Utilities Package #1

for ORCA/M for the Apple IIGS
or the Apple IIGS Programmer's Workshop

Includes Source Code

By Mike Westerfield

The Byte Works Inc.

Limited Warranty - Subject to the below stated limitations, Byte Works, Inc., hereby warrants that the programs contained in this unit will load and run on the standard manufacturer's configuration for the computer listed for a period of ninety (90) days from date of purchase. Except for such warranty, this product is supplied on an "as is" basis without warranty as to merchantability or its fitness for any particular purpose. The limits of warranty extend only to the original purchaser.

Neither Byte Works, Inc., nor the authors of this program are liable or responsible to the purchaser and/or user for loss or damage caused, or alleged to be caused, directly or indirectly by this software and its attendant documentation, including (but not limited to) interruption of service, loss of business, or anticipatory profits.

To obtain the warranty offered, the enclosed purchaser registration card must be completed and returned to the Byte Works, Inc., within ten (10) days of purchase.

Important Notice - This is a fully copyrighted work and as such is protected under copyright laws of the United States of America. According to these laws, consumers of copywritten material may make copies for their personal use only. Duplication for any purpose whatsoever would constitute infringement of copyright laws and the offender would be liable to civil damages of up to $50,000 in addition to actual damages, plus criminal penalties of up to one year imprisonment and/or a $10,000 file.

This product is sold for use on a *single computer* at a *single location.* Contact the publisher for information regarding licensing for use at multiple- workstation or multiple-computer installations.

Source code for all programs included in this package are copyrighted and may not be duplicated or redistributed under any circumstances without prior written permission of the publisher.

ORCA/M is a trademark of the Byte Works, Inc. Apple is a registered trademark of Apple Computer, Inc.

Table of Contents

Getting Started 1
Installation Details 2
  How a Utility is Installed 2
  The INSTALL Program 2
The Utilities 3
  Standard Input and Standard Output 3
  File Names 4
CAL Calendar 5
CHECK Check a Disk 6
CMP Compare Binary Files 7
LOWER Convert to Lowercase 9
SORT Sort an ASCII File 10
  The Basic Sort 10
  The Quick Sort 11
  Options 12
  SORT -B file Ignore Leading Blanks 12
  SORT -D file Sort in Dictionary Order 13
  SORT -F file Fold Uppercase into Lowercase 13
  SORT -I file Ignore Non-printing Characters 13
  SORT -R file Reverse the Sense of the Compare 14
  SORT -V Write the Program Version and Copyright 14
  SORT -W rile Runs of White Space Compare Equal 14
STRIPC Strip Control Characters 15
STRIPW Strip Trailing White Space 16
TCMP Compare Text Files 17
  TCMP -B filel file2 Ignore Leading Blanks 19
  TCMP -F filel file2 Fold Uppercase into Lowercase 19
  TCMP -I filel file2 Ignore Non-Printing Characters 20
  TCMP -N filel file2 Print Line Numbers of Mismatches 20
  TCMP -S n filel file2 Resynchronize After n Lines Match 21
  TCMP -V filel file2 Write the Program Version and Copyright 21
  TCMP -W filel file2 Runs of White Space Compare Equal 21
TEE Split Input to Standard and Error Out 23
UNIQ Find Unique Lines 24
UPPER Convert to Uppercase 26
WC Word Count 27

Getting Started

This utility package serves several purposes. How you use it, and what you read in this manual, will depend on which of those purposes you purchased the package for. Whatever the purpose, though, start by making a backup of the disk that the utilities are on. The disk is not copy protected, so you can use any copy program (including COPY -D from the shell) to copy the disk. Protect your investment - don't skip this step!

With a backup made and safely tucked away, the first step is to install the utilities in your system. If you are using a hard disk, or if you have split things up on your 3.5" system disk, there will be plenty of room for the utilities. If you are working from a 3.5" disk exactly as it was shipped, you may want to free up some space first by deleting files you do not use, or by splitting the system up so it spans two disks. How you do this depends on which program (ORCA/M or APW) and which version you are using. Refer to the documentation for details.

Assuming you have enough free room on the disk that contains your utility prefix (120 blocks are enough), installation is really quite easy. The Utility Pack #1 disk has an EXEC file called INSTALL that installs the utilities automatically. If you are in a hurry, simply insert the disk in a drive (with your utilities in another drive) and type

```
PREFIX /UTILITY.PAC.1 INSTALL
```

The installation program will keep you advised of the actions it is taking. Details on what will happen are discussed below.

