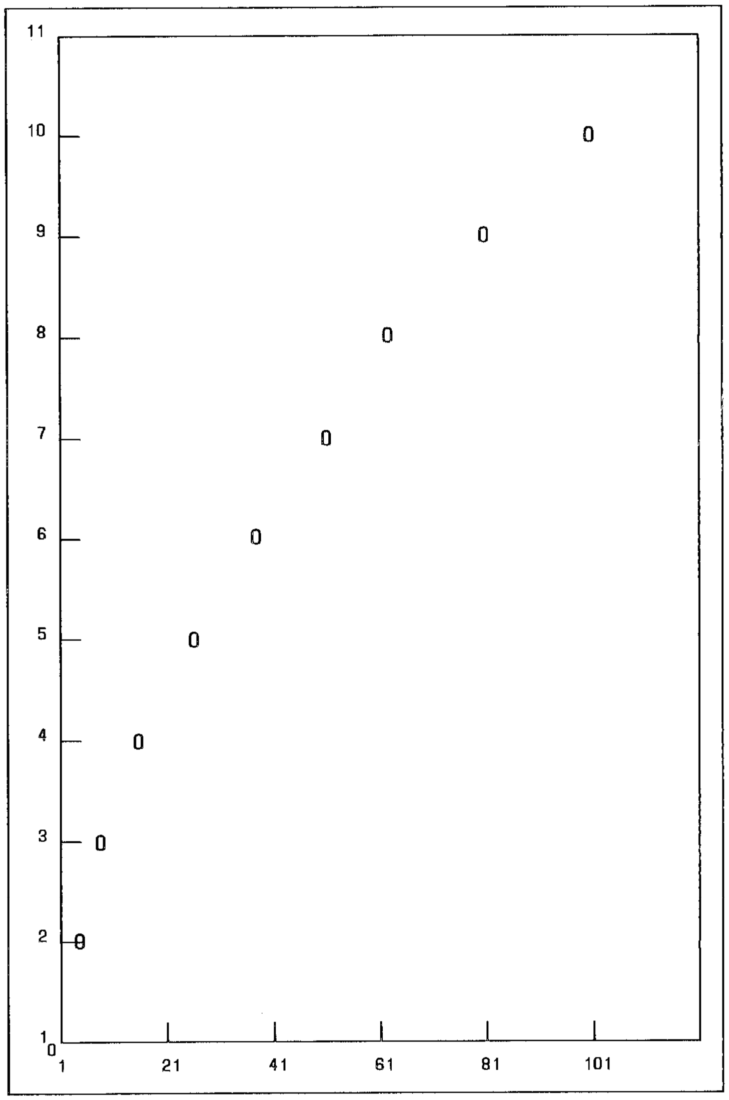


Figure 4-7. Plot of gas | af " x^2" | plot -F-,dg B | td

point — Empirical Cumulative Density Function Point

General

The **point** command is a summarizer node. The output is a linearly interpolated value from the empirical cumulative density function (e.c.d.f) for the input vector. By default, point returns the median (50 percent(%) point). If no vector is given, the standard input is assumed.

Command Format

```
point [-option(s)] [vector(s)]
```

Options:

f*n*	Return the (1/*n*)*100 percent point from the e.c.d.f.
p*n*	Return the *n* *100 percent point
n*n*	Return the *n*th element
s	The input is assumed to be sorted.

Command Example

The following is an example of finding the empirical cumulative density function point of vector **A**.

A = 10 20 30 40 50

```
^point -p.25 A<CR>

20
^point -p.50 A<CR>

30
^point -p.75 A<CR>

40
```

power — Power Function

General

The **power** function is a transformer node. The output is a vector with each element being a power of the corresponding element from the input vector(s). If no vector is given, the standard input is assumed.

Command Format

```
power [-option(s)] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line
p*n*	Input elements are raised to the *n*th power. If not given, 2 is used.

Command Example

The following is an example of finding the 4th power of the input vector **A**.

A = 10 20 30 40 50

```
^power -p4 A<CR>

10e+03  160e+03  810e+03 2.56e+06 6.25e+06
```

prime — Generate Prime Numbers

General

The **prime** command is a generator node that generates prime numbers. The output is a vector of number elements determined by the parameters low and high. The parameters are set by command options.

Command Format

```
prime [-option(s)]
```

Options:

c*n*	*n* elements per output line
h*n*	*n* = high
l*n*	*n* = low. If not given, low = 2.

Command Example

The following is an example of generating prime numbers from 5 through 20.

```
^prime -l5,h20<CR>
5 7 11 13 17 19
```

prod — Product

General

The **prod** command is a summarizer node that finds the product of each element in the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
prod [vector(s)]
```

Command Example

The following is an example of finding the product of the input vector **A**.

A = 10 20 30 40 50

```
^prod A<CR>

1.2e+07
```

ptog — Plot(5) Format to GPS Format

General

The **ptog** command transforms plot(5) format into a GPS. Input is taken from the file, if given; otherwise, it is taken from the standard input. Output is to the standard output.

Command Format

```
ptog [plot(5) file(s)]
```

Command Example

The following is an example of transforming a plot(5) format into a GPS.

1. Create a plot(5) format and direct it to a file.

```
^gas | af " x^2" | plot | gtop > B<CR>
```

2. Use **ptog** command to transform the plot(5) format to a GPS and display it on a TEKTRONIX 4014.

```
^ptog B | td<CR>
```

See Figure 4-8 for a result of the drawing.

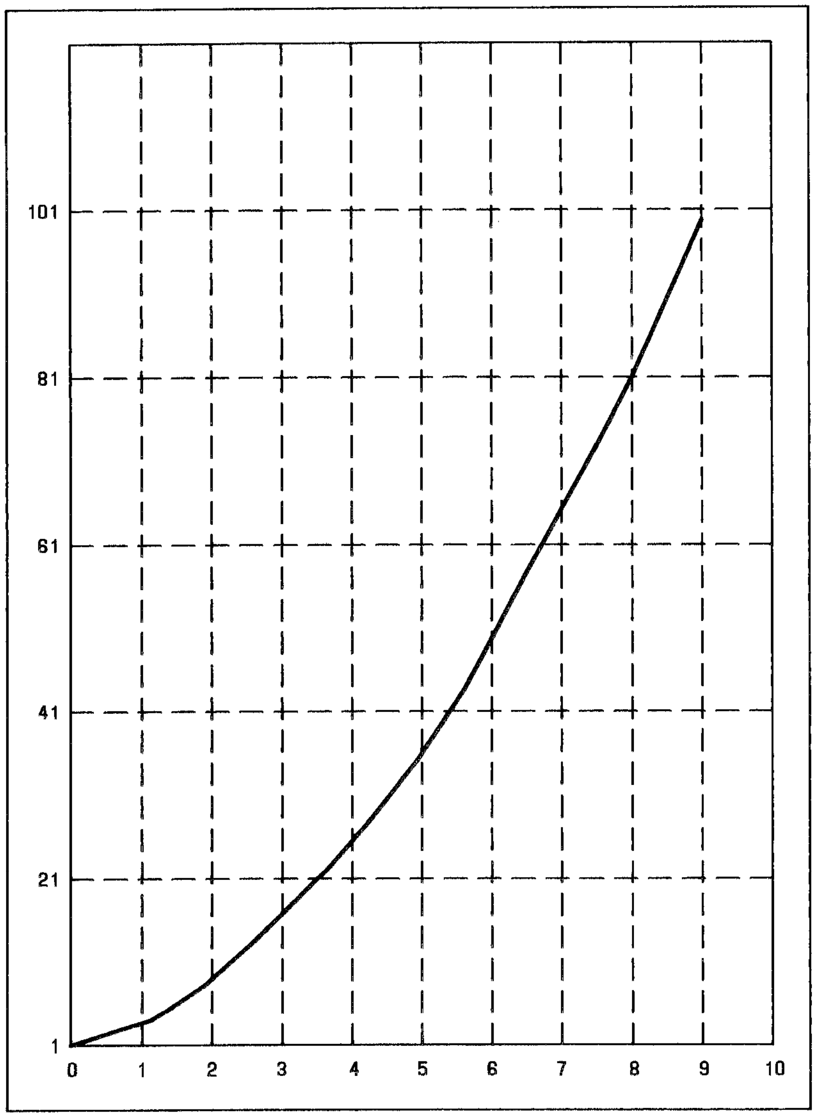


Figure 4-8. Plot of ptog B ¦ td

qsort — Quick Sort

General

The **qsort** command is a summarizer command that sorts the input vector in ascending order. If no vector is given, then the standard input is assumed.

Command Format

```
qsort [-option] [vector(s)]
```

Option:

c*n*	*n* is the number of output elements per line.

Command Example

The following is an example of sorting the vector **P**.

P = 60 50 40 30 20 10

```
^qsort P<CR>
10 20 30 40 50 60
```

quit — Terminate Session

General

The **quit** command terminates the current terminal session.

rand — Generate Random Sequence

General

The **rand** command is a generator node that generates random numbers. The output is a vector of number elements determined by the parameters low, high, multiplier, and seed. Random numbers are first generated in the range 0 to 1, initialized by the seed. Then if a multiplier is given, each number is multiplied accordingly. The parameters are set by command options.

Command Format

```
rand [-option(s)]
```

Options:

c*n*	*n* elements per output line
h*n*	*n* = high. If not given, high = 1
l*n*	*n* = low. If not given, low = 0
m*n*	*n* = multiplier. If not given, multiplier is determined from high and low
n*n*	*n* = number. If not given, number = 10
s*n*	*n* = seed. If not given, seed = 1.

Command Example

The following is an example of generating a vector of random numbers with the seed = 10 and high = 20.

```
^rand -s10,h20<CR>

2.77291 17.221  6.86361 5.41398 10.3073
1.36357 12.3063 16.8413 12.363  18.7445
```

rank — Rank of Vector

General

The **rank** command is a summarizer node that gives the number of elements in each input vector. If no vector is given, then the standard input is assumed.

Command Format

```
rank [vector(s)]
```

Command Example

The following is an example of finding the rank of vector **A**.

A = 10 20 30 40 50

```
^rank A<CR>

5
```

remcom — Remove Comments

General

The **remcom** command copies its input to its output with comments removed. Comments are as defined in C (such as, /* comment */). Input is from file(s), if given, otherwise it is from the standard input.

Command Format

```
remcom [file(s)]
```

Command Example

The following is an example of removing the comments from vector **G**.

```
G = 23  /*comments*/
   45  /*comments*/
```

```
^remcom G<CR>

23
45
```

root — Root Function

General

The **root** function is a transformer node. The output is a vector with each element being the root of the corresponding element from the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
root [-option(s)] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line
r*n*	*n* = root. If not given, root = 2.

Command Example

The following is an example of finding the square root of vector **A**.

A = 10 20 30 40 50

```
^root A<CR>
3.16778 4.47214 5.47723 6.32456 7.07107
```

round — Rounded Value

General

The **round** command is a transformer node. The output is the rounded value for each element of the input vector(s). If no vector is given, the standard input is assumed.

Command Format

```
round [-option(s)] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line
p*n*	*n* is the number of places following the decimal point rounded to the next number where *n* is in the range 0 to 9, 0 by default
s*n*	*n* is the number of significant digits rounded to the next number where *n* is in the range 0 to 9, 9 by default.

Command Example

An example of rounding each elements of the input vector **X**, follows:

X = 2.4 3.6 8.2

```
^round X<CR>

2 4 8
```

siline — Generate a Line Given Slope and Intercept

General

The **siline** command is a transformer node that generates a line from a given slope and intercept. The output is a vector of values slope times x plus intercept, where x takes on values from vector(s). If the *n* option is given, vector is the ascending positive integers. If neither the *n* option nor a vector is given, vector comes from the standard input.

Command Format

```
siline [-option(s)] [vector(s)]
```

Options:

c*n*	*n* is the number of output elements per line
i*n*	*n* is the intercept, 0 if not given
n*n*	*n* is the number of positive integers to be used for x
s*n*	*n* is the slope, 1 if not given.

Command Example

The following is an example of generating a line that has the slope of 2 and intercept of 1.

```
^siline -n10,s2,i1 | plot | td
```

- **siline -n10,s2,i1** generates the following data.

```
1   3    5    7    9
11  13   15   17   19
```

- **plot** generates a GPS of an x-y graph.
- **td** command displays a drawing on TEKTRONIX 4014 terminal.

See Figure 4-9 for a result of the drawing.

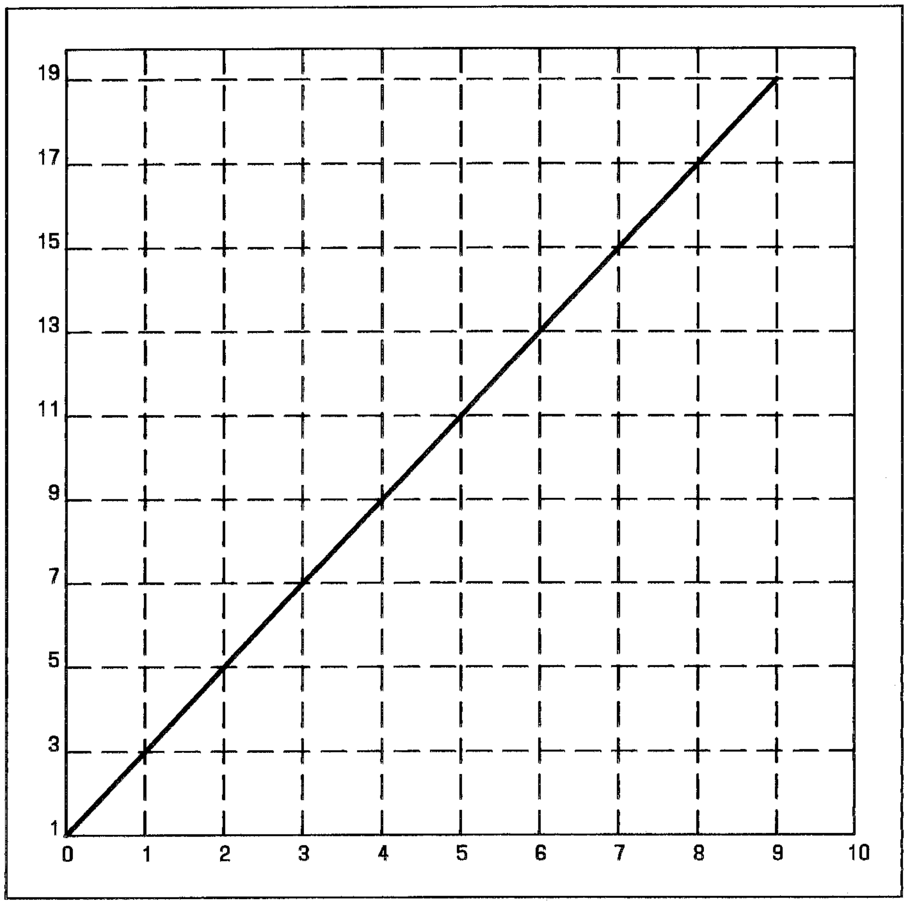


Figure 4-9. Plot of siline -n10,s2,i1 | plot ltd

sin — Sine Function

General

The **sin** command is a transformer node that takes the sine for each element of the input vector(s). Input is assumed to be in radians. If no vector is given, then the standard input is assumed.

Command Format

```
sin [-option] [vector(s)]
```

Option:

c*n*	*n* is the number of output elements per line.

Command Example

The following is an example of finding the sine of each element of the input vector **A**.

A = 10 20 30 40 50

```
^sin A<CR>
-.544071 .912945 -.988032 .745113 -.262375
```

spline — Interpolate Smooth Curve

General

The **spline** command takes pairs of numbers from the standard input as abscissas and ordinates of a function. It produces a similar set, which is almost equally spaced and includes the input set, on the standard output. The cubic spline output has two continuous derivatives, and has enough points to look smooth when plotted (for example, by **graph**).

Command Format

```
spline [options]
```

Options:

-a	Supplies abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.
-k	The constant k used in the boundary value computation: $y_0 = ky_1, y_n = ky_{n-1}$ is set by the next argument (default k = 0).
-n	Space output points so that n intervals occur almost between the lower and upper x limits (default n = 1000).
-p	Makes output periodic, such as, matching derivatives at the ends. First and last input values should normally agree.

-x	Next 1 (or 2) argument is lower (and upper) x limits. Normally, these limits are calculated from the data. Automatic abscissas start at the lower limit (default 0).

Command Example

The following is an example of using the **spline** command to produce an x-y graph from the **Z** vector.

```
Z = 1 2
   3 4
   5 6
   7 9
   10 11
```

```
^spline <Z | graph | ptog | td<CR>
```

Note: The **spline** command take the pairs of numbers from the input and generates points that will create a smooth curve when used with the **graph** command. Remember the **graph** command must use the **ptog** command to change the format from plot(5) to a GPS.

See Figure 4-10 for a result of the drawing.

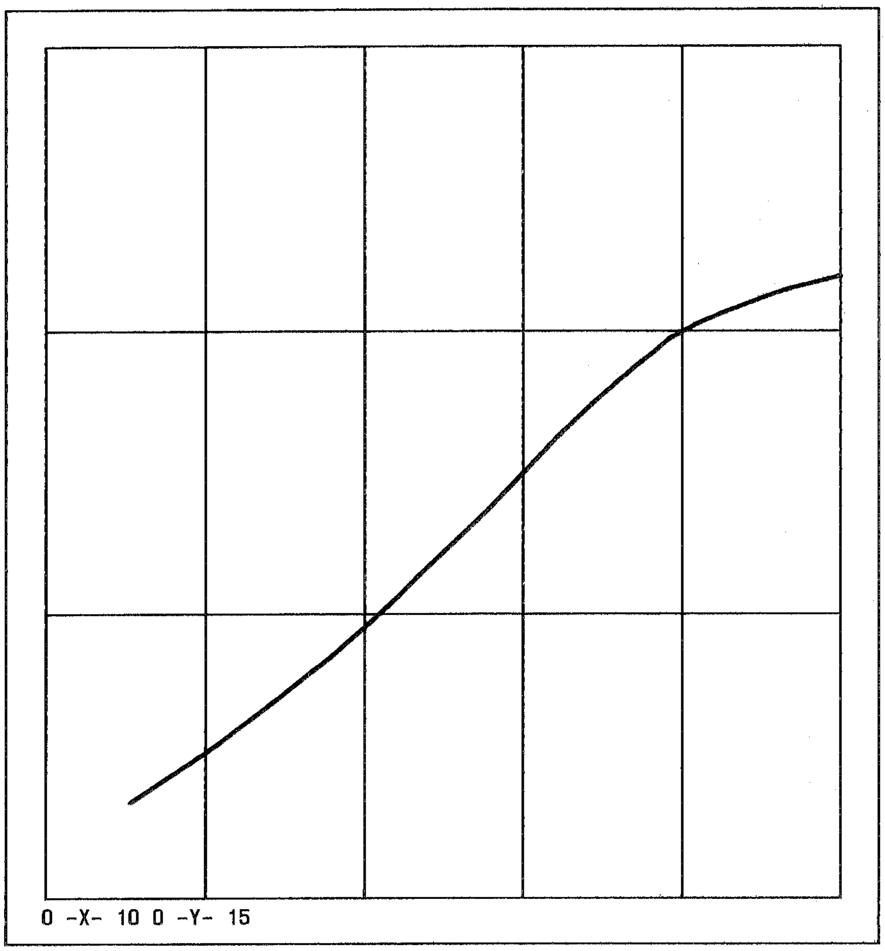


Figure 4-10. Plot of spline <Z ¦ graph ¦ ptog ¦ td

subset — Generate a Subset

General

The **subset** command is a transformer node that generates a subset. The output is elements selected from the input based on a key and option(s). If no vector is given, then the standard input is assumed.

Selection

If a master vector is given, then the key for the *i*th element of the input is the *i*th element of master, otherwise the key is the input element itself. In either case, *i* goes from start to terminate.

The input element is selected if the key is either above, below, or equal to pick, and not equal to leave. If neither above, below, nor pick is given, then the element is selected if it is not equal to leave.

Command Format

```
subset [-option(s)] [vector(s)]
```

Options:

a*n*	*n* = above
b*n*	*n* = below
c*n*	*n* elements per output line
F*vector*	*vector* is the master

i*n*	*n* = interval, default is 1
l*n*	*n* = leave
nl	Leave elements whose index is given in master
np	Pick elements whose index is given in master
p*n*	*n* = pick
s*n*	*n* = start, default is 1
t*n*	*n* = terminate, default is 32767.

Command Example

The following is an example of generating a **subset** from the master vector and another vector.

```
(master vector) xvector = 1  2  3  4  5  6  7  8  9
                yvector = 11 22 33 44 55 66 77 88 99
```

```
^subset -Fxvector,a5<CR>

66 77 88 99
```

td — Display GPS on a TEKTRONIX 4014

General

The **td** command displays the GPS on a TEKTRONIX 4014. The output is scope coded for a TEKTRONIX 4014 terminal. A viewing window is computed from the maximum and minimum points in the first file unless options are provided. If no file is given, then the standard input is assumed.

Command Format

```
td [-option(s)] [GPS file(s)]
```

Options:

r*n*	Window on GPS region *n*, *n* between 1 and 25, inclusively
u	Window on the entire GPS universe
e	Do not erase screen before initiating display.

tekset — Send Reset Character for TEKTRONIX 4014 Display Terminal

General

The **tekset** command clears the display screen, sets the display mode to alpha, and the characters to the smallest font.

Command Format

```
^tekset<CR>
```

title — Title a Vector or GPS

General

The **title** command is a translator node used to title a vector or a GPS. Input is taken from the file(s), if given, otherwise it is from the standard input.

Command Format

```
title [-option(s)] [file(s)]
```

Options:

b	Make the GPS title bold.
c	Retain lower case letters in title; otherwise, all letters are upper case.
l*string*	For a GPS, generate a lower title = string.
u*string*	For a GPS, generate an upper title = string.
v*string*	For a vector, title = string.

Command Example

The following is an example of titling a GPS.

1. Create a file containing a GPS.

```
^rand -n1000 | qsort | bucket | hist >Randplot<CR>
```

2. Title the GPS file (**Randplot**).

```
^title -l" lower title" ,u" upper title"  Randplot | td<CR>
```

See Figure 4-11 for a result of the drawing.

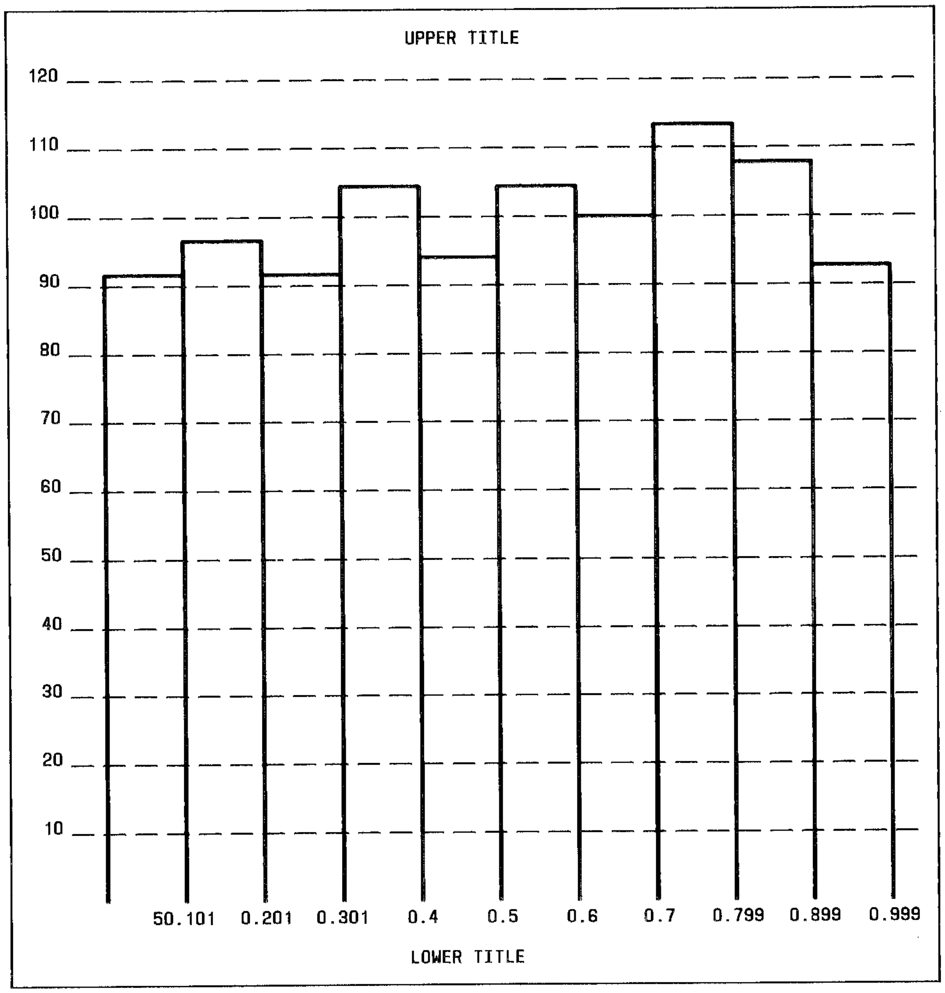


Figure 4-11. Plot of title -l" lower title" ,u" upper title" Randplot | td

total — Sum Total

General

The **total** command is a summarizer node. The output is the sum total of the elements in the input vector(s). If no vector is given, the standard input is assumed.

Command Format

```
total [vector(s)]
```

Command Example

The following is an example of finding the sum total of all the elements in vector **A**.

A = 10 20 30 40 50

```
^total A<CR>
150
```

tplot — Graphics Filter

General

The **tplot** command reads plotting instructions, plot(5), from the standard input; and in general, produce plotting instructions suitable for a particular terminal on the standard output. If no terminal is specified, the environment parameter **$TERM** is used. Known terminals are:

300	DASI 300
300S	DASI 300s
450	DASI 450.
4014	TEKTRONIX 4014
ver	Versatec D1200A. (This version of the plot places a scan-converted image in /usr/tmp/raster and sends the result directly to the plotter device, rather than to the standard output. The -e option causes a previously scan-converted file raster to be sent to the plotter.)

Command Format

```
tplot [-T terminal [-e raster] ]
```

Command Example

(See the **graph** example.)

ttoc — Make Textual Table of Contents

General

The output is the Textual Table of Contents (TTOC) generated by the .H macro of the nroff or troff raw data file of Document Workbench. If no file is given, then the standard input is assumed.

Command Format

```
ttoc [mm(1) file]
```

Command Example

The following is an example of outputting a Textual Table of Contents (**TTOC**).

1. Create a heading with text.

```
^vi txt<CR>
^.H 1 " example one" <CR>
 A line of text<CR>
.H 2 " example two" <CR>
 A second line of text<CR>
.H 3 " example three" <CR>
A third line of text<CR>
.H 3 " example four" <CR>
A fourth line of text<CR>
.H 2 " example five<CR>
A fifth line of text<CR>
.H 3 " example six<CR>
A sixth line of text<CR>
.H 1 " example seven<CR>
end
```

2. List a Textual Table of Contents (**TTOC**).

```
^ttoc txt<CR>

0.   " Table of Contents"
1.   " example one"  0
1.1  " example two"  0
1.1.1" example three"  0
1.1.2" example four"  0
1.2  " example five"  0
1.2.1" example six"  0
2.   " example seven"  0
```

3. Display the Textual Table of Contents (**TTOC**) command on the TEKTRONIX 4014.

```
^ttoc txt ! vtoc ! td<CR>
```

See Figure 4-12 for a result of the drawing.

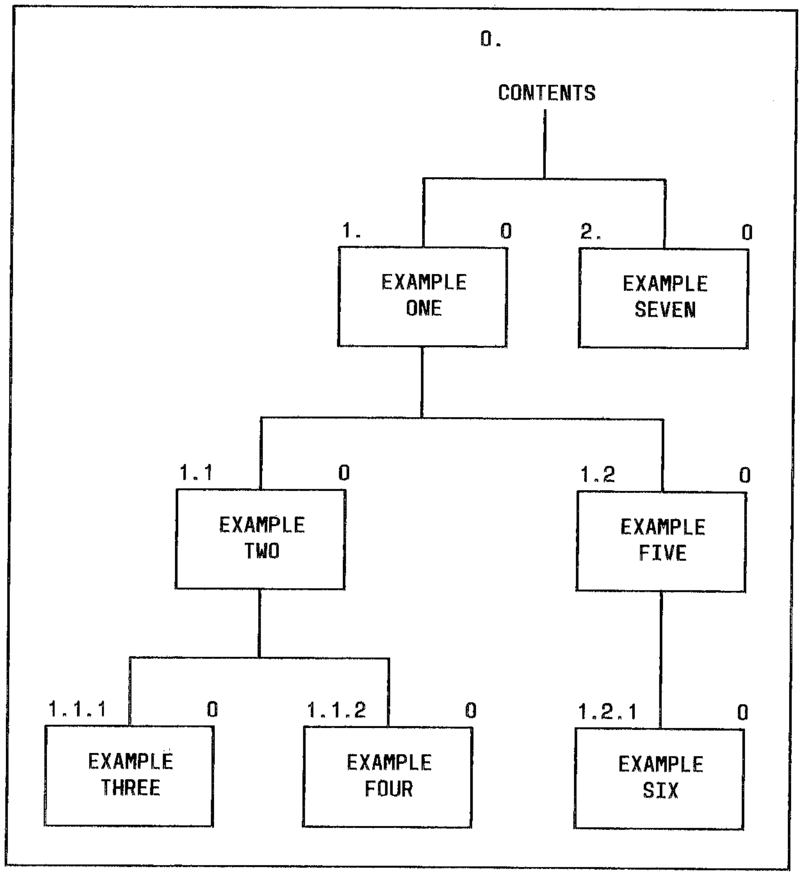


Figure 4-12. Plot of ttoc txt ¦ vtoc ¦ td

var — Variance

General

The **var** command is a summarizer node that finds the difference between the slope point and outer point. The output is the variance of the elements in the input vector(s). If no vector is given, then the standard input is assumed.

Command Format

```
var [vector(s)]
```

Command Example

The following is an example of finding the variance of the input vector **A**.

A = 10 20 30 40 50

```
^var A<CR>
250
```

vtoc — Visual Table of Contents

General

The output is a GPS that describes a Visual Table of Contents (**vtoc** or hierarchical chart) of the Textual Table of Contents (**TTOC**) entries from the input. If no file is given, then the standard input is assumed. **TTOC** entries have the form:

```
id [line weight,line style] " text"  [mark]
```

where

id	Is an alternating sequence of numbers and dots.
line weight	Is either **n** for narrow, **m** for medium, or **b** for bold.
line style	Is either **so** for solid, **do** for dotted, **dd** for dot-dashed, or **ld** for long-dashed.
text	Is a string of characters surrounded by quotes.
mark	Is a string of characters (surrounded by quotes if it contains spaces), with included dots being escaped.

Command Format

```
vtoc [-option(s)] [TTOC file]
```

Options:

c	Take the ext as entered (default is all upper case).
d	Connect the boxes with diagonal lines.
h*n*	Horizontal interbox space is *n*% of box width.
i	Suppress the box id.
m	Suppress the box mark.
s	Do not compact boxes horizontally.
v*n*	Vertical interbox space is *n*% of box height.

Command Example

(See the **TTOC** example.)

whatis — Brief Online Documentation

General

The **whatis** command prints a brief description of each command given. If no command is given, then the current list of the description command is printed. The **whatis** command prints out every description.

Command Format

```
whatis [-option] [name(s)]
```

Option:

o	Just print command options.

Command Example

The following is an example of using the **whatis** command.

```
^whatis bel<CR>

bel; send bel character to terminal

_B_e_l causes most terminals to sound an audible
tone, a useful nonvisual signal.
```

yoo — Pipe Fitting

General

The **yoo** command is a piping primitive that deposits the output of a pipeline into a file used in the same pipeline. Note that without **yoo**, this is not usually successful as it causes a read and write on the same file, simultaneously.

Command Format

```
yoo file
```

Command Example

The following is an example of using the **yoo** command.

```
^af " x^2"  ¦ plot ¦ yoo x<CR>
```

Chapter 5

GRAPHICS EDITOR

Chapter 5

GRAPHICS EDITOR

INTRODUCTION

The graphics editor (**ged**), is an interactive graphical editor used to display, edit, and build drawings on a TEKTRONIX 4014 display terminal. The drawings are represented as a sequence of objects in a token language known as a GPS (graphical primitive string). A GPS is produced by the drawing commands in the UNIX System Graphics such as **vtoc** and **plot**, as well as by **ged** itself.

Drawings are built from objects consisting of lines, arcs, and text. Using the editor, the objects can be viewed at various magnifications and from various locations. Objects can be created, deleted, moved, copied, rotated, scaled, and modified.

The examples in this chapter will illustrate how to build and edit simple drawings. Try them to become familiar with how the editor works, but keep in mind that **ged** is intended primarily to edit the output of other programs rather than to build drawings from scratch.

GETTING STARTED

To enter the graphics editor (**ged**), enter the following command while in the graphics shell:

```
^ged<CR>
```

After a moment the screen should be clearing except for the **ged** prompt, *, in the upper left corner. The * shows that **ged** is ready to accept a command.

```
*
```

Each command passes through a sequence of stages during which you describe what the command is to do. All commands pass through a subset of these stages:

1. Command line
2. Text
3. Points
4. Pivot
5. Destination.

As a rule, each stage is stopped by typing <*CR*>. The <*CR*> for the last stage of a command triggers execution.

COMMAND FORMAT

The simplest commands consist only of a *command line*. The command line is modeled after a conventional command line in the shell.

```
command name [–option(s)] filename
```

GRAPHICS EDITOR COMMAND DESCRIPTION

The graphics editor will echo the full name of all commands and wait for the rest of the command line. For example, **e** references the **e**rase command. As **e**rase consists only of stage1, typing *<CR>* causes the **e**rase command to clear the display screen,

```
*erase<CR>
```

bringing the editor back to the **ged** prompt, *.

Following the command name, *options* may be entered. Options control such things as the width and style of lines to be drawn or the size and orientation of the text. Most options have a default value that applies if a value for the option is not specified on the command line. The **s**et command allows examination and change of the default values. To see the current default values, type:

```
*set<CR>
```

The option value is one of three types: integer, character, or Boolean. Boolean values are represented by a + (for true) and a – (for false). A default value is modified by providing it as an option to the **s**et command. For example, to change the default text height to 300 units, type:

```
*set –h300<CR>
```

The following list will name each of the options default values. A complete description of each default option is discussed in the *Command Summary* section located at the rear of this chapter.

- **a** - angle
- **f** - factor
- **h** - height
- **s** - styletype
- **w** - withtype
- **e** - echo
- **k** - kopy
- **m** - midpoint
- **o** - out
- **p** - points
- **r** - rightpoint

- **t** - text
- **x** - x.

A question mark (**?**) is a command used to list the commands and options understood by **ged**. To generate the list, type the following:

```
*?<CR>
```

The delete key (**del**) on the 5620 DMD is used to abort a command. This is done by depressing the key after the command and before the carriage return *<CR>*. The following is an example of using this command:

```
*?<del>
```

Arguments on the command line, but not the command name, may be edited using the erase and kill characters from the shell. This applies whenever text is being entered.

CONSTRUCTING GRAPHICAL OBJECTS

Drawings are stored as a GPS in a display buffer internal to the editor. Typically, a drawing in **ged** is composed of instances of three graphical primitives: *arcs*, *lines*, and *text*.

GENERATING TEXT

To put a line of text on the display screen, use the **T**ext command.

First enter the *command line* (stage 1):

```
*Text<CR>
```

Next enter the text (stage 2):

```
a line of text<CR>
```

Next place the graphics cursor at the desired position on the screen. The graphics cursor is the point at which the lines intercept on the screen. It can be moved by using the mouse on the DMD.

