Program Title

SNR95 Version: 6.1
SNR32 Version: 6.1

Purpose

Multi-string simultaneous Search 'n' Replace program for Windows95

Release date

6/25/97

Author

Thomas A. Lundin
16267 Hudson Avenue
Lakeville, MN 55044
(612) 431-5805
email: 70523.262@compuserve.com

Registration

$60 per copy; volume and site licenses available. Registered version discards the opening shareware compliance screen.

Registration guarantees technical support; unregistered users will receive support on a time-available basis.

Fill out and return the ORDER.FRM file along with your payment to register your copy.

Description

SNR is a multi-string search-and-replace filter. Both text and binary files can be processed by the program. SNR will translate a file of any size that your system can handle.

SNR features many advanced conversion capabilities, such as:

- wild-card conversion patterns
- variable-length conversion patterns
- start-of-file and end-of-file conversions
- text field padding
- table chaining
- enhanced context flags

Up to 2,500 multi-character (m:n) equations can be entered into an SNR table, each of which can be a maximum of 4,999 characters in length. Equations may occupy a total of 64,000 characters. All search-and-replace operations are performed in a single pass. Context flag conversion allows toggling between two different output strings for the same input string. Wild card patterns allow unspecific character strings to be matched and converted.

SNR32 Program Operation (command-line)

SNR32 is the command-line version of SNR. It is designed to run in a DOS box under Windows95/NT. The command-line version allows SNR to be used in a .BATch file for automated file processing.

The command line invocation is:

snr [@]filename ext table[table2...] [/d] [/opathname] [/r]

1. ***filename*** is the name of the disk file you wish to convert. The DOS95 wildcard characters "*" and "?" are allowed here, and they will result in all files matching the name pattern to be converted at one time. An at-sign (@) in front of the filename indicates that the file contains a list of file names to convert. There are two ways of creating the list of file names: the first is to use the DOS95 DIR command and redirect its output to a file, like this:

   ```
   dir *.txt >dirfile
   ```

   This will create a disk file named "dirfile" which contains a directory list of all files with a ".TXT" extension. You can edit this file if you want, to weed out certain files you don't want and retain the others.

   You can also create a list file manually by just creating a file with a text editor and typing in the path and file names for each of the files you wish converted. Example:

   ```
   d:\data\test.txt
   c:\bigfiles\tapedat.asc
   c:\windows\win.ini
   ```

   The converted output files will appear in your current folder. You can optionally direct the output to another folder with the /o option, explained later.

   Windows95/NT long filenames are supported by SNR32. A long filename path is specified within surrounding quotes, like this:

   ```
   snr "\text files\my stuff\temp\email log file" "after conversion" table1.s
   ```

   In this example, the resulting output file will be named
   \text files\my stuff\temp\email log file.after conversion

2. ***ext*** is the filename extension you wish to assign each output file. The output files are created by adding this extension onto the name of the input file. If the original input file already contains an extension, the old extension is replaced by the new one. If the original input files does not contain an extension, the extension is added.

   If you are converting multiple input files from a DIRLIST, or from a wild card in the command line, all of the resulting output files will have the same extension. The choice of an extension is important; if you accidentally choose an extension which is already used by another file, you will overwrite the existing file, losing its data in the process. SNR will not warn you of impending overwrites. Use an extension which you are sure is unique.

Extensions can also be the DOS95 system names *stdout* and *stderr*. Using one of these system names as your output extension allows the output from SNR to be piped with the vertical bar character (|), or redirected with the right angle bracket (>). Be aware that redirecting your output to a disk file with the angle bracket operator will result in double carriage returns at the end of each line. This will not be apparent when viewed on the screen, but a text editor may display the file double-spaced. You can overcome this drawback by adding a line in your conversion table that converts a carriage return/line feed pair as a line feed only:

```
\0d\0a=\0a
```

This equation is not necessary if your intended use of stdout is to display a file to the screen without redirection.

Note that normal program error messages are also displayed through stdout, and this means that any redirected output will contain messages if errors occur.

Examples of piping and redirection are shown in the next section. Refer to a DOS manual for a thorough explanation of this subject.

3. ***tablename*** is the file name and optional path which contains your string translation *equations*. Although no restrictions are placed upon the tablename (aside from conforming to the Windows95 naming format), for sake of clarity it is suggested that you adopt a consistent naming scheme for them, (by convention, most people use an extension of ***.s**). Creating conversion equations is discussed in more detail later. You may specify more than one table at a time on the command line; the effect of this is that SNR will convert your input files through each table in sequence. The contents of the output files will be just as if you had run SNR a number of times in a row.

4. ***/d*** is an option which replaces the original input file with the converted data. The extension which you specify on the command line is used as a temporary intermediate file until the conversion is complete, at which time the program deletes the original input file and renames the temporary output file to the input name.

5. ***/o*** is an option which allows you to place the output files in some folder other than the current folder. The /o is followed immediately by the name of the folder path (no spaces in between).

6. ***/r*** is an option used only in conjuction with the /o option; it is meaningless by itself. /r forces all of the output files in the other folder to retain the same name as the original input files. At the end of the conversion run using this option, you will have two folders of files with the same names, but the data in the files in one of the folders will have been converted.

7. The options /o, /r, and /d may appear in a list file, after the name of the file to convert. Example:

```
d:\data\file.1 /od:\
d:\data\file.2
c:\text\may.doc
```

In the above example, all of the files in the list will be converted and stored on the root folder of drive D:. These option codes may NOT appear in a list file created by redirecting the DOS95 DIR command to a file.

Command Line Examples

```
C>snr tstfil.doc txt tst.s
```

The above command line will convert input file "tstfil.doc" through table "tst.s" and create output file "tstfil.txt".

```
C>snr @dirlist p1 sample.s
```

The above command line will convert a group of files listed in the file "dirlist" through table "sample.s" and create a group of output files with extensions of *.p1.

```
C>snr *.c xxx pass1.s pass2.s pass3.s /d
```

The above command line will convert a group of files matching the file name pattern "*.c" through the three tables "pass1.s", "pass2.s" and "pass3.s" in order, and the converted output files will replace the original input files. (The temporary intermediate files will have an extension of *.xxx.)

```
C>snr mytest.dat da2 pass1.s /od:\
```

The above command will convert the input file "mytest.dat" through table "pass1.s" and create an outfile named "mytest.da2" in the root folder of drive D:.

```
C>snr mytest.dat stdout pass1.s
```

The above command will perform the same conversion as the previous example, except that the output of the conversion will be displayed on the screen instead of being saved in a disk file.

```
C>snr mytest.dat stdout pass1.s|more
```

The effect of the above command will be to have the output displayed on the screen with a pause between screenfuls. Refer to a DOS manual for a thorough explanation of piping and redirection.