If you decide not to install the utilities, you can still use them. If the utilities are on the current prefix (if so, you will see them when you type CATALOG), you can use them as described here. If not, simply type the fell path name of the utility instead of the name of the utility.

While the main thrust of this package is to provide a set of useful utilities, you may have purchased it primarily for the source code. The source code for each of the utilities, as well as the macros used, are in subdirectories of the SOURCE directory. For example, the source code for the CAL utility is in the directory /UTILITY.PAC. 1 /SOURCE/CAL. The file CL is the source code itself, while CL.MACROS contain the macros used to assemble the CAL utility. To assemble this utility, type

```
PREFIX /UTILITY.PAC.1/SOURCE/CAL
ASML CL
```

As you explore the source, you will find many similarities between the programs. These similarities are a clue to how you can use these programs to write other utilities, quickly and easily.

Installation Details

How a Utility is Installed

There are three steps required to install a program that executes from the shell as a utility. The first step is to place the executable file in the utility subdirectory. This is the directory associated with prefix 6, which is listed when you type the SHOW PREFIX command. The next step is to copy the help file to the help subdirectory of the utility directory. The help file should have the same name as the utility itself, and should be a TXT file or SRC file. The help directory is 6/HELP. Finally, the command table must be updated to add the utility as a command.

As an example, let us assume that you have written a utility that will be called HAL. The executable file that is in your current directory is called HAL.EXE, and the help file is called HAL.HELP. To copy the files to the proper spot, you could type

```
COPY HAL.EXE 6/HAL
COPY HAL.HELP 6/HELP/HAL
```

Once the files are in place, you can edit the command table by typing

```
EDIT 4/SYSCMND
```

The numbered prefix 4 is always set to the system prefix while ORCA is running, so unless you have changed it since the last time ORCA was executed, this command will edit the command table that is loaded when you boot ORCA. Once in the editor, add the command HAL to the table (usually in alphabetical order). It should appear in column one of a new line. After at least one space, type a U, indicating that the new command is a utility. If you like, you can skip one more space and type in a comment telling what the utility does. With the new command in the command table, the shell will recognize it automatically after the next time you boot, or you can type

```
COMMANDS 4/SYSCMND
```

to read the new command table in immediately. If everything has been done correctly, the command will appear when you type HELP, the help file will be typed when you type HELP HAL, and the utility will run from any subdirectory when you type its name.

The INSTALL Program

The install utility does something very similar to the above process. First it copies the utilities and help files to the proper spot. To modify the command table, it actually uses some of the utilities themselves. The commands that must appear in the command table are collected on the disk in a file called SYSCMND. The installation program types this file and your command table with a single type command, which concatenates the two files. The output from this operation is piped to the SORT utility, which alphabetizes the list of files. As a check, just in case you have already installed the utilities once, the sorted file is then piped to the UNIQ utility to remove

duplicate lines. Finally, the output is written back to your original command file. The complete command looks like

```
TYPE 4/SYSCMND SYSCMND | BINARY/SORT | BINARY/UNIQ >4/SYSCMND
```

The last step is to install the new command table with the command

```
COMMANDS 4/SYSCMND
```

The Utilities

Each of the utilities in this package is described in detail on the following pages. Each of the descriptions starts with a header listing the name of the utility, what it is for, a model line, and a brief description of the parameters. After you have read the details about a utility once, the header (or the help file!) will often contain all of the information you need to refresh your memory about its use.

After the header is a detailed description of how to use the utility. This description tells what the utility does, exactly what each of the flags lets you control, and where appropriate, how the utility operates. For example, it is fairly obvious that the UPPER utility converts each lowercase character that is read into an uppercase character before writing it out, so the algorithm is not described in detail. At the other extreme, it is vitally important to know how text files are compared by the TCMP utility to be able to use it effectively, so a great deal of space is devoted to examples showing its subtleties. If a utility has any limitations, these are also discussed.

Standard Input and Standard Output

Many of the utilities process text information. These utilities can usually read their input from a file or from standard input. Reading the input from a file is fairly straight forward, and the reasons for doing so are fairly apparent. Why, though, should a utility be able to read information from standard input? The answer lies mostly in the fact that standard input can be redirected, as happens with pipes. In the section above that described how the installation procedure worked, a command appeared which concatenated two text files, sorted them, removed duplicated lines, and finally wrote the file back out. This was only possible because the utilities could read input from standard input.

When text utilities produce text output, they write the output to standard output. Normally, this is the computer's display screen, but you can use output redirection to redirect the output to a file, or pipelines to send the output on to the next program.