```
<position cursor><CR>
```

Positioning of the graphic cursor is done either with the thumbwheel knobs on the TEKTRONIX 4014 display terminal keyboard or with the mouse on the 5620 DMD. The *<CR>* establishes the location of the cursor to be the starting point for the text string. The **T**ext command ends at stage 3, so this *<CR>* shows the drawing of the text string.

The **Text** command accepts options to vary the angle, height, and line width of the characters, and to either center or right justify the text object. The text string may span more than one line by escaping the <*CR*> (i.e., \<*CR*>) to show continuation. To illustrate some of these capabilities, try the following:

```
*Text -r<CR>      (right justify text)
top\<CR>
right<CR>
<position cursor><CR>
*Text -a90<CR>  (rotate text 90 degrees)
lower\<CR>
left<CR>
<position cursor><CR>        (pick a point below and left of
                         the previous point)
```

Results of these commands are shown in Figure 5-1.

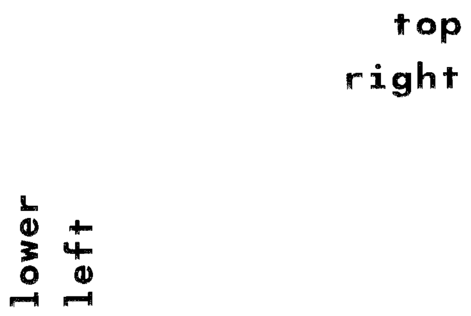


Figure 5-1. Generating Text Objects

DRAWING LINES

The **L**ines command is used to make objects built from a sequence of straight lines. It consists of stages 1 and 3. Stage 1 is straightforward:

```
*Lines [options]<CR>
```

The **L**ines command accepts options to specify line style and line width.

Stage 3, the entering of *points*, is more interesting. *Points* are referenced either with the graphic cursor or by name. We have already entered a point with the cursor for the **T**ext command. For the **L**ines command, it is more of the same. As an example, to build a triangle, type:

```
*Lines<CR>
<position cursor><SP>  (locate the first point)
<position cursor><SP>  (the second point)
<position cursor><SP>  (the third point)
<position cursor><SP>  (back to the first point)
<CR>            (end points, draw triangle)
```

Results of these commands are shown in Figure 5-2.

Typing <**SP**> enters the location of the crosshairs as a point. **Ged** identifies the point with an integer and adds the location to the current *point set*. The last point entered can be erased by typing #. The current point set can be cleared by typing @. On receiving the final <*CR*>, the points are connected in numerical order.

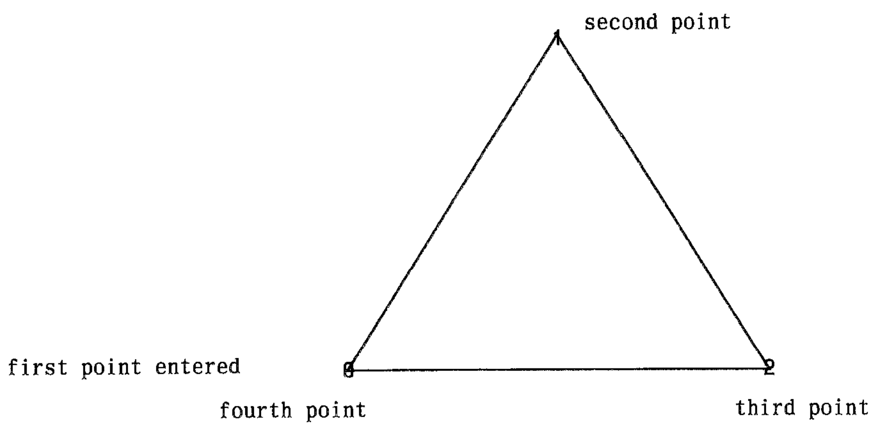


Figure 5-2. Building a Triangle

ACCESSING POINTS BY NAME

The points in the current point set may be referenced by name using the **$** operator. For instance, **$***n* references the point numbered *n*. By using **$**, the triangle above can be redrawn by entering:

```
*Lines<CR>
<position cursor><SP>
<position cursor><SP>
<position cursor><SP>
$0<CR>          (reference point 0)
<CR>
```

At the start of each command that includes stage 3, *points*, the current point set is empty. The point set from the previous command is saved and is accessible using the **.** operator. The **.** swaps the points in the previous point set with those in the current set. The = operator can be used to identify the current points. To illustrate, use the triangle just entered as the basis for drawing a quadrilateral:

```
*Lines<CR>
.      (access the previous set)
=        (identify the current points)
#        (erase the last point)
<position cursor><SP>  (add a new point)
$0<CR>      (close the figure)
<CR>
```

Results of these commands are shown in Figure 5-3.

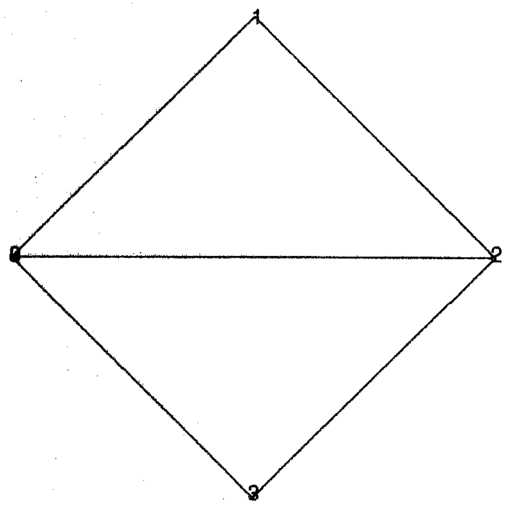


Figure 5-3. Accessing the Previous Point Set

Individual points from the previous point set can be referenced by using the **.** operator with **$**. The following example builds a triangle that shares an edge with the quadrilateral:

```
*Lines<CR>
$.1<CR>   (reference point 1 from the previous point set)
$.2<CR>   (reference point 2)
<position cursor><SP> (enter a new point)
$0<CR>    (or $.1, to close the figure)
<CR>
```

Results of these points are shown in Figure 5-4.

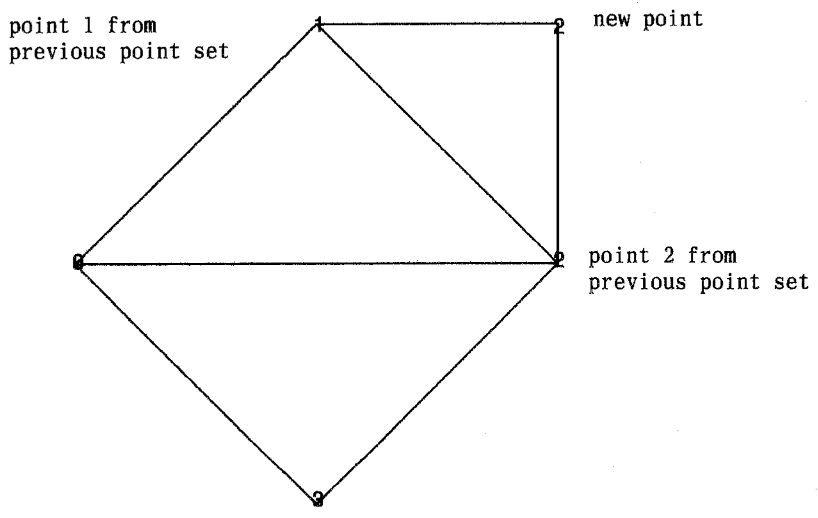


Figure 5-4. Referencing Points from Previous Point Set

A point can also be given a name. The > operator permits an upper case letter to be associated with a point just entered. A simple example is:

```
*Lines<CR>
<position cursor><SP>    (enter a point)
>A<CR>       (name the point A)
<position cursor><SP>
<CR>
```

In commands that follow, point **A** can be referenced using the **$** operator, as in:

```
*Lines<CR>
$A<CR>
<position cursor><SP>
<CR>
```

DRAWING CURVES

Curves are interpolated from a sequence of three or more points. The **Arc** command generates a circular arc given three points on a circle. The arc is drawn starting at the first point, through the second point, and ending at the third point. A circle is an arc with the first and third points coincidentally touching. Thus, one way to draw a circle is:

```
*Arc<CR>
<position cursor><SP>
<position cursor><SP>
$0<CR>
<CR>
```

Also, a circle can be generated by using the **Circ**le command. A simple example is:

```
*Circle<CR>
<position cursor><SP>    (specify the center)
<position cursor><CR>    (specify a point on the circle)
```

EDITING OBJECTS

Addressing Objects

An object is addressed by pointing to one of its *handles.* All objects have an *object-handle.* Usually the object-handle is the first point entered when the object was created. The **o**bject command marks the location of each object-handle with an **O**. For example, to see the handles of all the objects on the screen, type:

```
*objects -v<CR>
```

Some objects, **L**ines for example, also have *point-handles.* Typically, each of the points entered when an object is constructed becomes a point-handle. (An object-handle is also a point-handle.) The **p**oints command marks each of the point-handles.

A handle is pointed to by including it within a *defined area.* A defined area is generated either with a command line option or interactively using the graphic cursor. As an example, to delete one object that was created on the screen, type:

```
*Delete<CR>
<position cursor><SP>  (above and to the left of some
                        object-handle)
<position cursor><SP>  (below and to the right of the
                        object-handle)
<CR>                    (the defined area should include the
                        object-handle)
<CR>                    (if all is well, delete the object)
```

The defined area is outlined with dotted lines. The reason for the seemingly extra *<CR>* at the end of the **D**elete command is to provide an opportunity to stop the command (using *<del>* key) if the defined area is not right. Every command that accepts a defined area will wait for a confirming *<CR>*. The **n**ew command can be used to get a fresh copy of the remaining objects.

Defined areas are entered as **p**oints in the same way that objects are created. Actually, a defined area may be generated by giving anywhere from 0 to 30 points. Inputting zero points is particularly useful to point to a single handle. It creates a small defined area which is about the location of the terminating *<CR>*. Using a zero point defined area, the **D**elete command would be:

```
*Delete<CR>
<position cursor>    (center crosshairs on the object-handle)
<CR>              (end the defined area)
<CR>              (delete the object)
```

A defined area can also be given as a command line option. For example, to delete everything in the display buffer gives the **u**niverse option (**u**) to the **D**elete command. Note the difference between the commands **D**elete **–u**niverse and **e**rase. The **u**niverse option deletes all points in the buffer. The **e**rase command clears the screen.

Changing the Location of an Object

Objects are moved, using the **M**ove command. Create a circle using **A**rc, then move it as follows:

```
*Move<CR>
<position cursor><CR> (centered on the object-handle)
<CR>                (this establishes a pivot, marked with
                      an asterisk)
<position cursor><CR> (this establishes a destination)
```

The basic move operation relocates every point in each object within the defined area by the distance from the *pivot* to the *destination*. Here, the pivot was chosen to be the object-handle. So effectively, the object-handle was moved to the destination point.

Changing the Shape of an Object

The **B**ox command is a special case of generating lines. Given two points, it creates a rectangle such that the two points are at opposite corners. The sides of the rectangle lie parallel to the edges of the screen. To draw a box, type:

```
*Box<CR>
<position cursor><SP>
<position cursor><CR>
```

The **B**ox command generates point-handles at each vertex of the rectangle. Use the **p**oints command to mark the point-handles. The shape of an object can be altered by moving point-handles. The next example illustrates one way to double the height of a box (see Figure 5-5.)

```
*Move -p+<CR>
<position cursor><SP>   (left of the box, between the
                        top and bottom edges)
<position cursor><CR>   (right of the box, below the
                        bottom edge)
<position cursor><CR>   (on the top edge)
<position cursor><CR>   (directly below on the bottom
                        edge)
```

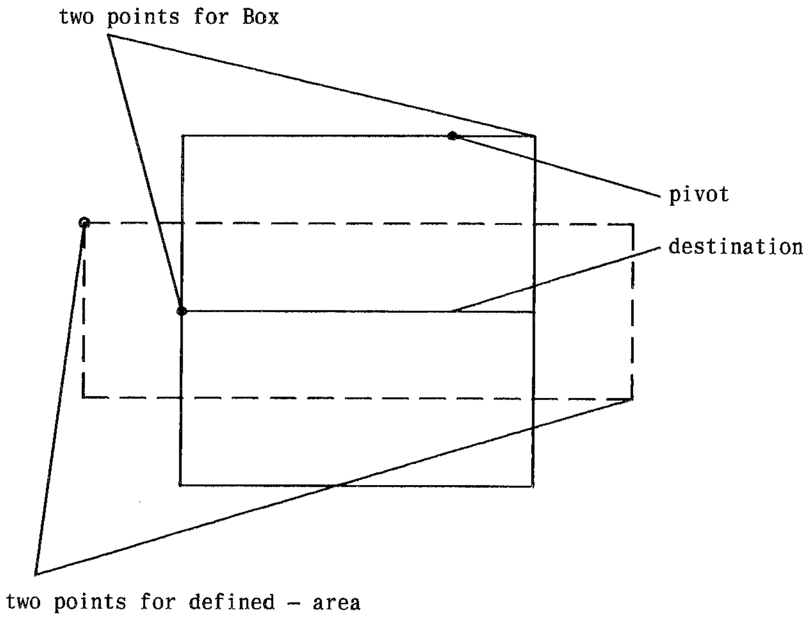


Figure 5-5. Growing a Box

When the **p**oints flag (**p**) is true, operations are applied to each point-handle addressed. In this example, the **p**oints flag was set to true using the command-line option **–p**+ causing each point-handle within the defined area to be moved the distance from the pivot to the destination. If **p** was false, only the object-handle would have been addressed.

Changing the Size of an Object

The size of an object can be changed using the **S**cale command. The **S**cale command scales objects by changing the distance from each handle of the object to the pivot by a factor. Put a line of text on the screen and try the following **S**cale commands (Figure 5-6):

```
*Scale –f200<CR>        (factor is in percent)
<position cursor><CR>   (point to object-handle)
<position cursor><CR>   (set pivot to rightmost character)
<CR>

*Scale –f50<CR>
.<CR>                   (reference the previous defined area)
<position cursor><CR>   (set pivot above a character
                        near the middle)
<CR>
```

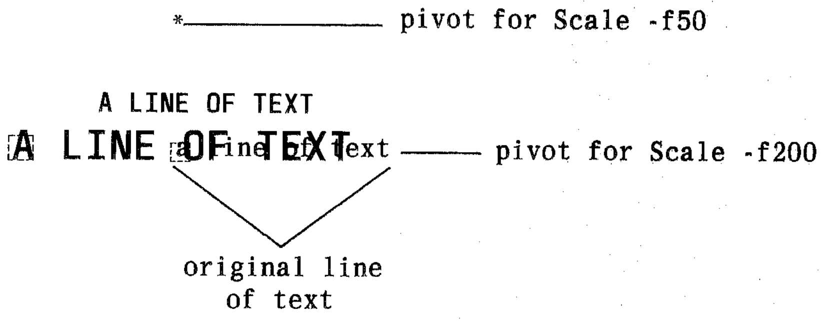


Figure 5-6. Scaling Text

A useful insight into the behavior of scaling is to note that the position of the pivot does not change. Also observe that the defined area is scaled to preserve its relationship to the graphical objects.

The size of objects can also be changed by moving point-handles. Generate a circle, this time using the **C**ircle command:

```
*Circle<CR>
<position cursor><SP>   (specify the center)
<position cursor><CR>   (specify a point on the circle)
```

The **C**ircle command generates an arc with the first and third point at the point specified on the circle. The second point of the arc is located 180 degrees around the circle. One way to change the size of the circle is to move one point-handle (using **M**ove **–p**+).

The following is an example of using **M**ove **–p**+:

1. Create a circle.

```
*Circle<CR>
<position cursor><SP>   (Point A Figure 5-7.)
<position cursor><CR>   (Point B Figure 5-7.)
```

2. Use the **p**oint command to mark the point-handles.

```
*point<CR>
<position cursor><CR>   (Point B Figure 5-7.)
<CR>
```

3. Move the circle by using the -p+ option.

```
*Move -p+<CR>
<position cursor><CR>   (point-handle, Point B)
<CR>   (pivot point B, Figure 5-7)
<position cursor><CR>   (Point C Figure 5-7.)
```

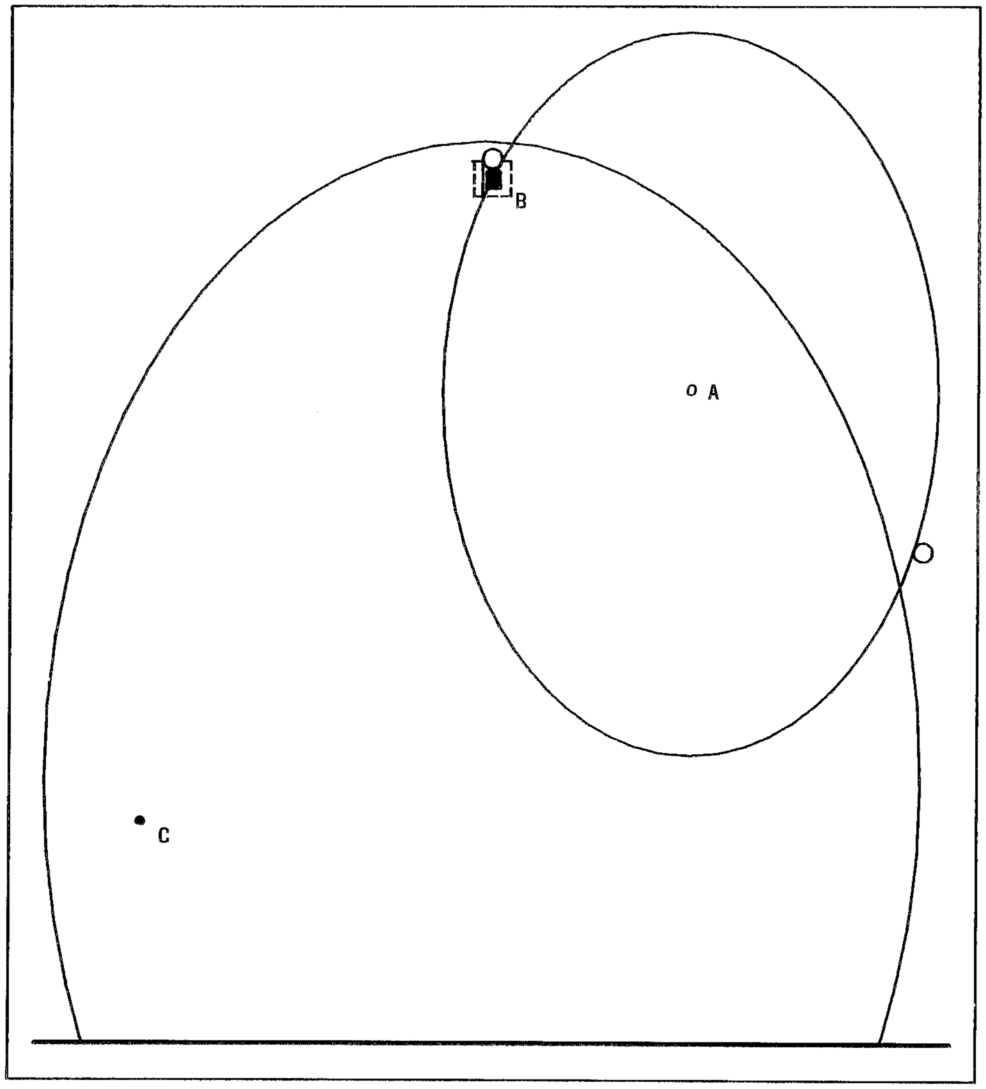


Figure 5-7. Example of Moving a Circle Using the Move –p+

The size of text characters can be changed via a third mechanism. Character height is the property of a line of text. The **Edit** command allows you to change the character height, shown in the following example:

1. Create a line of text.

```
*Text<CR>
A line of text.
<position cursor><CR>
```

   See Figure 5-8 (small print) for a result of the drawing.

2. Use the **Edit** command to enlarge the text.

```
*Edit –h1000<CR>   (increase height to 1000.)
<position cursor><CR>   (point to the object-handle point A)
<CR>
```

See Figure 5-8 (large print) for a result of the drawing.

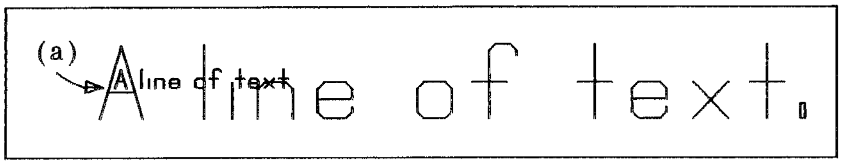


Figure 5-8. Example of Edit -h1000

Changing the Orientation of an Object

The orientation of an object can be altered using the **R**otate command. The **R**otate command rotates each point of an object about a pivot by an angle. Try the following rotations on a line of text (Figure 5-9).

```
*Rotate -a90<CR>  (angle is in degrees)
<position cursor><CR>  (point to object-handle)
<position cursor><CR>  (set pivot to rightmost
                 character)
<CR>

*Rotate -a-90<CR>
.<CR>    (reference previous defined area)
<position cursor><CR>    (set pivot to a character near
                  the middle)
<CR>
```

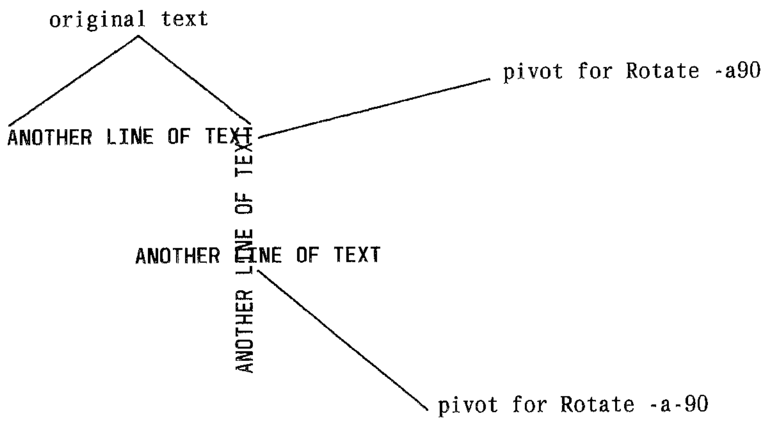


Figure 5-9. Rotating Text

Changing the Style and Width of Lines

In the current editor, objects can be drawn from lines in any of five styles: solid (**so**), dashed (**da**), dot-dashed (**dd**), long-dashed (**ld**), and three widths -- narrow (**n**), medium (**m**), and bold (**b**). Style is controlled by the **s** option and width by the **w**. The next example creates a narrow-dotted line:

```
*Lines -wn,sdo<CR>
<position cursor><SP>
<position cursor><SP>
<CR>
```

Using the **E**dit command, the line can be changed to bold, dot-dashed:

```
*Edit -wb, sdd<CR>
$.0<CR>    (reference the object-handle of the previous line)
<CR>      (complete the defined area)
<CR>
```

VIEW COMMANDS

All the objects drawn lie within a Cartesian plane, 65,534 units on each axis, known as the *universe*. Thus far, only a small portion of the universe has been displayed on the screen. The command:

```
*view -u<CR>
```

displays the entire universe.

WINDOWING

A mapping of a portion of the universe onto the display screen is called a *window.* The extent or magnification of a window is altered using the **zoom** command. To build a window that includes all the objects drawn, type:

```
*zoom<CR>
<position cursor><SP>     (above and to the left of all
                          the object)
<position cursor><CR>     (below and to the right, also
                          end points)
<CR>                      (verify)
```

Zooming can be either *in* or *out.* Zooming in, as with a camera lens, increases the magnification of the window. The area outlined by *points* is expanded to fill the screen. Zooming out decreases magnification. The current window is shrunk so that it fits within the defined area. The direction of the zoom is controlled by the sense of the **out** flag; **o** true means zoom out.

The location of a window is altered using the **view** command. **View** moves the window so that a given point in the universe lies at a given location on the screen.

```
*view<CR>
<position cursor><CR>     (locate a point in the universe)
<position cursor><CR>     (locate a point on the screen)
```

View also provides access to several predefined windows. As seen earlier, **view −u** displays the entire universe. The **view −h** command displays the *home-window.* The home-window is the window that encircles all the objects in the universe. The result is similar to that of the example using **zoom** that was given earlier.

Lastly, the **view** command permits selection of a window on a particular *region*. The universe is partitioned into 25 equal-sized regions. Regions are numbered from 1 through 25, beginning at the lower left and proceeding toward the upper right. Region 13, the center of the universe, is used as the default region by drawing commands such as **plot**(1) and **vtoc**(1).

OTHER COMMANDS

Interacting with Files

The **write** command saves the contents of the display buffer by copying it to a file:

```
*write filename<CR>
```

The contents of *filename* will be a GPS. Thus, it can be displayed using any of the device filters (such as, **td** (1)) or read back into **ged**.

A GPS is read into the editor using the **read** command:

```
*read filename<CR>
```

The GPS from *filename* is appended to the display buffer and then displayed. Because **read** does not change the current window, only some (or none) of the objects read may be visible.

A useful command sequence to view everything read **is**:

```
*read -e- filename<CR>
*view -h<CR>
```

The display function of **r**ead is inhibited by setting the **e**cho flag to false; **v**iew **-h** windows on and displays the full display buffer.

The **r**ead command may also be used to input text files. The form is:

```
read [-option(s)] filename<CR>
```

Followed by a single point to locate the first line of text. A text object is created for each line of text from *filename*. Options to the **r**ead command are the same as those for the **T**ext command.

EXAMPLE OF EDITING A GPS IN THE GRAPHICS EDITOR

Note: From the label command example in Chapter 4, you will observe that the label for the y-axis was not in the correct position. In this example, we will position the label in the correct position by using the graphics editor (**ged**).

1. Instead of displaying the drawing after appending the label file to the histogram, you must direct the output to a file so that it can be read into the graphics editor (**ged**).

```
^label -Flab,h,r90,y Randplot >Q<CR>
```

2. Enter the graphics editor (**ged**) and read the file **Q** into the graphics editor (**ged**). Then, observe the drawing by using the **view** command.

```
^ged<CR>
*read -e- Q<CR>
*view -h<CR>
```

The same drawing that is shown in Figure 4-5 will be seen.

3. Now, you can use the **m**ove command to position the label in the correct position.

```
*Move<CR>
<position cursor><CR>     (point A, Figure 5-10(a))
<CR>     (point A, Figure 5-10(a))
<position cursor><CR>     (point B, Figure 5-10(a))
```

4. Use the **n**ew command and observe that the label is in the correct position.

```
*new<CR>
```

The results of the drawing is shown in Figure 5-10(b)).

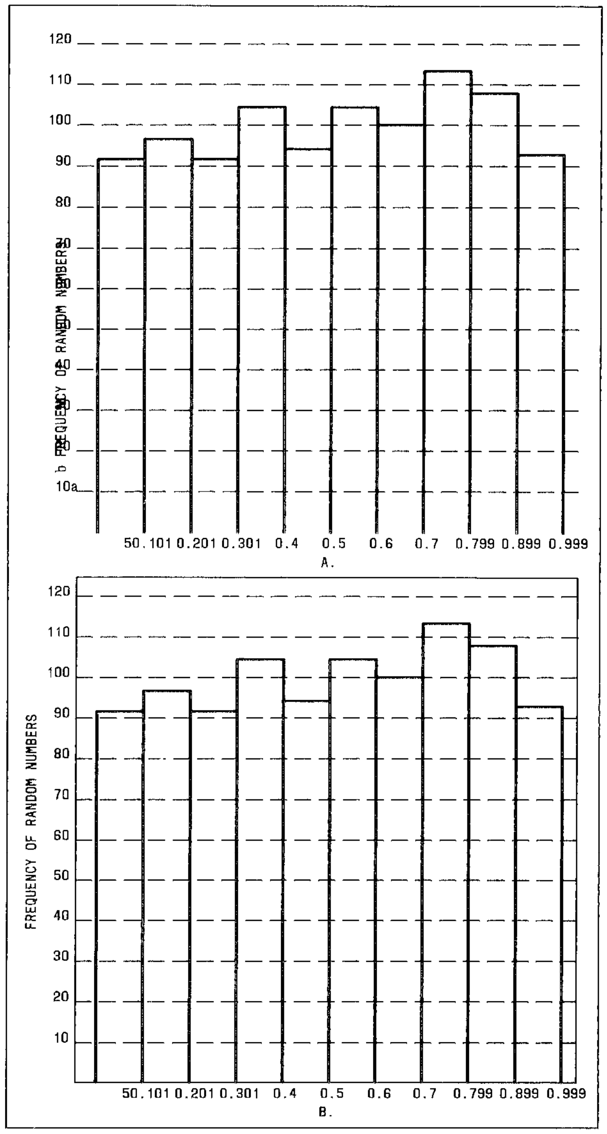


Figure 5-10. Example of Editing a GPS in the Graphics Editor

EXAMPLE OF CREATING MULTIDRAWINGS IN THE SAME UNIVERSE

The following is an example of creating an x-y graph and a histogram in the graphics shell; then placing both drawings in the same universe by using the graphics editor (**ged**).

1. Create an x-y graph and direct it to file **A**; then, display the drawing on a TEKTRONIX 4014 display terminal.

```
^gas -s0,t10 | af " x^2" | plot >A<CR>
^td A<CR>
```

   The results of the drawing is shown in Figure 5-11(a).

2. Create a file of 100 random numbers, then break that file of 100 random numbers into intervals and counts by using the **bucket** command, and direct it to file **C**.

```
^rand -n100 | title -v" 100 random numbers" | qsort | bucket >C<CR>
```

3. Create a histogram of file **C** and display it on a TEKTRONIX 4014 display terminal.

```
^hist C | td<CR>
```

   The results of the drawing is shown in Figure 5-11(b).

4. Now, direct the histogram to region 14.

```
^hist -r14 C >D<CR>
```

5. Enter the graphics editor and read in the two files containing the GPS for the x-y graph and histogram. Then, use the view command to observe the results.

