Contents

Contents

APPENDIX A: STATISTICAL FORMULAS

MENTOR (BUILDER/PRINTER)

All the tests in this section are performed in the print phase of the run based upon calculations executed in the table-building phase of the run as a result of using the DO_STATISTICS command.

Preliminaries

There are N cases indexed by j = 1, 2, . . . , N. Each case has a weight W_j and observations in none, some, or all of I groups; indexed by k = 1, 2, . . . , I and/or by m = 1, 2, . . . , I:

$$X_{1j}, \ldots, X_{Ij}$$

For any case, the weight W_j and/or any of the variables X_{kj} may be missing, either by design or by the happenstance of collection. In all formulas below, missing values are assumed to be excluded from the summations (the additional and cumbersome notation to show this is also omitted) and the index of summation is similarly omitted when the range is obvious. For summations involving the variable X_{kj}, we use the notation W_{kj} to indicate the weight W_j if X_{kj} is present. For two variables X_{kj} and X_{mj}, we use W_{kjm} to mean W_j if both X_{kj} and X_{mj} are present. Finally, we say X_{kjm} to mean X_{kj}, if X_{mj} is present.

Basic Sums and Statistics

After all data reading is completed, the following summations and counts are available:

UNIVARIATE:

Counts:	$F_k = \cdot W_{kj}$	$k = 1, \ldots, I$
Sums:	$S_k = \cdot W_{kj}X_{kj}$	
Sums of Squares:	$Z_k = \cdot W_{kj}X_{kj}^2$	

Squares of Weights: $Y_k = \cdot W_{kj}^2$

BIVARIATE:

Counts: $F_{km} = \Sigma W_{kjm}$ $k \neq m$

Sums: $S_{km} = \Sigma W_{kjm}X_{kjm}$

Sums of Squares: $Z_{km} = \Sigma W_{kjm}X_{kjm}^2$

Squares of Weights: $Y_{km} = \Sigma W_{kjm}^2$

Cross Products: $T_{km} = \Sigma W_{kjm}X_{kjm}X_{mjk}$

CAUTION: Only $F_{km} = F_{mk}$ is generally true. Otherwise, you have to be careful with the order of subscripts.

The following statistics are all available. **Note:** If processing is unweighted (i.e., $W_j = 1$ for $j = 1, \ldots, I$) then all the formulas reduce to their usual unweighted versions.

Mean: $M_k = S_k/F_k$

Adjusted sum of squares: $A_k = (Z_k - S_k^2/F_k)/(\Sigma_j W^2/\Sigma_j W) = (Z_k - S_k^2/F_k)/(Y_k/F_k)$

Effective sample size*: $E_k = F_k^2/Y_k$

Variance: $V_k = A_k/(E_k - 1)$

Standard deviation: $SD_K = \sqrt{V_K}$

Standard error: $SE_k = SD_k/\sqrt{E_K}$

Correlation:

$$R_{km} = \frac{T_{km} - S_{km} \cdot S_{mk}/F_{km}}{\sqrt{(Z_{km}) - (S_{km}2)/F_{km})(Z_{mk} - (S_{mk}2)/F_{km})}}$$

Bivariate effective

Sample size*: $E_{km} = F_{km}^2/Y_{km} = \sqrt{V_K/E_K}$

Sets of Variables: Newman-Keuls Preliminaries

Suppose β is a subset of the indices {1, 2, . . . , I}. We will need:

V_β, the pooled variance and

DF_β, the associated degrees of freedom.

Let $E_\beta = \Sigma_\beta E_k$ then,

Case One: The Xs are 0 - 1 variables (means are proportions).

Let $P_\beta = \Sigma_\beta S_k / \Sigma_\beta F_k$

$$V_\beta = \frac{P_\beta (1-P_\beta)}{1-1/E_\beta}$$

$DF_\beta = E_\beta - 1$

Case Two: The Xs are continuous (or categorical).

Let G_β = the number of indices in β

$$V_\beta = V_\beta = \frac{\Sigma_\beta A_K \bullet Y_k / F_k}{\Sigma_\beta (E_k - 1) Y_k / F_k}$$

$DF_\beta = E_\beta - G_\beta$

*For a discussion of this concept, see:

- *"Equivalent Sample Size" and "Equivalent Degrees of Freedom,"* Refinements for Inference Using Survey Weights Under Superpopulation Models by Richard F. Potthoff, Max A. Woodbury, and Kenneth G. Manton.

- *American Statistical Association Journal* or the *American Statistical Association,* June 1992, Vol.87, No.418, Theory and Methods, pages 383 – 396.

Then we define the Student differences Q_{km} k, m ε β and the associated degrees of freedom as:

$$Q_{km} = \frac{(M_k - M_m)}{\sqrt{\frac{V_\beta}{2} \cdot \left(\frac{1}{E_k} + \frac{1}{E_m} - \left(\frac{2}{1} \cdot \frac{R_{km} \cdot E_{km}}{(E_k \cdot E_m)} \right) \right)}}$$

$$DF_{km} = E_k + E_m - 2 \qquad E_{km} = 0$$

$$= E_k + E_m - E_{km} - 1 \qquad E_{km} > 0$$

MOTIVATION

To motivate the use of these formulas, consider the following collection of Normal random variables:

$$\begin{array}{llll} X_{1j} & j = 1, \ldots, N_1 & \sim & N(a_1, \sigma_1^2) \\ X_{2j} & j = 1, \ldots, N_c & \sim & N(a_2, \sigma_2^2) \\ X_{3j} & j = 1, \ldots, N_c & \sim & N(a_3, \sigma_3^2) \\ X_{4j} & j = 1, \ldots, N_4 & \sim & N(a_4, \sigma_4^2) \end{array}$$

with all variables independent except:

$$\text{cov}(X_2, X_3) = \nu\sigma_2\sigma_3$$

and define:

$$M_1 = \frac{\Sigma X_1 + \Sigma X_2}{N_1 + N_C}$$

$$M_4 = \frac{\Sigma X_3 + \Sigma X_4}{N_C + N_4}$$

Then M_1, M_4 and M_1 - M_4 are all Normal:

$$M_1 \sim N \quad (b_1 = \frac{N_1 a_1 + N_c a_2}{N_1 + N_c} \ , S_1 = \frac{N_1 \sigma 1^2 + N_c \sigma 2^2}{(N_1 + N_c)^2})$$

$$M_2 \sim N \quad (b_4 = \frac{N_c a_3 + N_4 a_4}{N_c + N_4} \ , S_4 = \frac{N_c \sigma 3^2 + N_4 \sigma 4^2}{(N_c + N_4)^2})$$

$$M_1 - M_4 \sim N \ (b_1 - b_4, S_1 + S_4 - \frac{2N_c \nu \sigma_2 \sigma_4}{(N_1 + N_c) + (N_c + N_4)})$$

In the following common cases, we can simplify the formulas and see more familiar forms.

1 No overlap, independent groups: $N_c = 0$

$$M_1 - M_4 \sim N \ (a_1 - a_4, \frac{\sigma 1^2}{N_1} + \frac{\sigma 4^2}{N_4})$$

2 Complete overlap, correlated observations: $N_1 = N_4 = 0$

$$M_1 - M_4 \sim N \ (a_2 - a_3, \frac{\nu \sigma_2 \sigma_4}{N_c})$$

3 M_4 is a subset of M_1: $N_1 = 0$, $\nu = 1$, $a_3 = a_4$, $\sigma_3 = \sigma_4$

$$M_1 - M_4 \sim \frac{N_4}{N_4 + N_c} \bullet N \ (a_2 - a_4 \ , \frac{\sigma 2^2}{N_c} + \frac{\sigma 4^2}{N_4})$$

(Note that tests of M_1 - M_4 are equivalent to tests of X_2 - X_4.)

The Newman-Keuls Procedure

Given a set of indices β, do the following with all the indices k, m ε β.

Step 1: Initial ordering. Put the indices in order as follows:

- If all variables are computationally independent, i.e., $E_{km} = 0$ for all k, m ε β, k (not equal to) m, then sort the indices in order by the means M_k
- Otherwise, sort the indices in order by the sums, i.e., $Sum_k = \sum_{\substack{1 \varepsilon \beta \\ m \neq k}} Q_{km}$

In either case, sort ties in any order. Set the lowest index to test to the first one, the highest index to test to the last one. Finally, calculate V_β.

Step 2: This step is applied to any contiguous subset of the sorted indices. We start with the first index in the subset (Lo) and the last (Hi). First, check ties:

2.1 If there are ties in the sort basis at the Lo end, then reorder, putting the group with the largest value of E lowest; similarly if there are ties at the high end, put the largest group highest.

2.2 If the Xs are 0 - 1 (proportions), recalculate V_β only on the indices between Lo and Hi and recalculate $Q_{Lo, Hi}$.

2.3 Count the number of groups N_g between Lo and Hi, and test $Q_{Lo, Hi}$ using a Newman-Keuls table with (DF_β, N_g) degrees of freedom.