For example, let's assume that you have a file which contains a mailing list. Naturally, you want to remove duplicates. This can be done by first sorting the file, so that all duplicates are next to each other, and then using the UNIQ utility, which, among other things, can remove adjacent duplicate lines from a file. Assuming the file is called MAIL, you can do this by sorting the file and piping the output to UNIQ The output from UNIQ which has all of the duplicates removed, is then sent to a file called MAIL2:

```
SORT MAIL | UNIQ >MAIL2
```

For the sake of speed, when you give a utility a file to work on, it reads the entire file into memory with one ProDOS read. The savings in speed are enormous, but there is an obvious disadvantage: if you do not have enough memory to hold the entire file, the utility will fail. Most of the utilities (SORT is not one of them) can work on a file one line at a time. If your memory is limited, you can force the utility to read the file one character at a time, processing the file as it goes, by redirecting the input from the file rather than giving it the file name directly. For example,

```
UNIQ MAIL2
```

will read in the entire MAIL2 file, then process it. This is fast, but requires lots of memory. The command

```
UNIQ <MAIL2
```

reads the lines in two at a time, processing the first line before reading the next. This takes more time, but requires very little memory. If you are not sure if you have enough memory, try it the first way first. No harm will be done if you have too little memory - you will simply get an error message. If that happens, use the second method instead.

File Names

These utilities support the full range of special file name features introduced by ORCA/M and ProDOS 16. You can use device names, wildcards, prefix numbers and .. for directory walking in the normal way. If you are unfamiliar with these options, see your user's manual for details.

CAL Calendar

CAL [-V] [month] year

_V	Write the program name and copyright.
month	The month to print a calendar for.
year	The year to print a calendar for.

The CAL utility writes a calendar to standard output. Calendars are always printed for an entire month. A sample calendar is shown below to illustrate the format used.

```
          July 1987

Sun Mon Tue Wed Thu Fri Sat
                1   2   3   4
  5   6   7   8   9  10  11
 12  13  14  15  16  17  18
 19  20  21  22  23  24  25
 26  27  28  29  30  31
```

If you leave off the *month* parameter, the utility will print a calendar for the entire year. Years are specified as a full four-digit year. Years must be greater than 1499, and less than 4000. For example, to print the calendar for the year of 1987, type

```
CAL 1987
```

If you specify a specific month, only the calendar for that month is printed. Months are specified as the first three characters of the name of the month. It is not an error if more than three characters are specified, but any extra characters are not checked for spelling. For example, to print the calendar for July 1987 (shown above), you would type

```
CAL JUL 1987
```

CHECK Check a Disk

```
CHECK [-S] [-V] device
```

-S	Don't print anything - set return codes only.
-V	Write the program name and copyright.
device	The device (by name or number) to check.

The check utility checks a disk for bad blocks. It does this by first reading block two, where ProDOS disks have a field telling how many blocks are on the device. Each block on the device is then read. If a read error occurs, the block number where the error occurred is written to error out.

The -S flag adds to the usefulness of this utility when used from an EXEC file. In an EXEC file, where branching on the result of the check is more important that the messages the utility normally prints, the -S flag can be used to turn off the text messages. The utility will set the status flag to zero if there are no bad blocks, to one if there are bad blocks, and to -1 (65535) if there is an error, such as a bad parameter.

CMP Compare Binary Files

```
CMP [-L] [-S] [-V] filel file2
```

-L	Print all differences, not just the first.
-S	Don't print anything - set return codes only.
-V	Write the program name and copyright.
file1	First file to compare.
file2	Second file to compare.

The CMP utility does a binary compare of the two input files. The method used is to read both files into memory, then scan the files from start to finish looking for a byte that differs. If such a byte is found, the displacement into the files where the differing byte is located is written, first in decimal format and then in hexadecimal format. The bytes are then written, with the byte from *filel* appearing first.

If there are no differing bytes, but one file is shorter than the other, a message to that effect is written.

If there are no differences, and the files are the same length, a message is written to let you know that the files are equal.

One use of CMP is in EXEC files, where text messages telling what happened are not as useful as an error return code, which could be tested. The CMP utility will return one of three status codes. If the files are identical, the status will be zero. If the files are not identical, the status will be 1. If there was some error, such as an illegal flag or a missing file, the status will be set to 65535 (- 1).