```
^ged<CR>
*read -e- D<CR>
*read -e- A<CR>
*view -u<CR>
```

   The results are shown in Figure 5-11(c).

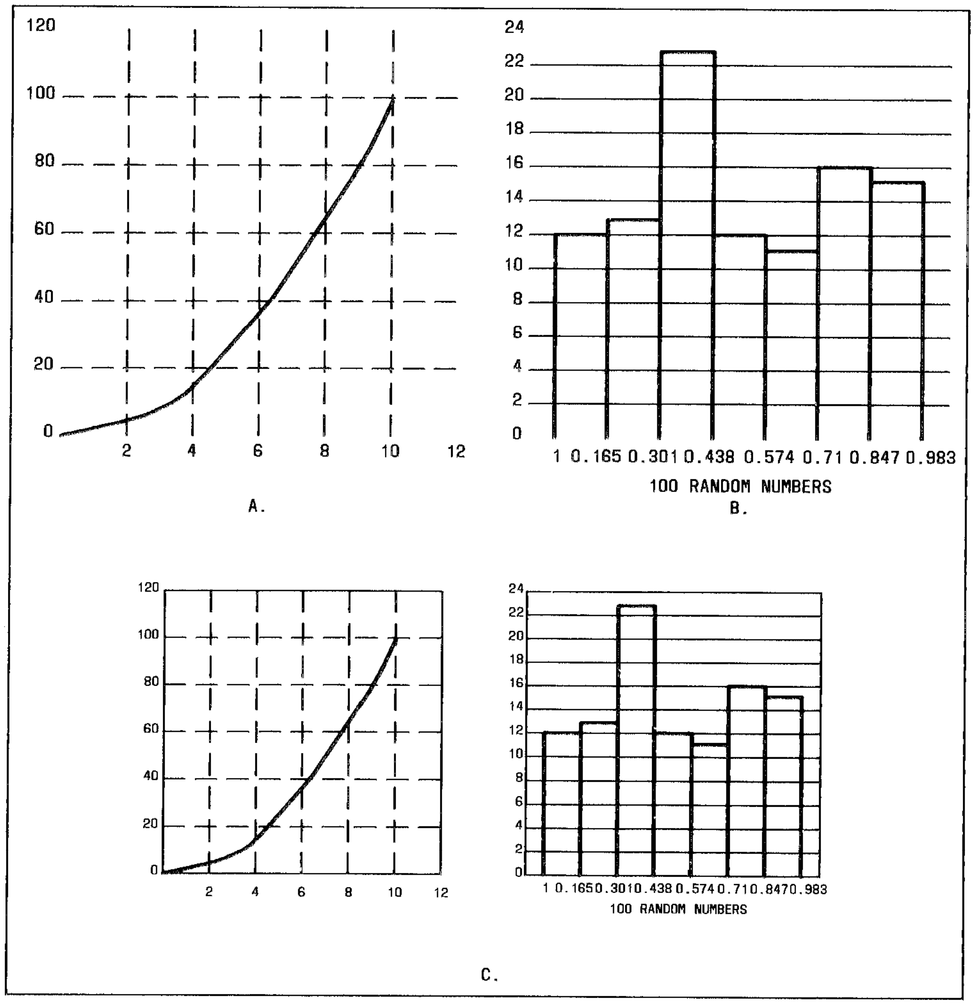


Figure 5-11. Creating a Multidrawing in the Same Screen

LEAVING THE GRAPHICS EDITOR

The **quit** command is used to end an editing session. As with the text editor **ed**, **quit** responds with **?** if the internal buffer has been modified since the last **write** command. A second **quit** command forces exit.

OTHER USEFUL INFORMATION

One-Line UNIX System Escape

As in **ed**, the **!** provides a temporary escape to the shell.

Typing Ahead

Most programs under the UNIX System allow input to be typed before the program is ready to receive it. In general, this is not the case with **ged**; characters typed before the appropriate prompt are lost.

Speeding up Things

Displaying the contents of the display buffer can be time consuming, particularly if much text is involved. The use of two flags to control what gets displayed can make life more pleasant:

- The **echo** flag controls echoing of new additions to the display buffer.
- The **text** flag controls whether text will be outlined or drawn.

COMMAND SUMMARY

In the summary, characters actually typed are printed in boldface. Command stages are printed in italics. Arguments surrounded by brackets (e.g., [...]) are optional. Parentheses surrounding arguments, separated by "or," means that exactly one argument must be given.

For example, the **D**elete command accepts the arguments **–u**niverse, **–v**iew, and *points.*

Construct Commands

Arc	[**–e**cho,**s**tyle,**w**idth] *points*
Box	[**–e**cho,**s**tyle,**w**idth] *points*
Circle	[**–e**cho,**s**tyle,**w**idth] *points*
Hardware	[**–e**cho] *text points*
Lines	[**–e**cho,**s**tyle,**w**idth] *points*
Text	[**–a**ngle,echo,**h**eight,**m**idpoint,**r**ightpoint, **t**ext,**w**idth] *text points*

Edit Commands

Delete	(**–** (**u**niverse or **v**iew) or *points*)
Edit	[**–a**ngle,echo,**h**eight,**s**tyle,**w**idth] (**–** (**u**niverse or **v**iew) or *points*)
Kopy	[**–e**cho,**p**oints,**x**] *points pivot destination*
Move	[**–e**cho,**p**oints,**x**] *points pivot destination*

Rotate	[–**a**ngle,**e**cho,**k**opy,**x**] *points pivot destination*
Scale	[–**e**cho,**f**actor,**k**opy,**x**] *points pivot destination*

View Commands

coordinates *points*	
erase	
new	
objects	(– (**u**niverse or **v**iew) or *points*)
points	(– (**l**abelled-points or **u**niverse or **v**iew) or *points*)
view	(– (**h**ome or **u**niverse or **r**egion) or [–**x**] *pivot destination*)
x	[–**v**iew] *points*
zoom	[–**o**ut] *points*

OTHER COMMANDS

quit	
read	[–**a**ngle,**e**cho,**h**eight,**m**idpoint,**r**ightpoint,**t**ext, **w**idth] *filename* [*destination*]
set	[–**a**ngle,**e**cho,**f**actor,**h**eight,**k**opy,**m**idpoint, **p**oints,**r**ightpoint,**s**tyle,**t**ext,**w**idth,**x**]
write	*filename*

!*command*

?

OPTIONS

Options specify parameters used to build, edit, and view graphical objects. If a parameter, used by a command, is not specified as an *option*, the default value for the parameter will be used. The format of command *options* is:

–*option* [,*option*]

where *option* is *keyletter*[*value*]. Flags take on the values of true or false, shown by + and –, respectively. If no value is given with a flag, true is assumed. Object options are:

angle*n*	Specify an angle of *n* degrees.
echo	When true, changes to the display buffer will be echoed on the screen.
factor*n*	Specify a scale factor of *n* percent.
height*n*	Specify height of text to be *n* universe-units (*n* greater than or equal to 0 and less than 1280).
kopy	The commands **S**cale and **R**otate can be used to either create new objects or to alter old ones. When the **k**opy flag is true, new objects are created.
midpoint	When true, use the midpoint of a text string to locate the string.
out	When true, reduce magnification during zoom.

points	When true, operate on points; otherwise, operate on objects.
rightpoint	When **true**, use the rightmost point of a text string to locate the string.
style*type*	Specify line style to be of the following *types*: **so** solid **da** dashed **dd** dot-dashed **do** dotted **ld** long-dashed
text	Most text is drawn as a sequence of lines. This can sometimes be painfully slow. When the **t**ext flag (**t**) is false, strings are outlined rather than drawn.
width*type*	Specify line width to be of the following *types*: **n** narrow **m** medium **b** bold
x	One way to find the center of a rectangular area is to draw the diagonals of the rectangle. When the **x** flag is true, defined areas are drawn with their diagonals.

Area options are:

home	References the home-window
region*n*	References the region *n*
universe	Reference the universe-window
view	Reference those objects currently in view.

SOME EXAMPLES OF USING THE ged

The following examples are used to illustrate use of the **ged**.

Example 1--Text Centered Within a Circle

```
*Circle<CR>
<position cursor><SP>     (establish center)
<position cursor><CR>     (establish radius)
*Text -m<CR>      (text is to be centered)
some text<CR>
$.0<CR>             (first point from previous set,
                    i.e., circle center)
<CR>
```

Figure 5-12 shows the output of these commands.

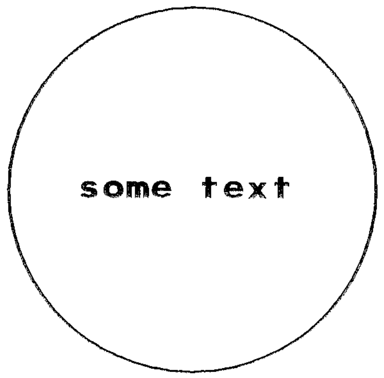


Figure 5-12. Text Centered Within a Circle

Example 2--Making Notes on a Plot

```
*! gas | plot -g >A<CR>  (generate a plot, put it in file A)
*read -e- A<CR>          (input the plot, but do not display it)
*view -h<CR>          (window on the plot)
*Lines -sdo<CR>          (draw dotted lines)
<position cursor><SP>  (0,6.5 y-axis)
<position cursor><SP>  (6.5,5.5)
<position cursor><SP>  (5.5,0 x-axis)
<CR>                     (end of Lines)
*set -h150,wn<CR>        (set text height to 150, line width to
                         narrow)
*Text -r<CR>  (right justify text)
threshold beyond that nothing matters<CR>
<position cursor><CR>  (set right point of text)
*Text -a-90<CR>  (rotate text negative 90 degrees)
threshold beyond that nothing matters<CR>
<position cursor><CR>  (set top end of text)
*x<CR>  (find center of plot)
<position cursor><SP>  (top left corner of plot)
<position cursor><CR>  (bottom right corner of plot)
*Text -h300,wm,m<CR>  (build title: height 300, weight
                   medium, centered)
SOME KIND OF PLOT<CR>
<position cursor><CR>  (set title centered above plot)
*view -h<CR>  (window on the resultant drawing)
```

Figure 5-13 shows the output of these commands.

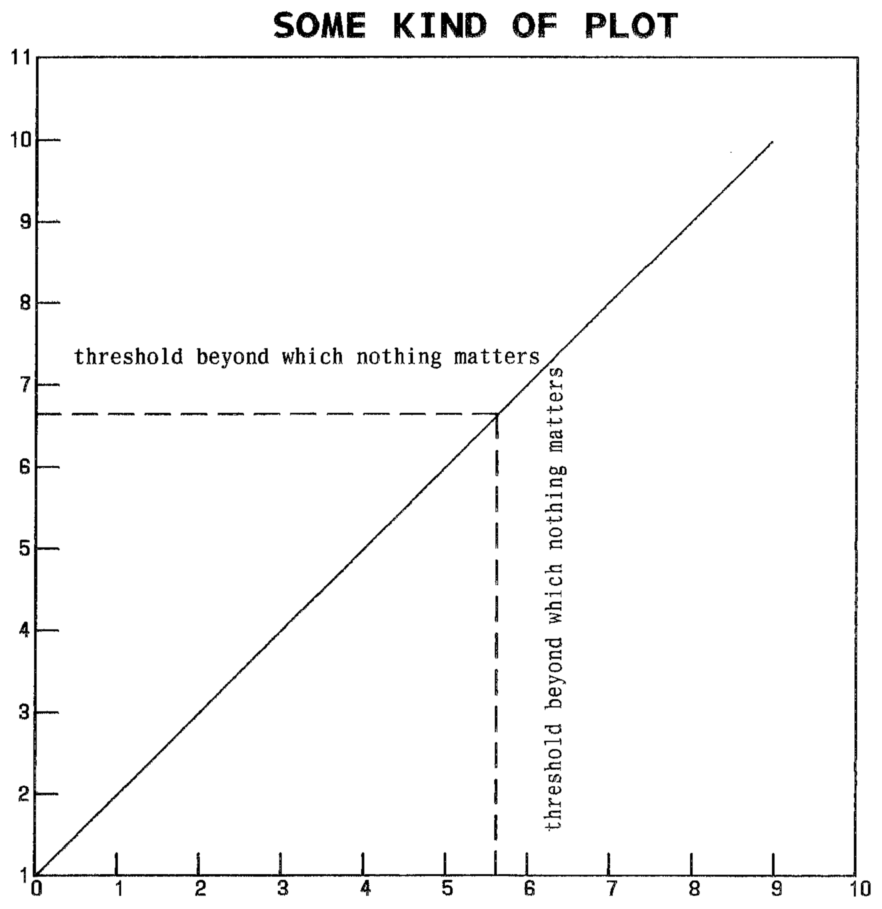
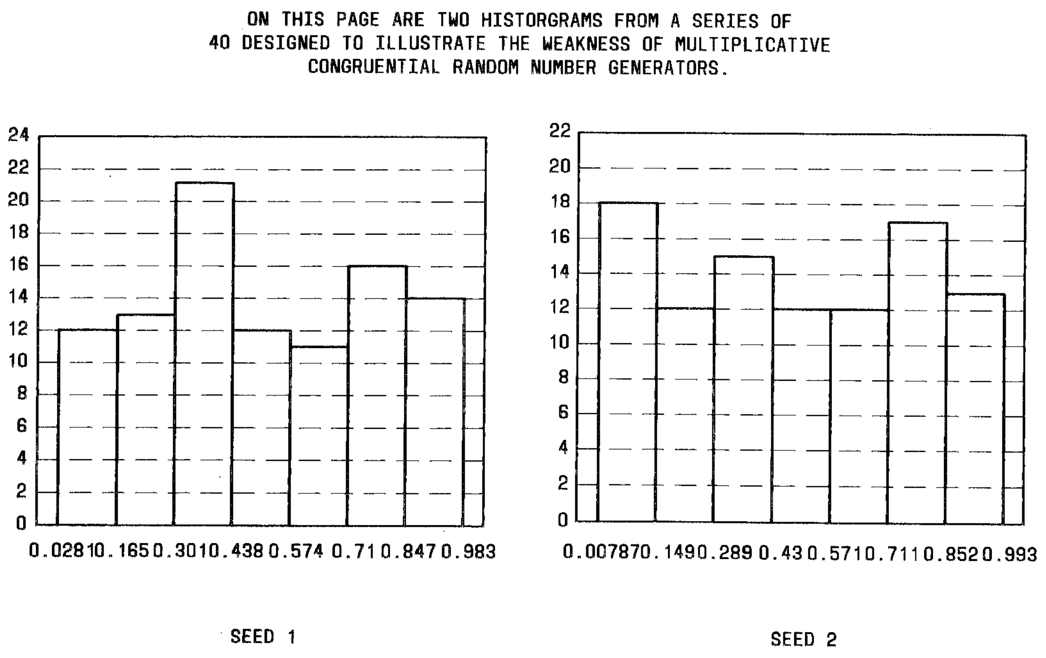


Figure 5-13. Making Notes on a Plot

Example 3--A Page Layout with Drawings and Text

```
*! rand -s1,n100 | title -v" seed 1" | qsort | bucket |
       hist -r12 >A<CR>        (put a histogram, region
             12, of 100 random numbers in file A)
*! rand -s2,n100 | title -v" seed 2" | qsort | bucket |
       hist -r13 >B<CR>        (put another histogram,
                          region 13, into file B)
*! ed<CR>  (create a file of text using the text editor)
a<CR>
On this page are two histograms<CR>
from a series of 40<CR>
designed to illustrate the weakness<CR>
of multiplicative congruential random number
generators.<CR>
.pl 3<CR>    (mark end of page)
.<CR>
w C<CR>    (put the text into file C)
151
q<CR>
*! nroff C | yoo C<CR>    (format C, leave the output
                         in C)
*view -u<CR>        (window on the universe)
*read -e- A<CR
*read -e- B<CR>
*view -h<CR>        (view the two histograms)
*read -h300,wn,m C<CR>   (text height 300, line weight
                         narrow, text centered)
<position cursor><CR>        (center text over two plots)
*view -h<CR>        (window on the resultant drawing)
```

Figure 5-14 shows the output of these commands.



ON THIS PAGE ARE TWO HISTORGRAMS FROM A SERIES OF
40 DESIGNED TO ILLUSTRATE THE WEAKNESS OF MULTIPLICATIVE
CONGRUENTIAL RANDOM NUMBER GENERATORS.

Figure 5-14. Page Layout with Drawings and Text

Replace this

page with the

HELP

tab separator.

AT&T

AT&T 3B2 Computer
UNIX™ System V Release 2.0
Help Utilities Guide

CONTENTS

Chapter 1

INTRODUCTION

Chapter 1

INTRODUCTION

GENERAL

This guide describes command formats (syntax) and use of the Help Utilities provided with your AT&T 3B2 Computer. The commands and procedures described in this guide are for someone who needs help in using the **UNIX*** System. This utilities is an interactive, menu-driven facility that provides information on the UNIX System. There are four major sections or modules in the Help Utilities. These modules will be described in later chapters of this guide.

* Trademark of AT&T

FEATURE DESCRIPTION

The Help Utilities is an optional 3B2 Computer add-on feature. It allows you to easily get a variety of information about the UNIX System while on the computer.

Feature highlights include:

1. **Starter Module** - general UNIX System information
2. **Glossary Module** - definitions of UNIX System terms and symbols
3. **Locate Module** - function-related commands
4. **Usage Module** - how to use some UNIX System commands
5. **System Administration Operation**.

HOW COMMANDS ARE DESCRIBED

In the command format discussions, the following symbology and conventions are used to define the command syntax.

- The basic command is shown in bold type. For example: **command** is in bold type
- Arguments that you must supply to the command are shown in a special type. For example: **command** *argument*
- Command options and arguments that do not have to be supplied are enclosed in brackets ([]). For example: **command** *[optional arguments]*

- The pipe symbol (¦) is used to separate arguments when one of several forms of an argument can be used for a given argument field. The pipe symbol can be thought of as an exclusive OR function in this context. For example:
  command *[argument1 ¦ argument2]*

In the sample command discussions, the lines that you input are ended with a carriage return. This is shown by using *<CR>* at the end of the lines.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

The following conventions are used to show your terminal input and the system output:

```
This style of type is used to show system generated
responses displayed on your screen.
```

This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.

These bracket symbols, < > identify inputs from the keyboard that are not displayed on your screen, such as: *<CR>* carriage return, *<CTRL d>* control d, *<ESC g>* escape g, passwords, and tabs.

This style of italic type is used for notes that provide you with additional information.

GUIDE ORGANIZATION

This guide is structured so you can easily find information without having to read the entire text. The remainder of this guide is organized as follows:

- Chapter 2, "HELP MENU," describes the **help** command and the top level menu of the Help Utilities.
- Chapter 3, "STARTER MODULE," describes the **starter** option and its lower level menu screens.
- Chapter 4, "GLOSSARY MODULE," describes the **glossary** option and its lower level menu screens.
- Chapter 5, "LOCATE MODULE," describes the **locate** option and its lower level menu screens.
- Chapter 6, "USAGE MODULE," describes the **usage** option and its lower level menu screens.
- Chapter 7, "ADMINISTRATION UTILITIES," describes the administration commands used to add or change information in the Help Utilities.

Chapter 2

HELP MENU

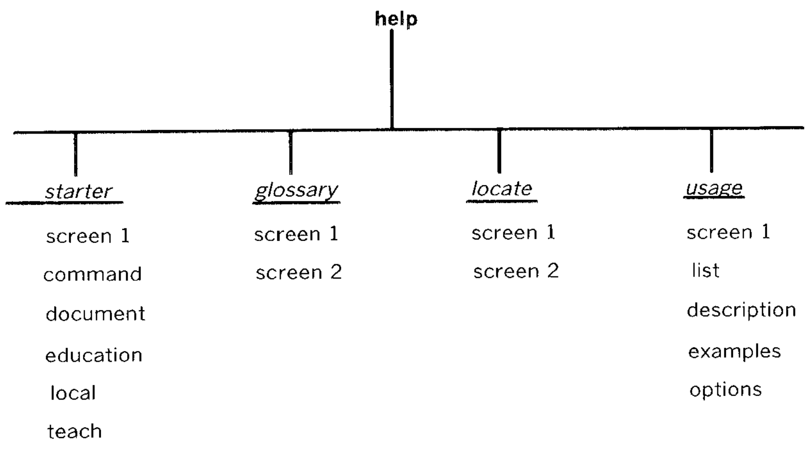
Chapter 2

HELP MENU

GENERAL

This chapter describes the **help** command and the top level menu of the Help Utilities. The top level menu is the *help menu.* There are several menus in the modules of the Help Utilities. In these menus there may be one or several screens of other menus or information. These layers of menus and information screens are connected together and branch out like the branches of a tree.

HELP UTILITIES TREE



SETTING UP YOUR TERMINAL

There are a few things you might have to do for the Help Utilities to work to its full potential with your terminal. Some of the modules have screens that are more than one page long. These screens have options that allow you to move forward and backward, one page at a time. Important information in some screens is also made more noticeable by highlighting words or letters. This highlighting does not show up on all terminals or on terminals that are defined wrong in your **.profile**. The following information will help you to set up your terminal so that it will work as well as it can with the Help Utilities.

If the **TERM** variable is not set in your **.profile** file, the default terminal will be a 450 hard-copy terminal. You must set the TERM variable if your terminal is not a 450 hard-copy terminal. All the functions of help will not be available if your TERM variable is set wrong or not set at all. For instance, the paging back and forth of some screens will not be available on some terminals. The paging options **n** and **b** are available in some of the screens that are more than one page long. The help facility also requires that the tabs are set on your terminal. If the tabs are not set, the data displayed may look like garbage.

The **SCROLL** variable may also be set in your **.profile** according to your preference. The default for **SCROLL** is **yes** (SCROLL = yes). If the SCROLL variable is set to **no** (SCROLL = no) and then exported, the screen will be cleared before printing the next screen of data. If you want the screen to scroll, you must set SCROLL = yes and then export SCROLL, or you can simply delete the SCROLL variable from your .profile. You should also delete the exporting of SCROLL if you delete the variable SCROLL. The lines that should be in your .profile are shown below:

```
SCROLL=yes
export SCROLL

or

SCROLL=no
export SCROLL
```

Option	Description
s	Enters starter screen 1
l	Enters locate screen 1
u	Enters usage screen 1
g	Enters glossary screen 1
q	Quits and exits to shell.

Bypassing the Help Menu

There may be several steps in the process of reaching the information for which you are searching. You may skip the step of entering the help menu once you become familiar with the lower level modules. For example, if you want to enter the starter screen 1, you can skip the help screen by entering:

```
help starter<CR>
```

or you can enter:

```
starter<CR>
```

Illegal Entries

Error messages are sometimes printed when you enter an unknown or illegal character while in any menu of the Help Utilities. The Help Utilities is an interactive system and will tell you when you make certain mistakes. If an illegal character is entered while in a menu, the following error message is printed.

```
_ is an invalid choice.  Enter a choice shown above.
Enter choice > _
```

Now you may enter a legal choice.

LOWER LEVEL MODULES

There are four lower level modules each with a set of menus with options for selecting various other menus or information. The information in these modules will be described later.

Module Menus

These menus are interactive, which means the computer will prompt you with a list of choices. Then, you must decide what to do. If a mistake is made, the computer will print an error message and allow you to make another choice. In this way, the computer will teach you how to use the Help Utilities.

> ***Note:*** The **h** (help) option of the lower level module screens is only available if you enter that module from the help menu.

Module Contents

- **Starter Module** - contains general information on the UNIX System and information for beginners.
- **Glossary Module** - contains a list of terms and symbols and the definitions of those terms and symbols.

- **Locate Module** - is a means of identifying UNIX System commands by their function.
- **Usage Module** - contains information about specific UNIX System commands with descriptions, options, and examples demonstrating some typical uses.

Chapter 3

STARTER MODULE

Chapter 3

STARTER MODULE

GENERAL

This chapter describes the **starter** module and its lower level screens. These screens contain general information for beginners. There are six screens in the starter module, each with its own options. All these options require only a single-character entry followed by a carriage return for the operation to start.

STARTER MODULE SCREENS

The *starter module* contains the following:

- **Command screen** - lists some of the basic commands and terms for a beginner to learn.
- **Document screen** - is a list of important basic UNIX System documents.
- **Education screen** - is a list of training centers for UNIX System courses, including addresses.
- **Local screen** - lists the name, location, and telephone number of your local system administrator.
- **Teach screen** - lists information about available on-line teaching aids.

Entering Starter Screen 1

The *starter screen 1* can be entered from the help menu as explained in Chapter 2. You can also enter the starter screen directly from the shell command level. Whether you enter directly from the shell or from the help menu, you will be put in starter screen 1. From there you can choose an option for one of the other starter screens. Each screen of the starter module can be entered from the starter screen 1 by using the appropriate option.

The *starter screen 1* menu entry methods are shown below:

To enter, type: **help**<*CR*> then enter option **s**<*CR*>
or type: **help starter**<*CR*>
or type: **starter**<*CR*>

Starter Screen 1 Options

The options of the *starter screen 1 menu* are shown below:

Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
c (command)	Displays command screen
d (document)	Displays document screen
e (education)	Displays education screen
l (local)	Displays local screen
t (teach)	Displays teach screen.

Entering Starter Command Screen

The *starter command screen* is entered from starter screen 1. The *starter command screen* menu entry method is shown below:

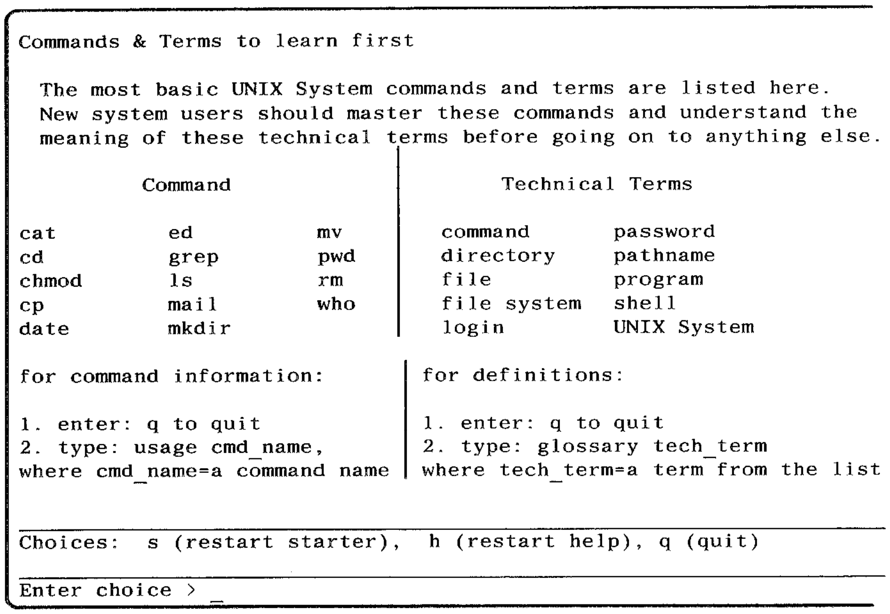
To enter, type: **c**<*CR*>

Starter Command Screen Options

The options of the *starter command menu* are shown below:

Options	Descriptions
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
s (starter)	Returns to starter screen 1.

Below is an example of the *starter command* screen:



```
Commands & Terms to learn first

  The most basic UNIX System commands and terms are listed here.
  New system users should master these commands and understand the
  meaning of these technical terms before going on to anything else.

          Command                     |        Technical Terms
                                      |
cat          ed          mv           |  command        password
cd           grep        pwd          |  directory      pathname
chmod        ls          rm           |  file           program
cp           mail        who          |  file system    shell
date         mkdir                    |  login          UNIX System

for command information:              | for definitions:

1. enter: q to quit                   | 1. enter: q to quit
2. type: usage cmd_name,              | 2. type: glossary tech_term
where cmd_name=a command name         | where tech_term=a term from the list

Choices:  s (restart starter),  h (restart help), q (quit)

Enter choice > _
```

You can use the commands and terms in this screen to get familiar with using the Help Utilities and learn the commands and terms at the same time.

Entering Starter Documents Screen

The *starter documents screen* is entered from starter screen 1. The *starter documents screen* menu entry method is shown below:

To enter, type: **d**<*CR*>

Starter Documents Screen Options

The options of the *starter documents menu* are shown below:

Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
s (starter)	Returns to starter screen 1.

The following is an example of a few of the documents listed in the documents screen:

— Basics for UNIX System Users
— Using the File System
— Screen Editor (vi) Tutorial
— Shell Tutorial
— File System Hierarchy/Pathnames
— Text Formatters Reference.

Entering Starter Education Screen

The *starter education screen* is entered from starter screen 1. The *starter education screen* menu entry method is shown below:

To enter, type: **e**<*CR*>

Starter Education Screen Options

The options of the *starter education menu* are shown below:

Options	Description
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
s (starter)	Returns to starter screen 1.

The following is an example of how information about a training center is listed in the education screen.

Name:	AT&T Technologies Dublin Training Center
Location:	Dublin OH
Information:	Call: 800-TRAINER
Audience:	AT&T personnel and commercial customers

Entering Starter Local Screen

The *starter local screen* is entered from starter screen 1. The *starter local screen* menu entry method is shown below:

To enter, type: **l**<*CR*>

Starter Local Screen Options

The options of the *starter local menu* are shown below:

Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
s (starter)	Returns to starter screen 1.

Below is an example of the type of information found in the local screen:

```
Local System Information:

   The following is an example of information for your system
   that is to be supplied by your system administrator:

         SYSTEM ADMINISTRATOR: Chip Logic
         PHONE NUMBER: 555-8669
         SYSTEM NAME: 3B2
         PROCESSOR TYPE: WE3200

Choices:  s (restart starter),  h (restart help),  q (quit)

Enter choice > _
```

Entering Starter Teach Screen

The *starter teach screen* is entered from starter screen 1. The *starter teach screen* menu entry method is shown below:

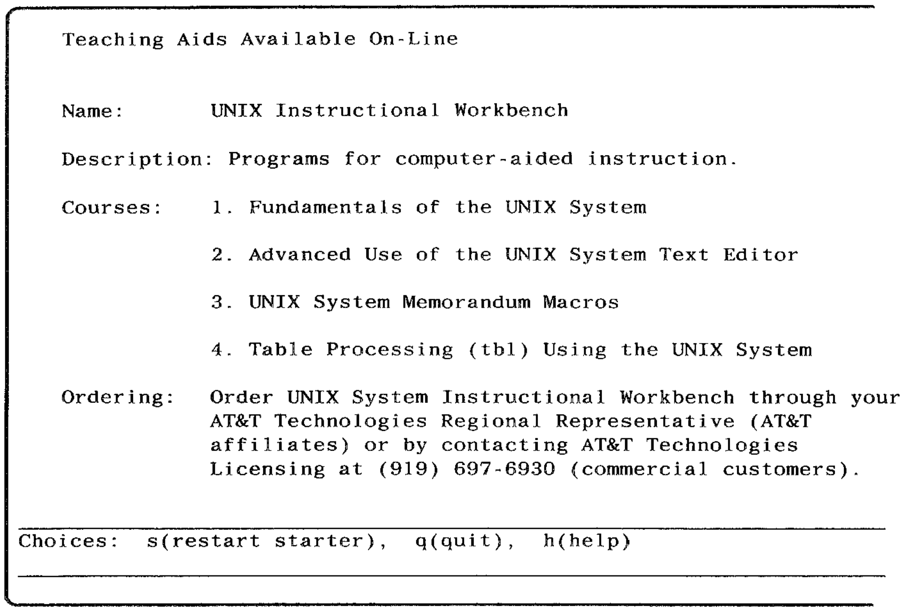
To enter, type: **t**<*CR*>

Starter Teach Screen Options

The options of the *starter teach menu* are shown below:

Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
s (starter)	Returns to starter screen 1.

Below is an example of the teach screen:



```
   Teaching Aids Available On-Line


   Name:       UNIX Instructional Workbench

   Description: Programs for computer-aided instruction.

   Courses:    1. Fundamentals of the UNIX System

               2. Advanced Use of the UNIX System Text Editor

               3. UNIX System Memorandum Macros

               4. Table Processing (tbl) Using the UNIX System

   Ordering:   Order UNIX System Instructional Workbench through your
               AT&T Technologies Regional Representative (AT&T
               affiliates) or by contacting AT&T Technologies
               Licensing at (919) 697-6930 (commercial customers).

________________________________________________________________
Choices:  s(restart starter),  q(quit),  h(help)
________________________________________________________________
```

Chapter 4

GLOSSARY MODULE

Chapter 4

GLOSSARY MODULE

GENERAL

This chapter describes the **glossary** module and its lower level screens. There are two screens of the glossary module, each with its own options. Most of these options require only a single character entry followed by a carriage return for the operation to start. The option *UNIX term* is the only exception.

GLOSSARY MODULE SCREENS

The *glossary module* contains the following:

- **Glossary screen 1** - A list of *UNIX terms* that are defined in *glossary screen 2.*

- **Glossary screen 2** - The definition of the terms listed in *glossary screen 1.*

Entering Glossary Screen 1 (Terms)

The *glossary screen 1* can be entered from the help menu as explained in Chapter 2. You can also enter the glossary screen directly from the shell command level. If you enter from the shell, you can go directly to glossary screen 2 by entering a *UNIX term* as the last argument on the command line. If you leave off the *UNIX term* argument you will be put in glossary screen 1. From there you can choose an option for one of the other glossary screens.

UNIX term can be a word or phrase special to the UNIX System (file, shell, link, or mode). The term is entered at the bottom of glossary screen 1; then, glossary screen 2 is automatically entered, and the definition is printed on the screen.

The *glossary screen 1* menu entry methods are shown below:

To enter, type: **help**<*CR*> then enter option **g**<*CR*>
    or type: **help glossary**<*CR*>
    or type: **glossary**<*CR*>

Glossary Screen 1 Options

The options of the *glossary screen 1 menu* are shown below.

Option	**Description**
h (help)	Returns to help screen 1 (if you entered from the help menu)
q (quit)	Quits and exits to shell
n (next)	Goes to next page
b (back)	Goes back one page
UNIX term	Goes to glossary screen 2 and displays definition of the *UNIX term* entered.

Below are a few of the terms listed in glossary screen 1:

#	$
*	&
HOME	UNIX
block	command line
field	file
group	link
login	mode
owner	pathname
pipe	prompt
root	root directory
shell	string
user ID	white space

Entering Glossary Screen 2 (Definitions)

The *glossary screen 2* menu entry methods are shown below:

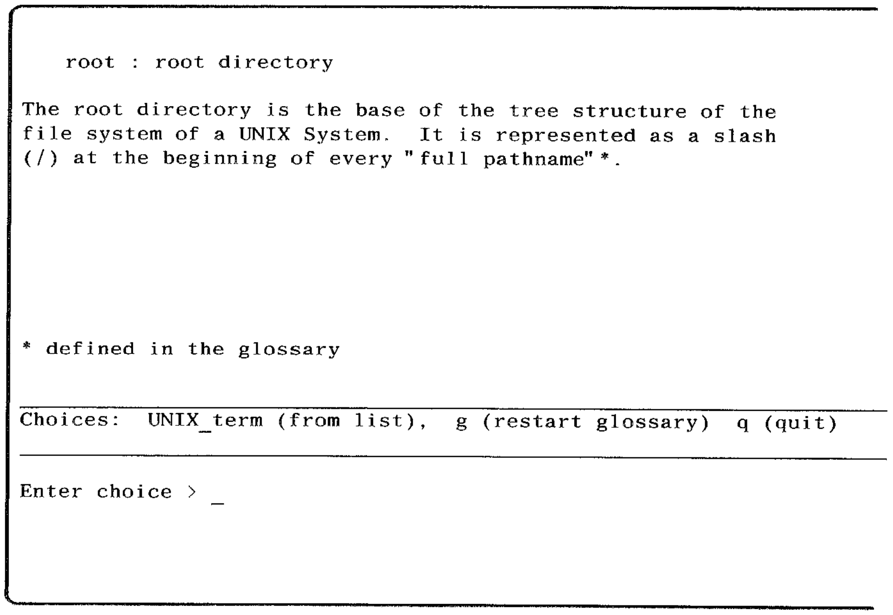
To enter, type: **help glossary** *UNIX term<CR>*
or type: **glossary** *UNIX term<CR>*
or type: **UNIX term***<CR>* (from glossary screen 1)

Glossary Screen 2 Options

The options of the *glossary screen 2 menu* are shown below:

Option	Description
h (help)	Returns to help screen 1 (if you entered from the help menu)
q (quit)	Quits and exits to shell
g (glossary)	Returns to glossary screen 1.
n (next)	Goes to next page
b (back)	Goes back one page.

Below is an example of a description found in glossary screen 2. The definition shown is for the terms **root** or **root directory**.



```
    root : root directory

The root directory is the base of the tree structure of the
file system of a UNIX System.  It is represented as a slash
(/) at the beginning of every "full pathname"*.