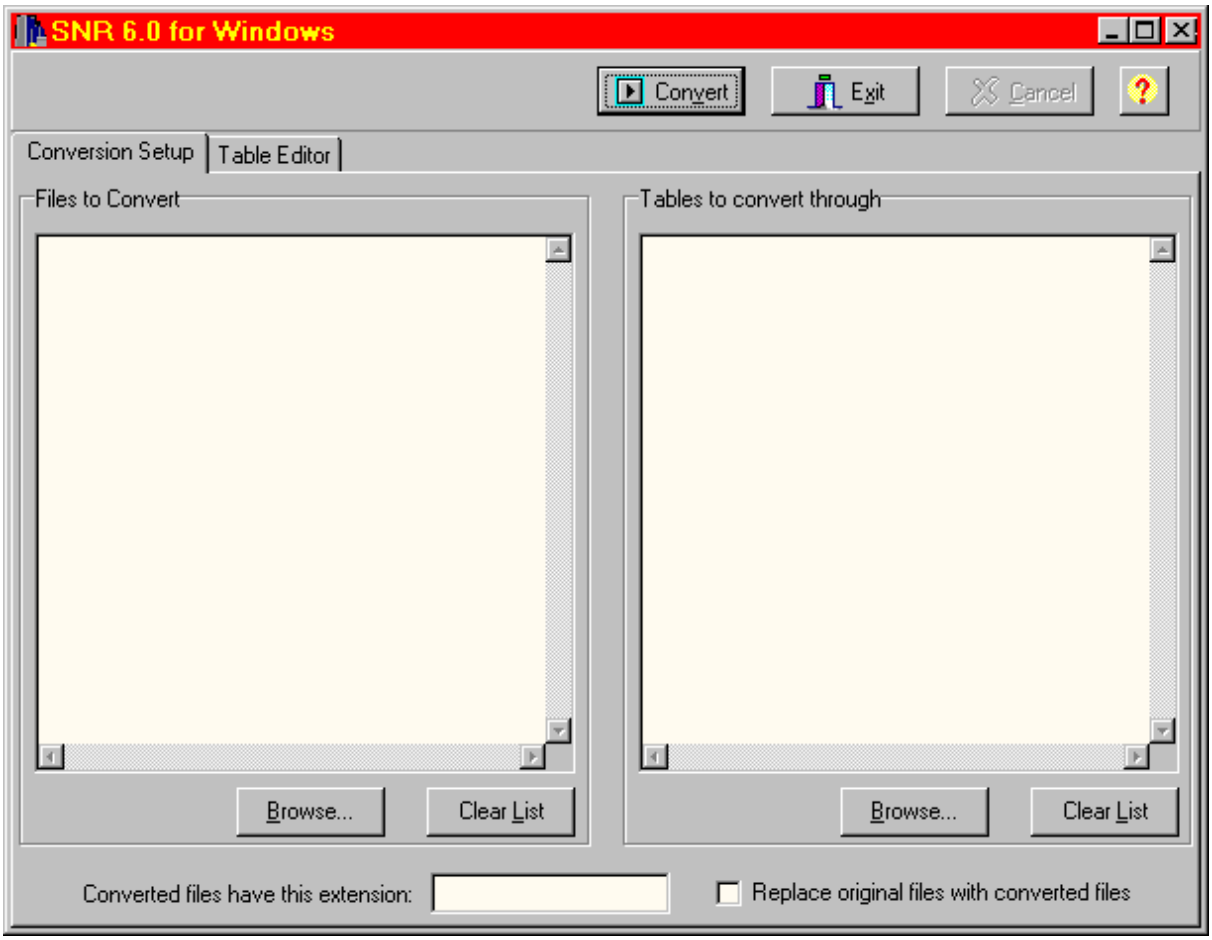
```
C>snr "\test files\mytest*.*" "test conv" pass1.s
```

The above command will convert all files in the "test files" folder that begin with "mytest" and create output files with the extension of "test conv".

SNR95 Program Operation (GUI)

SNR95 is the graphical user inteface version of SNR, designed to run under Windows95 or Windows NT.

SNR starts up on the **Conversion Setup** screen (shown below).



Files to Convert

This box lists all of the files that will be processed when the **Convert** button is clicked. You can enter the file names manually by clicking your cursor in this box and typing them in, or you can click the **Browse** button and select a group of files from a file selection dialog box. Click the **Clear List** button if you want to empty this list box and load a new set of files for conversion.

You can manipulate the file list by cutting, pasting and deleting the lines. Select a line or group of lines with the mouse by clicking from the left edge of a filename and dragging the mouse down (or press Shift-Down Arrow from the keyboard). After you have selected the file(s), right-click the mouse to open a pop-up menu with Cut, Copy, Paste and Delete commands, or use the keyboard commands Ctrl-X to Cut, Ctrl-C to Copy, Ctrl-V to Paste, and Del to Delete.

Tables to convert through

This box lists the conversion tables which will be used to perform the actual conversion. Conversion tables contain lists of equations which specify a search string and its replacement. Usually, you'll list only one table in this box. If you list more than one table in this box, files from the left list box will

each be converted through all of the tables in sequence listed in the right list box. You can enter the file names manually by clicking your cursor in this box and typing them in, or you can click the **Browse** button and select a group of tables from a file selection dialog box. Click the **Clear List** button if you want to empty this list box and load a new set of tables.

You can manipulate the table list by cutting, pasting and deleting the lines. Select a line or group of lines with the mouse by clicking from the left edge of a filename and dragging the mouse down (or press Shift-Down Arrow from the keyboard). After you have selected the table(s), right-click the mouse to open a pop-up menu with Cut, Copy, Paste and Delete commands, or use the keyboard commands Ctrl-X to Cut, Ctrl-C to Copy, Ctrl-V to Paste, and Del to Delete.

Converted files have this extension

Each resulting converted file is given a new extension, specified in this edit box. This new extension will replace the original extension, if there was one, or it will be added to the file name if there was no extension originally. Extensions in Windows95 can be multiple words.

Replace original files with converted files

If this box is checked, the original file names will be kept intact, but the contents of the original files will be replaced by the converted files. ***The data in the original files is irretrievable.*** Regardless of the status of this check box, the **Converted files have this extension** edit box must have a value in it. The new extension is used temporarily to hold the result of the conversion before being assigned the original file name.

Convert button

Clicking this button will begin the conversion. As each file is converted, its name will be displayed to the left of the Convert button.