For example, suppose that you want to write an EXEC file to insure two files are identical. If they match, the duplicate is to be deleted, but if they do not match, you want to leave both files intact for further examination. The EXEC file would take as input the names of the two files to compare, deleting the second one if they match. The EXEC file, then, is

```
CMP {1} {2}
IF {STATUS} == 0
      DELETE {2}
END
```

Two flags control options available with this utility. If you are using CMP in an EXEC file, and would rather not have CMP write anything to standard output, even if the files differ, use the -S flag. Normally, only the first differing byte is printed. If you want all differences to be listed, and not just the first, use the -L flag.

Finally, some comment is in order about the method used to compare the files. The intent with CMP is to provide a utility that can verify that two files are identical as quickly as possible. This

is the reason for reading the files in before starting the compare - the fewer calls to ProDOS, the faster the utility will run. This has two minor disadvantages. The first is that, if there is a difference early in the files, it takes longer to find the difference than if a character by character compare were used. The second is that there must be enough free memory to hold both of the files at the same time.

See TCMP for a utility that is set up to compare text files, rather than binary files.

LOWER Convert to Lowercase

```
LOWER [-V] [file ... ]
```

-V	Write the program name and copyright.
file	Take the input from *file*.

Characters are read from a file or standard input and written to standard output. Any character that appears in the input file as an uppercase alphabetic character (i.e. 'A' to 7) is converted to the equivalent lowercase character before being written out.

There are three ways to provide input to the utility. The first is to take the input from the keyboard, which happens if the utility is called with no file name and without redirecting input. To exit the utility, type CTRL-@. The second is to take the input from a file by redirecting the input, or by piping the input from a previous program. Finally, you can type the name of a file. While the last two methods generally result in the same output, there are internal differences in the way the input is handled. When input comes from a file, as in

```
LOWER MYFILE
```

the utility loads the entire file into memory with a single ProDOS call, then processes it. This is generally about twice as fast as redirecting the input, as in

```
LOWER <MYFILE
```

The advantage of redirecting input is that, since the entire file is not loaded into memory, you do not have to have enough memory to hold the file.

If more than one file name is supplied on the command line, as in

```
LOWER MYFILE1 MYFILE2
```

the files are concatenated, and processed as if they were a single file.

SORT Sort an ASCII File

```
SORT [-B] [-D] [-F] [-I] [-R] [-V] [-W] [file]
```

-B	Ignore leading blanks.
-D	Dictionary order (ignore all but alphanumeric and spaces).
-F	Fold uppercase into lowercase.
-I	Ignore non-printing characters.
-R	Reverse the sense of the comparisons.
-V	Write the program name and copyright.
-W	Runs of white space compare equal.
file	Take the input from file.

Due to the fact that it has so many options, the SORT utility is one of the hardest to explain. It is not, however, hard to use. The explanation will be divided into three sections. In the first section, we will look at basically what the sort utility does, and how to use it. The next section covers the type of sort used, along with the limitations and advantages that this method implies. Finally, we will look at the options, describing how each option changes the basic sort performed by this utility.

The Basic Sort

The SORT utility takes ASCII text as input, reading it from the file specified, or if no file is listed, from standard input. Whether the input comes from a file or from standard input, the entire file is read into memory before the sort starts, so there must be enough free memory to hold the file. The utility then sorts the file in ASCII character order.

To understand fully what SORT will do to a particular file, we must look at what happens in a few specific cases. First, if two lines are different in length, but they are identical up to the last character of the shorter line, the shorter line is placed first in the file. Another important case is when a line contains tabs. While tabs look like spaces when you are in the editor or when the file is typed, they are not. A tab is treated like any other character when the file is sorted.

As an example, consider the following file.

```
Now is the time
for all good men
to come to the aid
of their computer.

And now, some numbers:
1
   1
 10
!Wow!
```

When sorted, the file will be

```
   1
 10
!Wow!
1
And now, some numbers:
Now is the time
for all good men
of their computer.
to come to the aid
```

Note that the first line is blank.

The sort utility assumes that all lines in the file are less than or equal to 255 characters. If the -B, - D, -F, -1 and -W flags are not used, lines up to 65535 characters can be sorted. If any of these flags are used, any characters after the first 255 characters in a line are ignored.

The Quick Sort

While we will not go into what the quick sort is, or how to implement it, it is important to know about some of its advantages and disadvantages for you to understand this utility. The quick sort is a sort that functions recursively, by dividing a file into successively smaller pieces, and then sorting the smaller pieces. The way the quick sort algorithm is implemented in this utility, if you are sorting a perfectly random file, the recursion depth will be

ln(lines)/ln(2)

where *lines* is the number of lines in the file, and In is the natural logarithm function. In plain English, this means that if you double the number of lines in your file, the recursion depth increases by one. As we pointed out, this formula holds for a perfectly sorted file. It is also approximately correct for a randomly sorted file. In extreme cases, though, it is possible for the recursion depth to equal the number of lines.