2.4 If the difference is not significant, go to Step 4; otherwise, mark the difference (Lo, Hi) as significant, along with all other groups that are tied on both the sort basis and sample size.

Step 3: Raise the Lo value to above the ties and go back to Step 2 if $Lo \neq Hi$.

Step 4: Lower the Hi value to below the ties and go back to Step 2.

Statistical Testing In Mentor

All tests generated by the **DO_STATISTICS=** command are tested as follows:

1 **Tests Between Columns:** Use the Newman-Keuls procedure except when:

1.1 $E_k/E_m > 2$ for some k, m ε β or

1.2 The user specified separate tests.

In case 1.1, calculate V_β and Q_{km} k, m ε β and test each Q separately using DF_{km}, G_β as degrees of freedom.

In case 1.2, calculate V_β, Q_{km} for S = {k,m} (i.e., for every pair) and test each Q_{km} separately using DF_{km}, 2 as degrees of freedom.

(We use the Student Newman-Keuls range table. In this case, a version of the conventional T-table.)

2 **Tests Between Rows:** These are the same as a T-test with independent groups, i.e., the same as 1.1 with two indices and $T_{km}=0$. For preference tests, unknowns are split evenly between the two groups for reporting, but excluded from the statistical testing.

3 Difference Tests:

Write $D = e_1x_1 + e_2x_2 \ldots + e_mx_m$ with $\left(\sum_j\right)\left(\sum_m\right)\frac{E_{km}e_ke_m}{E_kE_m}$

$e_k = 1$ or -1 for $k = 1, \ldots, I$.

For example, $D = (x_1 - x_2) - (x_3 - x_4) = x_1 - x_2 - x_3 + x_4$

then let $D_1 = \Sigma\, e_k m_k$

Continuous: $V = V_{\{1,2,\ldots,I\}}$ •

Proportion: $V = V_{\{1,2,\ldots,I\}}$

and test $D_1/\sqrt{v/2}$ with degrees of freedom $(DF_{\beta,}2)$

TABLE-BUILDING PHASE

The tests in this section are calculated entirely in the table-building phase. They may be subsequently printed or not. But no further computation is done in the printing phase.

NOTATIONS AND MISCELLANEOUS FACTS

Suppose $(X_1, Y_1), \ldots, (X_n, Y_n)$ are independent $\sim N(\mu, \Phi)$

with $E[(X_i)] = \mu_x$ $E[(X_i - \mu_x)^2] = \sigma_x^2$ $i = 1, \ldots, N$

$E[Y_i] = \mu_y$ $E[(Y_i - \mu_y)^2] = \sigma_y^2$

$$E\left[\frac{(X_i - \mu_x)}{\sigma_x} \bullet \frac{(\mu_y)}{\sigma_y}\right] = \nu$$

Let $W_1, W_2, \ldots, W_n$ be real numbers with $W_i > 0 \qquad i = 1, \ldots, N$

Define $F = \cdot W_i$

$$S_x = \cdot W_i X_i \qquad S_y = \cdot W_i Yi$$

$$V_x = \cdot W_i X_i^2 - \frac{S_x 2}{F} \qquad V_y = \Sigma W_i Y_i^2 - \frac{S_y 2}{F}$$

$$C = \Sigma W_i X_i Y_i - \frac{S_x S_y}{F}$$

then $S_x \sim N(F\mu_x \; \Sigma W_i^2 \sigma_x^2) \qquad S_y \sim N(F\mu_y, \Sigma W_i^2 \sigma_y^2)$

so $E[S_x] = F\mu_x \qquad E[S_y] = F\mu_y$

$$E[V_x] = \sigma_x^2 \left(F - \frac{\Sigma W_i 2}{F}\right) \quad E[V_y] = \sigma_y^2 \left(F - \frac{\Sigma W_i 2}{F}\right)$$

$$E[C] = \nu \bullet \sigma_x \sigma_y$$

ESTIMATE FOR MEANS, STANDARD DEVIATIONS, STANDARD ERRORS AND CORRELATIONS

General Strategy: If $E[f(x_1, \ldots, x_n, y_1, \ldots, y_n)] = g$

then estimate g with f, i.e., $\hat{g} = f(x_1, \ldots, x_n, y_1, \ldots, y_n)$

so that $\hat{g}$ is, at least, unbiased. This easily gives

$$\hat{\mu}x = \frac{S_x}{F} \qquad \hat{std}_x = \sqrt{\hat{\sigma}_x 2} = \sqrt{\frac{v_x}{F - \frac{\Sigma(W_i)2}{F}}}$$

$$\hat{\mu}y = \frac{S_y}{F} \qquad \hat{std}_y = \sqrt{\hat{\sigma}_y 2} = \sqrt{\frac{v_y}{F - \frac{\Sigma(W_i)2}{F}}}$$

(Note: σ_x^2 is unbiased, but $\hat{std}_x$ is not.)

For Standard Error, if $\hat{se}_x = \sqrt{E[(\hat{\mu}_x - E[\hat{\mu}_x])^2]}$

then $\hat{se}_x = \sqrt{\frac{\Sigma W_i 2}{F^2}\sigma_x 2}$ so,

$$\hat{se}_x = \sqrt{\frac{\hat{\sigma}_x 2}{\left(\frac{F^2}{\Sigma W_1 2}\right)}} \qquad \hat{se}_y = \sqrt{\frac{\hat{\sigma}_x 2}{\left(\frac{F^2}{\Sigma W_i 2}\right)}}$$

(Note**:** As with $\hat{std}$, $\hat{se}$ is not unbiased, but its square is.)

More Estimates

This formula $\hat{v} = \frac{C}{\sqrt{\hat{std}_x \hat{std}_y}}$

is not unbiased, though built in an obvious way from unbiased estimators.

This formula $\hat{t} = \frac{\hat{\mu}_x - \hat{\mu}_y}{\sqrt{\hat{se}_x 2 - 2\hat{v}se_x se_y + \hat{se}_x 2}}$

has a t-distribution when $W_1 = W_2 =, \ldots, W_n$, but is otherwise different by an unknown amount.

FORMULAS FOR STATISTICS CREATED DURING TABLE BUILDING (ROW=)

Suppose X_i, W_i, i = 1, . . . , N are the values and weights for a set of N cases. (If the table is not weighted, it is the same as if $W_i = 1$ for all cases.)

The FREQUENCY $F = \Sigma W_i$

The SUM $S = \Sigma W_i X_i$

The MEAN $\overline{X} = S/F$

$$N = \frac{F^2}{\Sigma W^2} \qquad N - 2 = \frac{F^2 - \Sigma W^2}{\Sigma W^2}$$

The STANDARD DEVIATION $sd = \sqrt{\frac{\Sigma x^2 w - \frac{S^2}{F}}{F - \frac{\Sigma w^2}{F}}}$

The STANDARD ERROR se = $se = \frac{std}{\sqrt{\frac{F^2}{\Sigma w^2}}}$

Note: If all the weights W_i are the same value, whatever it might be so long as its bigger than zero, then you get exactly the same values for , sd, se weighted or not weighted. This is also true for T and DEPT on the following page.

The T-TEST: Suppose you have two groups, each with its own Mean and Standard Error.

	Group 1	**Group 2**
Mean	$\bar{X}_1$	$\bar{X}_2$
Standard Error (SE)	se_1	se_2

$$T = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{(se_1)^2 + (se_2)^2}}$$

THE DEPENDENT T-TEST: Suppose you have two values - X, Y for each person. Each has a Mean and Standard Error (using the same weights).

	Group X	**Group Y**
Mean	$\bar{X}$	$\bar{Y}$
Standard Error (SE)	se_x	se_y

You also need $$CXY = \frac{\Sigma XYw - \frac{\Sigma Xw \Sigma Yw}{\Sigma w}}{\left(\Sigma w - \frac{\Sigma w^2}{\Sigma w}\right)\left(\frac{(\Sigma w)^2}{\Sigma w^2}\right)}$$

$$DEPT = \frac{\bar{X} - \bar{Y}}{(se_x)^2 - 2CXY + (se_y)^2}$$

FORMULAS FOR STATISTICS CREATED DURING THE PRINT PHASE (EDIT=)

ANOVA

The tests in this section are calculated in the print phase based entirely upon the numbers that were created in the table-building phase. For each ANOVA to be done, think about the table cut down to just the columns and rows to be used:

ROWS	Column 1	Column 2, ...	Column C	WEIGHTS
1	N_{11}	...	N_C1	W_1
?	N_{ij}	...	...	...
...	...	...	...	...
R	N_{1R}	...	N_{CR}	W_R

For each column i: $Z_{1i} = \left(\sum_{j}\right) N_{ij} \bullet W_j$

$$Z_{2i} = \left(\sum_{j}\right) N_{ij}$$

OVERALL:

$$N_T = \left(\sum_{ij}\right) N_{ij}$$

$$SS_T = \left(\sum_{ij}\right) N_{ij} W_j^2 - \frac{\left(\sum_{ij} N_{ij} W_j\right)^2}{N_\Gamma}$$

$$SS_B = \left(\sum_{ij}\right) N_{ij} W_j^2 - i \frac{Z_{ij}^2}{Z_{2i}}$$

$$DF_N = C - 1$$

$$DF_D = N_T - DF_N$$

$$SS_W = SS_T - SS_B$$

$$F = (SS_B/DF_N)/(SS_W/DF_N)$$

$$MS_N = SS_B/DF_N$$

$$MS_D = SS_W/DF_N$$

T AND Z TESTS

These tests assume columns. In the special case that one group is the total column, then the test is column versus total-column.