* defined in the glossary

Choices:  UNIX_term (from list),  g (restart glossary)  q (quit)

Enter choice > _
```

Chapter 5

LOCATE MODULE

Chapter 5

LOCATE MODULE

GENERAL

This chapter describes the **locate** module and its lower level screens. There are two screens in the locate module, each with its own options. All these options require only a single-character entry followed by a carriage return for the operation to start.

LOCATE MODULE SCREENS

The *locate module* contains the following:

- **Locate screen 1** - A description of **locate**, and a list of related commands for each *keyword* with a short example of how the function-related keywords are used.
- **Locate screen 2** - A description of related commands.

A *keyword* is a word (not a command) that is related to a group of commands through its function. **Print** is a word that can be related by function to such commands as: pr, ls, or cat. So, if you need to know what command will do a function (such as: hunt for an item), you enter that word (**hunt**). All the commands that do a similar or related function to **hunt** will be listed on the screen. You can then choose the command that will best suit your needs. If you need more information on **hunt** or other commands, you may enter the **usage** screen or reference the *AT&T 3B2 Computer User Reference Manual.*

Entering Locate Screen 1

The *locate screen 1* menu entry methods are shown below:

To enter, type: **help**<*CR*> then enter option **l**<*CR*>
or type: **help locate**<*CR*>
or type: **locate**<*CR*>

Locate Screen 1 Options

The options of the *locate screen 1 menu* are shown below:

Options	**Descriptions**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
k (*keyword*)	Goes to locate screen 2 (Enter *keyword(s)* after new prompt).

Entering Locate Screen 2

The *locate screen 2* menu entry methods are shown below:

To enter, type: **help locate** *keyword(s)<CR>*
or type: **locate** *keyword(s)<CR>*
or type: **k**<*CR*> (from locate screen 1)

Locate Screen 2 Options

The options of the *locate screen 2 menu* are shown below:

Options	**Descriptions**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
k (*keyword*)	Enter *keyword(s)* after new prompt (displays commands found using the keyword entered)
c (command)	Enter *command name*, then enter option [d, e, o] (links to usage screens) d displays the usage description screen e displays the usage examples screen o displays the usage option screen
n (next)	Goes to next page (on some terminals)
b (back)	Goes back one page (on some terminals).

Note: The **c** option goes to the Usage Module and the appropriate screen.

LOCATE EXAMPLE

Below is an example of the locate screen:

```
locate:  Find UNIX System Commands with keywords


   Give locate a list of one or more keywords related to the work
   you want to do.  It will print a list of UNIX System
   commands whose actions are related to the keywords.
   For example, the keyword list:

   print file could produce the list:   The cat (concatenate) command
                                        The ls (list) command
                                        The pr (print) command

   To use locate, enter a k. When the " Enter keywords" prompt
   appears, enter the keywords on one line, separated by blank spaces.

      choices   description

        k       Enter a list of keywords

        q       Quit

Enter choice > _
```

If you enter a **k** as the choice above, the following prompt will be displayed:

```
Enter keyword(s) >
```

If you enter **print** as the keyword at the keyword prompt in locate screen 1, the following would be displayed in locate screen 2:

```
The cat (concatenate) command
The echo command
The ls (list) command
The pr (print) command
The pwd (print working directory) command
The tail command


--------------------------------------------------------------------
Choices:   c (cmd info),  k (new keywords),  q (quit)
--------------------------------------------------------------------

Enter choice >
```

Chapter 6

USAGE MODULE

PAGE

Chapter 6

USAGE MODULE

GENERAL

This chapter describes the **usage** module and its lower level screens. There are five screens in the usage module, each with its own options. All these options require only a single-character entry followed by a carriage return for the operation to start.

USAGE MODULE SCREENS

The *usage module* contains the following:

- **Usage Screen 1** - A menu of the possible options.
- **Usage Command List Screen** - A list of all UNIX System commands supported by the Help Utilities.
- **Usage Command Description Screen** - A description of the commands listed in the command list screen.
- **Usage Command Examples Screen** - Examples of how to use each of the commands listed.
- **Usage Command Options Screen** - All the available options for each of the commands listed.

Entering Usage Screen 1

The **usage** module is used to get information on specific commands. Once you enter usage screen 1, you may print out a list of supported commands or get information on a specific command. To get information on a command, just choose option **c** and enter the command after the second prompt. You must then choose what type of information you want. You can get a description, an example, or possible options of the command. After you enter this choice, you will be allowed to make one of several choices.

The *usage screen 1* menu entry methods are shown below:

To enter, type: **help**<*CR*> then enter option **u**<*CR*>
    or type: **help usage**<*CR*>
    or type: **usage**<*CR*>

Usage Screen 1 Options

The options of the *usage screen 1 menu* are shown below:

Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
p (print)	Prints usage list screen command list
c (command)	Enter *command name*, then enter option [d, e, o] d displays the description screen e displays the examples screen o displays the option screen.

Entering Usage List Screen

The *usage list screen* menu entry method is shown below:

To enter, type: **up**<*CR*> (at usage screen 1)

Usage List Screen Options

The options of the *usage list screen 1 menu* are shown below:

Options	Description
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
c (command)	Enter *command name*, then enter option [d, e, o] d displays the description screen e displays the examples screen o displays the option screen.

Entering Usage Description Screen

The *usage description screen* menu entry methods are shown below:

To enter, type: **help usage** *command name<CR>*
  or type: **usage** *command name<CR>*
  or type: **d**<*CR*> from usage examples screen
  or type: **d**<*CR*> from usage options screen

You may also enter the description screen from other screens. After entering **c** as the choice in the appropriate screen, you must enter a **command name**. You may then enter the description screen by choosing the **d** option when you are asked to enter a choice. The following methods show how to enter from each of the other screens.

To enter, type: **d**<*CR*> from usage screen 1
  or type: **d**<*CR*> from usage list screen
  or type: **d**<*CR*> from locate screen 2

Usage Description Screen Options

The options of the *usage description screen menu* are shown below:

Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
p (print)	Prints usage list screen
c (command)	Enter *command name*, then enter option [e, o] e displays the examples screen o displays the option screen
e (example)	Displays example screen
o (option)	Displays option screen
l (locate)	Returns to the locate command list (if you entered usage from the locate module).

Entering Usage Example Screen

The *usage example screen* menu entry methods are shown below:

To enter, type: **help usage -e** *command name<CR>*
  or type: **usage -e** *command name<CR>*
  or type: **e**<*CR*> from usage description screen
  or type: **e**<*CR*> from usage options screen

You may also enter the example screen from other screens. After entering **c** as the choice in the appropriate screen, you must enter a **command name**. You may then enter the example screen by choosing the **e** option when you are asked to enter a choice.

The following methods show how to enter from each of the other screens:

To enter, type: **e**<*CR*> from usage screen 1
or type: **e**<*CR*> from usage list screen
or type: **e**<*CR*> from locate screen 2

Usage Example Screen Options

The options of the *usage example screen menu* are shown below:

Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
p (print)	Prints usage list screen
c (command)	Enter *command name*, then enter option [d, o] d displays the description screen o displays the option screen
d (description)	Displays description screen
o (option)	Displays option screen
l (locate)	Returns to the locate command list (if you entered usage from the locate module).

Entering Usage Options Screen

The *usage options screen* menu entry methods are shown below:

To enter, type: **help usage -o** *command name<CR>*
  or type: **usage -o** *command name<CR>*
  or type: **o**<*CR*> from description screen
  or type: **o**<*CR*> from example screen

You may also enter the options screen from other screens. After entering **c** as the choice in the appropriate screen, you must enter a **command name**. You may then enter the options screen by choosing the **o** option when you are asked to enter a choice. The following methods show how to enter from each of the other screens:

To enter, type: **o**<*CR*> from usage screen 1
  or type: **o**<*CR*> from usage list screen
  or type: **o**<*CR*> from locate screen 2

Usage Options Screen Options

The options of the *usage options screen menu* are shown below:

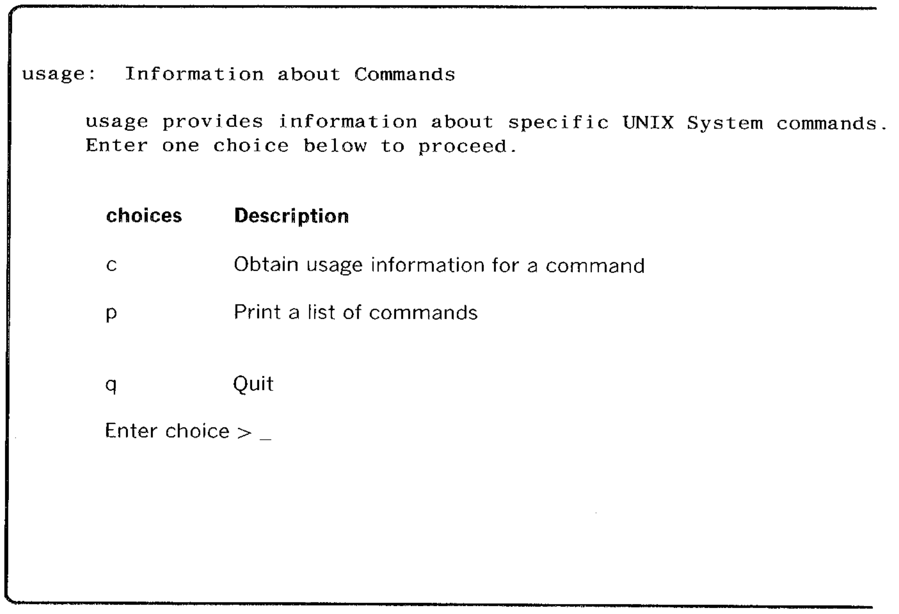
Options	**Description**
h (help)	Returns to help screen 1
q (quit)	Quits and exits to shell
p (print)	Prints usage list screen
c (command)	Enter *command name*, then enter option [d, e] d displays the description screen e displays the examples screen
d (description)	Displays description screen
e (example)	Displays example screen
l (locate)	Returns to locate command list (if you entered usage from the locate module)
n (next)	Goes to next page
b (back)	Goes back one page.

USAGE MODULE EXAMPLES

The following examples may not be an exact copy of the data contained in your help database. Because, the local administrator may add, change, or delete information in the database. The content of individual screens may be different from what is shown here.

Usage Screen 1 Example

Below is an example of **usage screen 1**



```
usage:  Information about Commands

     usage provides information about specific UNIX System commands.
     Enter one choice below to proceed.

       choices    Description

       c          Obtain usage information for a command

       p          Print a list of commands


       q          Quit

       Enter choice > _
```

Usage List Screen Example

The following is an example of the **usage list screen**

```
usage:  Information about Commands

The following commands are currently included in help:

cat     cd      chmod   cp       cut        date
echo    egrep   fgrep   file     find       glossary
grep    help    ln      locate   ls         mail
mesg    mkdir   mv      news     pr         ps
pwd     rm      sleep   sort     starter    stty
tabs    tail    tee     time     touch      tty
uname   usage   wall    wc       who        write

Choices:  c (enter cmd),  q (quit)

Enter choice > _
```

If you enter a **c** as the choice above, the following prompt will be displayed:

```
Enter command name > _
```

After you enter a *command name* (cmd_name) at the prompt above, the following prompt will be displayed:

```
Enter d (description),  e (examples), or  o (options) > _
```

After entering an option at the prompt above, the chosen screen will be displayed.

Usage Description Screen Example

Below is an example of the **usage description** screen for the **cat** command:

```
cat

Syntax Summary:  cat [-u] [-s] [-v [-t] [-e] ] file_name ...

        where:
                     file_names are simple file names, relative
                        pathnames, or full pathnames.

Description:
          cat is shorthand for "concatenate."  It prints the
          contents of the file[s] specified as its argument[s].
          If more than one file is specified, cat will print
          each one in sequence on the standard output.
          See also:  cp(1), pg(1), pr(1).

______________________________________________________________________
Choices:  o(options),  e(examples),  c(enter cmd),  p(print list),
 q(quit)
______________________________________________________________________

Enter choice > _
```

You may now enter a choice, and the appropriate screen will be displayed (unless you enter a **q**).

Usage Examples Screen Example

If you enter an **e** as the option at the prompt in the **cat description** screen, the following **cat examples** screen will be displayed:

```
cat   : Examples

                         cat textfile
     --> Prints, on the standard output, the contents of the
        file textfile in the current working directory.

                         cat /etc/passwd /etc/group > groupfile
     --> Prints the contents of /etc/passwd and then the contents
        of /etc/group and redirects the output to the file groupfile.

                         cat /usr/src/cmd/* | grep stdio.h
     --> Prints the lines that contain the pattern stdio.h from
        the files in the directory /usr/src/cmd.

----------------------------------------------------------------------
Choices:  o(options),  d(description),  c(enter cmd),  p(print list),
q(quit)
----------------------------------------------------------------------
Enter choice > _
```

Usage Options Screen Example

If you enter an **o** as the option at the prompt in the **cat examples** screen, the following **cat options** screen will be displayed:

```
cat   : Options

-u                output is not placed in temporary storage before
                  printing (unbuffered)
-s                suppresses messages about non-existent files
-v                causes non-printed characters to be printed (except tabs,
                  newlines and form-feeds) (e.g. ^G is printed for the bell
                  character)
-t                causes tabs to be printed as ^I (used only with -v)
-e                causes a $ character to be printed at the end of
                  each line (used only with -v)



Choices:  e(examples),  d(description),  c(enter cmd),  p(print list),
  q(quit)

Enter choice > _
```

Chapter 7

ADMINISTRATION UTILITIES

PAGE

Chapter 7

ADMINISTRATION UTILITIES

GENERAL

The Help Administration Utilities are interactive tools used by UNIX System administrators. The Administration Utilities enable administrators to add, change, or delete information in the Help database. The ability to monitor the use of the Help Utilities is also allowed for users with **root** or **bin** as a login.

Entering the 'helpadm' Menu

To use the **helpadm** command and change help data, you must be logged in as **root**, **bin**, or you must be a member of the group bin login. If anyone else tries to change help data, an error message is printed at their terminal, and they will be returned to the shell from which they entered the **Helpadm** utilities. The **helpadm** menu of options can be displayed by doing the following:

To enter, type: **helpadm**<*CR*>

'helpadm' Menu

The options of the **helpadm menu** are listed below:

Options	**Description**
1	To make changes to data in the starter module (only **root** logins may change starter data)
2	To make changes to data in the glossary module (only **root** or **bin** logins may change starter data)
3	To make changes to description, examples, options, and keyword data of the locate and usage modules (only **root** or **bin** logins may change starter data)
4	To stop monitoring the use of help (may be set by **root** or **bin** logins only)
5	To start monitoring the use of help (may be set by **root** or **bin** logins only)
q	Quits and exits **helpadm** menu, and returns you to the shell command level.

After entering the **helpadm** menu and selecting an option, you will be allowed to make the chosen action. The possible actions allowed in each menu and how to make those actions are described next.

CHANGING THE HELP DATABASE

There are three areas that you may enter, to make changes to the database. The three areas are:

- Starter
- Glossary
- Command information.

Only one person may change data in any one area at a time. If more than one person tries to change data in an area, a message is printed at the second or later person's terminal and they will be returned to the shell. If this happens, you should try to make your changes at another time.

> ***Note:*** The default editor for editing the help database is the **ed** editor.

Changing Starter Information

After you enter the starter database, a menu of the starter screen is displayed at your terminal. A prompt is printed at the bottom of the screen requesting you to enter your choice of screens. After you respond, you are placed in the editor specified by the EDITOR shell variable in your **.profile**. The default editor is **ed**. However, you may change the editor that you default to by defining the EDITOR shell variable in your **.profile** and then exporting it. You must be logged in as **root** to change information in the starter database.

Exiting the Editor

When you have finished editing the data in a screen, you will be asked if you are satisfied with the screen you have changed. If you are not satisfied, you can re-edit the screen. When you have responded that you are satisfied with the screen, you are asked if you want the changes entered into the database. To have the changes entered, you must respond by typing **y** for yes. If you do not type **y**, the database remains unchanged. This prompting sequence is used for most of the screens when exiting the edit mode.

Example of Changing Starter Data

The following is an example of how to change data in the starter database. To begin, enter:

helpadm<*CR*>

After entering **helpadm**, the first helpadm menu will be displayed. The **helpadm** menu is shown below:

```
     helpadm:  UNIX System On-Line help Administrative Utilities

These software tools will enable the administrator to change
information in the help facility's database, and to monitor
use of the help facility.


   choices    description

   1          starter

   2          glossary

   3          commands

   4          prevent recording use of help facility

   5          record use of the help facility

   q          quit


   Enter choice > _
```

If you enter **1** (starter) as the choice in the helpadm menu, the following starter menu will be displayed.

```
                helpadm:  starter

Which screens of starter do you want to make changes to?


   choices    description

   c          commands screen

   d          documents screen

   e          education screen

   l          local screen

   t          teach screen

   q          quit


   Enter choice > _
```

If you do not have permission to change starter information, the following message will be displayed; and you will be placed back in the shell from which you entered **helpadm**. Remember, only **root** can make changes to the starter database.

```
You do not have permission to change this starter screen.
Exiting with no changes to the help facility database.
```

Changing Glossary Information

After you enter the **glossary** database, you will be prompted for the term to be added, changed, or deleted. After you enter the option, you will be asked for the term which you are going to add, change, or delete. Once you enter the term, you will be placed into the default editor to edit the term. When you have finished editing, you can exit the editing process by using the procedure explained in the " Exiting the Editor" section of this manual.

> ***Note:*** If the screen being edited is more than 17 lines, you will be put back into the editor to shorten the definition.

Adding a Term

If you wish to add a term to the glossary, you should choose the add option when prompted for the option. When you enter a glossary term that is not already in the glossary, you will be prompted with a message that states the entered term is a new term, and assumes you are adding the term. At this point you will be placed in the editor specified by the EDITOR shell variable. You may then add the definition of the chosen term. The definition cannot be more than 17 lines long. When you have finished editing, you can exit the editing process by using the procedure explained in the " Exiting The Editor" section of this manual.

> ***Note:*** You may only change or delete glossary definitions if you have write permission for the **definition** file. If you do not have write permission for the definition file, an error message is printed at your terminal, and you will be returned to the shell.

Changing a Term

To change an existing term, you must enter the term when prompted for it. Once you enter an existing term, you will be prompted to choose between changing or deleting the term. If you wish to change the term, you should choose the modify option. After you enter the option, you will be placed into the default editor specified by the EDITOR shell variable.

You may then make your changes to the definition of the chosen term. When you have finished editing, you can exit the editing process by using the procedure explained in the " Exiting the Editor" section of this manual. The terms definition cannot be more than 17 lines long.

Deleting a Term

To delete an existing term, you must enter the term when prompted for it. Once you enter an existing term, you will be prompted to choose between changing or deleting the term. To delete the term, you should choose the delete option. If you choose the delete option, you will be asked again if you want to delete the term. For the term to be deleted, you must respond by typing **y** for yes. If you do not type **y**, the database remains unchanged. The following is an example of deleting a term after you have entered the glossary database from the helpadm menu.

```
Enter the name of the glossary term to be added/modified/deleted > list <CR>
list is already included in the glossary.
Do you want to m(modify) its definition or d(delete) it from the glossary?

Enter choice (m or d ) > d<CR>
Are you sure you want to delete list from the glossary?

Enter choice (y or n) > y<CR>
list deleted from glossary.
```

Example for Changing Glossary Terms

If you enter **2** (glossary) as the choice in the first helpadm menu, the following prompt will be displayed:

```
Enter the name of the glossary term to be added/modified/deleted> _
```

If you edit the definition for **shell**, which is an existing term in the glossary, the following message will be printed:

```
Editing definition for shell
```

You will then be placed in the editor mode specified by the EDITOR export variable defined in your **.profile**. After you have finished editing the term chosen, you will be prompted with the following message:

```
Are you satisfied with this definition (y or n)? > _
```

If you enter **n** at the prompt above, the following message will be displayed, and you will be placed back into the editor again:

```
Re-editing definition for shell
```

After you have finished editing again, you will be prompted with the following message:

```
Are you satisfied with this definition (y or n)? > _
```

If you enter **y** at the prompt above, the following message will be displayed.

```
Do you want the data you have entered to be added to the
help facility database (y or n)? > _
```

If you enter **y**, the changes will be entered in the database, and you will be returned to the shell. If you enter **n**, the following message will be displayed:

```
Exiting with no changes to the help database.
```

You will now be returned to the shell.

Adding a term to the glossary data base is similar to changing data of a term. After entering a **2** in the first menu for the glossary option, the following prompt will be displayed:

```
Enter the name of the glossary term to be added/modified/deleted> _
```

You will then be placed into the editor to add the definition for the term. When you have finished editing, you must answer yes to the following prompts to have the data added to the database:

```
Are you satisfied with this definition (y or n)? > _
Do you want the data you have entered to be added to the help
facility database (y or n)? > _
```

When you have answered **y** to these prompts, the following message will be displayed:

```
Modifications to the help glossary complete.
```

Modifying Command Information

You must be logged in as **root** or **bin** to enter the command area of the Help Utilities database. If anyone else tries to change command data, an error message is printed at their terminal, and they will be returned to the shell. After you enter the command area database, you may add, change, or delete the following four types of command information:

- Description information
- Examples information
- Keyword information
- Options information.

You will be asked to enter the name of the command whose data is to be added, changed, or deleted. If the command is not in the **help** database, the facility assumes that the command is to be added to the database.

Note: You may only change or delete a command description file if you have write permission for the file. If you do not have write permission for the description file, an error message is printed at your terminal, and you will be returned to the shell.

Adding Command Information

If a command is to be added to the database, then you must enter information for description, example, options, and keywords when you are prompted.

If you are adding description, options, or examples information, you will be placed in the editor specified by the EDITOR shell variable. You can then make the additions to the database. Adding keyword information will be described later.

When you have finished editing, you will be asked if you are satisfied with the screen you have changed. If you are not satisfied, you can re-edit the screen. When you have responded that you are satisfied with the screen, you will be asked if you want the changes entered into the database. To have the changes entered, you must respond by typing **y** for yes. If you do not type **y**, the database remains unchanged.

You will automatically be allowed to edit the keyword list for a new command. There must be at least one keyword in the list before you will be allowed to exit the keyword list. A keyword must be a single word. When you have finished adding keywords, you must enter a period on a line by itself. You will then be asked if you are satisfied with the list. Once you respond to this prompt, you will be asked if you want the data you enter to be added to the database. To have the changes entered, you must respond by typing **y** for yes. If you do not type **y**, the database remains unchanged. This will complete the editing process for adding a command to the database. The administration facility will add the information and print out a message for each type of screen as the data is added for the new command. A message will also be displayed when all changes have been done.

Changing Command Information

If command information is to be changed, you will be given the option of choosing what part is to be changed: description, option, examples, or keywords. When you respond, you will be placed in the editor specified by the EDITOR shell variable. You can then make the changes to the database.

When you have finished editing, you will be asked if you are satisfied with the screen you have changed. If you are not satisfied, you can re-edit the screen. When you have responded that you are satisfied with the screen, you will be asked if you want the changes entered into the database. To have the changes entered, you must respond by typing **y** for yes. If you do not type **y**, the database remains unchanged.

When changing keywords in the keyword list, the list is printed on the terminal. Then, you are asked if you want to delete any keywords. If you do not want to delete any keywords, you enter a period on a line by itself. This will stop deletions and start additions. You will be prompted for a new keyword. A keyword must be a single word. To stop the adding process, you must enter a period on a line by itself. At this point, you will be given the option of adding more keywords or stopping.

Deleting Command Information

If the command already exists in the database, you will be given the choice of deleting the commands information when the command name is entered. If the command is to be deleted, all the command information: description, options, examples, and keywords will be deleted at the same time. You must respond by typing **y** when the prompt is displayed. You will then be asked if you are sure you want to delete the information.

The following is an example of removing the **ls** command:

```
Enter the name of the command to be added/modified/deleted > ls<CR>
The ls command is already included in help.


Do you want to delete information on ls (y or n)? > y<CR>
Are you sure you want to remove keywords, description, option,
and example information for ls (y or n)? > y<CR>
Keywords, description, option, and example information for
ls have been deleted from the help facility database.
```

Example for Modifying Command Data

The following is an example for adding a command to the help database. After you enter a **3** (command) as the choice in helpadm menu, the following will be displayed for adding the **split** command:

```
Enter the name of the command to be added/modified/deleted > split<CR>

New Command: split

This command is not currently included in the help facility.
You must enter a COMPLETE set of command data, including
description and syntax information, option information, usage
examples, and a keyword set, if ANY of the data you enter
are to be added to the help facility's database. You will
be asked to explicitly request that this data be included in
the help facility database at the end of this session.

Editing Description Screen For split
```

At this point you may add the description for the command being added to the database. Once the editing has been completed, the following prompt is displayed. You should then respond by entering a **y** or an **n**.

```
Are you satisfied with this screen (y or n)? > y<CR>
Description Screen Completed.
Editing Options Screen for split
```

At this point you may add the options for the command being added to the database. Once the editing has been completed, the following prompt is displayed. You should then respond by entering a **y** or an **n**.

```
Are you satisfied with this screen (y or n)? > y<CR>
Options Screen Completed.
Editing Examples Screen for split
```

At this point you may add the examples for the command being added to the database. Once the editing has been completed, the following prompt is displayed. You should then respond by entering a **y** or an **n**.

```
Are you satisfied with this screen (y or n)? > y<CR>
Examples Screen Completed.
Making Keyword List for split
Enter a single keyword for the command after each colon (:).
To stop adding keywords, enter a period (.).

                    :
                    :
```

After entering the keywords as described and ending with a period, the keyword list you entered will be displayed on the terminal. The following prompts will be displayed after the keyword list:

```
Are you satisfied with this list? (y or n) > y<CR>
Do you want the data you have entered to be added to the help
facility database? (y or n) > y<CR>
```

Once you enter a **y** to have the data entered into the database, the following messages will be displayed as each type screen is written to memory.

```
Description Screen for split Updated
Options Screen for split Updated
Examples Screen for split Updated
Keyword List for split Updated.
Modifications to help database complete.
```

You will now be returned to the shell from which you entered the helpadm command.

The following is an example of changing data of the **list** command in the help database. After you enter a **3** (command) as the choice in the helpadm menu, the following will be displayed for changing the list command.

```
Enter the name of the command to be added/modified/deleted > list<CR>
The list command is already included in help.

Do you want to delete information on list (y or n)? n<CR>
What part of the command data do you wish to edit?

Enter d(desc), o(options), e(examp), k(keywds), or q(quit) > d<CR>
Editing Description Screen for list
```

After editing the description of the command chosen the following prompt will be displayed.

```
Are you satisfied with this screen (y or n)? > y<CR>
What part of the command data do you wish to edit?
Enter d(desc), o(options), e(examp), k(keywds), or q(quit) > d<CR>
```

You may continue editing the different parts of the command in this way until you are satisfied with the results. When you have finished, you should enter a **q** (quit) at the following prompt and return to the shell:

```
Enter d(desc), o(options), e(examp), k(keywds), or q(quit) > q<CR>
```

Monitoring the Use of Help

The option of monitoring help has been provided if you want to keep a record of how the help facility is used. The monitor function can be turned off and on from the helpadm menu. Monitoring will not take place unless you turn on the monitor function. To set the option for monitoring you must enter the first helpadm menu and select option **5**. Option **4** of the menu turns the monitor function off, and option **5** turns the monitor function on. After entering the option, you will be returned to the shell level. However, if the **LOGNAME** variable is not exported in your **.profile**, the monitoring function may not work properly.

Setting the monitor function to record the use of help creates a file called **HELPLOG**. A complete record of who uses the help command and every action taken while in the help facility is contained in this file. You may read this file to see the actions taken by anyone who uses the help facility to see such things as:

- Which commands are referenced.
- Who uses help.
- What mistakes are made using help.
- Which module is used most.
- What part of a command is referenced most often (options, descriptions, or examples).

Below is an example of the **HELPLOG** file:

```
login=bin       uname=wr3b2a      date=Fri July 19 10:46:03 1985
name=locate     response='l'      status=OK
name=locate     response='d'      status=ERROR
name=getkey     response='k'      status=OK
name=Keysrch    response='list'   status=OK
name=quit       response='q'      status=OK
login=bin       uname=wr3b2a      date=Fri July 19 10:47:03 1985
```

The **HELPLOG** file will occasionally need to be cleaned up. You can execute **helpclean** to clean out the HELPLOG file. The helpclean file is an executable file that removes the data in the HELPLOG file but does not remove the file. The data in the HELPLOG file is actually copied to the **oHELPLOG** file, and a new HELPLOG file is created. If you execute the helpclean command twice in succession, both the HELPLOG and the oHELPLOG files will be cleaned out. There are other ways in which you can cleanup the HELPLOG files. Some suggestions on other cleanup methods may be found in the *AT&T 3B2 Computer User Reference Manual.*

RECOMMENDATIONS FOR FORMATTING DATA

To help users of **helpadm** to input data in a consistent format, the following guidelines are recommended.

General Rules for All Types of Help Screens

- Data will be presented exactly as it is entered— so, it should be typed the way you would like it to appear.
- Use " \S" to cause words to be highlighted (displayed in reverse video) on the screen when it is displayed. The **helpadm** Utilities automatically puts the name(s) of the term highlighted into the buffer being edited when adding a new term. If the word is within the text, leave spaces before and after. For example:

  ```
  text \S word \S  text
  ```

  If the word is not within text, spaces are not needed. For example:

  ```
  text

  \Sword\S

  text
  ```

- The " \S" characters will not appear when the text is printed. Therefore, if you want words in a column to line up, you must account for any " \S" characters around the words.

Guidelines for Glossary Screens

- A blank line at the beginning of the screen is optional.
- Highlight the name of the term.

- Start typing the term in column 1.
- If more than one term has the same definition and is being defined on the same glossary screen, then separate the names of the terms by a colon. For example:

  ```
  \S term1 : term2 \S
  ```

- Leave a blank line before beginning the text of a definition.
- The text of a definition is entered in paragraph form.
- You may start in column 1 or indent.
- Any examples in the text should be on a separate line.
- The examples should be centered and preceded and followed by blank lines, for example:

  ```
  text

      example

  continuation of text
  ```

Guidelines for Description Screens

- A blank line at the beginning of the screen is optional.
- Highlight the name of the command.
- Start typing in column 1 (the **helpadm** Utilities automatically puts the name of the command highlighted into the edited buffer when adding a new command).
- Leave a blank line.

- Enter a syntax summary and an explanation of terms in the syntax summary, as follows:

  ```
  Syntax Summary: ............
                 .............

          where: <explanation of terms
                 in syntax summary>
  ```

  When typing the words " Syntax Summary," start typing in column 1.

- Leave a blank line.

- Type " Description:" in column 1, and then skip to the next line and type the description of the command indented, for example:

  ```
  Description:

          text indented
  ```

Guidelines for Options Screens

- A blank line at the beginning of the screen is optional.

- Highlight the name of the command and the word " Options" .

- Start typing the text in column 1, for example:

  ```
  \S command-name \S: Options
  ```

  (the **helpadm** Utilities automatically puts the name of the command highlighted into the screen).

- Leave a blank line.

- If the command has options, then type each one with an explanation, for example:

  ```
  \S option \S    explanation
        .           .
        .           .
        .           .
  ```

- If there are no options, then type the following:

  There are no options to \Scommand-name\S.

Guidelines for Examples Screens

- A blank line at the beginning of the screen is optional.
- Highlight the name of the command and the word " Examples" .
- Start typing the command in column 1, for example:

  \S command-name \S : Examples

  The **helpadm** Utilities automatically puts the name of the command highlighted into the screen.

- Leave a blank line.
- The example should be entered on a separate line and highlighted. The explanation should begin on the next line, for example:

  \S command-name \S

  text of explanation.

- Each example should be separated by one blank line.

Replace this

page with the

INTER-PROCESS COMMUNICATION

tab separator.

AT&T

AT&T 3B2 Computer
UNIX™ System V Release 2.0
Inter-Process Communication
Utilities Guide

CONTENTS

Chapter 1

INTRODUCTION

Chapter 1

INTRODUCTION

GENERAL

This guide describes the Inter-Process Communication (IPC) Facilities and Utilities available with the AT&T 3B2 Computer.

Facilities

Facilities are uniquely identifiable software mechanisms that processes (executing programs) create, control, or operate on. These software mechanisms "facilitate " or handle IPC.

There are three types of IPC facilities. These three types of IPC facilities are the heart of IPC . Each type of IPC facility allows a particular method of communication between or among cooperating processes. These methods of communication are named as follows:

- Messages
- Semaphores

- Shared memory.

Processes create, control, or operate on facilities by using system calls. Each type of IPC facility has three categories of system calls associated with it. These system calls are normally imbedded in **C** Language programs to do the following functions:

- Getting the facility
- Controlling the facility
- Operating on the facility.

There are nine IPC **UNIX*** System manual pages associated with these system calls, three manual pages for each of the three types of IPC:

msgget()	msgctl()	msgop()
semget()	semctl()	semop()
shmget()	shmctl()	shmop()

The first three letters of the IPC UNIX System manual page names represent the type of IPC: **msg** for message, **sem** for semaphore, and **shm** for shared memory. The last three letters of the names represent the action to do: **get** for getting the facility, **ctl** for controlling the facility, and **op** for operating on the facility.

These names are the system call names with two exceptions. The **msgop()** and **shmop()** UNIX System manual page names are not used to invoke the system calls. They both have two different system call names for their operations.

* Trademark of AT&T

For **msgop()** they are:

- **msgsnd()**, message send
- **msgrcv()**, message receive.

For **shmop()** they are:

- **shmat()**, shared memory attach
- **shmdt()**, shared memory detach.

The naming of the **msgop()** and **shmop()** UNIX System manual pages is for consistency and ease of reference.

Utilities

There are two IPC utilities (commands) that run under the UNIX System. These commands are used for the following:

- Checking the status of IPC facilities
- Removing IPC facilities.

The mnemonic names for these commands are as follows:

- **ipcs**
- **ipcrm**.

The first three letters of these commands (**ipc**) represent ''inter-process communication.'' The remaining letters denote what the command is used for: **s** stands for status, and **rm** stands for remove. These commands give you a direct interface to the IPC facilities.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

GUIDE ORGANIZATION

This guide is structured so you can easily find desired information without having to read the entire text. The remainder of this document is organized as follows:

- Chapter 2, "OVERVIEW OF IPC FACILITIES," gives an overview of each type of IPC facility. This overview allows you to understand how the types of IPC facilities work and what they can do for you.
- Chapter 3, "MESSAGES," describes the message type of IPC. The prerequisites (calling sequence) before invoking each system call and the return values for each system call are explained. A verified program listing to exercise each system call is explained.
- Chapter 4, "SEMAPHORES," describes the semaphore type of IPC. The prerequisites (calling sequence) before invoking each system call and the return values for each system call are explained. A verified program listing to exercise each system call is explained.
- Chapter 5, "SHARED MEMORY," describes the shared memory type of IPC. The prerequisites (calling sequence) before invoking each system call and the return values for each system call are explained. A verified program listing to exercise each system call is explained.
- Chapter 6, "SYSTEM TUNABLE PARAMETERS," describes the IPC system tunable parameters. The maximum or default value initially set for each tunable parameter is given. When one tunable parameter affects another parameter, the interrelationship is explained.

- Chapter 7, "COMMAND DESCRIPTIONS," contains tutorial information for using the **ipcs** and **ipcrm** utilities. The system call programs described in the MESSAGES, SEMAPHORES, and SHARED MEMORY chapters were used to develop the facilities shown in the examples.

- Appendix, "IPC ERROR CODES," explains the standard system call error numbers as they apply to IPC. They are categorized by the type of IPC and associated system calls.

Chapter 2

OVERVIEW OF IPC FACILITIES

Chapter 2

OVERVIEW OF IPC FACILITIES

The UNIX System V Release 2.0 Operating System supports three types of Inter-Process Communication (IPC):

- Messages
- Semaphores
- Shared Memory.

This chapter contains a general discussion of each type of IPC. Following chapters contain detailed discussions of the associated system calls for each type of IPC. If you are unfamiliar with IPC facilities, the organization of this guide should enable you to understand and use the facilities first before using the **ipcs** and **ipcrm** utilities.

MESSAGES

The *message* type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can do two operations:

- Sending
- Receiving.

Before a message can be sent or received by a process, a process must have the UNIX System generate the necessary software mechanisms to handle these operations. A process does this by using the **msgget()** system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for all other processes, including itself. Later, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl()** system call. However, the creator always remains the creator as long as the facility exists. Other processes with permission can use **msgctl()** to do various other control functions.

Processes that have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, simplistically, a process that is attempting to send a message can wait until the process that is to receive the message is ready and vice versa. A process that specifies that execution is to be suspended is performing a ''blocking message operation.'' A process that does not allow its execution to be suspended is performing a ''nonblocking message operation.''

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- It is successful
- It receives a signal
- The facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (-1) is returned to the process, and an external error number variable **errno** is set accordingly. Examples of message system calls are contained in Chapter 3, "MESSAGES."

System tunable parameters that define the maximum UNIX System resources that are initially set for this type of IPC are given in Chapter 6 of this guide, "SYSTEM TUNABLE PARAMETERS." These parameters are also pointed out where they affect the usage of a system call in the "MESSAGES" chapter.

SEMAPHORES

The *semaphore* type of IPC allows processes (executing programs) to communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the UNIX System is able to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a system tunable parameter limit, SEMMSL=25. Semaphore sets are created by using the **semget()** system call.

The process performing the **semget()** system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set, including itself. This process can later relinquish ownership of the set or change the operation permissions using the **semctl()**, semaphore control, system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl()** to do other control functions.

Provided a process has alter permission, it can manipulate the semaphore(s). Each semaphore within a set can be manipulated in two ways with the **semop()** system call:

- Incremented
- Decremented.

To increment a semaphore, an unsigned positive integer value of the desired magnitude is passed to the **semop()** system call. To decrement a semaphore, a minus (-) signed value of the desired magnitude is passed.

The UNIX System insures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process that is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is done by passing a value of zero to the **semop()** system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a "nonblocking semaphore operation." A known error code (-1) is returned to the process, and the external **errno** variable is set accordingly.

The blocking semaphore operation, simplistically, allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the UNIX System until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **semop()**, semaphore operation, system call.

> ***Note:*** When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is one less than the total in the set.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when a "blocking semaphore operation" on a semaphore in the set cannot be completed successfully; no changes are made until they can all be made. For example, if a process has successfully completed three of six operations on a set of ten semaphores but is "blocked" from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the "blocked" operation, including the blocked operation, can specify that at such time that all operations can be performed successfully, that the operation be undone. Otherwise, the operations are performed and the semaphores are changed or one "nonblocking operation" is unsuccessful and none are changed. All this is commonly referred to as being "atomically performed."

The ability to undo operations requires the UNIX System to maintain an array of "undo structures" corresponding to the array of semaphore operations to be performed. Each semaphore operation that is to be undone has an associated adjust variable used for undoing the operation, if necessary.

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly. The detailed usage of these system calls is contained in Chapter 4, "SEMAPHORES."

System tunable parameters that define the maximum UNIX System resources that are initially set for this type of IPC are given in Chapter 6 of this guide, "SYSTEM TUNABLE PARAMETERS." They are also pointed out where they affect the usage of a system call in the "SEMAPHORES" chapter.

SHARED MEMORY

The *shared memory* type of IPC allows two or more processes (executing programs) to share memory and consequently the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis that is 3B2 Computer memory management hardware dependent.

This sharing of memory provides the fastest means of exchanging data between processes.

A process initially creates a shared memory segment facility using the **shmget()** system call. On creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) on attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

- **shmat()** — shared memory attach
- **shmdt()** — shared memory detach.

Shared memory attach allows processes to associate themselves with the shared memory segment, if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl()** system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can do other functions on the shared memory segment using the **shmctl()** system call.

System calls make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly. The detailed usage of these system calls is contained in Chapter 5, "SHARED MEMORY."

System tunable parameters that define the maximum UNIX System resources that are initially set for this type of IPC are given in Chapter 6 of this guide, "SYSTEM TUNABLE PARAMETERS." They are also pointed out where they affect the usage of a system call in the "SHARED MEMORY" chapter.

Chapter 3

MESSAGES

PAGE

Chapter 3

MESSAGES

The *message* type of Inter-Process Communication (IPC) allows processes to communicate through the exchange of data. This data is exchanged in discrete portions called messages. They are exchanged by sending or receiving; see the " OPERATIONS FOR MESSAGES " section in this chapter about sending or receiving messages.

GENERAL

Before a message can be sent or received, a uniquely identified **message queue** and **data structure** must be created. The unique identifier created is called the message queue identifier (**msqid**); it is used to identify or reference the associated message queue and data structure. Figure 3-1 illustrates the relationships among the msqid, message queue, and data structure.

The message queue is used to store (header) information about each message that is being sent or received. This information includes the following for each message:

- Pointer to the next message on queue
- Message type
- Message text size
- Message text address.

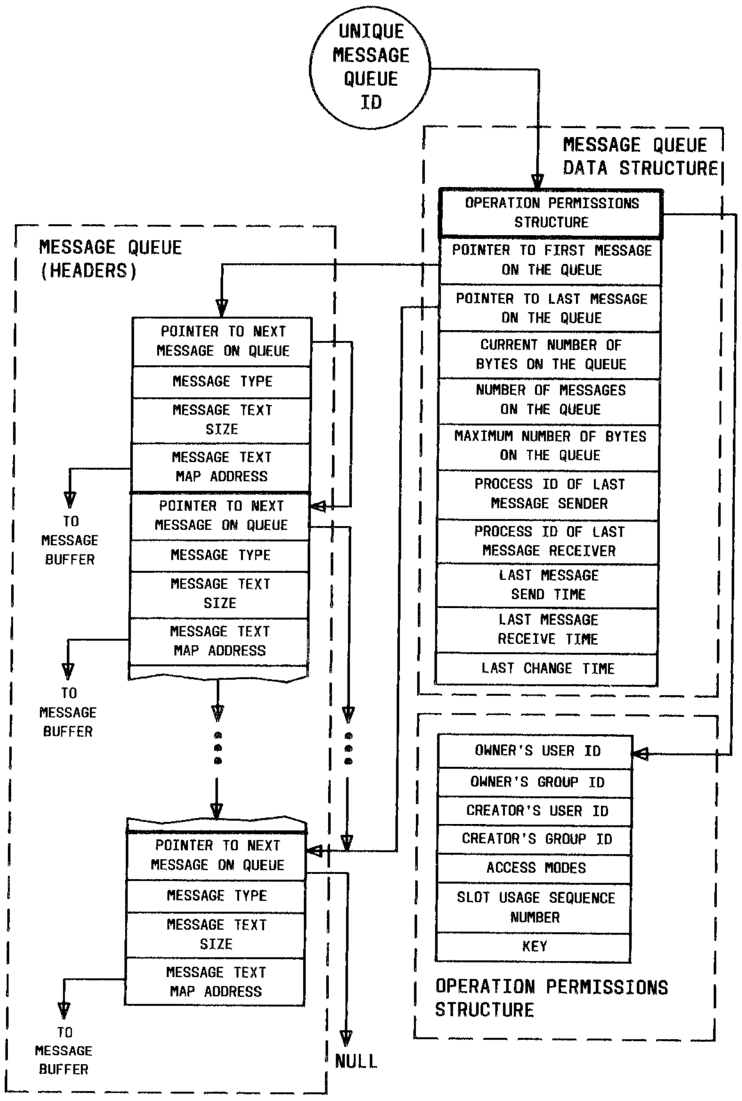

UNIQUE MESSAGE QUEUE ID
MESSAGE QUEUE DATA STRUCTURE
OPERATION PERMISSIONS STRUCTURE
POINTER TO FIRST MESSAGE ON THE QUEUE
POINTER TO LAST MESSAGE ON THE QUEUE
CURRENT NUMBER OF BYTES ON THE QUEUE
NUMBER OF MESSAGES ON THE QUEUE
MAXIMUM NUMBER OF BYTES ON THE QUEUE
PROCESS ID OF LAST MESSAGE SENDER
PROCESS ID OF LAST MESSAGE RECEIVER
LAST MESSAGE SEND TIME
LAST MESSAGE RECEIVE TIME
LAST CHANGE TIME
MESSAGE QUEUE (HEADERS)
POINTER TO NEXT MESSAGE ON QUEUE
MESSAGE TYPE
MESSAGE TEXT SIZE
MESSAGE TEXT MAP ADDRESS
TO MESSAGE BUFFER
NULL
OWNER'S USER ID
OWNER'S GROUP ID
CREATOR'S USER ID
CREATOR'S GROUP ID
ACCESS MODES
SLOT USAGE SEQUENCE NUMBER
KEY
OPERATION PERMISSIONS STRUCTURE

Figure 3-1. Message IPC Organization

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue.

- Operation permissions data (operation permission structure)
- Pointer to first message on the queue
- Pointer to last message on the queue
- Current number of bytes on the queue
- Amount of messages on the queue
- Maximum number of bytes on the queue
- Process Identification (PID) of last message sender
- PID of last message receiver
- Last message send time
- Last message receive time
- Last change time.

Note: All include files discussed in this guide are located in the */usr/include* or */usr/include/sys* directories.

The **C** Programming Language data structure definition for the message information contained in the message queue is as follows:

```
struct msg {
        struct msg      *msg_next;  /* ptr to next message on q */
        long            msg_type;   /* message type */
        short           msg_ts;     /* message text size */
        short           msg_spot;   /* message text map address */
};
```

It is located in the **/usr/include/sys/msg.h** header file.

Likewise, the structure definition for the associated data structure is as follows:

```
struct msqid_ds {
        struct ipc_perm  msg_perm;     /* operation permission struct */
        struct msg       *msg_first;   /* ptr to first message on q */
        struct msg       *msg_last;    /* ptr to last message on q */
        ushort           msg_cbytes;   /* current # bytes on q */
        ushort           msg_qnum;     /* # of messages on q */
        ushort           msg_qbytes;   /* max # of bytes on q */
        ushort           msg_lspid;    /* pid of last msgsnd */
        ushort           msg_lrpid;    /* pid of last msgrcv */
        time_t           msg_stime;    /* last msgsnd time */
        time_t           msg_rtime;    /* last msgrcv time */
        time_t           msg_ctime;    /* last change time */
                                       /* Times measured in sec since */
                                       /* 00:00:00 GMT, Jan. 1, 1970 */
};
```

It is located in the **#include <sys/msg.h>** header file also. Note that the **msgperm** member of this structure uses **ipc_perm** as a template. Thus, the breakout is shown in Figure 3-1 for the operation permissions data structure.

The definition of the **ipc_perm** data structure is as follows:

```
struct ipc_perm {
        ushort  uid;      /* owner's user id */
        ushort  gid;      /* owner's group id */
        ushort  cuid;     /* creator's user id */
        ushort  cgid;     /* creator's group id */
        ushort  mode;     /* access modes */
        ushort  seq;      /* slot usage sequence number */
        key_t   key;      /* key */
};
```

It is located in the #**include** <**sys/ipc.h**> header file; it is common for all IPC facilities.

The **msgget()** system call is used to perform two tasks when only the IPC_CREAT flag is set in the **msgflg** argument that it receives:

- To get a new msqid and create an associated message queue and data structure for it
- To return an existing msqid that already has an associated message queue and data structure.

The task performed is determined by the value of the **key** argument passed to the msgget() system call.

For the first task, if the key is not already in use for an existing msqid, a new msqid is returned with an associated message queue and data structure created for the key. This occurs provided no system tunable parameters would be exceeded.

There is also a provision for specifying a key of value zero that is known as the private key (IPC_PRIVATE = 0); when specified, a new msqid is always returned with an associated message queue and data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the msqid is all zeros.

For the second task, if a msqid exists for the key specified, the value of the existing msqid is returned. If you do not desire to have an existing msqid returned, a control command (IPC_EXCL) can be specified (set) in the msgflg argument passed to the system call. The details of using this system call are discussed in the " Using Msgget " section of this chapter.

When performing the first task, the process that calls msgget becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator; see the " CONTROLLING MESSAGE QUEUES " section in this chapter. The creator of the message queue also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, message operations [**msgop()**] and message control [**msgctl()**] can be used.

> ***Note:*** *Msgop()* is not a system call.

Message operations, as mentioned previously, consist of sending and receiving messages. System calls are provided for each of these operations; they are **msgsnd()** and **msgrcv()**. Refer to the " OPERATIONS FOR MESSAGES " section in this chapter for details of these system calls.

Message control is done by using the **msgctl()** system call. It permits you to control the message facility in the following ways:

- To determine the associated data structure status for a message queue identifier (msqid)
- To change operation permissions for a message queue
- To change the size (msg_qbytes) of the message queue for a particular msqid
- To remove a particular msqid from the UNIX System along with its associated message queue and data structure.

Refer to the " CONTROLLING MESSAGE QUEUES " section in this chapter for details of the msgctl() system call.

GETTING MESSAGE QUEUES

This section gives a detailed description of using the **msgget()** system call along with an example program illustrating its use.

Using Msgget

The synopsis of the msgget() UNIX System manual page is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/msg.h>

int  msgget (key, msgflg)
key_t  key;
int msgflg;
```

All these include files that are located in the **/usr/include/sys** directory of the UNIX System.

The following line in the synopsis:

```
int msgget (key, msgflg)
```

informs you that msgget() is a function with two *formal* arguments that returns an integer *type* value, on successful completion (msqid). The next two lines:

```
key_t  key;
int msgflg;
```

declare the types of the formal arguments. Key_t is declared by a *typedef* in the types.h header file to be a long integer. Therefore, key and msgflg are integers (*int*) which both occupy 32-bits each in the 3B2 Computer.

The integer returned from this function on successful completion is the message queue identifier (msqid) that was discussed in the "GENERAL" section of this chapter.

As declared, the process calling the msgget() system call must supply two *actual* arguments to be passed to the formal key and msgflg arguments.

The value passed to key must be a unique integer type hexadecimal value or zero (IPC_PRIVATE = 0) if a new msqid with an associated message queue and data structure is desired; it must be an existing key to return its msqid. This is true when only the IPC_CREAT flag is set in the msgflg argument.

Unique keys can be determined in several ways. The **STDIPC()**, standard interprocess communication package, subroutine is one method to generate unique keys to avoid undesired interference between processes. Another way could be to use the **makekey()** command. Picking a key at random is also possible but less desirable. If the key is IPC_PRIVATE, only the owner/creator process usually uses the facility.

> ***Note:*** Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

The value passed to the msgflg argument must be an integer type octal value and it will specify the following:

- Access permissions
- Execution modes
- Control fields (commands).

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the msgflg argument. They are collectively referred to as "operation permissions." Figure 3-2 reflects the numeric values for the valid operation permissions codes.

OPERATION PERMISSIONS	NUMERIC VALUE
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 3-2. Operation Permissions Codes

A specific numeric value is derived by adding the numeric values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). These values are represented in octal. There are constants located in the msg.h header file that can be used for the user (OWNER). They are as follows:

```
MSG_R                              0400
MSG_W                              0200
```

Control commands are predefined constants (represented by all uppercase letters). Figure 3-3 contains the names of the constants that apply to the msgget() system call along with their values. They are also referred to as flags and are defined in the ipc.h header file.

CONTROL COMMAND	VALUE
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 3-3. Control Commands (Flags)

The value for msgflg is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is done by bitwise ORing (¦) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

```
                       OCTAL VALUE      BINARY VALUE

  IPC_CREAT      =     0 1 0 0 0     0 000 001 000 000 000
¦ Read by User   =     0 0 4 0 0     0 000 000 100 000 000

  msgflg         =     0 1 4 0 0     0 000 001 100 000 000
```

The msgflg value can be easily set by using the names of the flags with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT ¦ 0400));

msqid = msgget (key, (IPC_CREAT ¦ IPC_EXCL ¦ 0400));
```

As specified by the msgget() UNIX System manual page, success or failure of this system call depends on the argument values for key and msgflg or system tunable parameters. The system call will attempt to return a new msqid if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0)
- Key does not already have a msqid associated with it, and (msgflg & IPC_CREAT) is "true" (not zero).

The key argument can be set to IPC_PRIVATE in the following ways:

```
msqid = msgget (IPC_PRIVATE, msgflg);

                OR

     msqid = msgget ( 0 , msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the MSGMNI system tunable parameter causes a failure regardlessly. The MSGMNI system tunable parameter determines the maximum amount of unique message queues (msqid's) in the UNIX System.

The second condition is satisfied if the value for key is not already associated with a msqid and the bitwise ANDing of msgflg and IPC_CREAT is "true" (not zero). This means that the key is unique (not in use) within the UNIX System for this facility type and that the IPC_CREAT flag is set (msgflg ¦ IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
      msgflg = x 1 x x x   (x = don't care)
& IPC_CREAT = 0 1 0 0 0

      result = 0 1 0 0 0   (not zero)
```

Since the result is not zero, the flag is set or "true." MSGMNI applies here also, just as for condition one.

IPC_EXCL is another control command used with IPC_CREAT to exclusively have the system call fail if, and only if, a msqid exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) msqid when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new msqid is returned if the system call is successful.

Refer to the msgget() UNIX System manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **msgget()** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the UNIX System manual page for msgget() (lines 4-8). Note that the errno.h header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired key
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **msgflg** argument
- **msqid**—used for returning the message queue identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal key, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 15-32).

> ***Note:*** All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the opperm_flags variable (lines 36-51).

The system call is made next, and the result is stored at the address of the msqid variable (line 53).

Since the msqid variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 55). If msqid equals -1, a message indicates that an error resulted, and the external **errno** variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The example program for the msgget() system call follows. It is suggested that the source program file be named "msgget.c" and that the executable file be named "msgget."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the message get, msgget(),
**system call capabilities.*/

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>
#include    <errno.h>

/*Start of main C language program*/
main()
{
    key_t key;              /*declare as long integer*/
    int opperm, flags;
    int msqid, opperm_flags;
    /*Enter the desired key*/
    printf("Enter the desired key in hex = ");
    scanf("%x", &key);

    /*Enter the desired octal operation
      permissions.*/
    printf("\nEnter the operation\n");
    printf("permissions in octal = ");
    scanf("%o", &opperm);

    /*Set the desired flags.*/
    printf("\nEnter corresponding number to\n");
    printf("set the desired flags:\n");
    printf("No flags                    = 0\n");
    printf("IPC_CREAT                   = 1\n");
    printf("IPC_EXCL                    = 2\n");
    printf("IPC_CREAT and IPC_EXCL      = 3\n");
    printf("            Flags           = ");
    /*Get the flag(s) to be set.*/
    scanf("%d", &flags);

    /*Check the values.*/
    printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
        key, opperm, flags);
        /*Incorporate the control fields (flags) with
          the operation permissions*/
        switch (flags)
        {
        case 0:     /*No flags are to be set.*/
            opperm_flags = (opperm | 0);
            break;
        case 1:     /*Set the IPC_CREAT flag.*/
            opperm_flags = (opperm | IPC_CREAT);
            break;
        case 2:     /*Set the IPC_EXCL flag.*/
            opperm_flags = (opperm | IPC_EXCL);
            break;
        case 3:     /*Set the IPC_CREAT and IPC_EXCL flags.*/
            opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
        }
        /*Call the msgget system call.*/
        msqid = msgget (key, opperm_flags);

        /*Perform the following if the call is unsuccessful.*/
        if(msqid == -1)
        {
            printf ("\nThe msgget system call failed!\n");
            printf ("The error number = %d\n", errno);
        }
        /*Return the msqid on successful completion.*/
        else
            printf ("\nThe msqid = %d\n", msqid);
        exit(0);
    }
```

CONTROLLING MESSAGE QUEUES

This section gives a detailed description of using the **msgctl()** system call along with an example program that allows all its capabilities to be exercised.

Using Msgctl

The synopsis of the msgctl() UNIX System manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

The msgctl() system call requires three arguments to be passed to it, and it returns an integer value.

On successful completion, a zero value is returned; and when unsuccessful, it returns a -1.

The msqid variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the msgget() system call.

The cmd argument can be replaced by one of the following control commands (flags):

- IPC_STAT—return the status information contained in the associated data structure for the specified msqid, and place it in the data structure pointed to by the *buf pointer in the user memory area

- IPC_SET—for the specified msqid, set the effective user and group identification, operation permissions, and the number of bytes for the message queue

- IPC_RMID—remove the specified msqid along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read permission is required to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using Msgget" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **msgctl()** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the UNIX System manual page for msgctl() (lines 5-9). Note in this program that **errno** is declared as an external variable, and therefore, the errno.h header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **bytes**—used to store the IPC_SET value for the number of bytes in the message queue (msg_qbytes)
- **rtrn**—used to store the return integer value from the system call
- **msqid**—used to store and pass the message queue identifier to the system call
- **command**—used to store the code for the desired control command so that further processing can be performed on it
- **choice**—used to determine what member is to be changed for the IPC_SET control command
- **msqid_ds**—used to receive the specified message queue indentifier's data structure when an IPC_STAT control command is performed
- ***buf**—a pointer passed to the system call that locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set.

Note that the **msqid_ds** data structure in this program (line 16) uses the data structure located in the msg.h header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the *buf pointer is declared to be a pointer to a data structure of the msqid_ds type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier that is stored at the address of the msqid variable (lines 19, 20). This is required for every msgctl() system call.

Then the code for the desired control command must be entered (lines 21-27), and it is stored at the address of the command variable. The code is tested to determine the control command for further processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out regardlessly; also, an error message is displayed and the **errno** variable is printed out (lines 108, 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111-114).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored at the address of the choice variable (line 60). Now, depending on the member picked, the program prompts for the new value (lines 66-95). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending on success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 100-103), and the msqid along with its associated message queue and data structure are removed from the UNIX System. Note that the *buf pointer is not required as an argument to perform this control command, and its value can be zero or NULL. Depending on the success or failure, the program returns the same messages as for the other control commands.

The example program for the msgctl() system call follows. It is suggested that the source program file be named "msgctl.c" and that the executable file be named "msgctl."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the message control, msgctl(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

/*Start of main C language program*/
main()
{
    extern int errno;
    int uid, gid, mode, bytes;
    int rtrn, msqid, command, choice;
    struct msqid_ds msqid_ds, *buf;
    buf = &msqid_ds;

    /*Get the msqid, and command.*/
    printf("Enter the msqid = ");
    scanf("%d", &msqid);
    printf("\nEnter the number for\n");
    printf("the desired command:\n");
    printf("IPC_STAT    =  1\n");
    printf("IPC_SET     =  2\n");
    printf("IPC_RMID    =  3\n");
    printf("Entry       =  ");
    scanf("%d", &command);

    /*Check the values.*/
    printf ("\nmsqid =%d, command = %d\n",
        msqid, command);
        switch (command)
        {
        case 1:     /*Use msgctl() to duplicate
              the data structure for
                      msqid in the msqid_ds area pointed
                      to by buf and then print it out.*/
            rtrn = msgctl(msqid, IPC_STAT,
                buf);
            printf ("\nThe USER ID = %d\n",
                buf->msg_perm.uid);
            printf ("The GROUP ID = %d\n",
                buf->msg_perm.gid);
            printf ("The operation permissions = 0%o\n",
                buf->msg_perm.mode);
            printf ("The msg_qbytes = %d\n",
                buf->msg_qbytes);
            break;
        case 2:     /*Select and change the desired
                      member(s) of the data structure.*/
        /*Get the original data for this msqid
             data structure first.*/
        rtrn = msgctl(msqid, IPC_STAT, buf);

        printf("\nEnter the number for the\n");
        printf("member to be changed:\n");
        printf("msg_perm.uid    = 1\n");
        printf("msg_perm.gid    = 2\n");
        printf("msg_perm.mode   = 3\n");
        printf("msg_qbytes      = 4\n");
        printf("Entry           = ");
        scanf("%d", &choice);
        /*Only one choice is allowed per
          pass as an illegal entry will
             cause repetitive failures until
          msqid_ds is updated with
             IPC_STAT.*/

        switch(choice){
        case 1:
            printf("\nEnter USER ID = ");
            scanf ("%d", &uid);
            buf->msg_perm.uid = uid;
            printf("\nUSER ID = %d\n",
                buf->msg_perm.uid);
            break;
        case 2:
            printf("\nEnter GROUP ID = ");
            scanf("%d", &gid);
            buf->msg_perm.gid = gid;
            printf("\nGROUP ID = %d\n",
                buf->msg_perm.gid);
            break;
        case 3:
            printf("\nEnter MODE = ");
            scanf("%o", &mode);
            buf->msg_perm.mode = mode;
            printf("\nMODE = 0%o\n",
                buf->msg_perm.mode);
            break;
        case 4:
            printf("\nEnter msq_bytes = ");
            scanf("%d", &bytes);
            buf->msg_qbytes = bytes;
            printf("\nmsg_qbytes = %d\n",
                buf->msg_qbytes);
            break;
        }
```

```
            /*Do the change.*/
            rtrn = msgctl(msqid, IPC_SET,
                buf);
            break;
        case 3:     /*Remove the msqid along with its
                          associated message queue
                          and data structure.*/
            rtrn = msgctl(msqid, IPC_RMID, NULL);
        }
        /*Perform the following if the call is unsuccessful.*/
        if(rtrn == -1)
        {
            printf ("\nThe msgctl system call failed!\n");
            printf ("The error number = %d\n", errno);
        }
        /*Return the msqid on successful completion.*/
        else
            printf ("\nMsgctl was successful for msqid = %d\n",
                msqid);
        exit (0);
    }
```

OPERATIONS FOR MESSAGES

This section gives a detailed description of using the **msgsnd()** and **msgrcv()** system calls, along with an example program that allows all their capabilities to be exercised.

Using Msgop

The synopsis of the **msgop()** UNIX System manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Sending a Message

The msgsnd() system call requires four arguments to be passed to it, and msgsnd() returns an integer value.

On successful completion, a zero value is returned; and when unsuccessful, msgsnd() returns a -1.

The msqid argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the msgget() system call.

The msgp argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The msgsz argument specifies the length of the character array in the data structure pointed to by the msgp argument. This is the length of the message. The maximum size of this array is determined by the MSGMAX system tunable parameter.

> ***Note:*** The *msg_qbytes* data structure member can be lowered from MSGMNB by using the *msgctl()* IPC_SET control command, but only the super-user can raise it afterwards.

The msgflg argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (msgflg & IPC_NOWAIT = 0); this would occur if the total amount of bytes allowed on the specified message queue are in use (msg_qbytes or MSGMNB), or the total system-wide amount of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call will fail and return a -1.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using Msgget" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Receiving Messages

The msgrcv() system call requires five arguments to be passed to it, and it returns an integer value.

On successful completion, a value equal to the number of bytes received is returned and when unsuccessful it returns a -1.

The msqid argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the msgget() system call.

The msgp argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The msgsz argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the msgflg argument.

The msgtyp argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

The msgflg argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (msgflg & IPC_NOWAIT = 0); this would occur if there is not a message on the message queue of the desired type (msgtyp) to be received. If the IPC_NOWAIT flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. Msgflg can also specify that the system call fail if the message is longer than the size to be received; this is done by not setting the MSG_NOERROR flag in the msgflg argument (msgflg & MSG_NOERROR = 0). If the MSG_NOERROR flag is set, the message is truncated to the length specified by the msgsz argument of msgrcv().

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using Msgget" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **msgsnd()** and **msgrcv()** system calls to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the UNIX System manual page for **msgop()** (lines 5-9). Note that in this program **errno** is declared as an external variable, and therefore, the errno.h header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **sndbuf**—used as a buffer to contain a message to be sent (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13)

  Note: The *msgbuf1* structure (lines 10-13) is almost an exact duplicate of the *msgbuf* structure contained in the msg.h header file. The only difference is that the character array for *msgbuf1* contains the maximum message size (MSGMAX) for the 3B2 Computer where in *msgbuf* it is set to one (1) to satisfy the compiler. For this reason *msgbuf* cannot be used directly as a template for the user-written program. It is there so you can determine its members.

- **rcvbuf**—used as a buffer to receive a message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13)
- ***msgp**—used as a pointer (line 13) to both the sndbuf and rcvbuf buffers
- **i**—used as a counter for inputing characters from the keyboard, storing them in the array, and keeping track of the message length for the msgsnd() system call; it is also used as a counter to output the received message for the msgrcv() system call
- **c**—used to receive the inputed character from the ''getchar()'' function (line 50)
- **flag**—used to store the code of IPC_NOWAIT for the msgsnd() system call (line 61)

- **flags**—used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the msgrcv() system call (line 117)
- **choice**—used to store the code for sending or receiving (line 30)
- **rtrn**—used to store the return values from all system calls
- **msqid**—used to store and pass the desired message queue identifier for both system calls
- **msgsz**—used to store and pass the size of the message to be sent or received
- **msgflg**—used to pass the value of flag for sending or the value of flags for receiving
- **msgtyp**—used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a msqid_ds data structure is set up in the program (line 21) with a pointer that is initialized to point to it (line 22); this will allow the data structure members that are affected by message operations to be observed. They are observed by using the msgctl() (IPC_STAT) system call to get them for the program to print them out (lines 80-92 and lines 161-168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored at the address of the choice variable (lines 23-30). Depending on the code, the program proceeds as in the following "Msgsnd or Msgrcv" sections.

Msgsnd

When the code is to send a message, the msgp pointer is initialized (line 33) to the address of the send data structure, sndbuf. Next, a message type must be entered for the message; it is stored at the address of the variable msgtyp (line 42), and then (line 43) it is put into the mtype member of the data structure pointed to by msgp.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the mtext array of the data structure (lines 48-51). This will continue until an end of file is recognized, which for the "getchar()" function, is a control-d (^d) immediately following a carriage return (<*CR*>). When this happens, the size of the message is determined by adding one to the i counter (lines 52, 53) as it stored the message beginning in the zero array element of mtext. Keep in mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of msgsz.

The message is immediately echoed from the mtext array of the sndbuf data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the IPC_NOWAIT flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored at the address of the flag variable. If a 1 is entered, IPC_NOWAIT is logically ORed with msgflg; otherwise, msgflg is set to zero.

The msgsnd() system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed and should be zero (lines 73-76).

Every time a message is successfully sent, there are three members of the associated data structure that are updated. They are described as follows:

- **msg_qnum**—represents the total amount of messages on the message queue; it is incremented by one
- **msg_lspid**—contains the Process Identification (PID) number of the last process sending a message; it is set accordingly
- **msg_stime**—contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

For this reason, these members are displayed after every successful message send operation (lines 79-92).

Msgrcv

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The msgp pointer is initialized to the rcvbuf data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested, and it is stored at the address of msqid (lines 100-103).

The message type is requested, and it is stored at the address of msgtyp (lines 104-107).

The code for the desired combination of control flags is requested next, and it is stored at the address of flags (lines 108-117). Depending on the selected combination, msgflg is set accordingly (lines 118-133).

Finally, the number of bytes to be received is requested, and it is stored at the address of msgsz (lines 134-137).

The msgrcv() system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145-148). If successful, a message indicates so, and the number of bytes returned is displayed followed by the received message (lines 153-159).

When a message is successfully received, there are three members of the associated data structure that are updated; they are described as follows:

- **msg_qnum**—contains the number of messages on the message queue; it is decremented by one
- **msg_lrpid**—contains the Process Identification (PID) of the last process receiving a message; it is set accordingly
- **msg_rtime**—contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

The example program for the msgop() system calls follows. It is suggested that the program be put into a source file called ''msgop.c'' and then into an executable file called ''msgop.''

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the message operations, msgop(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

struct msgbuf1 {
    long    mtype;
    char    mtext[8192];
} sndbuf, rcvbuf, *msgp;

/*Start of main C language program*/
main()
{
    extern int errno;
    int i, c, flag, flags, choice;
    int rtrn, msqid, msgsz, msgflg;
    long mtype, msgtyp;
    struct msqid_ds msqid_ds, *buf;
    buf = &msqid_ds;

    /*Select the desired operation.*/
    printf("Enter the corresponding\n");
    printf("code to send or\n");
    printf("receive a message:\n");
    printf("Send              =  1\n");
    printf("Receive           =  2\n");
    printf("Entry             =  ");
    scanf("%d", &choice);

    if(choice == 1) /*Send a message.*/
    {
        msgp = &sndbuf; /*Point to user send structure.*/

        printf("\nEnter the msqid of\n");
        printf("the message queue to\n");
        printf("handle the message = ");
        scanf("%d", &msqid);
```

```
/*Set the message type.*/
printf("\nEnter a positive integer\n");
printf("message type (long) for the\n");
printf("message = ");
scanf("%d", &msgtyp);
msgp->mtype = msgtyp;

/*Enter the message to send.*/
printf("\nEnter a message: \n");

/*A control-d (^d) terminates as
  EOF.*/

/*Get each character of the message
  and put it in the mtext array.*/
for(i = 0; ((c = getchar()) != EOF); i++)
    sndbuf.mtext[i] = c;

/*Determine the message size.*/
msgsz = i + 1;

/*Echo the message to send.*/
for(i = 0; i < msgsz; i++)
    putchar(sndbuf.mtext[i]);

/*Set the IPC_NOWAIT flag if
  desired.*/
printf("\nEnter a 1 if you want the\n");
printf("the IPC_NOWAIT flag set:  ");
scanf("%d", &flag);
if(flag == 1)
    msgflg |= IPC_NOWAIT;
else
    msgflg = 0;

/*Check the msgflg.*/
printf("\nmsgflg = 0%o\n", msgflg);
```

```
            /*Send the message.*/
            rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
            if(rtrn == -1)
            printf("\nMsgsnd failed.  Error = %d\n",
                    errno);
            else {
                /*Print the value of test which
                      should be zero for successful.*/
                printf("\nValue returned = %d\n", rtrn);

                /*Print the size of the message
                  sent.*/
                printf("\nMsgsz = %d\n", msgsz);

                /*Check the data structure update.*/
                msgctl(msqid, IPC_STAT, buf);

                /*Print out the affected members.*/

                /*Print the incremented amount of
                  messages on the queue.*/
                printf("\nThe msg_qnum = %d\n",
                    buf->msg_qnum);
                /*Print the process id of the last sender.*/
                printf("The msg_lspid = %d\n",
                    buf->msg_lspid);
                /*Print the last send time.*/
                printf("The msg_stime = %d\n",
                    buf->msg_stime);
            }

        }

        if(choice == 2)  /*Receive a message.*/
        {
            /*Initialize the message pointer
              to the receive buffer.*/
            msgp = &rcvbuf;

            /*Specify the message queue that contains
                    the desired message.*/
            printf("\nEnter the msqid = ");
            scanf("%d", &msqid);
```

```
/*Specify the specific message on the queue
     by using its type.*/
printf("\nEnter the msgtyp = ");
scanf("%d", &msgtyp);

/*Configure the control flags for the
     desired actions.*/
printf("\nEnter the corresponding code\n");
printf("to select the desired flags: \n");
printf("No flags                   =  0\n");
printf("MSG_NOERROR                =  1\n");
printf("IPC_NOWAIT                 =  2\n");
printf("MSG_NOERROR and IPC_NOWAIT =  3\n");
printf("            Flags          =  ");
scanf("%d", &flags);

switch(flags) {
   /*Set msgflg by ORing it with the appropriate
           flags (constants).*/
case 0:
   msgflg = 0;
   break;
case 1:
   msgflg |= MSG_NOERROR;
   break;
case 2:
   msgflg |= IPC_NOWAIT;
   break;
case 3:
   msgflg |= MSG_NOERROR | IPC_NOWAIT;
   break;
}
```

```
        /*Specify the number of bytes to receive.*/
        printf("\nEnter the number of bytes\n");
        printf("to receive (msgsz) = ");
        scanf("%d", &msgsz);

        /*Check the values for the arguments.*/
        printf("\nmsqid =%d\n", msqid);
        printf("\nmsgtyp = %d\n", msgtyp);
        printf("\nmsgsz = %d\n", msgsz);
        printf("\nmsgflg = 0%o\n", msgflg);

        /*Call msgrcv to receive the message.*/
        rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

        if(rtrn == -1) {
            printf("\nMsgrcv failed. ");
            printf("Error = %d\n", errno);
        }
        else {
            printf ("\nMsgctl was successful\n");
            printf("for msqid = %d\n",
                msqid);

            /*Print the number of bytes received,
              it is equal to the return
              value.*/
            printf("Bytes received = %d\n", rtrn);

            /*Print the received message.*/
            for(i = 0; i<=rtrn; i++)
                putchar(rcvbuf.mtext[i]);
        }
        /*Check the associated data structure.*/
        msgctl(msqid, IPC_STAT, buf);
        /*Print the decremented amount of messages.*/
        printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
        /*Print the process id of the last receiver.*/
        printf("The msg_lrpid = %d\n", buf ->msg_lrpid);
        /*Print the last message receive time*/
        printf("The msg_rtime = %d\n", buf->msg_rtime);
    }
}
```

Chapter 4

SEMAPHORES

Chapter 4

SEMAPHORES

The *semaphore* type of Inter-Process Communication (IPC) allows processes (executing programs) to communicate through the exchange of integer values. Semaphores are created in sets of one or more and are used depending on the results of operations that are performed on them. See the " OPERATIONS ON SEMAPHORES " section of this chapter about the specific operations allowed.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this Chapter.