The reason that all of this is so important is that there is a maximum recursion depth that the utility can handle. The maximum depth that can be handled is 89, so that the utility can sort a presorted file up to 6e29 lines long. While that is plenty of capacity, it is possible, although rare, that the utility will fail. This would occur on files that were ordered very strangely to start with. If the file is ordered strangely enough, the utility could fail on an 89 line file.

With the possibility that the sort could fail on some files, you may wonder why we did not choose a more reliable sort, like the shell sort. After all, the shell sort does not fail on any length of file that it can read into memory. The reason is that the common shell sort requires a time proportional to N^2 to sort an N line file, while the quick sort requires a time proportional to

N*ln(N). For a short file, this is of little consequence, but if you are sorting, say, a 1000 line source file, the quick sort method would be about 144 times faster than a shell sort.

Options

A number of options give you close control over the method used to sort the file. These options can be used alone, or in combination. To get a firm grasp on what each option does, we will continue to sort the file shown in the example above.

SORT -B file Ignore Leading Blanks

The -B flag causes the sort utility to ignore leading blanks when comparing two lines. Note that the blanks are not actually removed from the lines, just ignored. Sorted this way, our example file would become

```
!Wow!
   1
1
 10
And now, some numbers:
Now is the time
for all good men
of their computer.
to come to the aid
```

When two lines compare equal, as with the two lines that contain the number 1, ties are broken by comparing the two lines without removing the leading blanks first. Since a blank is the first printable ASCII character, that means that in the absence of imbedded control characters, the line with the most leading blanks will appear first. Lines containing all blanks will always appear first in the sorted file (again, assuming no imbedded control characters). If two all blank lines appear, the shorter one appears first.

SORT -D file Sort in Dictionary Order

Dictionary order basically means that the utility ignores any character that is not alphabetic, numeric, or a space. In our sample, then, the exclamation marks in the !Wow! line are ignored.

```
   1
 10
1
And now, some numbers:
Now is the time
!Wow!
for all good men
of their computer.
to come to the aid
```

SORT -F file Fold Uppercase into Lowercase

The -F option performs a case-insensitive sort. This means that the 'A' and 'a' characters, which normally are not considered to be the same, are both treated as if they were the 'a' character. Sorted this way, the test file becomes

```
   1
 10
!Wow!
1
And now, some numbers:
for all good men
Now is the time
of their computer.
to come to the aid
```

SORT -I file Ignore Non-printing Characters

It is possible to have control characters or characters whose numeric value is over 126 in an ASCII file, despite the fact that they do not print on the screen. A common example is a file that contains form feed characters. These show up in the editor as inverse L characters. When printed to the screen, they cause the screen to clear and the cursor to be placed at the top left comer of the screen. The -I flag causes them to be ignored when sorting the file.

Since our example did not contain any non-printing characters, we will not list the results again here.

SORT -R file Reverse the Sense of the Compare

The -R flag causes the sort to work in reverse order. Our sample file would then be

```
to come to the aid
of their computer.
for all good men
Now is the time
And now, some numbers:
1
!Wow!
 10
   1
```

SORT -V Write the Program Version and Copyright

Before sorting the file, the version number and copyright notice are printed.

SORT -W rile Runs of White Space Compare Equal

White space is a term used to refer to all of the ASCII characters that have something to do with spacing the file, such as the space character itself, horizontal tabs, form feeds, and carriage returns. White space characters consist of the space and all ASCII characters whose numeric values are in the range 9 to 13. The -W flag causes all sequences of white space that occur in a line to be treated as if they were one space. It the sample file, this means that the multiple spaces in front of the 1 are treated as a single space. This does not actually change anything in our sample.

STRIPC Strip Control Characters

```
STRIPC [-V] [file...]
```

-V	Write the program name and copyright.
file	Take the input from *file*.

Characters are read from standard input. If the character is a printable ASCII character (ASCII code $20 to $7F), a carriage return ($OD), or a tab ($09), it is written to standard output All other characters are disposed of.