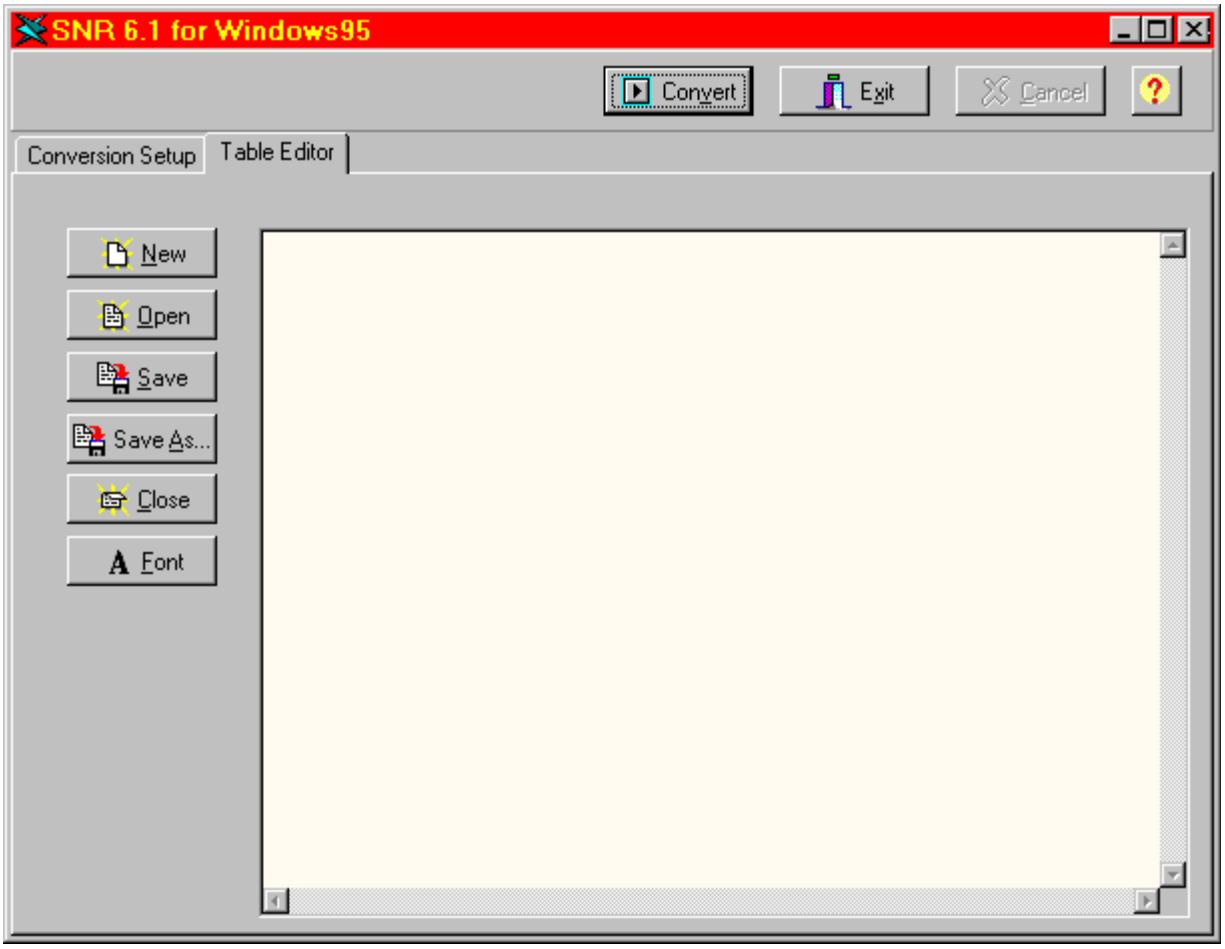
Exit button

Clicking this button will close the program.

Cancel button

Clicking this button will abort the conversion between files. Conversion for the current file in progress will first be completed.

If you click on the **Table Editor** tab, the following screen will appear:



This screen presents you with a simple editing program with which to open or create conversion tables. Tables you create or edit in this screen must be saved before being used in the **Conversion Setup** screen.

You do not need to use this screen to create your conversion tables; any ASCII text editor will do (such as Notepad.exe). This screen is provided as a convenience.

Creating Conversion Tables

SNR tables are ASCII text files which contain the search-and-replace equations used by the program. Any word processor or text editing program capable of loading and saving plain ASCII text files will be sufficient to create conversion tables. (SNR95 contains a built-in table editor for this purpose.)

Up to 2,500 of these equations can be entered in a single table, and each equation can consist of up to 4,999 total characters divided into a search side and a replacement side. Blank lines in a table will be ignored. The maximum size of a table is limited to 64,000 characters. These equations will be matched against the input file in a single pass, resulting in a converted output file.

EQUATION TYPES

This manual may refer to certain equation types from time to time. The equation types that are handled by SNR are as follows:

1:0 an equation comprising one character on the left side and no characters on the right side
1:1 an equation with one character on the left side and one character on the right side
1:n an equation with on character on the left side and two or more characters on the right side
m:0 an equation with two or more characters on the left side, and no characters on the right side
m:1 an equation with two or more characters on the left side, and one character on the right side
m:n an equation with two or more characters on the left side, and two or more characters on the right side

FORMAT OF EQUATIONS

The simplest form of conversion equation consists of literal search and replace text separated by an equals sign. A sample 1:1 equation would be:

```
A=a
```

The above equation would translate an upper-case 'A' to a lower-case 'a'.

A sample m:n equation would be:

```
Now is the time=NOW IS THE TIME
```

The above equation will translate the words "Now is the time" to all upper-case. Notice that spaces ***are significant*** characters in an equation, so don't use them carelessly if you don't mean to have them converted.

1:0 and m:0 equations ignore search strings (that is, throw it away on output). They are defined simply by leaving out a replacement string, like this:

```
Now is the time=
```

If you want to output word spaces at the end of a replacement string, but your text editor strips trailing spaces, you can define them as hex codes:

```
Now is the time =NOW IS THE TIME\20
```

In fact, any hex code can be formed from a backslash followed by two hex digits (0-9, a-f, A-F). You'd normally use hex codes to search or replace binary characters that can't be generated directly from the keyboard. For instance, a carriage return/line feed sequence (CRLF) can be specified like this:

```
\0d\0a\0d\0a=\0d\0a
```

The above equation will convert two CRLFs in a row to a single CRLF.

SNR RESERVED CHARACTERS

There are three ASCII characters which *must* be specified as hex codes in an SNR equation, since they have special meaning in their normal ASCII form. They are:

`Backslash (\),` which must be entered as `\5c`

`Equals (=),` which must be entered as `\3d`
`Asterisk (*),` which must be entered as `\2a`

If you forget to follow this notation, you will very likely trigger error messages when you attempt to run a conversion.

EXTRA-LONG EQUATIONS

SNR can handle equation lengths up to 4,999 characters. It's quite likely that you will never encounter this limit. Even so, your word processor or text editor may have a much smaller limit on the number of characters you can enter per line. Should you find you need to carry an equation over from one line to the next, you can end a line with **\+** as the last item in the line, and SNR will automatically add the next following line to the same equation. You can have as many continuation lines as you need to. Example:

```
Now is the time for all good men=NOW IS THE TIME    \+
\20FOR ALL GOOD MEN \ This is all one continued equation
```

Be mindful that any spaces that occur before the \+ continuation code are ignored (an exception to the all-spaces-are-significant rule). Any text that appears on the same line after the \+ code is ignored.

ENDING YOUR TABLES

No special marker is needed to signify the end of your equations; that is, SNR will stop loading equations when the end of a table is reached.

There is an optional code which may be used to force SNR to stop scanning for equations, which might be necessary if your word processor pads files with garbage or with an EOF character, which SNR would attempt to read as the start of a new equation. If you place a \\E on a line by itself after your last equation, it will prevent SNR from inadvertently reading past the intended last line of your equations. A sure bet that you need to place the \\E code at the end of your tables is if SNR occasionally aborts with the message:

```
Incomplete equation (missing right half):
```

...and you are unable to see that any of your equations is incomplete.

REPEATING CHARACTERS

SNR provides a shorthand notation for describing a large number of repeating characters. Rather than entering an equation that looks like this:

```
AAAAAAAAAAAAAAAAAAAAAAAAA=bbbbbbbbbbbbbbbbbbbb
```

...you can describe it like this:

```
*(25)A=*(20)b
```

Any character can be repeated using this notation. By changing the repeat value in parentheses, you accomplish the same thing as typing the repeat character that number of times. **NOTE:** SNR will

actually expand a shorthand notation into longhand notation before it begins conversion, so make sure you don't inadvertently exceed the 4,999-character equation limit with your repeat values.