GENERAL

Before semaphores can be used (operated on or controlled) a uniquely identified **data structure** and **semaphore set** (array) must be created. The unique identifier is called the semaphore identifier (**semid**); it is used to identify or reference a particular data structure and semaphore set. Figure 4-1 illustrates the relationships among the semid, data structure, and semaphore set.

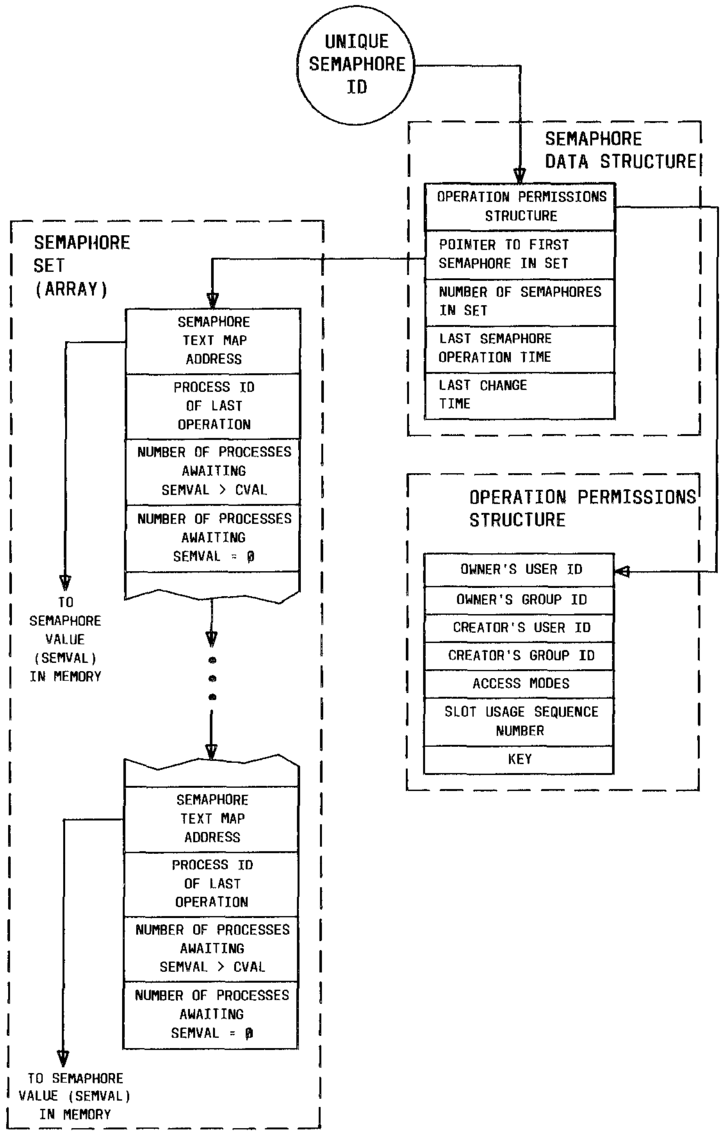

UNIQUE SEMAPHORE ID
SEMAPHORE DATA STRUCTURE
OPERATION PERMISSIONS STRUCTURE
POINTER TO FIRST SEMAPHORE IN SET
NUMBER OF SEMAPHORES IN SET
LAST SEMAPHORE OPERATION TIME
LAST CHANGE TIME
SEMAPHORE SET (ARRAY)
SEMAPHORE TEXT MAP ADDRESS
PROCESS ID OF LAST OPERATION
NUMBER OF PROCESSES AWAITING SEMVAL > CVAL
NUMBER OF PROCESSES AWAITING SEMVAL = Ø
TO SEMAPHORE VALUE (SEMVAL) IN MEMORY
SEMAPHORE TEXT MAP ADDRESS
PROCESS ID OF LAST OPERATION
NUMBER OF PROCESSES AWAITING SEMVAL > CVAL
NUMBER OF PROCESSES AWAITING SEMVAL = Ø
TO SEMAPHORE VALUE (SEMVAL) IN MEMORY
OPERATION PERMISSIONS STRUCTURE
OWNER'S USER ID
OWNER'S GROUP ID
CREATOR'S USER ID
CREATOR'S GROUP ID
ACCESS MODES
SLOT USAGE SEQUENCE NUMBER
KEY

Figure 4-1. Semaphore IPC Organization

The semaphore set contains a predefined amount of structures in an array, one structure for each semaphore in the set. The amount of semaphores (**nsems**) in a semaphore set is user selectable. The following members are in each structure within a semaphore set:

- Semaphore text map address
- Process Identification (PID) performing last operation
- Amount of processes awaiting the semaphore value to become greater than its current value
- Amount of processes awaiting the semaphore value to equal zero.

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set as follows:

- Operation permissions data (operation permissions structure)
- Pointer to first semaphore in the set (array)
- Amount of semaphores in the set
- Last semaphore operation time
- Last semaphore change time.

Note: All include files discussed in this guide are located in the */usr/include* or */usr/include/sys* directories.

The **C** Programming Language data structure definition for the semaphore set (array member) is as follows:

```
struct sem {
        ushort  semval;         /* semaphore text map address */
        short   sempid;         /* pid of last operation */
        ushort  semncnt;        /* # awaiting semval > cval */
        ushort  semzcnt;        /* # awaiting semval = 0 */
};
```

It is located in the #**include** <**sys/sem.h**> header file.

Likewise, the structure definition for the associated semaphore data structure is as follows:

```
struct semid_ds {
   struct ipc_perm sem_perm; /* operation permission struct */
   struct sem *sem_base; /* ptr to first semaphore in set */
   ushort          sem_nsems;  /* # of semaphores in set */
   time_t          sem_otime;  /* last semop time */
   time_t          sem_ctime;  /* last change time */
};
```

It is also located in the #**include** <**sys/sem.h**> header file. Note that the **sem_perm** member of this structure uses **ipc_perm** as a template. Thus, the breakout is shown in Figure 4-1 for the operation permissions data structure.

The ipc_perm data structure is the same for all IPC facilities, and it is located in the #**include** <**sys/ipc.h**> header file. It is shown in the " GENERAL " section of Chapter 3, " MESSAGES."

The **semget** system call is used to do two tasks when only the IPC_CREAT flag is set in the **semflg** argument that it receives:

- To get a new semid and create an associated data structure and semaphore set for it
- To return an existing semid that already has an associated data structure and semaphore set.

The task performed is determined by the value of the **key** argument passed to the semget system call.

For the first task, if the key is not already in use for an existing semid, a new semid is returned with an associated data structure and semaphore set created for it provided no system tunable parameter would be exceeded.

There is also a provision for specifying a key of value zero (0) that is known as the private key (IPC_PRIVATE = 0); when specified, a new semid is always returned with an associated data structure and semaphore set created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the semid is all zeros.

When performing the first task, the process that calls semget becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the " CONTROLLING SEMAPHORES " section in this chapter. The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a semid exists for the key specified, the value of the existing semid is returned. If it is not desired to have an existing semid returned, a control command (IPC_EXCL) can be specified (set) in the semflg argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (nsems) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use 0 for nsems. The details of using this system call are discussed in the " Using Semget " section of this chapter.

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop**] and semaphore control [**semctl**] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to do these operations. It is called semop. Refer to the " OPERATIONS ON SEMAPHORES " section in this chapter for details of this system call.

Semaphore control is done by using the semctl system call. These control operations permit you to control the semaphore facility in the following ways:

- To return the value of a semaphore.
- To set the value of a semaphore.
- To return the Process Identifier (PID) of the last process performing an operation on a semaphore set.
- To return the number of processes waiting for a semaphore value to become greater than its current value.
- To return the number of processes waiting for a semaphore value to equal zero.
- To get all semaphore values in a set and place them in an array in user memory.
- To set all semaphore values in a semaphore set from an array of values in user memory.
- To place all data structure member values, status, of a semaphore set into user memory area.
- To change operation permissions for a semaphore set.
- To remove a particular semid from the UNIX System along with its associated data structure and semaphore set.

Refer to the " CONTROLLING SEMAPHORES " section in this chapter for details of the semctl system call.

GETTING SEMAPHORES

This section contains a detailed description of using the **semget** system call along with an example program illustrating its use.

Using Semget

The synopsis of the **semget** is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>

int  semget (key, nsems, semflg)
key_t  key;
int nsems, semflg;
```

All these include files are located in the **/usr/include/sys** directory of the UNIX System.

The following line in the synopsis:

```
int semget (key, nsems, semflg)
```

informs you that semget is a function with three *formal* arguments that returns an integer *type* value, on successful completion (semid). The next two lines:

```
key_t  key;
int nsems, semflg;
```

declare the types of the formal arguments. Key_t is declared by a *typedef* in the types.h header file to be a long integer. Therefore key, nsems, and semflg are integers (*int*) that occupy 32 bits each in the 3B2 Computer.

The integer returned from this system call on successful completion is the semaphore set identifier (semid) that was discussed in the " GENERAL " section of this chapter.

As declared, the process calling the semget system call must supply three *actual* arguments to be passed to the formal key, nsems, and semflg arguments.

The value passed to key must be a unique integer type hexadecimal value or zero (IPC_PRIVATE = 0) if a new semid with an associated data structure and semaphore set is desired; it must be an existing key to return its semid. This is true when only the IPC_CREAT flag is set in the semflg argument.

Unique keys can be determined in several ways. The **STDIPC**, standard inter-process communication package, subroutine is one method to generate unique keys to avoid undesired interference between processes. Another way could be to use the **makekey** command, see the manual pages for the **STDIPC** and **makekey** commands. Picking a key at random is also possible but less desirable. If the key is IPC_PRIVATE, only the owner/creator process usually uses the facility.

The value passed to the semflg argument must be an integer type octal value and will specify the following:

- Access permissions
- Execution modes
- Control fields (commands).

Access permissions determine the read/alter attributes and execution modes determine the user/group/other attributes of the semflg argument. They are collectively referred to as " operation permissions." Figure 4-2 reflects the numeric values for the valid operation permissions codes.

OPERATION PERMISSIONS	NUMERIC VALUE
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

Figure 4-2. Operation Permissions Codes

A specific numeric value is derived by adding the numeric values for the operation permissions desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). These values are represented in octal. There are constants located in the sem.h header file that can be used for the user (OWNER). They are as follows:

SEM_R	0400
SEM_A	0200

Control commands are predefined constants (represented by all uppercase letters). Figure 4-3 contains the names of the constants that apply to the semget system call along with their values. They are also referred to as flags and are defined in the ipc.h header file.

CONTROL COMMAND	VALUE
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 4-3. Control Commands (Flags)

The value for semflg is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is done by bitwise ORing (¦) them with the operation permissions; the bit positions and values for the control commands to those of the operation permissions make this possible. It is illustrated as follows:

```
                          OCTAL VALUE       BINARY VALUE

  IPC_CREAT         =     0 1 0 0 0     0 000 001 000 000 000
¦ Read by User      =     0 0 4 0 0     0 000 000 100 000 000

  semflg            =     0 1 4 0 0     0 000 001 100 000 000
```

The semflg value can be easily set by using the names of the flags with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT ¦ 0400));

semid = semget (key, nsems, (IPC_CREAT ¦ IPC_EXCL ¦ 0400));
```

As specified by the **semget** success or failure of this system call depends on the *actual* argument values for key, nsems, semflg or system tunable parameters. The system call will attempt to return a new semid if a following condition is true:

- Key is equal to IPC_PRIVATE (0)

- Key does not already have a semid associated with it, and (semflg & IPC_CREAT) is " true " (not zero).

The key argument can be set to IPC_PRIVATE in the following ways:

```
            semid = semget (IPC_PRIVATE, nsems, semflg);

                               OR

              semid = semget ( 0, nsems, semflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified.

Exceeding the SEMMNI, SEMMNS, or SEMMSL system tunable parameters will cause a failure regardlessly. The SEMMNI system tunable parameter determines the maximum amount of unique semaphore sets (semid's) in the UNIX System. The SEMMNS system tunable parameter determines the maximum amount of semaphores in all semaphore sets system wide. The SEMMSL system tunable parameter determines the maximum amount of semaphores in each semaphore set.

The second condition is satisfied if the value for key is not already associated with a semid, and the bitwise ANDing of semflg and IPC_CREAT is " true " (not zero). This means that the key is unique (not in use) within the UNIX System for this facility type and that the IPC_CREAT flag is set (semflg ¦ IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
           semflg = x 1 x x x   (x = don't care)
      & IPC_CREAT = 0 1 0 0 0

           result = 0 1 0 0 0   (not zero)
```

Since the result is not zero, the flag is set or " true ." SEMMNI, SEMMNS, and SEMMSL apply here also, just as for condition one.

IPC_EXCL is another control command used with IPC_CREAT to exclusively have the system call fail if, and only if, a semid exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) semid when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new semid is returned if the system call is successful. Any value for semflg returns a new semid if the key equals zero (IPC_PRIVATE) and no system tunable parameters are exceeded.

Refer to the **semget** manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained in the manual page.

Example Program

P The example program in this section is a menu driven program that allows all possible combinations of using the **semget** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the manual page for semget (lines 4-8). Note that the errno.h header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

- **key**—used to pass the value for the desired key
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **semflg** argument
- **semid**—used for returning the semaphore set identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal key, an octal operation permissions code, and the control command combinations (flags) that are selected from a menu (lines 15-32).

> ***Note:*** All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the opperm_flags variable (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57), and its value is stored at the address of nsems.

The system call is made next, and the result is stored at the address of the semid variable (lines 60, 61).

Since the semid variable now contains a valid semaphore set identifier or the error code (-1), it is tested to see if an error occurred (line 63). If semid equals -1, a message shows that an error resulted and the external **errno** variable is displayed (lines 65, 66). Remember that the external **errno** variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the semget system call follows. It is suggested that the source program file be named " semget.c " and that the executable file be named " semget."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the semaphore get, semget(),
**system call capabilities.*/

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <errno.h>

/*Start of main C language program*/
main()
{
    key_t key;      /*declare as long integer*/
    int opperm, flags, nsems;
    int semid, opperm_flags;

    /*Enter the desired key*/
    printf("\nEnter the desired key in hex = ");
    scanf("%x", &key);

    /*Enter the desired octal operation
         permissions.*/
    printf("\nEnter the operation\n");
    printf("permissions in octal = ");
    scanf("%o", &opperm);

    /*Set the desired flags.*/
    printf("\nEnter corresponding number to\n");
    printf("set the desired flags:\n");
    printf("No flags                   = 0\n");
    printf("IPC_CREAT                  = 1\n");
    printf("IPC_EXCL                   = 2\n");
    printf("IPC_CREAT and IPC_EXCL     = 3\n");
    printf("            Flags          = ");
    /*Get the flags to be set.*/
    scanf("%d", &flags);

    /*Error checking (debugging)*/
    printf ("\nkey =0x%x, opperm = 0%o,
    flags = 0%o\n",
        key, opperm, flags);
    /*Incorporate the control fields (flags) with
         the operation permissions.*/
    switch (flags)
    {
    case 0:    /*No flags are to be set.*/
        opperm_flags = (opperm | 0);
        break;
    case 1:    /*Set the IPC_CREAT flag.*/
        opperm_flags = (opperm | IPC_CREAT);
        break;
    case 2:    /*Set the IPC_EXCL flag.*/
        opperm_flags = (opperm | IPC_EXCL);
        break;
    case 3: /*Set the IPC_CREAT and IPC_EXCL
                 flags.*/
        opperm_flags = (opperm | IPC_CREAT |
        IPC_EXCL);
    }

    /*Get the number of semaphores for this set.*/
    printf("\nEnter the number of\n");
    printf("desired semaphores for\n");
    printf("this set (25 max) = ");
    scanf("%d", &nsems);
    /*Check the entry.*/
    printf("\nNsems = %d\n", nsems);
    /*Call the semget system call.*/
    semid = semget(key, nsems, opperm_flags);
    /*Perform the following if the call is
    unsuccessful.*/
    if (semid == -1)
    {
      printf("The semget system call failed!\n");
      printf("The error number = %d\n", errno);
    }
    /*Return the semid on successful completion.*/
    else
        printf("\nThe semid = %d\n", semid);
    exit(0);
}
```

CONTROLLING SEMAPHORES

This section contains a detailed description of using the **semctl** system call along with an example program that allows all its capabilities to be exercised.

Using Semctl

The synopsis of the **semctl** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
        int val;
        struct semid_ds *buf;
        ushort array[];
} arg;
```

The semctl system call requires four arguments to be passed to it, and it returns an integer value.

The semid argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the semget system call.

The semnum argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set.

> ***Note:*** When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The cmd argument can be replaced by a following control command (flags):

- GETVAL—return the value of a single semaphore within a semaphore set.
- SETVAL—set the value of a single semaphore within a semaphore set.
- GETPID—return the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set.
- GETNCNT—return the number of processes waiting for the value of a particular semaphore to become greater than its current value.
- GETZCNT—return the number of processes waiting for the value of a particular semaphore to be equal to zero.
- GETALL—return the values for all semaphores in a semaphore set.
- SETALL—set all semaphore values in a semaphore set.
- IPC_STAT—return the status information contained in the associated data structure for the specified semid, and place it in the data structure pointed to by the *buf pointer in the user memory area; **arg.buf** is the union member that contains the value of buf.
- IPC_SET—for the specified semaphore set (semid), set the effective user/group identification and operation permissions.
- IPC_RMID—remove the specified (semid) semaphore set along with its associated data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to do an IPC_SET or IPC_RMID control command. Read/alter permission is required as applicable for the other control commands.

The arg argument is used to pass the system call the appropriate union member for the control command to be performed:

- arg.val
- arg.buf
- arg.array

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the " Using Semget " section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **semctl** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the manual page for semctl (lines 5-9). Note that in this program **errno** is declared as an external variable, and therefore the errno.h header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. Those declared for this program and their purpose are as follows:

- **semid_ds**—used to receive the specified semaphore set identifier's data structure when an IPC_STAT control command is performed
- **c**—used to receive the inputed values from the " scanf " function (line 117) when performing a SETALL control command
- **i**—used as a counter to increment through the union arg.array when displaying the semaphore values for a GETALL (lines 97-99) control command, and when initializing the arg.array when performing a SETALL (lines 115-119) control command
- **length**—used as a variable to test for the number of semaphores in a set against the i counter variable (lines 97, 115)
- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer from the system call that depends on the control command or a -1 when unsuccessful
- **semid**—used to store and pass the semaphore set identifier to the system call
- **semnum**—used to store and pass the semaphore number to the system call

- **cmd**—used to store the code for the desired control command so that further processing can be performed on it

- **choice**—used to determine what member (uid, gid, mode) for the IPC_SET control command that is to be changed

- **arg.val**—used to pass the system call a value to set (SETVAL) or to store (GETVAL) a value returned from the system call for a single semaphore (union member)

- **arg.buf**—a pointer passed to the system call that locates the data structure in the user memory area where the IPC_STAT control command is to place its return values, or where the IPC_SET command gets the values to set (union member)

- **arg.array**—used to store the set of semaphore values when getting (GETALL) or initializing (SETALL) (union member).

Note that the **semid_ds** data structure in this program (line 14) uses the data structure located in the sem.h header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The arg union (lines 18-22) serves three purposes in one. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing them as if they were regular structure members. Note that the array is declared to have 25 elements (0 through 24). This number corresponds to the maximum amount of semaphores allowed per set (SEMMSL), a system tunable parameter.

The next important program aspect to observe is that although the *buf pointer member (arg.buf) of the union is declared to be a pointer to a data structure of the semid_ds type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

Now that all the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier that is stored at the address of the semid variable (lines 25-27). This is required for all semctl system calls.

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored at the address of the cmd variable. The code is tested to determine the control command for further processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored at the address of the semnum variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message shows this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external **errno** variable (lines 191-193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored at the address of the semnum variable (line 58). Next, a message prompts for the value to what the semaphore is to be set, and it is stored as the arg.val member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64-67), and the PID of the process performing the last operation is displayed. Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored at the address of the semnum variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored at the address of the semnum variable (line 82). Then the system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the number of semaphores in the set (line 91). Next, the system call is made and, on success, the arg.array union member contains the values of the semaphore set (line 96). Now, a loop is entered that displays each element of the arg.array from zero to one less than the value of length (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending on success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop that takes values from the keyboard and initializes the arg.array union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120-122). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out regardlessly; also an error message is displayed, and the **errno** variable is printed out (lines 191, 192).

If the IPC_SET control command is selected (code 8), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the semctl system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored at the address of the choice variable (line 154). Now, depending on the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the IPC_RMID control command (code 10) is selected, the system call is performed (lines 183-185). The semid along with its associated data structure and semaphore set is removed from the UNIX System. Depending on success or failure, the program returns the same messages as for the other control commands.

The example program for the semctl system call follows. It is suggested that the source program file be named " semctl.c " and that the executable file be named " semctl."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the semaphore control, semctl(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include     <stdio.h>
#include     <sys/types.h>
#include     <sys/ipc.h>
#include     <sys/sem.h>

/*Start of main C language program*/
main()
{
    extern int errno;
    struct semid_ds semid_ds;
    int c, i, length;
    int uid, gid, mode;
    int retrn, semid, semnum, cmd, choice;
    union semun  {
        int val;
        struct semid_ds *buf;
        ushort array[25];
    } arg;

    /*Initialize the data structure pointer.*/
    arg.buf = &semid_ds;

    /*Enter the semaphore ID.*/
    printf("Enter the semid = ");
    scanf("%d", &semid);

    /*Choose the desired command.*/
    printf("\nEnter the number for\n");
    printf("the desired cmd:\n");
    printf("GETVAL      =  1\n");
    printf("SETVAL      =  2\n");
    printf("GETPID      =  3\n");
    printf("GETNCNT     =  4\n");
    printf("GETZCNT     =  5\n");
    printf("GETALL      =  6\n");
    printf("SETALL      =  7\n");
    printf("IPC_STAT    =  8\n");
    printf("IPC_SET     =  9\n");
    printf("IPC_RMID    =  10\n");
    printf("Entry       =  ");
    scanf("%d", &cmd);
```

```
        /*Check entries.*/
        printf ("\nsemid =%d, cmd = %d\n\n",
            semid, cmd);

        /*Set the command and do the call.*/
        switch (cmd)
        {
        case 1: /*Get a specified value.*/
            printf("\nEnter the semnum = ");
            scanf("%d", &semnum);
            /*Do the system call.*/
            retrn = semctl(semid, semnum, GETVAL, 0);
            printf("\nThe semval = %d\n", retrn);
            break;
        case 2: /*Set a specified value.*/
            printf("\nEnter the semnum = ");
            scanf("%d", &semnum);
            printf("\nEnter the value = ");
            scanf("%d", &arg.val);
            /*Do the system call.*/
            retrn = semctl(semid, semnum, SETVAL,
              arg.val);
            break;
        case 3: /*Get the process ID.*/
            retrn = semctl(semid, 0, GETPID, 0);
            printf("\nThe sempid = %d\n", retrn);
            break;
        case 4: /*Get the number of processes
              waiting for the semaphore to
              become greater than its current
                      value.*/
           printf("\nEnter the semnum = ");
           scanf("%d", &semnum);
           /*Do the system call.*/
           retrn = semctl(semid, semnum, GETNCNT, 0);
           printf("\nThe semncnt = %d", retrn);
           break;
       case 5: /*Get the number of processes
             waiting for the semaphore
                     value to become zero.*/
           printf("\nEnter the semnum = ");
           scanf("%d", &semnum);
           /*Do the system call.*/
           retrn = semctl(semid, semnum, GETZCNT, 0);
           printf("\nThe semzcnt = %d", retrn);
           break;
        case 6: /*Get all the semaphores.*/

            /*Get the number of semaphores in
              the semaphore set.*/
            retrn = semctl(semid, 0, IPC_STAT, arg.buf);
            length = arg.buf->sem_nsems;
            if(retrn == -1)
                goto ERROR;
            /*Get and print all semaphores in the
              specified set.*/
            retrn = semctl(semid, 0, GETALL, arg.array);
            for (i = 0; i < length; i++)
            {
                printf("%d", arg.array[i]);
                /*Separate each
                  semaphore.*/
                printf("%c", ' ');
            }
            break;
        case 7: /*Set all semaphores in the set.*/

            /*Get the number of semaphores in
              the set.*/
            retrn = semctl(semid, 0, IPC_STAT, arg.buf);
            length = arg.buf->sem_nsems;
            printf("Length = %d\n", length);
            if(retrn == -1)
                goto ERROR;
            /*Set the semaphore set values.*/
            printf("\nEnter each value:\n");
            for(i = 0; i < length ; i++)
          {
              scanf("%d", &c);
              arg.array[i] = c;
          }
          /*Do the system call.*/
          retrn = semctl(semid, 0, SETALL, arg.array);
          break;
```

```
case 8: /*Get the status for the semaphore
              set.*/

    /*Get the current status values and
      print them out.*/
    retrn = semctl(semid, 0, IPC_STAT, arg.buf);
    printf ("\nThe USER ID = %d\n",
        arg.buf->sem_perm.uid);
    printf ("The GROUP ID = %d\n",
        arg.buf->sem_perm.gid);
    printf ("The operation permissions = 0%o\n",
        arg.buf->sem_perm.mode);
    printf ("The number of semaphores in set = %d\n",
        arg.buf->sem_nsems);
    printf ("The last semop time = %d\n",
        arg.buf->sem_otime);
    printf ("The last change time  = %d\n",
        arg.buf->sem_ctime);
    break;
case 9:    /*Select and change the desired
              member of the data structure.*/

    /*Get the current status values.*/
    retrn = semctl(semid, 0, IPC_STAT, arg.buf);
    if(retrn == -1)
        goto ERROR;
    /*Select the member to change.*/
    printf("\nEnter the number for the\n");
    printf("member to be changed:\n");
    printf("sem_perm.uid    = 1\n");
    printf("sem_perm.gid    = 2\n");
    printf("sem_perm.mode   = 3\n");
    printf("Entry           = ");
       scanf("%d", &choice);

       switch(choice){
       case 1: /*Change the user ID.*/
           printf("\nEnter USER ID = ");
           scanf ("%d", &uid);
           arg.buf->sem_perm.uid = uid;
           printf("\nUSER ID = %d\n",
               arg.buf->sem_perm.uid);
           break;
        case 2: /*Change the group ID.*/
            printf("\nEnter GROUP ID = ");
            scanf("%d", &gid);
            arg.buf->sem_perm.gid = gid;
            printf("\nGROUP ID = %d\n",
                arg.buf->sem_perm.gid);
            break;
        case 3: /*Change the mode portion of
              the operation
                          permissions.*/
            printf("\nEnter MODE = ");
            scanf("%o", &mode);
            arg.buf->sem_perm.mode = mode;
            printf("\nMODE = 0%o\n",
                arg.buf->sem_perm.mode);
            break;
        }
     /*Do the change.*/
     retrn = semctl(semid, 0, IPC_SET, arg.buf);
        break;
    case 10:    /*Remove the semid along with its
                  data structure.*/
        retrn = semctl(semid, 0, IPC_RMID, 0);
    }
    /*Perform the following if the call is unsuccessful.*/
    if(retrn == -1)
    {
ERROR:
    printf ("\n\nThe semctl system call
    failed!\n");

    printf ("The error number = %d\n", errno);
    exit(0);
 }
    printf ("\n\nThe semctl system call was successful\n");
    printf ("for semid = %d\n", semid);
    exit (0);
}
```

OPERATIONS ON SEMAPHORES

This section contains a detailed description of using the **semop** system call along with an example program that allows all its capabilities to be exercised.

Using Semop

The synopsis of the **semop** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

The semop system call requires three arguments to be passed to it, and it returns an integer value.

On successful completion, a zero value is returned and when unsuccessful it returns a -1.

The semid argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the semget system call.

The sops argument is a pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- The semaphore number
- The operation to be performed
- The control command (flags).

The **sops declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. Sembuf is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the **#include <sys/sem.h>** header file.

The nsops argument specifies the length of the array (the number of structures in the array). The maximum size of this array is determined by the SEMOPM system tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each semop system call.

The semaphore number determines the particular semaphore within the set on what operation is to be performed.

The operation to be performed is determined by the following:

- A positive integer value means to increment the semaphore value by its value.
- A negative integer value means to decrement the semaphore value by its value.
- A value of zero means to test if the semaphore is equal to zero.

The following operation commands (flags) can be used:

- IPC_NOWAIT—this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for what IPC_NOWAIT is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not. See " blocking/nonblocking semaphore operations " in Chapter 2.

- SEM_UNDO—this operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the IPC_NOWAIT flag set. That is, the blocked operation waits until it can do its operation; and when it and all succeeding operations are successful, all operations with the SEM_UNDO flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is done by using an array of adjust values for the operations that are to be undone when the blocked operation and all further operations are successful.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **semop** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that the system calls provide.

This program begins by including the required header files as specified by the manual page for **msgop** (lines 5-9). Note that in this program **errno** is declared as an external variable, and therefore, the errno.h header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since the declarations are local to the program. The variables declared for this program and their purpose are as follows:

- **sembuf**[10]—used as an array buffer (line 14) to contain a maximum of ten sembuf type structures; ten equals SEMOPM, the maximum amount of operations on a semaphore set for each semop system call
- ***sops**—used as a pointer (line 14) to sembuf[10] for the system call and for accessing the structure members within the array
- **rtrn**—used to store the return values from the system call
- **flags**—used to store the code of the IPC_NOWAIT or SEM_UNDO flags for the semop system call (line 60)
- **i**—used as a counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79)
- **nsops**—used to specify the number of semaphore operations for the system call—must be less than or equal to SEMOPM
- **semid**—used to store the desired semaphore set identifier for the system call.

First, the program prompts for a semaphore set identifier that the system call is to do operations on (lines 19-22). Semid is stored at the address of the semid variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored at the address of the nsops variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (nsops) to be performed for the system call, so nsops is tested against the i counter for loop control. Note that sops is used as a pointer to each element (structure) in the array, and sops is incremented just like i. Sops is then used to point to each member in the structure for setting them.

After the array is initialized, all its elements are printed out for feedback (lines 78-85).

The sops pointer is set to the address of the array (lines 86, 87). Sembuf could be used directly, if desired, instead of sops in the system call.

The system call is made (line 89), and depending on success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the semctl GETALL control command.

The example program for the semop system call follows. It is suggested that the source program file be named " semop.c " and that the executable file be named " semop."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the semaphore operations, semop(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
/*Start of main C language program*/
main()
{
    extern int errno;
    struct sembuf sembuf[10], *sops;
    char string[];
    int retrn, flags, sem_num, i, semid;
    unsigned nsops;
    sops = sembuf; /*Pointer to array sembuf.*/

    /*Enter the semaphore ID.*/
    printf("\nEnter the semid of\n");
    printf("the semaphore set to\n");
    printf("be operated on = ");
    scanf("%d", &semid);
    printf("\nsemid = %d", semid);

    /*Enter the number of operations.*/
    printf("\nEnter the number of semaphore\n");
    printf("operations for this set = ");
    scanf("%d", &nsops);
    printf("\nnosops = %d", nsops);

    /*Initialize the array for the
      number of operations to be performed.*/
    for(i = 0; i < nsops; i++, sops++)
    {

        /*This determines the semaphore in
          the semaphore set.*/
     printf("\nEnter the semaphore\n");
     printf("number (sem_num) = ");
     scanf("%d", &sem_num);
     sops->sem_num = sem_num;
     printf("\nThe sem_num = %d", sops->sem_num);
```

```
        /*Enter a (-)number to decrement,
          an unsigned number (no +) to increment,
          or zero to test for zero.  These values
          are entered into a string and converted
          to integer values.*/
        printf("\nEnter the operation for\n");
        printf("the semaphore (sem_op) = ");
        scanf("%s", string);
        sops->sem_op = atoi(string);
        printf("\nsem_op = %d\n", sops->sem_op);

        /*Specify the desired flags.*/
        printf("\nEnter the corresponding\n");
        printf("number for the desired\n");
        printf("flags:\n");
        printf("No flags                    = 0\n");
        printf("IPC_NOWAIT                  = 1\n");
        printf("SEM_UNDO                    = 2\n");
        printf("IPC_NOWAIT and SEM_UNDO     = 3\n");
        printf("            Flags           = ");
        scanf("%d", &flags);

        switch(flags)
        {
        case 0:
            sops->sem_flg = 0;
            break;
        case 1:
            sops->sem_flg = IPC_NOWAIT;
            break;
        case 2:
            sops->sem_flg = SEM_UNDO;
            break;
        case 3:
            sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
            break;
        }
        printf("\nFlags = 0%o\n", sops->sem_flg);


       }
```

```
    /*Print out each structure in the array.*/
    for(i = 0; i < nsops; i++)
    {
        printf("\nsem_num = %d\n",
          sembuf[i].sem_num);
        printf("sem_op = %d\n", sembuf[i].sem_op);
        printf("sem_flg = %o\n", sembuf[i].sem_flg);
        printf("%c", ' ');
    }

       sops = sembuf; /*Reset the pointer to
                        sembuf[0].*/

       /*Do the semop system call.*/
       retrn = semop(semid, sops, nsops);
       if(retrn == -1) {
           printf("\nSemop failed. ");
           printf("Error = %d\n", errno);
       }
       else {
        printf ("\nSemop was successful\n");
        printf("for semid = %d\n", semid);

        printf("Value returned = %d\n", retrn);
       }
  }
```

Chapter 5

SHARED MEMORY

Chapter 5

SHARED MEMORY

The *shared memory* type of Inter-Process Communication (IPC) allows processes (executing programs) to communicate by explicitly setting up access to a common virtual address space. The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the UNIX System at any point in time.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this Chapter.