There are three ways to provide input to the utility. The first is to take the input from the keyboard, which happens if the utility is called with no file name and without redirecting input. To exit the utility, type CTRL-@. The second is to take the input from a file by redirecting the input, or by piping the input from a previous program. Finally, you can type the name of a file. While the last two methods generally result in the same output, there are internal differences in the way the input is handled. When input comes from a file, as in

```
STRIPC      MYFILE
```

the utility loads the entire file into memory with a single ProDOS call, then processes it. This is generally about twice as fast as redirecting the input, as in

```
STRIPC      <MYFILE
```

The advantage of redirecting input is that, since the entire file is not loaded into memory, you do not have to have enough memory to hold the file.

If more than one file name is supplied on the command line, as in

```
STRIPC      MYFILE1 MYFILE2
```

the files are concatenated, and processed as if they were a single file.

STRIPW Strip Trailing White Space

```
STRIPW [-V] [file ... ]
```

-V	Write the program name and copyright.
file	Take the input from *file*.

Lines are read from standard in. The end of each line is marked with a carriage return (ASCII code $OD). All white space that appears at the end of the line is removed, and the resulting line is written to standard output.

There are three ways to provide input to the utility. The first is to take the input from the keyboard, which happens if the utility is called with no file name and without redirecting input. To exit the utility, type CTRL-@. The second is to take the input from a file by redirecting the input, or by piping the input from a previous program. Finally, you can type the name of a file. While the last two methods generally result in the same output there are internal differences in the way the input is handled. When input comes from a file, as in

```
STRIPW MYFILE
```

the utility loads the entire file into memory with a single ProDOS call, then processes it. This is generally about twice as fast as redirecting the input, as in

```
STRIPW <MYFILE
```

The advantage of redirecting input is that, since the entire file is not loaded into memory, you do not have to have enough memory to hold the file.

If more than one file name is supplied on the command line, as in

```
STRIPW MYFILE1 MYFILE2
```

the files are concatenated, and processed as if they were a single file.

TCMP Compare Text Files

```
TCMP [-B] [-F] [-I] [-N] [-S n] [-V] [-W] file1 file2
```

-B	Ignore leading blanks.
-F	Fold uppercase into lowercase.
-I	Ignore non-printing characters.
-N	Print line numbers of mismatches.
-S n	Resynchronize after n lines match.
-V	Write the program name and copyright.
-W	Runs of white space compare equal.
filel	First file to compare.
file2	Second file to compare.

The TCMP utility compares two ASCII files. The files are first read into memory, and then compared, one line at a time. If two lines differ, the utility attempts to resynchronize by skipping lines in one or both files. Lines are skipped in such a way that the fewest possible number of total lines are skipped. When five lines from each file match, the utility prints the lines skipped and continues comparing the files from that point as if they were equal.

To get a better grasp on all of this, we will look at several examples. In the first, we will compare two files that have a single line that differs between them.

File 1:	File 2:
Shakespeare	Shakespeare
Dickinson	Dickiinson
Farley	Farley
Asimov	Asimov
Locke	Locke
Plato	Plato
Heinlein	Heinlein

Since the two files differ - Dickinson is misspelled in the second file - the utility will detect the difference and write

```
---- Files differ:

-- File 1:

Dickinson

-- File 2:

Dickinson
```

Note that even though the files are of different length, the utility does not report the rest of the file as being different, as the CUP utility would. The key point to remember is that at least five lines must match before the utility believes the files are in step again. For example, if two of the names were different, as in

File 1:	File 2:
Shakespeare	Shakespeare
Dickinson	Dickiinson
Farley	Farley
Asimov	Isaac
Locke	Locke
Plato	Plato
Heinlein	Heinlein

the utility will report the entire last section of the files as different:

```
---- Files differ:

-- File 1:

Dickinson
Farley
Asimov
Locke
Plato
Heinlein

-- File 2:

Dickiinson
Farley
Isaac
Locke
Plato
Heinlein
```

As was mentioned, the files can actually have extra lines, as the files below do.

File 1:	File 2:
Shakespeare	Shakespeare
Dickinson	Farley
Farley	Isaac
Asimov	Locke
Locke	Plato
Plato	Heinlein
Heinlein	

Here, the difference is that the first file has a line for Dickinson, while that name is missing from the second file. If you are not expecting the results, they may not seem correct at first glance, since one file has no information at all listed:

```
---- Files differ:

-- File 1:

Dickinson

-- File 2:
```

TCMP -B filel file2 Ignore Leading Blanks

The -B flag causes leading blanks to be ignored when the lines are compared. Thus, the following sample files would compare equal if this option were used.