T - Test

$$T = \frac{\bar{X}_1 - \bar{Y}_2}{\sqrt{(SE_1)^2 + (SE_2)^2}}$$

Z - Test

$$Z = \frac{\bar{X}_1 - \bar{Y}_2}{\sqrt{\frac{1}{N_2} \bullet \frac{\Sigma X_1^{\,2} - \frac{(\Sigma X_1)^2}{N_1}}{N_1} + \left(\frac{1}{N_2} \bullet \frac{\Sigma X_2^{\,2} - \frac{(\Sigma X_2)^2}{N_2}}{N_2} \right)}}$$

Like T, but does not adjust N to N-1

WALKER T-TEST

Calculate $F=\left(\frac{STD_1}{STD_2}\right)^2$ with $DF = N_1 - 1, N_2 - 2$

If F is significant at .05 use $D = (SE_1)^2 + (SE_2)^2$

Otherwise, use $D=((\Sigma X_1^2 - + \Sigma X_2^2 - \frac{(\Sigma X_2)^2}{N_2} /(N_1 + N_2)\left(\frac{1}{N_1} + \frac{1}{N_2}\right)$

Then $T = \frac{\overline{X}_1 - \overline{Y}_2}{\sqrt{D}}$

RANK SUM/WILCOXEN TEST

1 Rank the cells in the cumulative sample. Call the ranks $R_1, R_2, \ldots$, etc.

2 Call the sample size of the smaller group N_1. For this group, let:

$$T = \Sigma N_1 R_1$$

3 Call the size of the bigger group N_2. $N = N_1 + N_2$

4 Normalize T with

$$E(T) = \frac{N_1(N_1 + N_2 + 1)}{2}$$

$$VAR(T) = \frac{N_1 N_2(N_1 + N_2 + 1)}{12}$$

5 Correction for ties: Call the ties T_1, T_2, . . . , etc.

$$\text{VAR (T)} = \frac{N_1 N_2}{N(N-1)}\left(\frac{(N_3 - N)}{12} - \Sigma\frac{(T_1^{\,3} - T_1)}{12}\right)$$

MORE REFERENCES:

THE KRUSKAl WALLIS TEST

For a discussion of this statistical test, refer to:

- Syndey Siegel and N. John Castellan, Jr., "*NONPARAMETRIC STATISTICS FOR THE BEHAVIORAL SCIENCES*," Chapter 8, pages 207-215.

THE ANOVA SCAN AND FISHER TESTS

- Sir Maurice Kendall, Sc.D., F.B.A. and Alan Stuart, D.Sc. (econ.), *"THE ADVANCED THEORY OF STATISTICS,"* Volume 3, Design and Analysis and Time-Series, Third Editon, pages 43-46.

CHI-SQUARE TESTS

For a discussion of Chi-Square tests, refer to either:

- Wilfred J. Dixon and Frank J. Massey, Jr., "*INTRODUCTION TO STATISTICAL ANALYSIS,"* Third Edition, Chapter 13, Enumeration Statistics, pages 237-243
- John T. Roscoe, *"FUNDAMENTAL RESEARCH STATISTICS FOR THE BEHAVIORAL SCIENCES,"* Chapter 29, Chi-Square Tests of Independence, pages 196-203.

APPENDIX B: TILDE COMMANDS

SYNTAX AND OPTIONS

This section describes the syntax and options to Mentor tilde commands. Tilde commands are preceded by the tilde (~) character. These commands can be invoked at any time. While the tilde character must be the first character on the line, it does not have to be in the first column.

Many tilde commands have subcommands that have their own syntaxes and sets of options. These commands are called *keywords*. Once you enter a tilde command, you do not need to enter it again for any of its keywords. This set of related commands is known as a *tilde block*. You are in a tilde block until you issue another tilde command.

Syntax:

```
~COMMAND
keyword
keyword
~NEXT TILDE BLOCK COMMAND
```

The exception to this block rule is ~FREQUENCY, a command that requires that all options must be specified on the same line.

Meta commands (i.e., >REPEAT), file references (&filename), and comment characters (") can also be used in any command block. See the *Utilities* manual for information commands.

All tilde command blocks can be commented out (that is, the all the commands in that block will be ignored by Mentor) by placing a minus sign after the tilde and before the command. This can be useful to quickly comment out a block of commands.

Example:

```
~-DEFINE
```

In this case, the command, any options in the block will be ignored. This does *not* affect meta commands in the block.

Tilde commands can be assigned a label and then referenced later. This is often used with branching (see ~GO_TO for more information).

Example:

```
~label: DEFINE
```

Many keywords in a tilde block can be turned off by putting a minus sign before the command. They will be noted.

Example:

```
~SET -AUTOMATIC_TABLES
```

This will turn off automatic table generation.

Most commands can be abbreviated; supported abbreviations are shown in parentheses after the command description and are often used in the examples. Underscores (_) between words are to make the commands easier to read, and are optional. For example, you can use ~MAKEREADCONTROL for MAKE_READ_CONTROL. See the *Utilities* manual *Appendix B: ALLOWED ABBREVIATIONS* for more information on abbreviations.

If you are using more than one option for a command, you can separate them with spaces, commas, or both.

Example:

```
~INPUT  NEWFILE,LENGTH=5/80,STOP=120
```

FILE NAME EXTENSIONS

You can specify your own file names, up to the maximum allowed by your operating system: DOS=eight plus three-character extension; UNIX=255 characters. If you use the command >longfilenames, dos will also allow 255-character names. If you put the filename inside quotes, you can use special characters in the name like spaces.

Mentor adds and checks for the following extensions on some commands:

Command	File Type	Extension	Note
~input/~output	Data	TR	if filetype not specified
>printfile	Print	PRT	
>usedb/>createdb	Database	DB	

You can direct Mentor not to add or check for these extensions by using $filename in your spec files (e.g. $myfile). You can also use the meta command >-CFMC_FILE_EXTENSIONS. This makes $filename the default for any filename you supply.

Example:

```
~ INPUT $DATA.JAN
```

Example:

```
&STUDY.DEF
```

See Appendix D: CfMC Conventions of the *Utilities* manual for a list of all of the file extensions that are generated by CfMC software.

Finally, if you put a minus sign before a file name, Mentor will not echo the lines of the file to the screen or list file as it reads them.

Example:

```
&-STUDY.DEF
```

The following terms are used in the definitions in this appendix:

Convention	Description
[col.wid]	A field in the data starting at the location *COL* and continuing for a given *WID* number of columns. [col1-col2] is acceptable
Datavar	Any simple variable that references data. It can be a label, QQ#, or direct data reference (varname[col.wid]).
Database	Any datavar or combination of datavars that use functions, joiners or mathematical operators.
Function	A keyword used to modify the meaning of a datavar or expression, usually to get special kinds of numbers.
IDvar	A user-defined variable that refers to the location of the case ID in the data.
Joiner	A keyword used to combine datavars.
Label	The name of a simple variable defined using PREPARE type or data variable syntax.
Loc	Column location in the data
Procedure	A set of commands that will all be executed on each case when the procedure is called. Commonly called a proc.
QQnum	A variable created in ~PREPARE when compiling a questionnaire, which refers to the question number of the variable.
Text variable	A user-defined pr program-generated variable that contains textual labeling.
Varname	Refers to the name given to any user-defined variable.

See the *Utilities* manual for a more complete glossary of CfMC terms.

~ADJUST (ADJ)

Copies an existing CfMC data file to a new data file (these files have an extension of TR). The new data file can have a different case length.

Syntax:

~ADJ outfile="comment", infile, filein

Options:

outfile

The name of the output data file.

="comment"

An optional one- to 21- character comment that displays whenever the file is opened.

infile

Name of the data file you are copying. Mentor assumes the file has a TR extension, you do not need to include it.

fileinfo

Optional information you can supply about the input or output file.

They are:

INPUT_TEXT_LOCATION=col.wid (INTEXTLOC)

Specifies the location of the input text area. If you do not specify an output text location, then the values for input are used.

NEW_LENGTH=# (NEWLEN)

Length of new file. # can be a number 1-9999, or a record/column reference (i.e., 3/80 for 240 columns).

NUMBER_OF_CASES=# (NUMC)

Makes an indexed directory of the case IDs for direct case access when you build a new file; useful if you will be making many modifications to the file. The default is that files are built without a directory, and are read sequentially to find cases.