GENERAL

Before sharing of memory can be realized, a uniquely identified **shared memory segment** and **data structure** must be created. The unique identifier created is called the shared memory identifier (**shmid**); it is used to identify or reference the associated data structure. Figure 5-1 illustrates the relationships among the shmid, segment descriptor, and data structure.

The data structure includes the following for each shared memory segment:

- Operation permissions
- Segment size
- Segment descriptor
- Process identification performing last operation
- Process identification of creator
- Current amount of processes attached
- In memory the amount of processes attached
- Last attach time
- Last detach time
- Last change time.

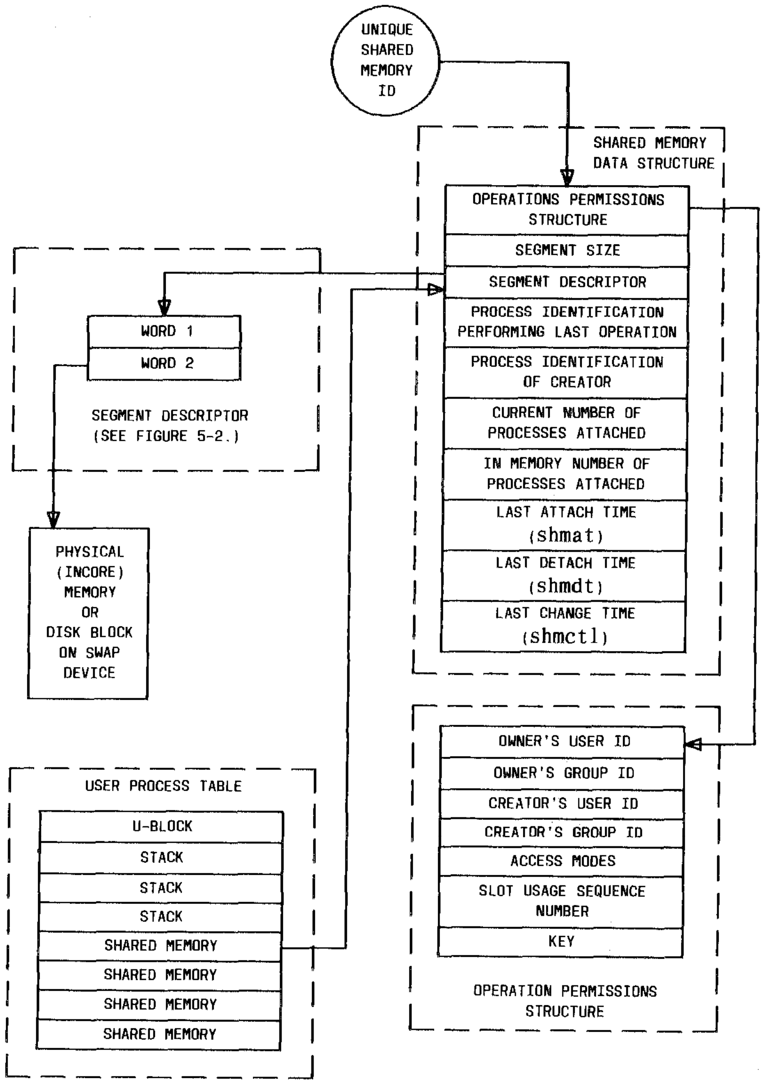

UNIQUE SHARED MEMORY ID
SHARED MEMORY DATA STRUCTURE
OPERATIONS PERMISSIONS STRUCTURE
SEGMENT SIZE
SEGMENT DESCRIPTOR
PROCESS IDENTIFICATION PERFORMING LAST OPERATION
PROCESS IDENTIFICATION OF CREATOR
CURRENT NUMBER OF PROCESSES ATTACHED
IN MEMORY NUMBER OF PROCESSES ATTACHED
LAST ATTACH TIME (shmat)
LAST DETACH TIME (shmdt)
LAST CHANGE TIME (shmctl)
WORD 1
WORD 2
SEGMENT DESCRIPTOR (SEE FIGURE 5-2.)
PHYSICAL (INCORE) MEMORY OR DISK BLOCK ON SWAP DEVICE
USER PROCESS TABLE
U-BLOCK
STACK
STACK
STACK
SHARED MEMORY
SHARED MEMORY
SHARED MEMORY
SHARED MEMORY
OWNER'S USER ID
OWNER'S GROUP ID
CREATOR'S USER ID
CREATOR'S GROUP ID
ACCESS MODES
SLOT USAGE SEQUENCE NUMBER
KEY
OPERATION PERMISSIONS STRUCTURE

Figure 5-1. Shared Memory IPC Organization

The C Programming Language data structure definition for the shared memory segment data structure is as follows:

```
/*
**    There is a shared mem ID data structure for
**    each segment in the system.
*/

struct shmid_ds {
    struct ipc_perm     shm_perm;       /* oper permission struct */
    int                 shm_segsz;      /* segment size */
    sde_t               shm_seg;        /* segment descriptor */
    ushort              shm_lpid;       /* pid of last shmop */
    ushort              shm_cpid;       /* pid of creator */
    ushort              shm_nattch;     /* current # attached */
    ushort              shm_cnattch;    /* in memory # attached */
    time_t              shm_atime;      /* last shmat time */
    time_t              shm_dtime;      /* last shmdt time */
    time_t              shm_ctime;      /* last change time */
};
```

Note that the **shm_perm** member of this structure uses **ipc_perm** as a template. Thus, the breakout is shown in Figure 5-1 for the operation permissions data structure.

The ipc_perm data structure is the same for all IPC facilities, and it is located in the #**include** <**sys/ipc.h**> header file. It is shown in the "GENERAL" section of Chapter 3, "MESSAGES."

The **shm_seg** member of this data structure is defined by a **typedef** in the **/usr/include/sys/types.h** file. The definition is as follows:

```
typedef struct _SDE {                  /*  segment descriptor  */
/*                                                             */
/*  +--------+-------------+--+--------+ +-------------------------------+  */
/*  | access  | maxoff  | | flags  | | address                 |  */
/*  +--------+-------------+--+--------+ +-------------------------------+  */
/*       8          14       2     8                  32                     */
/*                                                                          */
/*                                       +-------------------------+-+-+-+  */
/*                          (V0):        |                         |N |W |S |  */
/*                                       +-------------------------+-+-+-+  */
/*                                                   29              1 1 1  */
                                                                         */
  unsigned int access    :   8;                   /* Access rights      */
  unsigned int maxoff    :  14;                   /* Segment's max offset */
  unsigned int           :   2;                           /* Reserved          */
  unsigned int flags     :   8;                           /* Descriptor flags */
  union {
          unsigned int address;
          struct {
            unsigned int          : 29;
            unsigned int lock     :  1;           /* "N" bit  */
            unsigned int shmswap  :  1;               /* "W" bit  */
            unsigned int alloc    :  1;           /* "S" bit  */
          } V0;
    } wd2;
} sde_t;             /* old name: SDE      */
```

Figure 5-2 represents the shared memory segment descriptor pictorially.

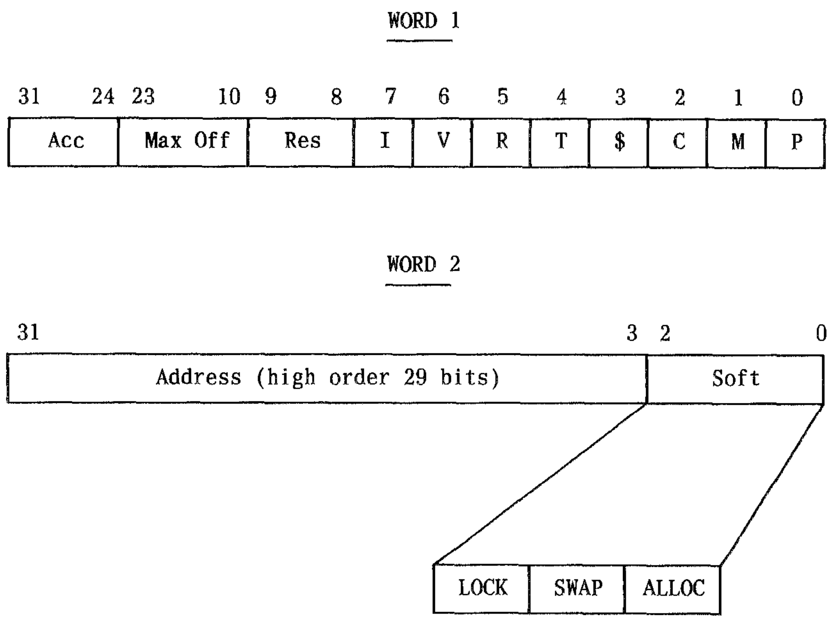


Figure 5-2. Shared Memory Segment Descriptor

Figure 5-3 is a table that shows the shared memory state information.

SHARED MEMORY STATES			
LOCK BIT	SWAP BIT	ALLOCATED BIT	IMPLIED STATE
0	0	0	Unallocated Segment
0	0	1	Incore
0	1	0	Unused
0	1	1	On Disk
1	0	1	Locked Incore
1	1	0	Unused
1	0	0	Unused
1	1	1	Unused

Figure 5-3. Shared Memory State Information

The implied states of Figure 5-3 are as follows:

- **Unallocated Segment**—the segment associated with this segment descriptor has not been allocated for use.

- **Incore**—the shared segment associated with this descriptor has been allocated for use. Therefore, the segment does exist and is currently resident in memory.

- **On Disk**—the shared segment associated with this segment descriptor is currently resident on the swap device.

- **Locked Incore**—the shared segment associated with this segment descriptor is currently locked in memory and will not be a candidate for swapping until the segment is unlocked. Only the super-user may lock and unlock a shared segment.

- **Unused**—this state is currently unused and should never be encountered by the normal user in shared memory handling.

The **shmget** system call is used to do two tasks when only the IPC_CREAT flag is set in the **shmflg** argument that it receives:

- To get a new shmid and create an associated shared memory segment data structure for it
- To return an existing shmid that already has an associated shared memory segment data structure.

The task performed is determined by the value of the **key** argument passed to the shmget system call.

For the first task, if the key is not already in use for an existing shmid, a new shmid is returned with an associated shared memory segment data structure created for it provided no system tunable parameters would be exceeded.

There is also a provision for specifying a key of value zero that is known as the private key (IPC_PRIVATE = 0); when specified, a new shmid is always returned with an associated shared memory segment data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the shmid is all zeros.

For the second task, if a shmid exists for the key specified, the value of the existing shmid is returned. If it is not desired to have an existing shmid returned, a control command (IPC_EXCL) can be specified (set) in the shmflg argument passed to the system call. The details of using this system call are discussed in the "Using Shmget" section of this chapter.

When performing the first task, the process that calls shmget becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "CONTROLLING SHARED MEMORY" section in this chapter. The creator of the shared memory segment also determines the initial operation permissions for it.

Once an uniquely identified shared memory segment data structure is created, shared memory segment operations [**shmop**] and control [**shmctl**] can be used.

> ***Note:*** *Shmop* is not a system call.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat** and **shmdt**. Refer to the "OPERATIONS FOR SHARED MEMORY" section in this chapter for details of these system calls.

Shared memory segment control is done by using the shmctl system call. It permits you to control the shared memory facility in the following ways:

- To determine the associated data structure status for a shared memory segment (shmid)
- To change operation permissions for a shared memory segment
- To remove a particular shmid from the UNIX System along with its associated shared memory segment data structure
- To lock a shared memory segment in memory
- To unlock a shared memory segment.

Refer to the "CONTROLLING SHARED MEMORY" section in this chapter for details of the shmctl system call.

GETTING SHARED MEMORY SEGMENTS

This section gives a detailed description of using the **shmget** system call along with an example program illustrating its use.

Using Shmget

The synopsis of the **shmget** is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

int  shmget (key, size, shmflg)
key_t  key;
int size, shmflg;
```

All these include files are located in the **/usr/include/sys** directory of the UNIX System.

The following line in the synopsis:

```
int shmget (key, size, shmflg)
```

informs you that shmget is a function with three *formal* arguments that returns an integer *type* value, on successful completion (shmid). The next two lines:

```
key_t  key;
int, size, shmflg;
```

declare the types of the formal arguments. Key_t is declared by a *typedef* in the types.h header file to be a long integer. Therefore, key, size, and shmflg are integers (*int*) that occupy 32 bits each in the 3B2 Computer.

The integer returned from this function on successful completion is the shared memory identifier (shmid) that was discussed in the ''GENERAL'' section of this chapter.

As declared, the process calling the shmget system call must supply three *actual* arguments to be passed to the formal key, size, and shmflg arguments.

The value passed to key must be a unique integer type hexadecimal value or zero (IPC_PRIVATE = 0) if a new shmid with an associated shared memory segment data structure is desired; it must be an existing key to return its shmid. This is true when only the IPC_CREAT flag is set in the shmflg argument.

Unique keys can be determined in several ways. The **STDIPC**, standard inter-process communication package, subroutine is one method to generate unique keys to avoid undesired interference between processes. Another way could be to use the **makekey** command, see the **STDIPC** and **makekey** manual pages. Picking a key at random is also possible but less desirable. If the key is IPC_PRIVATE, only the owner/creator process usually uses the facility.

The value passed to the shmflg argument must be an integer type octal value and will specify the following:

- Access permissions
- Execution modes
- Control fields (commands).

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the shmflg argument. They are collectively referred to as "operation permissions." Figure 5-4 reflects the numeric values for the valid operation permissions codes.

OPERATION PERMISSIONS	NUMERIC VALUE
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 5-4. Operation Permissions Codes

A specific numeric value is derived by adding the numeric values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). These values are represented in octal. There are constants located in the shm.h header file that can be used for the user (OWNER). They are as follows:

```
SHM_R          0400
SHM_W          0200
```

Control commands are predefined constants (represented by all uppercase letters). Figure 5-5 contains the names of the constants that apply to the shmget system call along with their values. They are also referred to as flags and are defined in the ipc.h header file.

CONTROL COMMAND	VALUE
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 5-5. Control Commands (Flags)

The value for shmflg is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is done by bitwise ORing (¦) them with the operation permissions; the bit positions and values for the control commands to those of the operation permissions make this possible. It is illustrated as follows:

```
                          OCTAL VALUE       BINARY VALUE

  IPC_CREAT        =      0 1 0 0 0      0 000 001 000 000 000
¦ Read by User     =      0 0 4 0 0      0 000 000 100 000 000

  shmflg           =      0 1 4 0 0      0 000 001 100 000 000
```

The shmflg value can be easily set by using the names of the flags with the octal operation permissions value:

```
shmid = shmget (key, size, (IPC_CREAT ¦ 0400));

shmid = shmget (key, size, (IPC_CREAT ¦ IPC_EXCL ¦ 0400));
```

As specified by the **shmget** manual page, success or failure of this system call depends on the argument values for key, size, and shmflg or system tunable parameters. The system call will attempt to return a new shmid if a following condition is true:

- Key is equal to IPC_PRIVATE (0)
- Key does not already have a shmid associated with it, and (shmflg & IPC_CREAT) is "true" (not zero).

The key argument can be set to IPC_PRIVATE in the following ways:

```
shmid = shmget (IPC_PRIVATE, size, shmflg);

               OR

    shmid = shmget ( 0 , size, shmflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the SHMMNI system tunable parameter causes a failure regardlessly. The SHMMNI system tunable parameter determines the maximum amount of unique shared memory segments (shmid's) in the UNIX System.

The second condition is satisfied if the value for key is not already associated with a shmid and the bitwise ANDing of shmflg and IPC_CREAT is "true" (not zero). This means that the key is unique (not in use) within the UNIX System for this facility type and that the IPC_CREAT flag is set (shmflg | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
          shmflg = x 1 x x x    (x = don't care)
    & IPC_CREAT = 0 1 0 0 0

          result = 0 1 0 0 0    (not zero)
```

Since the result is not zero, the flag is set or "true." SHMMNI applies here also, just as for condition one.

IPC_EXCL is another control command used with IPC_CREAT to exclusively have the system call fail if, and only if, a shmid exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) shmid when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new shmid is returned if the system call is successful. Any value for shmflg returns a new shmid if the key equals zero (IPC_PRIVATE).

The system call will fail if the value for the size argument is less than SHMMIN or greater than SHMMAX. These tunable parameters specify the minimum and maximum shared memory segment sizes.

Refer to the **shmget** manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **shmget** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the manual page for shmget (lines 4-7). Note that the errno.h header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired key
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **shmflg** argument

- **shmid**—used for returning the message queue identification number for a successful system call or the error code (-1) for an unsuccessful one
- **size**—used to specify the shared memory segment size.

The program begins by prompting for a hexadecimal key, an octal operation permissions code, and finally for the control command combinations (flags) that are selected from a menu (lines 14-31).

> ***Note:*** All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the opperm_flags variable (lines 35-50).

A display then prompts for the size of the shared memory segment, and it is stored at the address of the size variable (lines 51-54).

The system call is made next, and the result is stored at the address of the shmid variable (line 56).

Since the shmid variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 58). If shmid equals -1, a message shows that an error resulted and the external **errno** variable is displayed (lines 60, 61).

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the shmget system call follows. It is suggested that the source program file be named "shmget.c" and that the executable file be named "shmget."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the shared memory get, shmget(),
**system call capabilities.*/

#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/shm.h>
#include    <errno.h>

/*Start of main C language program*/
main()
{
    key_t key;        /*declare as long integer*/
    int opperm, flags;
    int shmid, size, opperm_flags;
    /*Enter the desired key*/
    printf("Enter the desired key in hex = ");
    scanf("%x", &key);

    /*Enter the desired octal operation
      permissions.*/
    printf("\nEnter the operation\n");
    printf("permissions in octal = ");
    scanf("%o", &opperm);

    /*Set the desired flags.*/
    printf("\nEnter corresponding number to\n");
    printf("set the desired flags:\n");
    printf("No flags                    = 0\n");
    printf("IPC_CREAT                   = 1\n");
    printf("IPC_EXCL                    = 2\n");
    printf("IPC_CREAT and IPC_EXCL      = 3\n");
    printf("            Flags           = ");
    /*Get the flag(s) to be set.*/
    scanf("%d", &flags);

    /*Check the values.*/
    printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
        key, opperm, flags);
    /*Incorporate the control fields (flags) with
      the operation permissions*/
    switch (flags)
    {
    case 0:    /*No flags are to be set.*/
        opperm_flags = (opperm | 0);
        break;
    case 1:    /*Set the IPC_CREAT flag.*/
        opperm_flags = (opperm | IPC_CREAT);
        break;
    case 2:    /*Set the IPC_EXCL flag.*/
        opperm_flags = (opperm | IPC_EXCL);
        break;
    case 3:    /*Set the IPC_CREAT and IPC_EXCL flags.*/
        opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
    }

    /*Get the size of the segment in bytes.*/
    printf ("\nEnter the segment");
    printf ("\nsize in bytes = ");
    scanf ("%d", &size);

    /*Call the shmget system call.*/
    shmid = shmget (key, size, opperm_flags);

    /*Perform the following if the call is unsuccessful.*/
    if(shmid == -1)
    {
      printf ("\nThe shmget system call failed!\n");
      printf ("The error number = %d\n", errno);
    }
   /*Return the shmid on successful completion.*/
   else
       printf ("\nThe shmid = %d\n", shmid);
   exit(0);
}
```

CONTROLLING SHARED MEMORY

This section gives a detailed description of using the **shmctl** system call along with an example program that allows all its capabilities to be exercised.

Using Shmctl

The synopsis of the **shmctl** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

The shmctl system call requires three arguments to be passed to it, and shmctl returns an integer value.

On successful completion, a zero value is returned; and when unsuccessful, shmctl returns a -1.

The shmid variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the shmget system call.

The cmd argument can be replaced by one of following control commands (flags):

- IPC_STAT—return the status information contained in the associated data structure for the specified shmid and place it in the data structure pointed to by the *buf pointer in the user memory area

- IPC_SET—for the specified shmid, set the effective user and group identification, and operation permissions

- IPC_RMID—remove the specified shmid along with its associated shared memory segment data structure
- SHM_LOCK—lock the specified shared memory segment in memory, must be super-user
- SHM_UNLOCK—unlock the shared memory segment from memory, must be super-user.

A process must have an effective user identification of OWNER/CREATOR or super-user to do an IPC_SET or IPC_RMID control command. Only the super-user can do a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to do the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using Shmget" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **shmctl** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the manual page for shmctl (lines 5-9). Note in this program that **errno** is declared as an external variable, and therefore, the errno.h header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer value from the system call
- **shmid**—used to store and pass the shared memory segment identifier to the system call
- **command**—used to store the code for the desired control command so that further processing can be performed on it
- **choice**—used to determine what member for the IPC_SET control command that is to be changed
- **shmid_ds**—used to receive the specified shared memory segment indentifier's data structure when an IPC_STAT control command is performed
- ***buf**—a pointer passed to the system call that locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET

command gets the values to set.

Note that the **shmid_ds** data structure in this program (line 16) uses the data structure located in the shm.h header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the *buf pointer is declared to be a pointer to a data structure of the shmid_ds type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid shared memory segment identifier that is stored at the address of the shmid variable (lines 18-20). This is required for every shmctl system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored at the address of the command variable. The code is tested to determine the control command for further processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-86). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out regardlessly; also an error message is displayed and the **errno** variable is printed out (lines 148, 149). If the system call is successful, a message shows this along with the shared memory segment identifier used (lines 151-154).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored at the address of the choice variable (line 99). Now, depending on the member picked, the program prompts for the new value (lines 105-127). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 128-130). Depending on success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 132-135), and the shmid along with its associated message queue and data structure are removed from the UNIX System. Note that the *buf pointer is not required as an argument to do this control command and its value can be zero or NULL. Depending on the success or failure, the program returns the same messages as for the other control commands.

If the SHM_LOCK control command (code 4) is selected, the system call is performed (lines 137,138). Depending on the success or failure, the program returns the same messages as for the other control commands.

If the SHM_UNLOCK control command (code 5) is selected, the system call is performed (lines 140-142). Depending on the success or failure, the program returns the same messages as for the other control commands.

The example program for the shmctl system call follows. It is suggested that the source program file be named "shmctl.c" and that the executable file be named "shmctl."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the shared memory control, shmctl(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/shm.h>

/*Start of main C language program*/
main()
{
    extern int errno;
    int uid, gid, mode;
    int rtrn, shmid, command, choice;
    struct shmid_ds shmid_ds, *buf;
    buf = &shmid_ds;

    /*Get the shmid, and command.*/
    printf("Enter the shmid = ");
    scanf("%d", &shmid);
    printf("\nEnter the number for\n");
    printf("the desired command:\n");
    printf("IPC_STAT     =  1\n");
    printf("IPC_SET      =  2\n");
    printf("IPC_RMID     =  3\n");
    printf("SHM_LOCK     =  4\n");
    printf("SHM_UNLOCK   =  5\n");
    printf("Entry        =  ");
    scanf("%d", &command);

    /*Check the values.*/
    printf ("\nshmid =%d, command = %d\n",
        shmid, command);
    switch (command)
    {
        case 1:     /*Use shmctl() to duplicate
             the data structure for
                 shmid in the shmid_ds area pointed
                 to by buf and then print it out.*/
           rtrn = shmctl(shmid, IPC_STAT,
               buf);
           printf ("\nThe USER ID = %d\n",
               buf->shm_perm.uid);
           printf ("The GROUP ID = %d\n",
               buf->shm_perm.gid);
           printf ("The creator's ID = %d\n",
               buf->shm_perm.cuid);
           printf ("The creator's group ID = %d\n",
               buf->shm_perm.cgid);
           printf ("The operation permissions = 0%o\n",
               buf->shm_perm.mode);
           printf ("The slot usage sequence\n");
           printf ("number = 0%x\n",
               buf->shm_perm.seq);
           printf ("The key= 0%x\n",
               buf->shm_perm.key);
           printf ("The segment size = %d\n",
               buf->shm_segsz);
           printf ("Segment Descriptor:\n");
           printf ("access = %o\n",
               buf->shm_seg.access);
           printf ("maximum offset = 0%x\n",
               buf->shm_seg.maxoff);
           printf ("flags = %o\n",
               buf->shm_seg.flags);
           printf ("address = 0%x\n",
               buf->shm_seg.address);
           printf ("lock = %o\n",
               buf->shm_seg.wd2.V0.lock);
           printf ("shmswap = %o\n",
               buf->shm_seg.wd2.V0.shmswap);
         printf ("alloc = %o\n",
             buf->shm_seg.wd2.V0.alloc);
         printf ("The pid of last shmop = %d\n",
             buf->shm_lpid);
         printf ("The pid of creator = %d\n",
             buf->shm_cpid);
```

```
            printf ("The current # attached = %d\n",
                buf->shm_nattch);
            printf ("The in memory # attached = %d\n",
                buf->shm_cnattch);
            printf ("The last shmat time= %d\n",
                buf->shm_atime);
            printf ("The last shmdt time= %d\n",
                buf->shm_dtime);
            printf ("The last change time= %d\n",
                buf->shm_ctime);
            break;
        case 2:    /*Select and change the desired
                   member(s) of the data structure.*/

            /*Get the original data for this shmid
                 data structure first.*/
              rtrn = shmctl(shmid, IPC_STAT, buf);

              printf("\nEnter the number for the\n");
              printf("member to be changed:\n");
              printf("shm_perm.uid    = 1\n");
              printf("shm_perm.gid    = 2\n");
              printf("shm_perm.mode   = 3\n");
              printf("Entry           = ");
              scanf("%d", &choice);
              /*Only one choice is allowed per
                pass as an illegal entry will
                    cause repetitive failures until
                shmid_ds is updated with
                    IPC_STAT.*/

              switch(choice){
              case 1:
                  printf("\nEnter USER ID = ");
                  scanf ("%d", &uid);
                  buf->shm_perm.uid = uid;
                  printf("\nUSER ID = %d\n",
                      buf->shm_perm.uid);
                  break;
              case 2:
                  printf("\nEnter GROUP ID = ");
                  scanf("%d", &gid);
                  buf->shm_perm.gid = gid;
                  printf("\nGROUP ID = %d\n",
                      buf->shm_perm.gid);
                  break;
            case 3:
                printf("\nEnter MODE = ");
                scanf("%o", &mode);
                buf->shm_perm.mode = mode;
                printf("\nMODE = 0%o\n",
                    buf->shm_perm.mode);
                break;
            }
            /*Do the change.*/
            rtrn = shmctl(shmid, IPC_SET,
                buf);
            break;
        case 3:    /*Remove the shmid along with its
                      associated
                      data structure.*/
            rtrn = shmctl(shmid, IPC_RMID, NULL);
            break;
        case 4: /*Lock the shared memory segment*/
            rtrn = shmctl(shmid, SHM_LOCK, NULL);
            break;
        case 5: /*Unlock the shared memory
                      segment.*/
            rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
            break;
        }
        /*Perform the following if the call is unsuccessful.*/
        if(rtrn == -1)
        {
          printf ("\nThe shmctl system call failed!\n");
          printf ("The error number = %d\n", errno);
        }
       /*Return the shmid on successful completion.*/
        else
          printf ("\nShmctl was successful for shmid = %d\n",
              shmid);
        exit (0);
    }
```

OPERATIONS FOR SHARED MEMORY

This section gives a detailed description of using the **shmat** and **shmdt** system calls, along with an example program that allows all their capabilities to be exercised.

Using Shmop

The synopsis of the **shmop** is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;

int shmdt (shmaddr)
char *shmaddr
```

Attaching a Shared Memory Segment

The shmat system call requires three arguments to be passed to it, and it returns a character pointer value.

The system call can be cast to return an integer value. On successful completion, this value will be the address in core memory where the process is attached to the shared memory segment and when unsuccessful it will be a -1.

The shmid argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the shmget system call.

The shmaddr argument can be zero or user supplied when passed to the shmat system call. If it is zero, the UNIX System picks the address of where the shared memory segment will be attached. If it is user supplied, the address must be a valid address that the UNIX System would pick.

The following illustrates some of the typical address ranges for the 3B2 Computer:

```
0xc00c0000
0xc00e0000
0xc0100000
0xc0120000
```

Note that these addresses are in chunks of 20,000 hexadecimal. It would be wise to let the operating system pick addresses to improve portability.

The shmflg argument is used to pass the SHM_RND and SHM_RDONLY flags to the shmat system call.

Further details are discussed in the example program for shmop. If you have problems understanding the logic manipulations in this program, read the "Using Shmget" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Detaching Shared Memory Segments

The shmdt system call requires one argument to be passed to it, and shmdt returns an integer value.

On successful completion, zero is returned; and when unsuccessful, shmdt returns a -1.

Further details of this system call are discussed in the example program. If you have problems understanding the logic manipulations in this program, read the "Using Shmget" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **shmat** and **shmdt** system calls to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was written for clarity and not program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that they provide.

This program begins by including the required header files as specified by the manual page for **shmop** (lines 5-9). Note that in this program that **errno** is declared as an external variable, and therefore, the errno.h header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **flags**—used to store the codes of SHM_RND or SHM_RDONLY for the shmat system call
- **addr**—used to store the address of the shared memory segment for the shmat and shmdt system calls
- **i**—used as a loop counter for attaching and detaching

- **attach**—used to store the desired amount of attach operations
- **shmid**—used to store and pass the desired shared memory segment identifier
- **shmflg**—used to pass the value of flags to the shmat system call
- **retrn**—used to store the return values from both system calls
- **detach**—used to store the desired amount of detach operations.

This example program combines both the shmat and shmdt system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

Shmat

The program prompts for the number of attachments to be performed, and the value is stored at the address of the attach variable (lines 17-21).

A loop is entered using the attach variable and the i counter (lines 23-70) to do the specified amount of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored at the address of the shmid variable (line 29). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored at the address of the addr variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37-44), and the code representing the flags is stored at the address of the flags variable (line 45). The flags variable is tested to determine the code to be stored for the shmflg variable used to pass them to the shmat system call (lines 46-57). The system call is made (line 60). If successful, a message stating so is displayed along with the attach address (lines 66-68). If unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

Shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored at the address of the detach variable (line 76).

A loop is entered using the detach variable and the i counter (lines 78-95) to do the specified amount of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored at the address of the addr variable (line 84). Then, the shmdt system call is performed (line 87). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the shmop system calls follows. It is suggested that the program be put into a source file called "shmop.c" and then into an executable file called "shmop."