File 1:	File 2:
Shakespeare	Shakespeare
Dickinson	Dickinson
Farley	Farley
Asimov	Asimov
Locke	Locke
Plato	Plato
Heinlein	Heinlein

TCMP -F filel file2 Fold Uppercase into Lowercase

Lowercase and their uppercase counterparts compare as the same character. Thus, with this option, the following two files will compare as equal.

File 1:	File 2:
shakespeare	Shakespeare
Dickinson	Dickinson
Farley	Farley
Asimov	Asimov
Locke	Locke
Plato	Plato
Heinlein	Heinlein

TCMP -I filel file2 Ignore Non-Printing Characters

Non-printing characters are removed from the lines before they are compared. The non-printing characters are all characters outside of the ASCII range $20 (the space) to $7E (the -), with the exception of $09 (the tab character), which is considered to be printable.

TCMP -N filel file2 Print Line Numbers of Mismatches

With the -N option, when differences occur, the line number where the difference occurs is also listed. This can be useful in tracking down the exact position of a difference in a file. For example, if you compare these two files:

File 1:	File 2:
Shakespeare	Shakespeare
Dickinson	Dickiinson
Farley	Farley
Asimov	Asimov
Locke	Locke
Plato	Plato
Heinlein	Heinlein

and the -N option is used, the output looks like this:

```
---- Files differ:

-- File 1:

    2 Dickinson

-- File 2:

    2 Dickinson
```

You can then use the editor repeat count to edit the file and jump to the differing line, or use the TYPE command with the +N flag to see the lines that surround the differing line.

TCMP -S n filel file2 Resynchronize After n Lines Match

Normally, the utility resynchronizes after five lines that match are found. This has been found to be a good number for comparing assembly language files - with this value, it can generally locate an entire subroutine that has been inserted into the program. Depending on what you are searching for, you may want to have the utility resynchronize after more or fewer lines match. The -S flag allows you to select the number of lines that match before the utility considers the file to be in step again.

Some cautionary notes are in order. The more lines you ask to match, the longer it will take the utility to scan a pair of files. If you use a value that is too small, the utility may not be able to detect when several lines are inserted into a file.

As an example, the following command will search the two files, resynchronizing after three lines match.

```
TCMP -S 3 FILE1 FILE2
```

TCMP -V filel file2 Write the Program Version and Copyright

Before printing any differences that may exist between the two files, the version number and copyright notice are printed.

TCMP -W filel file2 Runs of White Space Compare Equal

This flag causes sequences of white space (spaces, tabs, line feeds, form feeds) to be compared as if they were a single space. Note that at least one space must exist for the comparison to match. For example, the files

File 1:

```
Shakespeare
 Dickinson
Farley
Asimov
Locke
Plato
Heinlein
```

File 2:

```
Shakespeare
         Dickinson
Farley
Asimov
Locke
Plato
Heinlein
```

would be considered to be equal if the -W flag is used, but the files

File 1:	File 2:
Shakespeare	Shakespeare
Dickinson	Dickinson
Farley	Farley
Asimov	Asimov
Locke	Locke
Plato	Plato
Heinlein	Heinlein

are not equal, since there is no space at all in the first file.

TEE Split Input to Standard and Error Out

```
TEE [-V] [file ... ]
```

-V	Write the program name and copyright.
file	Take the input from *file*.

Characters are read from standard input. Each character is written to both standard output and error output.

There are three ways to provide input to the utility. The first is to take the input from the keyboard, which happens if the utility is called with no file name and without redirecting input. To exit the utility, type CTRL-@. The second is to take the input from a file by redirecting the input, or by piping the input from a previous program. Finally, you can type the name of a file. While the last two methods generally result in the same output, there are internal differences in the way the input is handled. When input comes from a file, as in

```
TEE  MYFILE
```

the utility loads the entire file into memory with a single ProDOS call, then processes it. This is generally about twice as fast as redirecting the input, as in

```
TEE  <MYFILE
```

The advantage of redirecting input is that, since the entire file is not loaded into memory, you do not have to have enough memory to hold the file.

If more than one file name is supplied on the command line, as in

```
TEE  MYFILE1  MYFILE2
```

the files are concatenated, and processed as if they were a single file.

UNIQ Find Unique Lines

```
UNIQ [-C] [-D] [-U] [-V] [file ... ]
```

-C	Write a frequency count.
-D	Print only lines with duplicates.
-U	Print only lines that are unique.
-V	Write the program name and copyright.
file	Take the input from *file*.