This is usually used if you are doing a database application, or when the file will be used by Survent with Coding or Phone Says Data Record mode, which update existing cases.

The program makes enough directory entries for the "Number of cases * 3". This is because if you make modifications that make the case significantly bigger, a new case is built and a new directory entry is used. You will use approximately 42 bytes of space for each case you specify in the NUMC= command.

If NUMC is set to -1, then no directory will be made for the file, even if one was in the file being copied; otherwise, the directory is copied into the new file.

OUTPUT_TEXT_LOCATION=col.wid (OUTTEXLOC)

Specifies the location of new text area. You must also specify the INPUT_TEXT_LOCATION with this option.

If nothing is specified about the text area location then the case length is adjusted per the INPUT and OUTPUT file lengths, and the case is written to the OUTPUT file in the standard way. Otherwise, the System constant, TEXT_AREA_STATUS is called to check the status of the text area.

If TEXT_AREA_STATUS is greater than 3, an error is generated and the output file is not kept.

A case is made up of three parts: data before the text area; the text area itself; and data after the text area. Each of these areas is dealt with as a separate piece. The size of each piece (in and out) is examined on a piece-by-piece basis:

- If a piece is smaller, then the columns lost in that piece are checked to make sure they are blank.
- If the piece is the area before or after the text and the area is not blank, then a warning is generated indicating where the area is not blank.

- If the piece is the text area, and it is not blank, then an error is generated and the output file is not kept.

Next, each piece is moved to its new place. If the new piece is longer, then the expanded area is blanked. After the three pieces are moved, the back and forward text pointers need to be adjusted. If a forward pointer would be in a place where no text pointers may now be, an error is generated and the output file is not kept. This should only happen if the forward pointer was in an area that became smaller. Otherwise, the forward and back pointers are adjusted

for their new location and the data is written to the output file.

See *Mentor Volume I: Chapter 9"Case Reading Constants"* for a description of TEXT_AREA_STATUS.

Related Commands:

`~INPUT and ~OUTPUT NUMBER_OF_CASES=`

~CLEANER (CLN)

Use ~CLEANER to clean (modify) data. With ~CLEANER, you can access a data file on a case-by-case basis, or define and test procedures to check and/or clean data across all cases. ~CLEANER allows you to display data (in ASCII or other formats), or to look at and/or modify tables. See below for explanations on how to reference individual data to entire tables.

If you are unfamiliar with ~CLEANER, you can do simple cleaning with the CLEANIT utility. See the *Utilities* manual for a description of CLEANIT.

Most commands used individually in the ~CLEANER block can also be used in procedures defined in ~DEFINE PROCEDURE=, which you can also use to perform operations across all cases. See ~DEFINE PROCEDURE= for additional information.

Syntax:

`~CLN`

```
keyword
keyword
~next tilde block command
```

To save any modifications to the data, a data file which can be updated must be opened using the ~CLEANER FILE command, ~OUTPUT, or ~INPUT with the ALLOW_UPDATE option invoked. If ~OUTPUT is used, changes will only occur if you use the WRITE_CASE command. If there are multiple data files open, ~CLEANER uses the primary file by default.

The program prompts the user to confirm any changes made to the data unless ~SET PRODUCTION_MODE is invoked (see ~SET or ~CLEANER SET) or the data file is open with the ~CLEANER FILE command.

There are also commands that cannot be used directly in the ~CLEANER block, although you can call in procedures while in the ~CLEANER block. See ~DEFINE PROCEDURE= for more information.

You can specify more than one CLEANER command on a single line, separated by semi-colons (;). Any REDO command will re-execute the entire line.

Example:

```
CleaNer--> DA 5.2; DA 10.2; MA 10.2
CleaNer--> NEXT_REDO
```

The above example would execute the same set of commands on the next case.

Categories of ~Cleaner Commands

Here are the major categories of ~CLEANER commands:

DATA REFERENCES AND INTERACTIVE COMMANDS

You can display columns in ASCII or column binary format, change the data, or find cases with specific information in the ~CLEANER block. You can move from file to file and from case to case.

These commands are further broken down into those that will work only in ~CLEANER, and those that will also work in ~DEFINEd procedures.

These commands work in ~CLEANER**,** not in procedures:

DEFINE	HUNT
DROP	NEXT_REDO
FILE	REDO
FIND	RESTORE
FIND_FLAGGED	SET
FIND_FLAGGED_REDO	UPDATE
FIND_REDO	VIEW

These commands work in ~CLEANER or in ~DEFINEd procedures, but are most often used in the ~CLEANER block:

BACKUP	MODIFY_ASCII
DISPLAY_ASCII	MODIFY_BINARY
DISPLAY_BINARY	MODIFY_COLUMN
DISPLAY_COLUMN	MODIFY_TEXT
DISPLAY_TEXT	NEXT
ERASE_TEXT	SHOW

These commands work in ~CLEANER or in ~DEFINEd procedures, but are most often used in procedures:

ALTER
ASSIGN_DELETE_FLAG
ASSIGN_ERROR_FLAG

BLANK

CHECK
CHECK_COLUMNS
CHOOSE_FILE
CLEAN
CLEAR_ERROR_FLAG

DO_SET
DO_META
DO_TABLES
DUMP_VARIABLES

EDIT
END_WHEN
ENTER
ERROR
EXECUTE_ANY
EXECUTE_DATA
EXECUTE_EOF

FIXUP

GOTO

HALT

INTERVIEW
MAKE_DATA
MODIFY

NEW_CASE
NO_UPDATE
NULL

OK_COLUMNS

PAUSE
PRINT_LINES
PRINT_TO_DATA
PUT_ID

SAY
SKIP_TO
TERMINAL_PRINT
TERMINAL_SAY
TRANSFER

UNDELETE
UPSHIFT

WHEN TOP
WHEN BOTTOM
WRITE_CASE

YES_UPDATE

These commands create and alter tables. Tables are arrays of numbers generally defined by a row definition and a column definition. The complete array is the columns crossed with the rows. Each cell of the array contains a number or blank.

Tables can also be created by reading data and filling the cells with counts from the data in the ~EXECUTE block, or using ~DEFINE TABLE=. ~CLEANER table operations are usually used to modify tables made in the ~EXECUTE block by reading data; for instance, you may wish to add the numbers from two data-created tables.

These table-related commands are:

CREATE_TABLES	PRINT_TABLES
EXECUTE	SHOW_TABLES
FILL	STORE_TABLES
LOAD_TABLES	UNLOAD_TABLES
MODIFY	

!procname

Executes the named procedure on the current case. A typical application of this is to do a ~EXECUTE PROCEDURE= to generate a cleaning error listing, then call up a dirty case and modify it. Immediately after, you can test to see if that case is now clean by doing a !procname.

ALTER (ALT)

This modifies data using data variables (usually defined with ~DEFINE PREPARE=). The question is displayed as a Survent interviewing screen, with the current data displayed on the bottom line of the screen.

Syntax:

```
ALT label
```

The user can elect to change the data or not. Here are the possible responses on the Survent screen:

RES	Restores data to original.
BLK	Blanks the columns.
TERM	Terminates the process and goes to the end of the file.
code1,code2	Enter new codes.
+code1,code2	Add new codes to existing codes (CAT question).
-code1,code2	Remove some codes from existing codes (CAT question).

The program will display the type of question and its current value before the Survent screen is presented and after the modification has been made. ALTER shows you the data in the variable and then presents you with the Survent screen unconditionally.

Related Commands:

See EDIT and ENTER, and MODIFY_ASCII/MODIFY_BINARY for simple data modification without the controlling Survent screen.

ASSIGN_DELETE _FLAG (DELETE)

Flags a case for deletion when the input file is closed. The input file must be opened with the ~INPUT ALLOW_UPDATE option before cases can be flagged as deleted. Deleted cases can be recovered by using the USE_DELETED option with either the ~COPY or ~INPUT command.

Syntax:

```
DELETE
```

The case is still in memory after being flagged; DROP immediately drops the case from memory. This means you cannot use the RESTORE command to undo the last set of changes made to the case.

Related Commands:

UNDELETE and DROP

ASSIGN_ERROR_FLAG (ASSIGNF)

Turns on the error flag for the current case. Later, you can reference cases flagged with IF ERROR_FLAG in a procedure or cleaning run.

Syntax:

ASSIGNF

Related Commands:

CLEAR_ERROR_FLAG

BACKUP

Moves the user backwards the number of cases specified and displays the case identifier. You can then DISPLAY or MODIFY the data in that case. See Mentor, See *Mentor, Volume I, 2.5.1 QUICK REFERENCE: Cleaning Commands and Examples* for interactive cleaning examples.

Syntax:

BACKUP #

You must use ~SET SEQUENTIAL READ for BACKUP to work, you will get an error if it is not set.