COMMENTING YOUR EQUATIONS

SNR provides for writing comments in your tables. Comments are notes that you make to yourself as you are writing a table, to help describe the purpose of an equation. Comments are not a convertible part of an equation -- they are simply ignored by the SNR equation processor. Comments can be entered in a table as lines by themselves, or set on the same line to the right of an equation, as in this example:

```
\ This is a comment line by itself.
\   A comment consists of a single backslash
\   followed by one or more spaces.

\0d\0a=\0d\0a    \ this will ensure that existing CRLF pairs are
                 \ left untouched
\0d=\0d\0a       \ this equation will convert an isolated CR
                 \ into a CRLF
\0a=\0d\0a       \ this equation will convert an isolated LF
                 \ into a CRLF
\\E
```

When a comment follows an equation on the same line, any spaces that occur before the comment are ignored; *this is an exception to the all-spaces-are-significant rule.*

If you're a programmer, or if you've used a macro language of some kind, you already know the value of program comments. If you're new to the field of user programming, you should immediately get into the habit of commenting your work. Believe me, you'll be glad you did, because a time will come when you have to make a change to a table you created months past, and you'll be left wondering what you did and why you did it if you haven't commented the equations.

EQUATION ORDERING

SNR will automatically sort equations by how long they are when it assembles a table in memory. SNR does not process equations in sequential order when it converts a file, so the order in which you enter your equations is largely immaterial, except when you are using wild card patterns in your equations. This will be explained further on.

Equation processing is completed in a *non-recursive* fashion. That is, once an equation has been matched and a replacement made, the replaced data is not converted again through another equation, even if it would match one. All converted text is passed onto the output file once.

Context Flags

There are probably instances where you'd like to toggle between two different replacement strings for the same search string. The context flag allows you to do this. Some example uses would be:

- to compress multiple spaces from a document
- to ignore any text between two codes

- to make one code alternate as two different codes
- to turn all-upper-case text into upper and lower

There are undoubtedly many other uses.

There are 16 context flags which can be used, and each context flag has two states: ***on*** and ***off***. The context flags are identified by the numbers `0-9` and the letters `a-f` (for a total of 16 individual flags).

When SNR begins execution, the context flags' states are all ***off***. A flag's state can be tested in a search, and its state can be set or reset in a replacement. A context flag is represented in an equation by an asterisk, followed by the number or letter of the flag, followed by a one or zero (for ***on*** or ***off***).

A context flag is the last item entered in a search or replacement string. For example:

```
ABC*00=abc*01
ABC*01=XYZ*00
```

In the above two equations, if the string "ABC" is read from the input file, and context flag 0 is ***off***, then the string "abc" is written to the output file, and context flag 0 is set ***on***. If, on the other hand, the string "ABC" is read from the input file, and context flag 0 is ***on***, then the string "XYZ" is written to the output file, and context flag 0 is reset ***off***. For example, if data from the input file looks like this:

```
ABCDEFG ABCDEFG ABCDEFG ABCDEFG
```

...our small example table would convert it to this:

```
abcDEFG XYZDEFG abcDEFG XYZDEFG
```

Context flags are global in scope; this means that each flag is accessible by many different equations in one table, and any equation can test or set the flags. But beware of unwanted side effects -- remember that only one equation at a time will control the state of a flag, and its state may change between two equations that complement each other, resulting in misconversion. For example:

```
ABC*00=abc*01
ABC*01=XYZ*00
EFG*00=efg*01
EFG*01=ZYX*00
```

The above equations are similar to the previous example, except that we have defined two new equations that test and set a single context flag. So, given this example input data:

```
ABCDEFG ABCDEFG ABCDEFG ABCDEFG
```

...the result would convert to this:

```
abcDZYX abcDZYX abcDZYX abcDZYX
```

The context flag is being toggled by only two equations:

```
ABC*00=abc*01  <--- this one
ABC*01=XYZ*00
EFG*00=efg*01
EFG*01=ZYX*00  <--- and this one
```

If, on the other hand, the input data stream looked like this:

```
ABCDEFG DEFGABC ABCDEFG DEFGABC
```

...the result would look like this:

```
abcDZYX DefgXYZ abcDZYX DefgXYZ
```

This conflict between two sets of equations grappling over the same context flag can be alleviated by using a different flag number for one of the sets:

```
ABC*00=abc*01
ABC*01=XYZ*00
EFG*10=efg*11
EFG*11=ZYX*10
```

Now the equations will have independent flags, and the data stream will be processed using each equation separately. Now, this data stream:

```
ABCDEFG ABCDEFG ABCDEFG ABCDEFG
```

...will be converted to this:

```
abcDefg XYZDZYX abcDefg XYZDZYX
```

The thing to keep in mind if you are going to use a context flag is: it is ***off*** when the program begins; you need to enter an equation to set it ***on***; and then you may need to have some equation that resets it ***off***.

See some of the sample tables for examples in the use of context flags.

Special Context Flag Codes

Certain types of conversions involving context flags can be very tedious to define. Take the case of a table whose task is to compress two or more adjacent blank spaces into a single space. Using our context flags, we would start out by setting a flag when we first encounter a space:

```
\20*00=\20*01
```

This equation states that if a space is encountered and flag 0 is ***off***, it will be converted to one space and flag 0 ***on***. Next we define what happens to a space when flag 0 is ***on***:

```
\20*01=
```

This equation states that a space and flag 0 ***on***, will be ignored. That takes care of removing adjacent spaces, but there's something missing. The way we have it set up now, *every* subsequent space will be removed, even between words, because we have not defined the case where flag 0 ever gets set ***off***. Without something to set flag 0 ***off***, only the second equation will hold true.

Since we have defined the case where adjacent spaces will be ignored, we need to define the case where some character other than a space will reset flag 0 so we can retain the single interword space mentioned earlier.

Here's where the tedium comes in. The complete solution is to make every character *except* a space set flag 0 ***off***, like this:

```
\00=\00*00
\01=\01*00
\02=\02*00
\03=\03*00
\04=\04*00
\05=\05*00
    ...
\7c=\00*00
\7d=\00*00
\7e=\00*00
    ...
```

...and so on. Basically, you'd need 255 similar equations to finish the task.

Specifically for these types of conversions, SNR provides two specialty context flag codes: `*ic` and `*ig`.

IGNORE CONSECUTIVE

The Ignore Consecutive (`*ic`) flag code is used on the right-hand side of the conversion equation. Its purpose is to allow you to specify any set of characters that will be ignored when encountered adjacently in a data stream. A side benefit of this notation is that SNR will automatically generate all of the other flag reset equations that are needed to complete the conversion. This vastly reduces the "clutter" that would otherwise be present in your table. For instance, here is our complete space compression table with the help of the `*ic` code:

```
\20=\20*ic\20
```

That's all there is to it. This equation reads: one space equals one space, but ignore consecutive subsequent spaces.

You may define more than one character after the `*ic` code. In fact, you may define as many characters after the `*ic` code as you care to. Each defined character will be ignored if they occur consecutively in the data stream after matching the string on the left side of the equation. Here's an example:

```
.\20.\20.=<ldr>*ic\20.
```

This equation will convert three periods separated by spaces into the string <ldr>, and ignore any subsequent consecutive periods and spaces. Thus, given the following input data:

```
Price. . . . . . . . . . . . . . . $25.00
```

...the equation would convert it to this:

```
Price<ldr>$25.00
```

IGNORE GENERAL

The other specialty context flag code is the Ignore General (*ig) code. This code appears on the right-hand side of an equation and its purpose is to set a trigger that causes all characters from the input file to be ignored, until an Ignore Cancel (*ix) flag code resets the trigger and allows characters to be converted again. Example:

```
[=*ig
]=*ix
```

These two equations will cause anything falling between a left and right square bracket to be ignored, including the brackets themselves. Other characters falling outside of these delimiter characters will be converted in normal fashion. **Note:** this does *not* ignore *equations* placed between these two equations; it is data from the *input file* that are processed and ignored. For example, given the following data:

```
Now is the time for [just about] all good men to
```

The above two equations would create this output result:

```
Now is the time for  all good men to
```

The *ig/*ix notation is the quickest way to remove enclosed groups of characters from a data stream.