UNIQ lets you remove adjacent duplicate lines from a file. The utility examines each line in the file, and removes any lines that exactly match an adjacent line. If the -C option is coded, each line that is written to standard output is preceded by a five-digit frequency count, telling how many times the line occurred in the original file.

Two options allow you to print only unique lines or only lines that were not unique. The first, -D, prints lines that were duplicated. It still only prints each line one time. The -U option suppresses lines that were duplicated, printing only those that were unique. These two options cannot be used together.

Note that only adjacent unique lines are removed. For example, if the input file is

```
Mike
Phil
Phil
Patty
Mike
```

and you run UNIQ on the file with the -C option, the output would be

```
1 Mike
2 Phil
1 Patty
1 Mike
```

The duplicate Phil is counted and removed, but since the two Mike's are not adjacent, they are both flagged as unique. If you want to find all duplications of a given line, run the file through the SORT utility first. For example, if the above file is called MYFILE, the command

```
SORT MYFILE | UNIQ -C
```

would give the output

```
2 Mike
1 Patty
2 Phil
```

There are three ways to provide input to the utility. The first is to take the input from the keyboard, which happens if the utility is called with no file name and without redirecting input. To exit the utility, type CTRL-@. The second is to take the input from a file by redirecting the input, or by piping the input from a previous program. Finally, you can type the name of a file. While the last two methods generally result in the same output there are internal differences in the way the input is handled. When input comes from a file, as in

```
UNIQ  MYFILE
```

the utility loads the entire file into memory with a single ProDOS call, then processes it. This is generally about twice as fast as redirecting the input, as in

```
UNIQ  <MYFILE
```

The advantage of redirecting input is that, since the entire file is not loaded into memory, you do not have to have enough memory to hold the file.

If more than one file name is supplied on the command line, as in

```
UNIQ  MYFILE1  MYFILE2
```

the files are concatenated, and processed as if they were a single file.

UPPER Convert to Uppercase

```
UPPER [-V] [file]
```

-V	Write the program name and copyright.
file	Take the input from *file*.

Characters are read from a file or standard input and written to standard output. Any character that appears in the input file as a lowercase alphabetic character (i.e. 'a' to Y) is converted to the equivalent uppercase character before being written out.

There are three ways to provide input to the utility. The first is to take the input from the keyboard, which happens if the utility is called with no file name and without redirecting input. To exit the utility, type CTRL-@. The second is to take the input from a file by redirecting the input, or by piping the input from a previous program. Finally, you can type the name of a file. While the last two methods generally result in the same output, there are internal differences in the way the input is handled. When input comes from a file, as in

```
UPPER MYFILE
```

the utility loads the entire file into memory with a single ProDOS call, then processes it. This is generally about twice as fast as redirecting the input, as in

```
UPPER <MYFILE
```

The advantage of redirecting input is that, since the entire file is not loaded into memory, you do not have to have enough memory to hold the file.

If more than one file name is supplied on the command line, as in

```
UPPER MYFILE1 MYFILE2
```

the files are concatenated, and processed as if they were a single file.

WC Word Count

```
WC [-C] [-L] [-V] [-W] [file ... ]
```

-C	Write the number of characters in the file.
-L	Write the number of lines in the file.
-V	Write the program name and copyright.
-W	Write the number of words in the file.
file	Take the input from *file*.

Counts the number of characters, words and lines in an ASCII file. Words are defined as any sequence of characters, separated by white space. Lines are counted by counting the number of carriage returns in the file. Characters are counted by counting all of the characters in the file except for carriage returns.

If no parameters are coded, lines, characters and words are all counted. If any option - including the -V flag - is coded, only those counts requested with a flag are written out. For example, if you code

```
WC -L MYFILE
```

the only thing printed will be the number of lines.

There are three ways to provide input to the utility. The first is to take the input from the keyboard, which happens if the utility is called with no file name and without redirecting input. To exit the utility, type CTRL-@. The second is to take the input from a file by redirecting the input, or by piping the input from a previous program. Finally, you can type the name of a file. While the last two methods generally result in the same output, there are internal differences in the way the input is handled. When input comes from a file, as in

```
WC MYFILE
```

the utility loads the entire file into memory with a single ProDOS call, then processes it. This is generally about twice as fast as redirecting the input, as in

```
WC <MYFILE
```

The advantage of redirecting input is that, since the entire file is not loaded into memory, you do not have to have enough memory to hold the file.

If more than one file name is supplied on the command line, as in

```
WC MYFILE1 MYFILE2
```

the files are concatenated, and processed as if they were a single file.