> ***Note:*** When compiling **C** programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
/*This is a program to illustrate
**the shared memory operations, shmop(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/shm.h>
/*Start of main C language program*/
main()
{
    extern int errno;
    int flags, addr, i, attach;
    int shmid, shmflg, retrn, detach;


    /*Loop for attachments by this process.*/
    printf("Enter the number of\n");
    printf("attachments for this\n");
    printf("process (1-4).\n");
    printf("        Attachments = ");

    scanf("%d", &attach);
    printf("Number of attaches = %d\n", attach);

    for(i = 1; i <= attach; i++) {
       /*Enter the shared memory ID.*/
       printf("\nEnter the shmid of\n");
       printf("the shared memory segment to\n");
       printf("be operated on = ");
       scanf("%d", &shmid);
       printf("\nshmid = %d\n", shmid);


       /*Enter the value for shmaddr.*/
       printf("\nEnter the value for\n");
       printf("the shared memory address\n");
       printf("in hexadecimal:\n");
       printf("            Shmaddr = ");
       scanf("%x", &addr);
       printf("The desired address = 0x%x\n", addr);
```

```
        /*Specify the desired flags.*/
        printf("\nEnter the corresponding\n");
        printf("number for the desired\n");
        printf("flags:\n");
        printf("SHM_RND                    = 1\n");
        printf("SHM_RDONLY                 = 2\n");
        printf("SHM_RND and SHM_RDONLY = 3\n");
        printf("            Flags          = ");
        scanf("%d", &flags);

        switch(flags)
        {
        case 1:
            shmflg = SHM_RND;
            break;
        case 2:
            shmflg = SHM_RDONLY;
            break;
        case 3:
            shmflg = SHM_RND | SHM_RDONLY;
            break;
        }
        printf("\nFlags = 0%o\n", shmflg);

        /*Do the shmat system call.*/
        retrn = (int)shmat(shmid, addr, shmflg);
        if(retrn == -1) {
            printf("\nShmat failed. ");
            printf("Error = %d\n", errno);
        }
       else {
          printf ("\nShmat was successful\n");
          printf("for shmid = %d\n", shmid);
          printf("The address = 0x%x\n", retrn);
        }
    }
    /*Loop for detachments by this process.*/
    printf("Enter the number of\n");
    printf("detachments for this\n");
    printf("process (1-4).\n");
    printf("        Detachments = ");

    scanf("%d", &detach);
    printf("Number of attaches = %d\n", detach);
    for(i = 1; i <= detach; i++) {
            /*Enter the value for shmaddr.*/
            printf("\nEnter the value for\n");
            printf("the shared memory address\n");
            printf("in hexadecimal:\n");
            printf("            Shmaddr = ");
            scanf("%x", &addr);
            printf("The desired address = 0x%x\n", addr);

            /*Do the shmdt system call.*/
            retrn = (int)shmdt(addr);
            if(retrn == -1)  {
                printf("Error = %d\n", errno);
            }
            else {
               printf ("\nShmdt was successful\n");
               printf("for address  = 0%x\n", addr);

            }
        }
    }
```

Chapter 6

SYSTEM TUNABLE PARAMETERS

Chapter 6

SYSTEM TUNABLE PARAMETERS

To effectively allocate the UNIX System resources to the AT&T 3B2 Computer Inter-Process Communications (IPC) facilities, system tunable parameters are used. System tunable parameters, as their name implies, can be tuned or changed to provide the most efficient UNIX System environment. However, these tunable parameters cannot be changed arbitrarily as they are interdependent. This chapter deals with the system tunable parameters for the IPC facilities.

GENERAL

System tunable parameters are initialized to their initial maximum or initial default values when the UNIX System is built. These values are contained in a directory named **/etc/master.d**. Only a process with an effective user identification of super-user (0) can change these values. The initial maximum or initial default values are given in the following sections of this chapter for each IPC facility. Additionally, information on how they interrelate is given.

MESSAGES

There are seven system tunable parameters for the *message* type facility. Each parameter and its initial value follows:

- MSGMAP-100
- MSGMAX-8192
- MSGMNB-16384
- MSGMNI-10
- MSGSSZ-8
- MSGTQL-40
- MSGSEG-1024.

The following sections describe each system tunable parameter and how they interrelate to each other.

MSGMAP

This parameter specifies the size (amount of entries) of the memory control map used to manage message segments. A warning message is sent to the **console** port if this value is insufficient to handle the message type facilities. The initial value for this parameter is a default. MSGMAP can be raised as required to accommodate the message facilities. Each map number represents eight (8) bytes.

MSGMAX

This parameter determines the maximum size of a message sent (msgsnd). The initial value is a default, but it can be raised to a maximum of 131,072 bytes (128 kilobytes) (see MSGSSZ and MSGSEG). When receiving a message (msgrcv), a value larger than this parameter can be used to insure receiving the whole message without truncation.

MSGMNB

This parameter specifies the bytes that each message queue can have for storing its message header information. The initial value is a default, and it can be tuned as desired to fit the application; each header requires 12 bytes, so keep it in multiples (regardless of the default, two raised to the fourteenth power). The OWNER of a facility can lower this value, but only the super-user can raise it afterwards (msg_qbytes).

MSGMNI

This parameter specifies the amount of message queue identifiers (msqid) system wide. MSGMNI, therefore, determines the amount of message queues that can be created (msgget) at any one time. The initial value is a default, and it can be tuned to fit the application.

MSGSSZ

This parameter determines the segment size used for storing messages in memory. Each message is stored in contiguous segments numbering enough to fit the message. The initial value of this parameter is a default. MSGSSZ can be tuned as desired to fit the application. Keep in mind that the larger the segments are, the probability of having more wasted memory at the end of each message increases. The product of this parameter and MSGSEG should be no larger than 131,072 bytes (128 kilobytes). The 128 kilobyte value is also equal to the maximum value for a single message to be sent, MSGMAX.

MSGTQL

This parameter specifies the maximum amount of message queue headers on all message queues system wide, and consequently the total amount of outstanding messages. Each header occupies 12 bytes; this relates to the length of a message queue (MSGMNB). The initial value of MSGTQL is a default. MSGTQL can be tuned to fit the application.

MSGSEG

This parameter specifies the amount of memory segments system wide for storing messages. The initial value is a default. The product of MSGSEG and MSGSSZ should be no larger than 131,072 bytes (128 kilobytes).

The following data structure is contained in the **/usr/include/sys/msg.h** header file:

```
struct msginfo {
        int     msgmap,  /* # of entries in msg map */
                msgmax,  /* max message size */
                msgmnb,  /* max # bytes on queue */
                msgmni,  /* # of message queue identifiers */
                msgssz,  /* msg segment size (word size multiple) */
                msgtql;  /* # of system message headers */
        ushort  msgseg;  /* # of msg segments (MUST BE < 32768) */
};
```

This data structure is initialized from the **/etc/master.d/msg** file when the UNIX System is initialized.

SEMAPHORES

There are nine system tunable parameters for the *semaphore* type of IPC facility. Each parameter and its initial value follows:

- SEMMAP-10
- SEMMNI-10
- SEMMNS-60
- SEMMNU-30
- SEMMSL-25
- SEMOPM-10
- SEMUME-10
- SEMVMX-32767
- SEMAEM-16384.

The following sections describe each system tunable parameter and how they interrelate to each other.

SEMMAP

This parameter specifies the size (amount of entries) of the memory control map used to manage semaphore sets. A warning message is sent to the **console** port if this value is insufficient to handle the semaphore type facilities. The initial value for this parameter is a default. SEMMAP can be raised as required to accommodate the semaphore facilities. Each map number represents eight (8) bytes.

SEMMNI

This parameter specifies the amount of semaphore set identifiers (semid) system wide. SEMMNI, therefore, determines the amount of semaphore sets that can be created (semget) at any one time. The initial value is a default, and SEMMNI can be tuned to fit the application. This parameter occupies 32 bytes.

SEMMNS

This parameter specifies the total amount of semaphores in all semaphore sets system wide. The initial value is a default. SEMMNS can be tuned to fit the application. This parameter occupies 8 bytes.

SEMMNU

This parameter specifies the amount of semaphore undo structures system wide. The initial value is a default, and SEMMNU can be tuned to fit the application. The size of each undo structure equals [8 x (SEMUME + 2)] bytes.

SEMMSL

This parameter specifies the maximum amount of semaphores that can be in one semaphore set. The initial value is a default. SEMMSL can be tuned to fit the application.

SEMOPM

This parameter specifies the maximum amount of semaphore operations allowed for each semop() system call. The initial value is a default. SEMOPM can be tuned to fit the application. This parameter occupies 8 bytes.

SEMUME

This parameter specifies the maximum amount of undo structures per semaphore set. The initial value is a default. SEMUME can be tuned to fit the application. Keep in mind that it would be better to be able to undo as many operations as allowed per semaphore set; make SEMUME equal SEMOPM. This parameter occupies 240 bytes.

SEMVMX

This parameter specifies the maximum value that any semaphore can be. That is, the next higher number would be negative.

SEMAEM

This parameter specifies the maximum value that a semaphore adjust on exit value can be. That is, when decrementing a semaphore, this is the most that can be added to the adjust value for undoing the operation. Note that this parameter value is one more than half of SEMVMX.

The following data structure is contained in the **/usr/include/sys/sem.h** header file:

```
struct  seminfo {
        int     semmap,         /* # of entries in semaphore map */
                semmni,         /* # of semaphore identifiers */
                semmns,         /* # of semaphores in system */
                semmnu,         /* # of undo structures in system */
                semmsl,         /* max # of semaphores per id */
                semopm,         /* max # of operations per semop call */
                semume,         /* max # of undo entries per process */
                semusz,         /* size in bytes of undo structure */
                semvmx,         /* semaphore maximum value */
                semaem;         /* adjust on exit max value */
};
```

This data structure is initialized from the **/etc/master.d/sem** file when the UNIX System is initialized.

Note that the **semusz** member is not listed in the /etc/master.d/sem file as it will vary depending on the semaphore facility use.

SHARED MEMORY

There are five system tunable parameters for the *shared memory* type of IPC facility. Each parameter and its initial value follows:

- SHMMAX-8192
- SHMMIN-1
- SHMMNI-8
- SHMSEG-4
- SHMALL-32

The following sections describe each system tunable parameter and how they interrelate to each other.

SHMMAX

This parameter specifies the maximum amount of bytes that can be in a shared memory segment.

SHMMIN

This parameter specifies the minimum amount of bytes that a shared memory segment can be.

SHMMNI

This parameter specifies the total amount of shared memory facilities that can be in the UNIX System at one time. It corresponds to the amount of unique identifiers (shmid) that can be generated.

SHMSEG

This parameter specifies the maximum amount of shared memory segments that any one process can attach itself to at any one time. The default value is 4. Its maximum value is 15.

SHMALL

This parameter specifies the total amount of assigned physical pages of memory that can be in the UNIX System at one time. A page of memory equals 2048 bytes.

The following data structure is contained in the **/usr/include/sys/shm.h** header file:

```
struct    shminfo {
          int       shmmax,    /* max shared memory segment size */
                    shmmin,    /* min shared memory segment size */
                    shmmni,    /* # of shared memory identifiers */
                    shmseg;    /* max attached shared memory segments per process */
          int       shmall;    /* maximum physical assigned simultaneously */
};
```

This data structure is initialized from the **/etc/master.d/shm** file when the UNIX System is initialized.

Chapter 7

COMMAND DESCRIPTIONS

PAGE

Chapter 7

COMMAND DESCRIPTIONS

GENERAL

This chapter gives usage information for the two Inter-Process Communication (IPC) Utilities. The two utilities are as follows:

- *ipcs* — Inter-Process Communication status
- *ipcrm* — Inter-Process Communication remove.

The following sections contain the usage information and examples for each command.

INTER-PROCESS COMMUNICATION STATUS

The **ipcs** command can be used in two ways:

- Without options
- With options.

Ipcs Without Options

When using **ipcs** without options, a short status format is displayed for all IPC facilities that are in the UNIX System at the time of command execution. The short status format consists of the following information for all types of facilities:

- T—type of the facility
- ID—the identifier for the facility
- KEY—the key used for creating the facility
- MODE—the operation permissions and flags
- OWNER—the login name of the owner of the facility
- GROUP—the group name of the owner of the facility.

The example that follows is the result of entering the following command line:

```
$ipcs<CR>
IPC status from /dev/kmem as of Fri July 19 15:14:45 1985
T   ID    KEY        MODE       OWNER   GROUP
Message Queues:
q     0 0x00000000 S-rw-------     hrp    other
q     1 0x0000000a -Rrw-rw----     hrp    other
q     2 0x00000001 --rw-rw-rw-     hrp    other
Shared Memory:
m     0 0x00000000 D-rw-------     hrp    other
m     1 0x0000000a -Crw-rw----     hrp    other
m     2 0x00000001 -Crw-rw-rw-     hrp    other
Semaphores:
s     0 0x00000000 --ra-------     hrp    other
s     1 0x0000000a --ra-ra----     hrp    other
s     2 0x00000001 --ra-ra-ra-     hrp    other
```

From looking at this example, you can see several points of interest.

First, note that the display is separated into **Message Queues**, **Shared Memory**, and **Semaphores**. Note also that there are common column headings for these facility types. These headings correspond to the short status format information that **ipcs** without options displays as previously discussed.

The codes for the type (T) of facility are **q**, **m**, and **s** for message queues, shared memory, and semaphores, respectively.

Identifiers (ID) are integers (zero and positive) that are returned when creating a facility using the msgget(), shmget(), and semget() system calls.

Keys (KEY) are either IPC_PRIVATE (0) or equal to the value passed to the msgget(), shmget(), or semget() system calls for the **key** argument when creating a new facility; they can be 0, hexadecimal values, or decimal values. See the example display.

Mode (MODE) gives the operation permissions for each type of facility along with flags for the message and shared memory facilities. The mode is represented by a sequence of eleven character fields.

For message queues, the first character field is an S if a process is blocked from sending a message to the facility, and the second character field is an R if a process is blocked from receiving a message from a facility.

For shared memory, the first character field is a D if the shared memory segment facility is to be removed when the last process attached to the segment detaches it, and the second character field is a C if the shared memory segment facility is to be cleared when the first attach is made.

For semaphores, these two fields are not used as **semncnt** and **semzcnt** serve the same purpose. See the /usr/include/sys/sem.h file.

The corresponding special flags are not set for message queues and shared memory when the character field is " - " . These first two character fields are always " - " for semaphores as they are not used.

Operation permissions use the remaining nine character fields. They are used in groups of three and from left-to-right they represent the permissions for OWNER, GROUP, and OTHER. Note that for message queues rw means read/write and for semaphores ra means read/alter. All fields not in use are depicted by a hyphen.

The OWNER column heading gives the owner name of the facility. Note that when using msgctl(), shmctl(), or semctl() to change ownership of a facility, a positive integer value is used to represent the owner. These values can be determined for a particular owner name by searching through the **/etc/passwd** file.

The GROUP column heading gives the group name of the owner. Changing the group is analogous to changing the owner.

Ipcs With Options

The options available for the **ipcs** command consist of facility type options and general options. The facility type options allow the short format status information to be displayed for just the facility type desired. The general options allow information about size, creator, usage, process identification, and time to be observed. The general options can be used with the facility type options to observe the general options for a particular facility type.

Facility Type Options

The options that allow the status of only a particular type of facility to be observed are as follows:

-q	Message Queue Type
-m	Shared Memory Type
-s	Semaphore Type

Proper formats for entering these options are as follows:

$ipcs -q<*CR*>	Message Queue Type
$ipcs -m<*CR*>	Shared Memory Type
$ipcs -s<*CR*>	Semaphore Type

The status can be displayed for selected facilities by putting the options on the same command line, separated by spaces.

The following display occurs if status is requested but no facilities exist.

```
ipcs<CR>
IPC status from /dev/kmem as of Fri July 19 09:31:16 1985
T   ID   KEY      MODE      OWNER   GROUP
Message Queues:
Shared Memory:
Semaphores:
```

General Options

The general options allow additional kinds of information to be displayed for all facility types or for specific facility types. In other words, these general options can be used with **ipcs** alone to obtain the desired information for all facility types, or they can be appended to the facility type options for specific facility type information. More than one of these general options can be specified on the command line as well.

The following options are available:

• -b	Biggest allowable size
• -c	Creator login name and group name
• -o	Outstanding usage
• -p	Process number
• -t	Time
• -a	All general options
• -C	Use a different corefile than /dev/kmem
• -N	Use a different namelist than /unix.

Of course, these options will reflect only the information applicable to each facility type.

The biggest allowable size information option is illustrated as follows:

```
$ipcs -b<CR>
IPC status from /dev/kmem as of Fri July 19 07:55:13 1985
T   ID    KEY        MODE        OWNER   GROUP QBYTES
Message Queues:
q    0 0x00000000 --rw-------   hrp    other  16384
q    1 0x0000000a --rw-rw----   hrp    other  16384
q    2 0x00000001 --rw-rw-rw-   root   other   500
T   ID    KEY        MODE        OWNER   GROUP  SEGSZ
Shared Memory:
m    0 0x00000000 -Crw-------   hrp    other   8192
m    1 0x0000000a -Crw-rw----   hrp    other   1024
m    2 0x00000001 -Crw-rw-rw-   root   other   8192
T   ID    KEY        MODE        OWNER   GROUP NSEMS
Semaphores:
s    0 0x00000000 --ra-------   hrp    other   25
s    1 0x0000000a --ra-ra----   hrp    other   25
s    2 0x00000001 --ra-ra-ra-   root   other    5
```

Notice that for the message queue type of facility, QBYTES is the biggest allowable size information that is returned; it has been lowered for ID 2 to be 500. They were all initialized to the value of the system tunable parameter that specifies the maximum allowed bytes on a queue, MSGMNB.

For the shared memory type of facility, SEGSZ is the biggest allowable size information returned. SEGSZ specifies the size in bytes of the shared memory segment. The maximum is 8192 bytes (SHMMAX), and the minimum is 1 (SHMMIN).

For the semaphore type of facility, NSEMS is the biggest allowable size information that is returned. These values were determined when the facilities were created. The nsems argument passed to semget() determines these values. Remember that the system tunable parameter SEMMSL determines the maximum semaphores in a set (25).

The creator information is illustrated as follows:

```
$ipcs -c<CR>
IPC status from /dev/kmem as of Fri July 19 07:56:15 1985
T   ID   KEY        MODE        OWNER GROUP CREATOR CGROUP
Message Queues:
q    0 0x00000000 --rw-------   hrp  other  hrp   other
q    1 0x0000000a --rw-rw----   hrp  other  hrp   other
q    2 0x00000001 --rw-rw-rw-   root other  hrp   other
Shared Memory:
m    0 0x00000000 -Crw-------   hrp  other  hrp   other
m    1 0x0000000a -Crw-rw----   hrp  other  hrp   other
m    2 0x00000001 -Crw-rw-rw-   root other  hrp   other
Semaphores:
s    0 0x00000000 --ra-------   hrp  other  hrp   other
s    1 0x0000000a --ra-ra----   hrp  other  hrp   other
s    2 0x00000001 --ra-ra-ra-   root other  hrp   other
```

The results are the same for all facility types in this case. The column headings CREATOR and CGROUP show the login name and group name of the creator, respectively. The corresponding positive integer values for these names can be determined by searching the **/etc/passwd** file. Remember, the creator of a facility always remains the creator while the owner and group can change.

The outstanding usage option is as follows:

```
$ipcs -o<CR>
IPC status from /dev/kmem as of Fri July 19 07:58:13 1985
T   ID   KEY      MODE      OWNER   GROUP CBYTES QNUM
Message Queues:
q    0 0x00000000 --rw-------  hrp   other   16   1
q    1 0x0000000a --rw-rw----  hrp   other    0   0
q    2 0x00000001 --rw-rw-rw-  root  other  359  14
T   ID   KEY      MODE       OWNER   GROUP NATTCH
Shared Memory:
m    0 0x00000000 D-rw-------   hrp   other    1
m    1 0x0000000a -Crw-rw----   hrp   other    0
m    2 0x00000001 D-rw-rw-rw-   root  other    5
Semaphores:
s    0 0x00000000 --ra-------  hrp   other
s    1 0x0000000a --ra-ra----  hrp   other
s    2 0x00000001 --ra-ra-ra-  root  other
```

For message queues, the CBYTES and QNUM column headings stand for the total amount of bytes in core memory for all messages and the total amount of messages, respectively, for each message queue. The sum of all CBYTES is associated with the product of the amount of segments, MSGSEG, and the size of the segments, MSGSSZ. QNUM is associated with the total amount of bytes allowed for headers on each queue, MSGMNB. The sum of all QNUMs is associated with the total amount of message headers system wide, MSGTQL.

For shared memory, NATTCH corresponds to the amount of processes attached to the facility.

The outstanding usage option does not apply to the semaphore type facility even though the short format status information is displayed for it.

The process number option is illustrated as follows:

```
$ipcs -p<CR>
IPC status from /dev/kmem as of Fri July 19 08:12:53 1985
T    ID    KEY        MODE       OWNER  GROUP LSPID LRPID
Message Queues:
q     0 0x00000000 --rw-------    hrp  other 2275  2281
q     1 0x0000000a --rw-rw----    hrp  other   0    0
q     2 0x00000001 --rw-rw-rw-    root other   0    0
Shared Memory:
m     0 0x00000000 --rw-------    hrp  other  158  2254
m     1 0x0000000a --rw-rw----    hrp  other 2208  2254
m     2 0x00000001 --rw-rw-rw-    root other  166  2252
Semaphores:
s     0 0x00000000 --ra-------    hrp   other
s     1 0x0000000a --ra-ra----    hrp   other
s     2 0x00000001 --ra-ra-ra-    root  other
```

For message queues, the LSPID and LRPID column headings represent the last process identifier that sent and received a message from the associated message queue, respectively.

For shared memory, LSPID and LRPID represent the last process identifier to attach and detach from the facility, respectively.

The process number option does not apply to the semaphore type facility even though the short format status information is displayed for it.

The time information option is illustrated as follows:

```
$ipcs -t<CR>
IPC status from /dev/kmem as of Fri July 19 08:15:57 1985
T  ID    KEY    MODE     OWNER GROUP  STIME    RTIME    CTIME
Message Queues:
q  0 0x00000000 --rw------- hrp  other  8:11:44  8:12:07 15:09:25
q  1 0x0000000a --rw-rw---- hrp  other no-entry no-entry 15:09:53
q  2 0x00000001 --rw-rw-rw- root  other no-entry no-entry  7:25:23
T  ID    KEY    MODE     OWNER GROUP  ATIME    DTIME    CTIME
Shared Memory:
m  0 0x00000000 --rw------- hrp  other  8:04:50  8:05:12 15:10:39
m  1 0x0000000a --rw-rw---- hrp  other  8:05:00  8:05:29  7:54:41
m  2 0x00000001 --rw-rw-rw- root  other  8:03:42 no-entry  7:26:30
T  ID    KEY    MODE     OWNER GROUP  OTIME    CTIME
Semaphores:
s  0 0x00000000 --ra------- hrp  other  8:14:45 15:11:56
s  1 0x0000000a --ra-ra---- hrp  other no-entry 15:12:14
s  2 0x00000001 --ra-ra-ra- root  other no-entry  7:25:57
```

The message queue type of facility has three new column headings for this option: STIME, RTIME, and CTIME. STIME represents the last time that a process sent a message. RTIME represents the last time a process received a message. CTIME represents the time of the facility creation or the last time changed with a msgctl() system call.

The shared memory type of facility has three headings also. ATIME represents the time of the last attach operation. DTIME represents the time of the last detach operation. CTIME is the time of the facility creation or the last time changed with a shmctl() system call.

The semaphore type of facility has two new column headings for this option: OTIME, and CTIME. OTIME represents the last time that a process performed operations on the associated semaphore set. CTIME represents the time of the facility creation or the last time changed with a semctl() system call.

The display all options keyletter is illustrated as follows:

> ***Note:*** The short status format information is not included in this example so the pertinent information will fit on the page. On the display screen, the status information will wrap around.

```
$ipcs -a<CR>
IPC status from /dev/kmem as of Fri July 19 08:17:09 1985
CREATOR CGROUP CBYTES QNUM QBYTES LSPID LRPID STIME    RTIME    CTIME
Message Queues:
hrp    other  0    0  16384  2275   2281  8:11:44  8:12:07 15:09:25
hrp    other  0    0  16384   0      0 no-entry no-entry 15:09:53
hrp    other  0    0   500    0      0 no-entry no-entry  7:25:23
CREATOR CGROUP NATTCH SEGSZ CPID  LPID   ATIME    DTIME   CTIME
Shared Memory:
hrp    other  0    8192    158  2254  8:04:50  8:05:12 15:10:39
hrp    other  0    1024   2208  2254  8:05:00  8:05:29  7:54:41
hrp    other  0    8192    166  2252  8:03:42 no-entry  7:26:30
CREATOR CGROUP NSEMS  OTIME    CTIME
Semaphores:
hrp    other  25   8:14:45 15:11:56
hrp    other  25  no-entry 15:12:14
hrp    other   5  no-entry  7:25:57
```

The -C and -N options allow all the preceding options to be used on a different *corefile* and *namelist.* These options are useful for performing **ipcs** on a coredump file (-C) or when more than one version of the UNIX System (-N) is installed. Since the status of facilities can change while **ipcs** is running, these options allow more control.

INTER-PROCESS COMMUNICATION REMOVE

The command used to remove IPC facilities is as follows:

```
ipcrm [options]
```

There are two ways to remove a selected IPC facility from the UNIX System:

- Using the facility identifier (ID)
- Using the facility key (KEY).

The following sections illustrate how to remove IPC facilities using their IDs and KEYs.

Removal by ID

The options that are available to remove a facility by its ID are as follows:

- -q *msqid*
- -m *shmid*
- -s *semid*

An example of its use is as follows:

```
$ipcrm -q2 -s1 -q0 -m1<CR>
```

Note that the options can be repeated and placed on the command line in any order. The result of this command line will be to remove message queues 2 and 0, semaphore set 1, and shared memory segment 1.

Removal by Key

> ***Note:*** The key used for **ipcrm** must be a decimal value. The **ipcs** command reports keys in hexadecimal, however.

The options available to remove a facility by its key use the same letters as for removal by ID except that they are capital letters. They are as follows:

- -Q *msgkey*
- -M *shmkey*
- -S *semkey*

An example using these options follows:

```
$ipcrm -Q0 -S10 -Q1 -M1<CR>
```

The result of this command is to remove the message queue facilities with keys of **0** and **1**, to remove the semaphore facility with the key of **a** (10), and to remove the shared memory segment with the key of **1**.

Appendix

IPC ERROR CODES

Appendix

IPC ERROR CODES

This appendix contains the error codes for the 3B2 Computer Inter-Process Communication (IPC) system calls. Positive integer error codes are set in the external **errno** variable when a system call is unsuccessful.

An error has occurred when an IPC system call returns a -1 value. The value of **errno** is only valid immediately following this occurrence.

Each error code number has a corresponding mnemonic name. In this appendix, error code numbers and mnemonic names are categorized by facility type and associated system calls. The error reasons as they apply to IPC system calls are given.

These error codes are the same as those on the **intro(2)** manual page found in the *AT&T 3B2 Computer Programmer Reference Manual.* The reasons for the errors given there are more general than the reasons given in this appendix as they are used for all system calls.

MESSAGE ERROR CODES

The IPC error codes for the message type facility are contained in this section.

Msgget()

Each possible error code number, along with its mnemonic and reason(s), that the msgget() system call returns is contained in Figure A-1.

IPC (MSGGET) ERROR CODES		
NUMBER	MNEMONIC	REASON
2	ENOENT	A key not already in use is passed to the system call, but the IPC_CREAT flag is not set.
13	EACCES	Operation permissions deny the calling process.
17	EEXIST	A key already in use is passed to the system call with the IPC_CREAT and IPC_EXCL flags set. This is the exclusive create ability.
28	ENOSPC	The system wide amount of message queue identifiers would be exceeded (MSGMNI).

Figure A-1. Msgget Error Codes

Msgctl()

Each possible error code number, along with its mnemonic and reason(s), that the msgctl() system call returns is contained in Figure A-2.

IPC (MSGCTL) ERROR CODES		
NUMBER	MNEMONIC	REASON
1	EPERM	The process does not have the effective user identification of OWNER/CREATOR (msg_perm.[c]uid) of the facility or super-user when an IPC_RMID or IPC_SET control command is specified.
		The process does not have the effective user identification of super-user when using IPC_SET to increase the number of bytes (msg_qbytes) for the specified message queue.
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (buf) passed to the system call does not point to the necessary data structure (msqid_ds) in the user memory area.
22	EINVAL	The message queue identifier (msqid) is invalid; the facility does not exist.
		The value of the control command (cmd) passed to the system call is not equal to IPC_STAT, IPC_SET, or IPC_RMID.

Figure A-2. Msgctl Error Codes

Msgop()

Each possible error code number, along with its mnemonic and reason(s), that the msgsnd() and msgrcv() system calls return is contained in Figures A-3 and Figure A-4, respectively.

IPC (MSGSND) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
4	EINTR	The process received a signal while it was performing a " blocking message operation" that was blocked. (IPC_NOWAIT is not set.)
11	EAGAIN	The process cannot send a message because there are not enough bytes on the message queue (msg_qbytes), or the total amount of messages on all message queues would be exceeded (MSGTQL) while the process is performing a " nonblocking message operation." (IPC_NOWAIT flag is set.)
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (msgp) passed to the system call does not point to the necessary data structure (msgbuf) in the user memory area. (The data structure contains the message type value and message text array.)

Figure A-3. Msgsnd Error Codes (Sheet 1 of 2)

IPC (MSGSND) ERROR CODES		
NUMBER	MNEMONIC	REASON
22	EINVAL	The message queue identifier (msqid) is invalid; the facility does not exist.
		The value of the message type (msgtyp) variable passed to the system call is less than 1.
		The message size (msgsz) value passed to the system call is less than zero or greater than the system imposed limit, MSGMAX, for maximum message size.
36	EIDRM	The facility that the system call is performing a " blocking message operation" on is removed while the process is blocked. (IPC_NOWAIT is not set.)
4	EINTR	The process received a signal while it was performing a " blocking message operation" that was blocked. (IPC_NOWAIT is not set.)

Figure A-3. Msgsnd Error Codes (Sheet 2 of 2)

IPC (MSGRCV) ERROR CODES		
NUMBER	MNEMONIC	REASON
7	E2BIG	The value passed to the system call for the message size (msgsz) to be received is less than the message size and the IPC_NOERROR flag is not set.
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (msgp) passed to the system call does not point to the necessary data structure (msgbuf) in the user memory area. (The data structure contains the message type value and message text array.)
22	EINVAL	The message queue identifier (msqid) is invalid; the facility does not exist.
		The value of the message size (msgsz) variable passed to the system call is less than zero.
35	ENOMSG	The specified message queue does not contain the desired message type (msgtyp), and the IPC_NOWAIT flag is set (msgflg).

Figure A-4. Msgrcv Error Codes

SEMAPHORE ERROR CODES

The IPC error codes for the semaphore type facility are contained in this section.

Semget()

Each possible error code number, along with its mnemonic and reason(s), that the semget() system call returns is contained in Figure A-5.

IPC (SEMGET) ERROR CODES		
NUMBER	MNEMONIC	REASON
2	ENOENT	A key not already in use is passed to the system call, but the IPC_CREAT flag is not set.
13	EACCES	Operation permissions deny the calling process.
17	EEXIST	A key already in use is passed to the system call with the IPC_CREAT and IPC_EXCL flags set. This is the exclusive create ability.
22	EINVAL	The number of semaphores (nsems) to be in the set is less than or equal to zero or greater than the system tunable parameter SEMMSL.
		The value passed to the system call for the number of semaphores (nsems) is greater than what is in the set.
28	ENOSPC	The system call would cause the maximum amount of semaphore identifiers (sets) system wide to be exceeded (SEMMNI).
		The system call would cause the maximum amount of semaphores in all sets to be exceeded (SEMNS).

Figure A-5. Semget Error Codes

Semctl()

Each possible error code number, along with its mnemonic and reason(s), that the semctl() system call returns is contained in Figure A-6.

IPC (SEMCTL) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
1	EPERM	The process does not have the effective user identification of OWNER/CREATOR (sem_perm.[c]uid) of the facility or super-user when an IPC_RMID or IPC_SET control command is specified.
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (arg.buf) passed to the system call does not point to the necessary union data structure (semun) in the user memory area.
22	EINVAL	The semaphore set identifier (semid) is invalid; the facility does not exist.
		The semaphore number (0 through 24, semnum) is less than zero or greater than the number of semaphores in the set (sem_nsems).
		The value of the control command (cmd) passed to the system call is invalid.
34	ERANGE	When setting a semaphore(s) value (SETVAL, SETALL), the system imposed maximum is exceeded (SEMVMX).

Figure A-6. Semctl Error Codes

Semop()

Each possible error code number, along with its mnemonic and reason(s), that the semop() system call returns is contained in Figure A-7.

IPC (SEMOP) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
4	EINTR	The process received a signal while it was performing a " blocking semaphore operation" that was blocked. (IPC_NOWAIT is not set.)
7	E2BIG	The value passed to the system call for the number of semaphore operations (nsops) to be performed exceeds the system tunable parameter SEMOPM.
11	EAGAIN	The process would be blocked from performing its semaphore operation, but the IPC_NOWAIT flag is set.
13	EACCES	Operation permissions deny the calling process.

Figure A-7. Semop Error Codes (Sheet 1 of 3)

IPC (SEMOP) ERROR CODES		
NUMBER	MNEMONIC	REASON
14	EFAULT	The pointer (sops) passed to the system call does not point to an array of data structures (sembuf) in the user memory area. (Each data structure in the array contains the semaphore number, the operation to be performed, and the control command flags.)
22	EINVAL	The semaphore set identifier (semid) is invalid; the facility does not exist.
		The maximum amount of undo entries per system call (SEMUME) system tunable parameter would be exceeded.
27	EFBIG	The semaphore number (sem_num) for a data structure in the array is less than zero or greater than or equal to the total semaphores in the set.

Figure A-7. Semop Error Codes (Sheet 2 of 3)

IPC (SEMOP) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
28	ENOSPC	The maximum amount of undo entries system wide (SEMMNU system tunable parameter) would be exceeded.
34	ERANGE	The result of the operation would cause the semaphore value to exceed the maximum value for a semaphore (SEMVMX).
		The result of the operation would cause the maximum undo data structure value for semaphore adjust (SEMAEM) to be exceeded.
36	EIDRM	The facility that the system call is performing a " blocking message operation" on is removed while the process is blocked. (IPC_NOWAIT is not set.)

Figure A-7. Semop Error Codes (Sheet 3 of 3)

SHARED MEMORY ERROR CODES

The IPC error codes for the shared memory type facility are contained in this section.

Shmget()

Each possible error code number, along with its mnemonic and reason(s), that the shmget() system call returns is contained in Figure A-8.

IPC (SHMGET) ERROR CODES		
NUMBER	MNEMONIC	REASON
2	ENOENT	A key not already in use is passed to the system call, but the IPC_CREAT flag is not set.
12	ENOMEM	There is not enough physical memory to fill the request.
13	EACCES	Operation permissions deny the calling process.
17	EEXIST	A key already in use is passed to the system call with the IPC_CREAT and IPC_EXCL flags set. This is the exclusive create ability.
22	EINVAL	The size of the shared memory segment passed to the system call is invalid. The size must be greater than or equal to 1 or less than or equal to 8192. See SHMMIN and SHMMAX.
		An identifier exists for the facility (key) but the shared memory segment size is less than the size (not zero) passed to the system call.
28	ENOSPC	The system wide amount of shared memory segment identifiers would be exceeded (SHMMNI).

Figure A-8. Shmget Error Codes

Shmctl()

Each possible error code number, along with its mnemonic and reason(s), that the shmctl() system call returns is contained in Figure A-9.

IPC (SHMCTL) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
1	EPERM	The process does not have the effective user identification of OWNER/CREATOR (shm_perm.[c]uid) of the facility or super-user when an IPC_RMID or IPC_SET control command is specified.
		The process does not have the effective user identification of super-user when using the SHM_LOCK or SHM_UNLOCK commands.
13	EACCES	Operation permissions deny the calling process.

Figure A-9. Shmctl Error Codes (Sheet 1 of 2)

IPC (SHMCTL) ERROR CODES		
NUMBER	MNEMONIC	REASON
14	EFAULT	The pointer (buf) passed to the system call does not point to the necessary data structure (shmid_ds) in the user memory area.
22	EINVAL	The shared memory identifier (shmid) is invalid; the facility does not exist.
		The value of the control command (cmd) passed to the system call is not equal to IPC_STAT, IPC_SET, IPC_RMID, SHM_LOCK, or SHM_UNLOCK.
		The command is SHM_UNLOCK but the specified shared memory segment is not locked in memory.

Figure A-9. Shmctl Error Codes (Sheet 2 of 2)

Shmop()

Each possible error code number, along with its mnemonic and reason(s), that the shmat() and shmdt() system calls return is contained in Figures A-10 and Figure A-11, respectively.

IPC (SHMAT) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
12	ENOMEM	There is not enough in core memory to accommodate the shared memory segment.
13	EACCES	Operation permissions deny the calling process.
22	EINVAL	The shared memory identifier (shmid) is invalid; the facility does not exist.
		The shared memory address (shmaddr) passed to the system call is not equal to zero and [shmaddr-(shmaddr modulus SHMLBA)] is an illegal address.
		The shared memory address (shmaddr) passed to the system call is not equal to zero, the SHM_RND flag is false, and the address is illegal.

Figure A-10. Shmat Error Codes

IPC (SHMDT) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
13	EACCES	Operation permissions deny the calling process.
22	EINVAL	The system call detaches the shared memory segment located at the specified address (shmaddr) from the process data segment.
		The address (shmaddr) passed to the system call is not the start of a shared memory segment.
24	EMFILE	The number of shared memory segments attached to the calling process would exceed the number allowed, SHMSEG.

Figure A-11. Shmdt Error Codes