Start-of-file and End-of-file Conversions

SNR provides for special conversions to occur before and after normal conversion of a data stream. These conversions can be useful in adding specific headers or trailers to a file, or in removing portions of the beginning and end of a file.

START-OF-FILE PRECONVERSION TRIGGER

To trigger an extra conversion at the start of the file, before any other characters are converted, use the \s start-of-file preconversion trigger. \s can be entered only as the first character on the left-hand side of an equation. It may be used alone, or may be followed by other characters. Example:

```
\s=<BEGIN>\0d\0a
```

The above equation will output the string <BEGIN> as the first item in a converted file.

```
\s=*ic\20
```

The above equation will ignore any group of consecutive spaces occurring at the beginning of a file.

```
\sNow is the time=
```

The above equation will ignore the words "Now is the time" when they occur at the beginning of a file.

END-OF-FILE POSTCONVERSION TRIGGER

The `\q` end-of-file postconversion trigger works similarly to the start-of-file preconversion trigger. When it is used in an equation, it will trigger a conversion at the very end of a file, after all other characters in the file have been converted. `\q` must always stand by itself on the left-hand side of an equation; other characters can not follow it. Example:

```
\q=[END OF JOB]\0d\0a
```

The above equation will output the string [END OF JOB] at the very end of a converted file.

The end-of-file postconversion trigger can also be used to truncate a file when a specific character is encountered in the data stream. (This character may occur anywhere within the file, not necessarily at the end.) This is very useful in cases where some programs add an end-of-file code to any files they write, but you don't want them there. For instance, the DOS COPY command adds a hex 1A to the end of a combined file when you use the "+" operator to copy multiple files into one.

To use this truncation feature, you need to do two things. First, you need to define a hex character that will be recognized specifically as the end-of-file character. At the beginning of your table, you must place the following setup code:

```
\\Qxx
```

...where **"xx"** is a two-digit hex value representing the character that will trigger the end of file postconversion trigger. For instance, the hex value **1A** is the standard DOS text end-of-file character (although it's not often used for that any more), so the setup code for that character would be `\\Q1A`. Second, you need to add an equation for the `\q` postconversion trigger to define what will happen when the end-of-file character is encountered:

```
\q=
```

This particular example ignores the end-of-file character when it is encountered in the data stream, and it terminates processing of the file at that point, thus truncating the file.

SPECIAL \s and \q INFORMATION

Because of the way SNR assembles and categorizes conversion equations, any table that contains a `\q` or `\s` equation must be of a type that contains at least one **1:n** or **m:n** equation (this is an equation type that has one or more characters on the left side of the equation, and two or more characters on the

right side of the equation). `\s` and `\q` equations will not work in a table that contains only **1:0** or **1:1** equation types, because the **1:0** and **1:1** tables use a special high-speed array translation routine that bypasses the standard equation processing routines.

A very simple way to coerce the table processor into categorizing a **1:1** table as an **m:n** type is to add the following equation to it:

```
\80\80=\80\80
```

This will have no adverse effect on the data stream (since it converts two hex 80 characters to themselves), but it will force SNR's table categorizing routine to see the table as an **m:n** type.

Wild Card Codes

Now we get to the fun stuff. While SNR's string conversion capabilities are quite powerful on their own, there are inevitably times when it is necessary to perform conversions based on ambiguous patterns of characters, rather than on literal characters themselves. An example here would be if you had a comma-delimited ASCII file that was created by a database program, and you noticed that none of the phone numbers contained hyphens in them.

It would be quite unfeasible to attempt to literally define all of the possible phone numbers without hyphens, and convert them into phone numbers with hyphens. You would end up with about 7 million equations that went something like this:

```
2220000=222-0000
2220001=222-0001
2220002=222-0002
     ...
9989997=998-9997
9989998=998-9998
```

...hopefully, you get the idea. To help prevent such exercises in futility, SNR provides a number of wild card codes than can be used in place of literal characters in conversion equations. The wild card codes allow you to search and replace on a set of characters rather than on a specific character. For instance, one of the wild card codes can be used to search for any of the numeric characters `0-9`. Another can be used to search for any of the alphabetic characters `A-Z`.

There are 12 wild card definitions in all:

\p any character (text or binary)
\n numeric (0-9)
\x full alphabetic (A-Z, a-z)
\u upper case alphabetic (A-Z)
\y lower case alphabetic (a-z)
\m alphanumeric (A-Z, a-z, 0-9)
\t punctuation
\g non-whitespace

```
\o   whitespace (space, tab, return, line feed,
     vertical tab, form feed)
\z   ASCII-only (c < ASCII 128)
\v   printable (c > ASCII 31)
\k   non-printable (c < ASCII 32)
```

The broadest of these in scope, \p, will match any character that is encountered in the data stream. The narrowest of these in scope, \o, will match only the few "whitespace" characters. The other wild card codes fall between these two extremes, and their associated character patterns should be fairly straightforward to understand.

On the search (left-hand) side of an equation, a wild card code can be used as a straight substitute wherever a literal or hex character would be used. For instance, if you wanted to delete any pattern of three numbers contained in parentheses, you would enter the following equation:

```
(\n\n\n)=
```

This would delete such occurrences as (612), (000), (123) and so on.

On the replacement (right-hand) side, however, wild card code usage requires some explanation. You can't simply use a wild card code in place of a literal character, as you can in the search string.

SNR allows you to rearrange the order of wild card characters during conversion, so we need some method of numbering the characters or otherwise identifying the positions of the wild card codes in the search string so they can be specified in arbitrary order in the replacement string.

This is done easily enough by convention: SNR sequentially numbers the characters in a search string, with the leftmost search character being position 0, the next character position 1, the next position 2, and so on to the end of the string. The equation:

```
(\n\n\n)=
```

...contains five characters: The left parenthesis is character 0, the wild card codes are characters 1, 2 and 3, and the right parenthesis is character 4. To use a wild card in a replacement string, then, we define the position of the character we wish to output, like this:

```
(\n\n\n)=\p1\p2\p3
```

The \p code, when used in a replacement string, is always followed by the position number of the character you wish to output. In the above example, when three numbers enclosed in parentheses are encountered in the data stream, the parentheses will be thrown out and only the three numbers will remain (in the same order in which they were defined in the search string). If we change the equation to read like this:

```
(\n\n\n)=\p3\p1\p2
```

...the three numbers will be output in a different order.

Let's return to the example we opened this section with -- that of replacing a series of phone numbers to add a hyphen between the third and fourth digits. Now that we have wild card codes available as a conversion tool, the task becomes much simpler:

```
\n\n\n\n\n\n\n=\p0\p1\p2-\p3\p4\p5\p6
```

Look at the equation: seven numbers on the search side will be replaced by the first three numbers, followed by a hyphen, followed by the last four numbers. If the data stream contained 4315805, the equation would convert it to 431-5805. We could even handle area codes:

```
\n\n\n\n\n\n\n\n\n\n=(\p0\p1\p2) \p3\p4\p5-\p6\p7\p8\p9
```

The equation as written defines 10 numbers on the search side to be replaced by: a left parenthesis, the first three numbers, a right parenthesis and a space, the next three numbers, a hyphen, and the last four numbers. If the data stream contained 6124315805, the equation would convert it to (612) 431-5805.