Related Commands:

DISPLAY, MODIFY, NEXT

BLANK (B)

Blanks the data columns contained in a specified data variable. If the datavar contains punch categories, only valid categories are blanked, not punches outside the category description.

Syntax:

```
B datavar or data location
```

Examples:

```
B GENDER
B [1.10] [1.10,30,50] [2/20.5]
```

Related Commands:

For Survent TEX type questions, see ERASE_TEXT.

~CLN CHECK

CHECK (CHK)

Evaluates a datavar against its data columns, and prints an error message if the data does not correspond with the datavar definition. The user can define a message in quotes to print. Otherwise a default error message will print.

Syntax:

```
CHK SUMMARY_ONLY <mod>datavar "error message"
```

Options:

SUMMARY_ONLY

Optional; causes only the summary portion of an error message to be printed in the cleaning report, instead of a message for each data case that has a problem.

<mod>

is an optional modifier that can be:

- data *can* be blank.

+ any additional punches are OK (punch type variable)

***** data *must* be blank.

See ~DEFINE VARIABLE= for an explanation of the variable modifiers *D, *F, *L, *P, *S and *Z under the heading *Mod.*

datavar

A single location data variable (col.wid or col-col or label)

“error message”

Optional; if not specified, a system-generated error message is printed. This is intended to be a one-line informational tag; \N (new line) is ignored.

Examples:Where the variable GENDER is defined as [6^1/2]:

`CHK GENDER`Only punches 1 or 2 are acceptable.

`CHK -GENDER`Punch 1, 2, or blank are acceptable.

`CHK +GENDER`All punches are acceptable, but must have 1 or 2 also.

`CHK [5^1-12,B]` 1-12 can be blank.

`CHK [5*P=1^1//5/12]`

1-5 and 12 (Y) must all be single punched. *P=1 is the default, meaning only one response is allowed.

CHK [5*P^1//5/(-)12]

1-5 can be multi-punch (*P), 12 (Y) is exclusive (-).

CHK [5*P=3^1//5/10/(-)12]

1-5 and 10 can be multi-punch (up to a maximum of 3 responses, *P=3), 12 (Y) is exclusive (-). Exclusive means if this response is present, then no other responses are allowed. *P= can be any number 1-255.

`CHK [5$]` Must be ASCII (not blank).

`CHK [5.15*P=3$]` Must be ASCII and this is the minimum number of columns that must be non-blank. *P= can be any number 1-127.

`CHK [5]` Must be numeric.

CHK [5.3#1//55/RF]

Must be a number 1-55 or the literal RF.

CHK [15.2,...,19*ZF#1//17/(-)99]

There can be one of 01-17 (leading zeros, *Z) in any of the locations (no duplicates, *F), 99 is exclusive. No leading or embedded blanks allowed. Replace *F with *L to allow duplicate responses.

```
CHK [10,...,30^1//5]
```

Check each location for a single punch 1-5 (e.g., rating scales).

CHK [5$] ‘’column five is either blank or not ASCII”

Column five must be ASCII (not blank), and use the error message in quotes rather than a system generated error message.

```
CHK SUMMARY_ONLY [5$]
```

Column five must be ASCII (not blank), but print only a summary of the errors rather than an error message for each case that has an error.

A variable from Survent will contain information concerning multi-punch and exclusive codes.

If you have a print file open with the meta command >PRINT_FILE, then messages generated by the CHECK statement will go to the open print file, including the error summary.

CHECK_COLUMNS (CHKCOLS)

In a procedure, means that columns not touched by these commands:

ALTER

CHECK

CLEAN

EDIT

ENTER

FIXUP

OK_COLUMNS

will be checked to verify they are blank. If they are not, Mentor gives an error message. This is appropriate when you expect anything that has not specifically been checked to be blank. Use this command only once in a cleaning run, at the end of your cleaning commands.

Syntax:

```
CHKCOLS
```

CHOOSE_FILE (CHOOSE)

Determines which data file you are directing commands toward. This allows you to access multiple data files in a procedure. To use CHOOSE_FILE, open the first data file with ~INPUT NUMBER_INPUT_BUFFERS=# option set. Then open the second (and subsequent) files with the ~INPUT NEW_BUFFER option.

Syntax:

```
CHOOSE "name"
```

Put an exclamation point before the file name to designate that this will be the primary file from this point forward.

Example: CHOOSE "!Tranfile"

This opens the file TRANFILE and executes all future commands in the procedure on that file until you issue another CHOOSE_FILE command.

Related Commands:

See ~EXECUTE PROCEDURE= name MULTI for other examples.

CLEAN (CLN)

Blanks the data location when the data does not correspond with the definition of the datavar. When an error is encountered, a program-generated message or optional user-defined error message is displayed.

Syntax:

```
CLN SUMMARY_ONLY (mod)datavar "error message"
```

Options:

```
SUMMARY_ONLY
```

Optional; causes only the summary portion of an error message to be printed in the cleaning report, instead of a message for each data case that has a problem.

<mod>

Optional modifier that can be:

-	data *can* be blank.
+	anything additional is OK (CAT type variable)
*	data *must* be blank.

datavar

is a single location data variable (col.wid or col-col or label)

"error message"

Optional; if not specified, a system-generated error message is printed. This is intended to be a one line informational tag; \N (new line) is ignored.

Example:

```
CLN [6.2#1//20] "The boat question does not have an
answer 1-20"
```

Related Commands:

CHECK and FIXUP

See Also: CHECK for several data location examples.

CLEAR_ERROR_FLAG (CLRF)

Clears error flags on the current case.

Syntax:

```
CLRF
```

Related Commands:

ASSIGN_ERROR_FLAG

COPY

Copies data from one location to another. COPY does not translate or modify data. Put the receiving location first. The receiving location must be the same column width of the sending location, or longer. The receiving location's data is replaced with the data of the sending location.

Syntax:

```
COPY to_datavar = from_datavar
```

Example:

```
COPY [5 - 7] = [10 - 12]
```

Copies data in columns 10 - 12 to columns 5 - 7.

Example:

```
COPY [50.2,52.2,54.2]=[22.2,20.2,18.2]
```

Copies 22.2 to 50.2, 20.2 to 52.2, and 18.2 to 54.2

This command writes over existing data.

CREATE_TABLES (CREATE)

Creates a table. CREATE_TABLES creates tables from scratch. You can create tables from a data file using ~EXECUTE TABLE= or STORE_TABLE. You can

combine cells from tables made in ~EXECUTE with those made with CREATE_TABLES.

Examples:

1) Create table with given dimensions and values:

```
CREATE t1(=2, =3) =0
```

t1	is the table name
2	Is the number of columns
3	is the number of rows
0	is the number to fill table cells with

The numbers (2,3,0) can also be referenced variables in the data.

The parentheses ( ), equal signs =, and comma are required.

2) Create a table as part of an expression:

```
CREATE  t2 = t1 + t3
```

t2 is the created table's name which has the dimensions of t1, and its cells will have the sum of the values of t1 + t3.

3) Copy a table and its elements

CREATE t5 != t2

!= causes the table elements (STUB, BANNER, etc.) to be part of the new table.

Related Commands:

See MODIFY <table> for more information on expression syntax.

DEFINE (DEF)

Allows you to define variables and expressions, or procedures from the ~CLN block. You can use this command to define a variable when you are displaying or modifying the same columns during interactive cleaning. Once you have defined a

variable or a procedure, locate the first case to satisfy your definition with either the FIND or HUNT command, or !procname to execute the procedure on the current case. See ~DEFINE VARIABLE= and PROCEDURE= for information on the rules for these definitions.

Your definition cannot include any ~DEFINE AXIS=$[] cross-case functions.

Syntax:

```
DEF name[data definition] or
name:expression or
PROCEDURE={ name:commands}
```

Example:

```
DEF PROCEDURE={REPAIR:
          DISPLAY_ASCII 40.2
          MODIFY_ASCII 40.2
          DISPLAY_ASCII 40.2
           }
```

Typing !REPAIR will execute this procedure on the current case: display the contents of the columns 40 and 41; display the columns in modify mode; then confirm the change by displaying them again. Thereafter, just typing FIND will re-execute the procedure on the next case. In this example if you do not modify the data, pressing **Enter** at the modify prompt will execute the procedure on the next case.

You can define and immediately execute a procedure by defining it with !procname: instead of DEF procname:. As soon as you press Enter after the closing brace (}), the procedure will execute beginning at the top of the data file (like HUNT).

Continue a variable or expression definition to subsequent lines by using an ampersand (&) at the end of the line to be continued.

Example:

DEF var1:[6^2] OR [34#20-30] AND & [5^1-5]

DISPLAY_ASCII (DA)

Displays the contents of a data location or variable in ASCII format. A template appears above the first 80 columns of data being displayed.

Syntax:

```
DA column.width
DA column-column
DA datavar
DA *
```

* displays the entire case in ASCII mode. ASCII is the default mode for DISPLAYing, so you can use D instead of DA.

Example:

```
DA 1.20

Display 1.20
0         1         2
12345678901234567890
--------------------
0053 SMITH       *3A
```

Related Commands:

DISPLAY_BINARY, DISPLAY_COLUMN, DISPLAY_TEXT, MODIFY_ASCII, MODIFY_BINARY, MODIFY_COLUMN, MODIFY_TEXT, and SHOW *.

DISPLAY_BINARY (DB)

Displays the contents of a data location or variable in binary format. All the punches in each column would be displayed. Possible punches are 1 to 9, 0, X, and Y.

Syntax:

```
DB column.width
DB column-column
DB datavar
```

Example:

DB QN8 (where QN8 is a variable in columns 40 and 41)

ColumnASCII Binary

1/40 S 2,10(‘0’)

1/41 3 3

If the variable is a converted variable from compiled Survent specifications then the display will include information from the Survent question structure. After the case ID and case number Mentor will print the following information from the Survent question structure: varname=question label (total responses:response code=punch code=response text)

Example:

```
ID:0016, #16:qn8 =qn8(18:02=2=Dependable/continuous
service,
10=10=Variety of entertainment,15=15=Other)
```

ColumnASCII Binary

1/40 S 2,10(‘0’)

1/41 3 3

Related Commands:

DISPLAY_ASCII, DISPLAY_COLUMN, DISPLAY_TEXT, MODIFY_ASCII, MODIFY_BINARY, MODIFY_COLUMN, MODIFY_TEXT, and SHOW *B

DISPLAY_COLUMN (DC)

Displays the contents of a data location or variable in column format.

Syntax:

```
DC column.width
DC column-column
DC datavar
```

Like DISPLAY_BINARY, DISPLAY_COLUMN displays binary punches, but as numbered codes relative to the first punch of the first column specified. There are 12 possible punches per column, so the five punch of the second column would display as a 17 code in DISPLAY_COLUMN mode.

Example:

```
DC QN8(where QN8 is a variable in columns 40 and 41)

ID: 0016, #16 1/40.2: 2,10,15
```

The numbered codes 2,10,15 refer to the punch positions relative to the first punch in column 40. In this example there are punches present in the second, tenth, and fifteenth punch positions (a two and a zero punch in column 40, and a three punch in column 41). Also see the examples under DISPLAY_BINARY.

Related Commands:

DISPLAY_ASCII, DISPLAY_BINARY, DISPLAY_TEXT, MODIFY_ASCII, MODIFY_BINARY, MODIFY_COLUMN, and MODIFY_TEXT

DISPLAY_TEXT (DT)

Displays the contents of a text type variable (usually collected with a Survent TEXT type question).

Syntax:

```
DT text pointer column
DT text variable
DT [loc$T]
DT *
```

The contents of a text type variable are stored in the specified text location in the file; the text pointer or variable points to the data area where the text is stored. These variables are used to save space when a case has a lot of textual responses,

and are usually used only in questionnaires generated by ~PREPARE and executed in Survent to collect interview data. Use DT * to display the entire text area.

Example:

```
DT DEMO2

ID: 0501, #1:
TEX: 4/71 <-> 5/2=Blue Cross/Blue Shield
```

4/71 is the location of the text pointer for DEMO2 and 5/2 is the start of the actual text.

Related Commands:

DISPLAY_ASCII, DISPLAY_BINARY, DISPLAY_COLUMN, MODIFY_ASCII, MODIFY_BINARY, MODIFY_COLUMN, and MODIFY_TEXT and system constant TEXT_AREA_STATUS.

DO_META (META)

Specifies a meta command, usually in a procedure. The meta command must be specified inside quotes and only one discontinued line is allowed. Leading or trailing blanks inside the quoted string are ignored. You do not need to supply the meta (>) symbol, the program will append it to the beginning of the quoted string.

Example:

```
~DEFINE PROCEDURE={proc1:
META "USE_DB SAMPL"
procedure commands and/or conditional expressions
META "CLOSE_DB"
}
```

When this procedure is executed, the db file SAMPL will open with the meta command >USE_DB for use during this procedure only. The next meta command >CLOSE_DB will close the file at the end of the procedure.

As with other procedure commands, you can execute a DO_META conditionally with an IF/ENDIF statement.

DO_SET (SET)

Specifies a ~SET command usually in a procedure. The command must be specified inside quotes and only one discontinued line is allowed. Leading or trailing blanks inside the quoted string are ignored.

Example:

```
~DEFINE
PROCEDURE={proc1:
SET "ERROR_REVIEW"
procedure commands and/or conditional expressions
}
```

When this procedure is executed, ~SET ERROR_REVIEW mode will be turned on.

You can execute a DO_SET conditionally with an IF/ENDIF statement.

DO_TABLES (TAB)

In conjunction with ~EXECUTE READ_PROCEDURE=, this allows you to make data modifications for the printed tables without changing the input or output data file. Specify DO_TABLES whenever you want the tables to be accumulated (usually at the end of the procedure).

Example:

```
~DEFINE PROCEDURE=Change:
IF [5^1]
MODIFY [1] = 8
ENDIF
DO_TABLES
}
```

```
...

~EXECUTE
READ_PROCEDURE=Change
COLUMN=TOTAL
ROW=[1^1//8]
TABLE=*
~END
```

DOWNSHIFT (DOWN)

Changes all alpha characters in the specified position to lower case.

Syntax:

```
DOWN datavar
```

Example:

```
DOWN gender
DOWN [1.80$]
```

Related Commands:

~CLN UPSHIFT, ~SET CASE_SENSITIVE

See Also:*Mentor Volume I*, Chapter 3: "Changing Case."

DROP

Flags a case for deletion when the input file is closed and drops the case from memory immediately. The input file must be opened with the ~INPUT ALLOW_UPDATE option before cases can be dropped from that file, or if you are writing an output file (using ~OUTPUT, etc.) the case will be dropped there.

Remove the delete case flag with the UNDELETE command.

Syntax: DROP

Cases that are flagged as deleted using the DROP or ASSIGN_DELETE_FLAG commands are not removed from the original data file and can be accessed by opening the data file with the ~INPUT USE_DELETED option. Cases flagged as deleted will not be written to subsequent copies of the original data file.

Related Commands:
~INPUT BACKUP, ~OUTPUT, and ~COPY -KEEP_DELETE

DUMP_VARIABLES (DUMP)

Writes (to the list file or print file) the data associated with each specified variable and the command necessary to remake the data. This is used so that a new data file can be created in case of conversion to other Mentor versions or operating systems, or resetting the data and variables to make room for new variables without overwriting any data.

A dump of ten variables per case on one hundred cases would produce a list file of 1000 lines, each line representing the data for each variable for each case, one at a time.

Syntax:

```
DUMP datavar
```

DUMP_VARIABLES will write out the following variables:

CAT and FLD questions `MAKE_DATA +datavar(response code(s))`
VAR and TEXT questions `MODIFY datavar = "response"`
NUM questions `MODIFY datavar = ###`

EDIT

Shows the current data in the variable and presents the Survent screen for modification *only* if the current data does not match the allowable responses of the defined variable. See ALTER for more information on responses at the Survent screen.

Syntax:

```
EDIT <mod>label
```

Options:

<mod> is an optional modifier that can be:

- \- data *can* be blank.
- * data *must* be blank.
- \+ anything additional is OK (CAT type variable)

If you use EDIT in a procedure with ~SET ERROR_REVIEW, EDIT prints an error message to the print file for every case that does not match the variable definition.

If you execute a procedure with either ~CLEANER FIND or HUNT, the Survent screen will *not* be presented. FIND and HUNT will just list all of the errors for that case. Use `!<procedure name>` to fix the errors with the Survent screens.

For simple data modification without the controlling screen interaction, use ~CLEANER MODIFY_ASCII/MODIFY_BINARY, etc.

Related Commands:

ALTER, ENTER

END_WHEN

End statement for the WHEN TOP/WHEN BOTTOM printing block. See example under WHEN TOP/ WHEN BOTTOM. (ENDWHEN)

ENTER

Allows you to add data to a case using the Survent data screen, and according to the variable you have defined. The screen is presented unconditionally. For simple data modification without the controlling screen interaction, see ~CLEANER MODIFY_ASCII/MODIFY_BINARY, etc.

Syntax:

```
ENTER label
```

See ALTER for more complete information on responses to the Survent data screen.

ENTER does not display the case ID, so you may want to include the command SAY CASE_ID when you use ENTER in a procedure.

Related Commands:

ALTER, EDIT

ERASE_TEXT (ET)

Erases the text pointer and releases the text area specified for the variable from text type data. This type of data is usually from TEX type questions collected with a Survent interview, but it can be created with the ~CLEANER TRANSFER command (e.g., TRANSFER [loc$T] = “string”). New or modified text data can use the text space released with this command.

Syntax:

ET loc.len or datavar

Example:

```
ET 10
```

This will erase the text pointer from column ten and the corresponding text in the text area.