As an aid to equation readability, repeated numbers of wild card codes can use the repeating-character notation described in the first section:

```
*(10)\n=(\p0\p1\p2) \p3\p4\p5-\p6\p7\p8\p9
```

This eliminates the need for you to specifically count how many `\n` codes you've entered in a search string. The above equation has exactly the same meaning as the previous equation. In a replacement string, you can also use the repeating-character notation, so long as you identify which position is being initially output:

```
*(10)\n=(*(3)\p0) *(3)\p3-*(4)\p6
```

Look at the equation; it defines 10 numeric characters to be replaced by: a left parenthesis, three numeric characters starting at position 0, a right parenthsis, a space, three numeric characters starting at position 3, a hyphen, and four numeric characters starting at position 6. This is exactly the same equation as the one defined three paragraphs ago, but it is shorter and perhaps easier to read.

Which notation you ultimately use -- long or short -- is purely a matter of personal taste and appropriateness for the equation at hand. Some equations will be more readable using the "long" notation, while others will be a natural fit for the "short" version. There is no need to dogmatically rely upon one or the other.

MIXING WILD CARD CODES

Wild card codes can be intermixed in an equation. For instance, the following equation will convert a comma, a space, two upper case letters, and five numbers into: a comma, a space, two upper case letters, two spaces, and five numbers:

```
, \u\u\n\n\n\n\n=, \p2\p3  *(5)\p4
```

So if this string of characters was encountered in the data stream:

```
, MN55044
```

...the equation would convert it to:

```
, MN  55044
```

CONVERTING SHIFT CASE

Aside from \p, two other wild card codes can be used in a replacement string. Their purpose is to alter the shift case of an alphabetic character. The codes are \u and \y, for upper and lower conversion. As an example, we can make the sample equation dealing with the state and zip code used earlier a lot more universal with these new replacement wild cards:

```
, \x\x\n\n\n\n\n=, \u2\u3  *(5)\p4
```

This new equation will convert a comma, a space, two alphabetic characters of undetermined case, and five numbers into: a comma, a space, two upper case characters, two spaces, and five numbers.

So if the following items were encountered in the data stream:

```
, MN55044
, Ny10022
, il60301
, nJ07680
```

...the equation would convert them to:

```
, MN  55044
, NY  10022
, IL  60301
, NJ  07680
```

Here is set of equations that will convert a text file that is entirely in upper case into one that consists of upper and lower case, in sentence style:

```
\u*00=\p0*01
\u*01=\y0
.=.*00
```

In this equation, when an upper case letter is encountered in the data stream, and flag 0 is ***off***, it is output as-is and flag 0 is set ***on***. If an upper case letter is encountered and flag 0 is ***on***, it is converted to a lower case letter. If a period is encountered, it is output as-is and flag 0 is set ***off***, so the next upper case letter will be output as-is. Any other characters in the file will be passed through as-is.

From this example data stream:

```
NOW IS THE TIME. FOR ALL GOOD MEN. TO COME TO THE AID.
```

...the equation would produce:

```
Now is the time. For all good men. To come to the aid.
```

How would you alter the above small table to convert a file from all upper case to initial upper case, where the first letter of each word is capitalized? Stated another way, what character appears before each word that could be used to control the flag? Here's the answer:

```
\u*00=\p0*01
\u*01=\y0
.=.*00
\20=\20*00
```

Just by adding the last line to the equation, a space will set flag 0 to ***off***, forcing the next character to be output as upper case.

Now the previous sample data stream will be converted to:

```
Now Is The Time. For All Good Men. To Come To The Aid.
```

WILD CARDS AND \s

You can use wild cards in conjunction with the `\s` start-of-file code to remove a number of characters from the beginning of a file:

```
\s*(256)\p=
```

The above equation will remove the first 256 characters from a file (regardless of what they are).

BEWARE OF SIDE EFFECTS!

Wild card codes are powerful tools. They can also be powerfully troublesome if not used with care. Keep in mind that these wild card codes have a hierarchy of pattern-matching: they range from highly inclusive to highly restrictive. If you have two or more wild card equations in your table, be keen to the possibility that one of the equations might end up taking precedence over another, leading to unintended conversions.

SNR bases its pattern-matching tests on the total length of search equations. That is, for any string in the data stream that could potentially match two or more table equations, the equation that is longest will be tried first. The next longest one will be next, and so on down the line until no further equations meet the pattern in the data stream item.

So, for any two equations that begin with the same character, the longer one will be tested before the shorter one. If the longer one happens to match the data stream pattern, it is replaced immediately, and none of the other equations will be tested.

If two wild card equations have the same search string length but begin with a different wild card code, the equation with the wild card code that is *narrower* in scope will be tested first. That is, if an equation beginning with `\x` and one beginning with `\v` are both the same length, the one beginning with `\x` will be tested against the data stream first, because the set of alphabetic characters is narrrower than the set of all printable characters.

Design your wild card search strings carefully to take this behavior into consideration, and your conversions will be more successful.

Variable-Length Conversions

Up to now, the conversions that we have been discussing have dealt with fixed-length patterns of characters: some characters in a search string get converted to some other characters in a replacement string, and we have been able to specifically count the number of characters on either side of the equation.

Now we are going to explore the method SNR has of defining repeating characters or wild card codes whose actual number is unknown. These types of equations are called "variable-length" conversions.

Variable-length conversions are used when you want to search on some identifiable set of characters -- maybe a line of text, maybe a phrase, maybe a command parameter -- but you don't know *how many* characters will be involved in the actual pattern.

For instance, a line of text might contain one word, or it might contain fifty words. An imbedded text command might contain one parameter, or it might contain twelve parameters. A comma-delimited database field might contain ten characters, or it might contain twenty-five characters.

The one thing that these types of text patterns -- a line, a parameter, a field -- have in common is that we wish to treat each of them as a unit, even though we don't know how many characters they will contain.

Using variable-length conversions, you can define these higher-level units of data and translate them, throw them away, or rearrange them as needed.

VARIABLE-LENGTH SEARCH STRING FUNDAMENTALS

A variable-length string is declared to be a set of a specific type of character (literal or wild card) that matches from zero to a stated maximum length of similar characters in the data stream.

Since variable-length search strings are basically unspecified lengths of similar characters, we can draw upon the notation that we prescribed earlier for the repeating-character strings:

```
*(40)\20
```

The above portion of a search string should be familiar to you as the definition for 40 spaces. To turn this into a variable-length definition, we add the `\^` variable-length code:

```
\^*(40)\20
```

This search string fragment now defines *up to* 40 spaces. Likewise, the following fragment defining a string of 25 alphabetic characters:

```
*(25)\x
```

...can be redescribed in variable-length terms like this:

```
\^*(25)\x
```

This search fragment now defines a string of *up to* 25 alphabetic characters.

It is important to adhere to the following rules when you use variable-length search strings:

- An equation cannot begin with a variable-length search string.
- Each variable-length search string must be followed by a fixed-length terminator string.

These restrictions are not quite so arbitrary as they may seem. As for the first rule, each equation must have *at least* one literal character or wild card code that begins the search pattern. The variable-length string can come right after this. As for the second rule, there must be something to tell SNR when the end of a variable-length string has been met; some specific character or set of characters must be defined that can act as a terminator to the variable-length string. The second rule also implies that the search side of an equation cannot end with a variable-length string.