Related Commands:

BLANK

ERROR (ERR)

If executed, ERROR prints a message or the contents of variables (or both), turns on the system constant ERROR_FLAG, and adds to the counts of the error summary. This is a good command for noting complicated conditions in data checking procedures. This command also prints the case ID.

Syntax:

```
ERR SUMMARY_ONLY datavar "error message"
```

Options:

SUMMARY_ONLY

Optional; causes only the summary portion of an error message to be printed in the cleaning report, instead of a message for each data case that has a problem.

datavar

is a single location data variable (col.wid or col-col or label)

"error message"

Optional; if not specified, Mentor generates a generic message. You custom error messages can be as long as you want and can include quoted strings, numbers, and/or variables. Use an ampersand (&) to extend to a new line. If you have a print file open with the meta command >PRINT_FILE then the message defined on an ERROR statement will go to the open print file. Error messages can appear both before or after the datavar.

Example:

```
IF [5^1] AND [5^2]
```

```
ERR "Had both sexes marked" [5$P] "And should not"
ENDIF
```

The syntax for the printing is the same as the SAY command. Data variables print in the syntax specified by their type:

Punch variables:

Syntax: [loc.len$P]

Prints punches from the columns

Example:

```
123/45/B/7
```

String or text variables:

Syntax: [loc.len$] or [loc.len$t]

Prints text

Example:

```
This is the string
```

Numeric variables:

Syntax: [col.wid]

Prints a number

Example

```
76
```

Categorical variables:

Syntax: varname (!CAT or !FLD question collected in Survent), or

[loc.len^1/2] or [loc.len#1/2] or

[loc.len^"text":1/"text":2] or [loc.len#"text":1/"text":2]

Prints text of the category when you use a varname, or the category only“text” when you supply your own text.

Examples:

variable qn2A - `qn2A[7^]="4" Good`

[7^4] - `qn2A[7^]="4"`

[7^”Are happy”:4/”not happy”:3] `qn2A[7^]= "4"Are happy`

EXECUTE (EXC)

Processes all commands on the line as if you were in the ~EXECUTE block. Usually this is used when doing table creation or modification and you want to set the column, row, or other table elements to be stored with a table.

Syntax:

EXC "keyword keyword keyword"

Example:

```
EXC "COL=AGE  EDIT=EDIT2"
```

This sets the current column to AGE and current edit to EDIT2. These would be stored with any table created with MODIFY A != B syntax, and would affect the printing of tables with PRINT_TABLE. The quotes (") are required.

EXECUTE_ANY (EXCANY)

Executes subsequent ~CLEANER commands on all cases (from current case to last case) and once again after the last case has been read. This is usually used in conjunction with memory (data accumulation across cases; see ~INPUT TOTAL_LENGTH= and ~OUTPUT). EXECUTE_ANY is overridden by the next EXECUTE_DATA or EXECUTE_EOF command.

Syntax: EXANY

Related Commands:
EXECUTE_DATA, EXECUTE_EOF

EXECUTE_DATA (EXDATA)

Executes subsequent ~CLEANER commands on all cases read (from current to last case). This is the default. EXECUTE_DATA is overridden by the next EXECUTE_ANY or EXECUTE_EOF command.

Syntax: EXDATA

Related Commands:
EXECUTE_ANY, EXECUTE_EOF

EXECUTE_EOF (EXEOF)

Executes subsequent ~CLEANER commands after the last case has been read. Usually used in conjunction with memory (data accumulation across cases). EXECUTE_EOF is overridden by the next EXECUTE_ANY or EXECUTE_DATA command.

Syntax: EXEOF

Related Commands:
EXECUTE_ANY, EXECUTE_DATA

FILE

Opens a data file, closing any open ~INPUT or ~OUTPUT files, and allows you to update the file automatically. ~INPUT ALLOW_UPDATE and ~SET PRODUCTION_MODE are automatically set when you open a data file with the FILE command.

Syntax: FILE datafile, options

Options:

Any ~INPUT options *except:* FILES=; JOIN=.

Example:

```
FILE data1, SELECT=[5^1]
TRANSFER [1.5$] = "00001"
```

As the data file is read, Mentor prints a message to the screen if the file contains deleted cases. ASCII and binary files can be displayed in ~CLEANER, but must be opened with ~INPUT, not the FILE command.

Related Commands:

~INPUT ALLOW_UPDATE, ~SET PRODUCTION_MODE

FILL

Causes a one-item axis to affect all the columns or rows of the corresponding receiving axis. Used in table modification when the receiving matrix is not the same dimensions as the sending matrix. If a sending axis is evenly divisible into a receiving axis, FILL will operate on the corresponding parts of the axis.

Syntax:

```
FILL <table or region> operator <table or region>
```

Options:

<table or region>

Table name or a region of a table.

operator

One of the allowed MODIFY <table> operators.

Example:

```
CREATE_TABLES T001(=4,=8) = 4
CREATE_TABLES T002(=2,=4) = 3
```

```
FILL T001 += T002
```

This would add three to each of the cells of table T001, since there are evenly divisible column and row axes.

```
FILL T001 -= T002(1 by ALL)
```

This would subtract column one of table T002 from all of the columns of table T001.

Related Commands:

MODIFY <table>

FIND (F)

Used with an expression, FIND searches for data that satisfies the expression, starting with the *next* available data case from the current position in the file. To continue the search, you can use FIND with no option; the program defaults to the last expression specified. See ~DEFINE VARIABLE= for information on defining variables and expressions.

FIND prints dots for every 10 cases read. If the expression is not found before the end of file, it rewinds to the beginning of the file. FIND or REDO will continued the search.

Syntax:

FIND expression

Example:

```
FIND [5^1]
```

This will find the next case with a ‘1’ punch in column five.

Used with a procedure, FIND executes the procedure on each case beginning with the next

available case. It finds the first case with an error for the procedure, and prints all the errors on that case, (not just the first one found). In this way, you can use the FIND command to find all of the errors, do your cleaning, and re-issue the FIND command to make sure no more errors are present. The FIND command will then position you on the next case that needs cleaning.

Syntax:

FIND procedure

Example:

```
DEFINE PROCEDURE={CKGENDER:
CHECK -GENDER "Punches 1 or 2 or blank are OK"}
FIND CKGENDER
```

This defines a procedure to check question GENDER for acceptable responses (minus (-) means can be blank) , and then evaluates the entire data file using that procedure. After you have fixed the current case, you can execute the procedure on it again by entering !CKGENDER.

Example:

```
DEFINE GENDER[5^1//3]
FIND [GENDER^1]
```

This will find the next case with a one punch in column 5. Enter FIND by itself to re-execute the last FIND variable definition given, in this case [GENDER^1], to find the next case that satisfies your variable definition.

Example:

```
DEFINE
PROCEDURE={GENDER:
CHECK [5^1//3/B] "Should be punches 1, 2, 3, or blank"
  }
FIND GENDER
```

This defines a procedure to check column five for valid punches including blank (indicated by B in the last category). FIND

GENDER evaluates the entire data file against the statements in this procedure, listing the cases with errors.

`!GENDER` executes the procedure on the current case.

Use `HUNT <expression>` or `HUNT <procname>` the first time it is executed, since HUNT begins at the top of the data file. Thereafter FIND will locate the next case that satisfies the expression or procedure. Remember once you have stated the name of the expression or procedure, you can give the FIND command by itself to find the next case.

Related Commands:

FIND_REDO, HUNT

FIND_FLAGGED (FF)

Finds the next case with an error flag set. The error flag is set by any CHECK, CLEAN, EDIT, or ERROR statement that is true for that case.

Syntax:

FF

Related Commands:

FIND_FLAGGED_REDO, FIND_REDO, HUNT

FIND_FLAGGED _REDO (FFR)

The error flag is set by any CHECK, CLEAN, EDIT, or ERROR statement that is true for that case. Finds the next case with the error flag set, and re-executes the last command on that case.

Syntax:

FFR

Related Commands:

FIND_FLAGGED, FIND_REDO, HUNT

FIND_REDO (FR)

Finds the next case with the last FIND condition set, and then executes the previous command.

Syntax:

```
FR
```

R elated commands:

FIND_FLAGGED, FIND_FLAGGED_REDO, HUNT

FIXUP (FIX)

Blanks the data location when the data does not correspond with the definition of the datavar. Since FIXUP erases existing data, use it with an ~OUTPUT file (rather than ~INPUT ALLOW_UPDATE). FIXUP does not generate messages when it encounters errors.

Syntax:

```
FIX (mod)datavar
```

Options:<mod>

Optional modifier that can be:

-	data *can* be blank.
+	anything additional is OK (CAT type variable)
*	data *must* be blank.

Related Commands:

CHECK, CLEAN

GOTO

GOTO is used to move forward to another place in a procedure. It goes to a label that can be put on any command in the procedure.