So here is a complete example of a valid variable-length search string:

```
\v\^*(80)\v\0d\0a=
```

Look at the equation. An almost-English description of it, from left to right, is: **a printable character followed by up to 80 more printable characters terminated by a carriage return/line feed to be converted to nothing.** The effect of this equation will be to throw away full lines that are less than 82 characters long, or to throw away the last 82 characters of a longer line.

Multiple variable-length strings can appear in an equation:

```
"\^*(30)\v","\^*(30)\v","\^*(30)\v"\0d\0a=
```

The above equation will search for: **a quote character followed by up to 30 printable characters followed by a quote-comma-quote terminator followed by up to 30 more printable characters followed by another quote-comma-quote terminator followed by up to 30 more printable characters terminated by a quote character and carriage return/line feed, and convert it to nothing.** You may recognize this particular pattern as being characteristic of the comma-delimited ASCII file that many database and mail-merge programs support.

The two examples we used above employed literal characters as terminator strings. It is equally valid to use wild card codes as terminator strings:

```
\n\^*(20)\n\t=
```

The above equation defines **a number followed by up to 20 more numbers followed by a punctuation character to be converted to nothing.**

When you use wild card codes as a terminator to a variable-length string which itself uses wild cards, remember the earlier admonition concerning the hierarchy of wild card pattern matching *(see Beware of Side Effects! in an earlier section).* It applies equally here. Specifically, a wild card terminator must be more restrictive in type than the variable-length wild card string it is terminating.

For instance, the following variable-length string is invalid:

```
\n\^*(10)\n\p=
```

The `\p` code used as the terminator has a higher inclusion value than the `\n` code. The result will be that this equation will generate incorrect output, if indeed it outputs anything at all.

REPLACEMENT STRING FUNDAMENTALS

The prior discussion concerned itself exclusively with the procedures for defining the search side of an equation. Now we will turn our attention to defining the replacement side of a variable-length equation.

Since we don't know the actual number of characters that a variable-length string will pick up, it is not possible to use the `\p` notation to identify precise character positions for output. Instead, we identify groups of variable-length strings by the positions of the *group itself*: the first variable-length string on the search side is number one, the next variable-length string in the equation is number two, and so on. (This is in contrast to the `\p` wild card code, which begins at zero.)

Up to 40 variable-length strings may be entered in a single equation.

The variable-length replacement strings are defined as `\^1, \^2, \^3` and so on. Let's take an example based on the comma-delimited ASCII file, used a few examples ago:

```
"\^*(30)\v","\^*(30)\v","\^*(30)\v"\0d\0a=\^1\09\^2\09\^3\0d\0a
```

While the search side of the equation remains the same as before, we have now defined a replacement side. The replacement will be **the first variable-length string followed by a tab code (the hex 09) followed by the second variable-length string followed by a tab code followed by the third variable-length string followed by a carriage return/line feed.** The effect of this equation will be to convert a comma-quote delimited file into a tab-delimited file. (Digression: for purposes of illustration this example is fine, but is there an easier way to convert a comma-delimited file into a tab-delimited file? The answer appears in a later section.)

You may have observed by insight that it is possible on the replacement side of the equation to change the order of output of the variable-length strings simply by rearranging them (for instance, as `\^3\^1\^2)`. You may also leave one or more of the definitions out of the replacement if you wish to throw them away.

USING WILD CARD CODES AS TERMINATORS

How do you output a wild card character that has been defined as a terminator to a variable-length string? You can't define it in the replacement by exact position, since the length of a variable-length string is not known at the time you write your equation. For this reason, SNR provides a notation that is not dependent on a character number to describe a variable-length terminator string. Since each variable-length string in an equation is numbered from 1 to 40, and by definition each variable-length

string *must* be followed by a terminator string, it makes sense that the terminator strings are also numbered from 1 to 40. A variable-length equation will always have *at least* one variable-length string and a terminator.

The notation for terminator strings on the replacement side is `\^:1, \^:2, \^:3` and so on (the added colon is the only difference between the terminator's notation and the notation for the variable-length string itself).

Now having some way of differentiating between variable-length strings and terminators, we can convert equations such as the following:

```
\u\^*(15)\y\u\^*(15)\y\t=\p0\^1\^:1\20\^2\^:2
```

Whew! Stay with me here, this actually makes sense. This contrived example serves to illustrate the placement and usage of the various codes. The search side defines **1) an upper case character, 2) followed by up to 15 lower case characters, 3) terminated by an upper case character, 4) followed by up to 15 more lower case characters, 5) terminated by a punctuation character.** The replacement side outputs **1) the first character of the string (the first `\u` on the search side), 2) followed by the first variable-length string, 3) followed by the first terminator, 4) followed by a space, 5) followed by the second variable-length string, 6) followed by the second terminator.**

For a concrete example, assume that this string occurs in the data stream:

```
SuperDuper! UnquestionablyAwesome.
```

Our example equation will convert it to this:

```
Super Duper! Unquestionably Awesome.
```

Let's try switching the order of a few elements in the equation:

```
\u\^*(15)\y\u\^*(15)\y\t=\^:1\^2\20\p0\^1\^:2
```

The search side remains the same; the replacement side has changed. Study it and see if you can follow the meaning. Perhaps this will help: given the same example data stream as above, the altered equation will produce this output:

```
Duper Super! Awesome Unquestionably.
```

At first blush, understanding the notation for variable-length strings seems as obtuse as, oh, programming in C++. Actually, if you concentrate on each of the individual character notations one at a time from left to right, and build up the equation that way, rather than trying to view the equation as a whole, it is easier to make sense of it.

PADDING VARIABLE-LENGTH STRINGS

A specialized use of variable-length replacement strings allows the strings to be padded out to their maximum width. Normally, of course, the strings are output to their "natural" width -- however many characters match the variable-length search string are what get output during replacement.

Padding the strings adds blank spaces to the left or right of the actual text of the variable-length string, and it has the effect of aligning the text left or right within a field whose width is specified in the variable-length search string.

To invoke variable-length padding, the alignment codes R and L are inserted before the variable-length string number in a replacement string. For instance, `\^L1` will left-align the first variable-length string; `\^R1` would right-align the first variable-length string. Here's how the codes would be used in an equation:

```
"\^*(30)\v","\^*(30)\v","\^*(30)\v"\0d\0a=\^L1\^L2\^R3\0d\0a
```

Taking our well-worn comma-delimited example yet again, pay particular attention to the replacement side. Those codes specify that **1) the first variable-length string will be output left-aligned to a width of 30 characters, 2) the second variable-length field will be output left-aligned to a width of 30 characters,** and **3) the third variable-length field will be output right-aligned to a width of 30 characters, 4) followed by a carriage return/line feed.** What we have done in this example is converted a file containing comma-delimited fields into a file containing fixed-length fields. Cool!

The widths of the fields can be changed simply by changing the width values in the search strings:

```
"\^*(35)\v","\^*(25)\v","\^*(40)\v"\0d\0a=\^L1\^L2\^R3\0d\0a
```

Here, the replacement side of the equation is the same as before, but the search side now contains different widths for each of the fields. This will affect the appearance of the output when the text is padded. If you left out the R and L alignment specifiers from the replacement side, the text strings would appear butted against.each other with no space between them.

PAD CHARACTER

By default, variable-length strings are padded with spaces when the R or L alignment codes are specified. You can select a different padding character by placing the setup code `\\Axx` at the beginning of your table, where ***xx*** is a hex value representing the character you wish to use as the padding character. Only one padding character may be defined per table, and it can not be changed partway through the table.

As an example, you could pad your variable-length text strings with periods by using the setup code:

```
\\A2E
```

Hints and Tips

The conversion capabilities of SNR are very sophisticated. Some of them are subtle, and require some study and experimentation to master. Here is a list of a few pointers that might help you avoid the pitfalls of inexperience.

1. ***Lots of smaller tables are better than one large table.***

It is sometimes tempting to create a huge, do-it-all conversion table that takes care of your conversion task all at once. Let me tell you from experience that it is much better to break down your conversion into a series of smaller tables that are run one after the other.

One reason behind this is that it is much simpler to pinpoint problems when you take your conversion a little bit at a time, and you won't waste your energy poking around a large table feebly trying to discover why your text isn't converting the way you thought it should.

Remember that SNR32 allows you to run up to 20 tables at one time on the command line anyway (and up to 50 tables in SNR95), giving you the same benefit as having one large table.

Another reason to break your task into smaller pieces is that some of the more advanced conversion models -- such as variable-length strings -- may have unwanted side effects on other equations or vice versa. It is safer to isolate the complex portions of a conversion task into their own tables to insure that bizarre conversion side effects are not introduced.

2. Use literal characters instead of wild cards wherever possible.

Wild cards are very powerful, but they require much more time to process than literal characters. For speed of conversion, it is preferable to define conversion equations using literal characters instead of wild cards.

If your conversion task can be represented by a few well-defined data patterns that are limited in scope, then use literal equations. If, on the other hand, you have a particular need to search a wide range of data patterns, then the wild card equation is your best choice.

Remember the question I posed back in the section that described variable-length equations? I asked if there was a better way than using variable-length strings to convert a comma-delimited text file into a tab-delimited text file. Of course there is; it would have been a rhetorical question otherwise. If you think about it a little bit, there is no real need to do any variable-length conversion between the delimiters at all. Only the delimiters themselves need to be converted (has the light bulb come on yet?). So rather than chewing up a lot of CPU cycles scanning the data stream for variable-length text, the following two equations will perform the conversion much faster:

```
"=
","=\09
```

An added benefit to using this table over the other one is that this version does not need to be modified when the number of fields changes, whereas the other one does. I guess this is one of those situations where **Less Is More**.

3. Use context flags rather than variable-length patterns when you are seeking to compress the data stream.

This is another issue related to efficiency. Variable-length conversion strings provide a means of reading an unspecified number of consecutive characters and eliminating them, in effect compressing the data stream. But equations using context flags such as `*ic` or `*ig` provide a much faster conversion than variable-length equations.

4. Debug your tables one equation at a time.

Debugging is a cross between an art and a science; it requires both discipline and inspiration to do it well. The first step in effective debugging is to actually find the trouble spot and focus your attention on it.

The easiest way to do debug SNR equations is to first place a comment code (backslash followed by a space) in front of every equation in your table, and then uncomment each equation one at a time, testing a sample of your data file each time you uncomment an equation. Eventually, you will hit upon the one equation that is fouling things up, and you will be able to alter it or isolate it by moving it to another table, where it can be dissected and analyzed apart from other equations (and possibly rewritten).

Once the trouble equation has been found, look at it carefully. Consider these possibilities:

- Are you sure the equation is structured according to the rules set forth in this document?
- Does each element of the equation logically match a likely pattern in the data stream?
- Are your wild card definitions too vague to match a specific pattern in the data stream?
- Are there any possible conflicts between elements of the equation that would render it invalid?
- Can the equation be restated equivalently in different terms, that is, using a different conversion model (for example, transforming wild cards into literals)?
- Is what you are trying to do reasonable or even possible in the SNR conversion paradigm? The stream-mode data conversion paradigm can not handle every possible transformation need.
- Have you done something like this before and made it work?
- Is there a similar equation shown in the sample tables?
- Could there be a bug in SNR?

Seeking the answers to these questions will help guide your troubleshooting steps in a logical, scientific manner.

5. Use lots of comments.

This point was brought up in the first section of the manual, but it bears repeating. **Use lots of comments.**

6. Peruse the provided example tables.

The range of possible uses for SNR, and the examples to support them, are beyond the scope of this manual. Many of the more complex and subtle uses of conversion strings can only be appreciated by experimentation or by looking through the example tables provided in the archive file.

The tables may help provide you with hints on syntax and usage that this manual was unable to. If you register the program, you will be entitled to telephone or on-line support (toll at your expense) and I will be happy to help you solve your conversion problems.

Miscellaneous

NOTES

System requirements are an IBM-compatible PC with a 486 CPU or higher, running Windows95 or Windows NT. (The program might run under Windows 3.1 with the Win32s operating system extension, but I have not tested this configuration, nor will I declare to support it.)

SNR was developed with Borland C++Builder 1.0.

DISCLAIMER

This program is distributed as shareware. Use it, copy it, upload it, give it to your friends. Please distribute only the complete program in archived form, including all document and sample files. No warranties, either expressed or implied, are given by the author or distributor of the program, and the user accepts all risk of damage arising out of the application and use of the program.

REGISTRATION

If you like SNR, please register your copy for $60. At that price, it's one of the bargains of the text processing world, and I'll bet you recoup that investment dozens, maybe hundreds, of times over in the work it will save you.

Registered users will receive a version of the program that does not have the compliance delay screen. The registered version *may* contain additional features that are not in the shareware version, but I don't promise this.

Please fill out and return with your payment the form in the file ORDERFRM.DOC.

Registered users are guaranteed a response to technical support questions by phone, mail, or CompuServe. Unregistered users will receive responses on a time-available basis, but I can't guarantee that I will respond to those requests.

REVISION HISTORY

6.00 & 6.1 - 5/31/97
32-bit version for Windows95 and Windows NT. Command-line and GUI versions. Baseline functionality remains the same as 5.08.

5.06 - 4/16/95
Added stdout, stderr and stdprn as valid output names. This permits SNR output to be redirected or piped.

5.02 - 5/26/93
Major rewrite. Added advanced conversion features such as wild card pattern matching, variable-length patterns, field padding, and so on. Allow file names to contain DOS wild cards, also. Allow multiple-table chaining on the command line. Allow output file subdirectory specification. Greatly expanded program capacities. Allow 16 context flags. Change default bit-masking from 7-level to 8-level as this is the more harmless option.

4.0 - 7/19/90
Removed OS/2 compatibility. Nobody used it. Allow 256 1:1 equations to be loaded in addition to 50 m:n equations. Added context flag.

3.01 - 11/3/89
Program recompiled for family mode, meaning it will run as-is under DOS, OS/2 real mode, and OS/2 protected mode.

Also improved I/O throughput on fast disk drives.

3.0 - 9/27/89
The program has been completely rewritten. The pesky bugs that were present in versions 1.5 and 2.0 have been eliminated.

All binary characters INCLUDING NULLS can now be searched and replaced. Nulls can also be entered as part of a larger search-and-replace equation.

An automatic "bit-stripping" feature has been added which removes the high-order bits from input characters, forcing 8-bit codes into 7-bit ASCII. The feature can be selectively disabled for binary files.

The equation length has been increased to 200 characters, which can be split into search and replace sides of any length, as long as the total does not exceed 200.

Equations no longer need to be in any specific order.

The average speed of the program has been vastly improved, especially for multiple and high-occurrence search and replaces.

The program no longer needs to read itself for runtime parameters.