Syntax:

```
GOTO label
```

Example:

```
IF GENDER(M)
GOTO FTM
ENDIF
...
FTM: PRINT_LINES "I am male"
...
```

You can have multiple conditions and markers to go to on one GOTO statement. In these cases, the conditions are evaluated, and the first TRUE condition sends the program to the corresponding label. If no category in the expression is true, or if the true category has no corresponding marker, the program will continue to the next line of the procedure.

Syntax:

```
GOTO (label1 label2 ... labelN) expression
```

Example:

```
GOTO (men women) GENDER
ERROR "not a man or a woman"
GOTO FINISH
men:    SAY CASE_ID "is a man"
GOTO FINISH
women:  SAY CASE_ID "is a woman"
finish: SAY "that's all"
```

An asterisk (*) in the GOTO signifies that if the corresponding condition is true, the program will just continue processing. If there are no matches, an alternate label can be used, which will be the label that the program goes to:

Example:

```
GOTO (label1 * ... labeln ; alt_label) expression
```

Related Commands:

```
~GO_TO
```

HALT

Will stop execution and quit the program. A datavar and/or string is required after the HALT command and will be the last item displayed before the program quits. See SAY for print syntax.

Syntax:

```
HALT datavar "string"
```

Example:

```
IF GENDER(1) AND GENDER(2)
HALT "Both Genders!" GENDER
ENDIF
```

Related Commands:

```
TERMINATE
```

HUNT (H)

Starts at the beginning of the data file, looks for the first case that satisfies the expression specified and stops on that case. HUNT prints dots for every 10 cases read. If the expression is not found before the end of file, it goes back to the beginning of the file. Use HUNT, FIND, or REDO to continue the search. HUNT always starts at the beginning of the data file, use FIND if you want to start from your current position in the file.

Once you have used an expression or procedure name with HUNT (or FIND), you can specify HUNT (or FIND) by itself to search for the same item again.

Syntax:

```
HUNT expression
```

Used with a procedure, HUNT executes the procedure on each case beginning with the first case in the data file. It finds the first case with an error for the procedure, and then prints all the errors on that case, (not just the first one found). In this way, you can use the HUNT command to find errors, do your cleaning, and re-issue the HUNT command to make sure no more errors are present. The HUNT command then will position you on the next case that needs cleaning.

Syntax:

```
HUNT procedure
```

See FIND for examples of defining procedures and then executing them with FIND or HUNT.

Related Commands:

```
FIND
```

INTERVIEW (INT)

Conducts a Survent interview and writes the data to the current open case. To append an empty case to the end of the data file and write the data to the new case, first use the NEW_CASE command. You must have a questionnaire file (QFF) created by ~PREPARE commands open (see ~QFF_FILE) and a data file open with ~INPUT ALLOW_UPDATE or ALLOW_NEW or ~CLN FILE.

Syntax:

```
INT
```

Example:

```
~QFF_FILE study1
```

```
~INPUT mydata,ALLOW_NEW
~CLN
NEW_CASE
INT
```

Related Commands:

```
~INPUT, NEW_CASE, ~PREPARE, ~QFF_FILE
```

LOAD_TABLES (LOAD)

Loads a table from a DB file into memory. You can then perform operations on the table, print the table, etc. (Tables are automatically loaded when referenced in the commands CREATE_TABLES, MODIFY, FILL, or TRANSFER.)

Syntax:

```
LOAD <tablename>
```

Related Commands:

```
UNLOAD_TABLES
```

MAKE_DATA (MKDATA)

Adds or removes binary punches to a case using a predefined category datavar or a datavar punch definition. Without a modifier, MAKEDATA blanks the receiving location before generating any data.

Syntax:

```
MAKEDATA <mod>datavar(response codes)
or
MAKEDATA <mod>[col.wid^punch1,punch2,punchn]
```

Options:

mod can be + or -

\+ adds the punches

- removes the punches

Example:

```
MAKEDATA +[5^1,5]
```

Adds punches 1 and 5 to column 5.

Example:

```
MAKEDATA [5^1]
```

This statement would blank punches 2-Y in column five before generating a one punch.

Related Commands:

```
MODIFY_BINARY
```

~CLN MODIFY

MODIFY <datavar> (M)

Modifies the receiving datavar depending on the value of an expression. You can do almost any kind of data transformation with this command. MODIFY must be used with caution, because it will execute all kinds of data transfers without warnings.

Use COPY to do direct copies of data from one place to another. Use TRANSFER to transfer data of the same type from one variable to another. Use PRINT_DATA to get special formats into the data, such as zero-fill, decimals, response codes, and subscripts.

See ~CLEANER MODIFY <table> to do table operations.

Syntax:

```
M datavar  operator  expression
```

Options:

`datavar`
The datavar can be a single- or multi-location data variable. If it is multi-location, the expression must return the same number of categories as locations in the datavar. If the datavar is too small to hold the result, a warning is displayed and the data is truncated, or '****' is put in the data for numeric overflows.

Operators

= Replaces data (blanks receiving categories first)

+= Adds data (does not blank receiving categories first)

-= Removes data

For punch variables, you can also use:

<= Puts punch in datavar if categories on both sides are true (AND condition)

|= Puts punch in datavar if one or the other category is true, but not both (XOR condition); otherwise blanks.

@= Puts punch in receiving datavar if sending datavar category is NOT true (NOT condition); otherwise blanks.

Expressions can be any valid expression, but should match by data type unless you are changing strings to punches, text variables to string variables, etc.

Remember that MODIFY is limited in its error checking; it allows most data transformations.

Here are the different data types and examples of MODIFY uses:

STRING TYPE VARIABLES

VAR [loc$] and TEXT [loc$T] type variables are modified by putting the new data in quotes. The number of characters to be put into the data must be less than or equal to the number of columns referenced by the variable to be modified. Brackets are optional for specifying the columns you wish to change, String references do not require a dollar sign ($) if it clear you are using a string.

Syntax:

```
M stringvar = "data"
```

Example:

```
M 40.10 = "Hello"
```

Puts 'Hello ' in locations 40-49.

NUMERIC VARIABLES

Numeric variables (col.wid) can be changed by entering a number or arithmetic expression after the equal sign. Brackets are optional for specifying the columns you wish to change, but brackets are required for 'from' locations.

Syntax:

```
M numvar = <arithmetic expression>
```

Example:

```
M 30.5 = 12
```

Puts the number 12 into the location 30.5

The number returned will always be right-justified in the field and preceded by blanks. To put decimals in numeric fields, use PRINT_TO_DATA. Zero-fill the field by using the variable modifier “*Z” on the receiving field.

Example:

```
M 10.5*Z = [20.2] * [32.2]
```

You can also use +=, -=, /=, and *= to add, subtract, divide, or multiply a location by another location.

Example:

```
M 2/17.2 += [2/3.2]
```

You can use %= to get the percentage of one number (or cells of a table) to another. Arithmetic expressions can include numeric functions, numeric data variables, and the operators +, -, /, *, and %. See *~DEFINE VARIABLE=, Mathematical Joiners And Operators* for more information.

Example:

```
M 2/10.5 = SUM([10.3,20,...,70]) / 5
```

This example takes the sum of the columns 10-12, 20-22, 30-32, etc. through 70-72, divides it by five, and puts the result in columns 2/10, 11, 12, 13, and 14.

Example:

```
M 2/10.5 *DZ = SUM([10.3,20,...,70]) / 5
```

Puts a two-decimal number into the data.

CATEGORY TYPE VARIABLES

With categorical data, including CAT and FLD questions from ~PREPARE specifications, if the category on the right is true, it affects the corresponding category on the left.

Example:

```
M [10^1/2/3/4] = [11^4/3/2/1]
```

If the two punch exists in column 11, it will put a 3 punch in column 10, since they are both the third category of the statement.

You can use the operators += and -= to add or subtract codes without affecting other codes in the column set. = by itself will overwrite existing data with the new data.

A system variable useful when modifying categories is CATEGORIES( ). CATEGORIES( ) allows you to reference a specific set of categories out of a list.

Syntax:

```
M catvar (+/-)= CATEGORIES(1,2,...,n)
```

Options:1,2,...,n
represents the relative position of the category in the list of the catvar.

Example:

```
M GENDER[2/23^1/3] = CATEGORIES(2)
```

This example changes the response to the question GENDER to the second category (the 3 code). Since Survent FLD type questions and ASCII category variables cannot share data space, you can only use = with them, not += or -=. Only one category is allowed on the right side of the equation. The field is blanked and then the new data is entered.

To convert a category variable to 01 coding, first write it out in blank/one format and at the same time define the receiving location as a [!numeric] variable to zero-fill the blank columns. This statement spreads a multiple punched single column into a range of columns.

Example:

```
M [!21,...,26] = [5^1//6]
```

This spreads the multiple-punched, single-column five into the range of columns 21 through 26. Any category present on the right will turn on a one punch (the

default) in the corresponding column on the left. Assuming the second and fourth categories are present in column five, a one punch in columns 21 and 24 would be recorded. The other columns will be recorded as zero.

You can not use `M [21,...,26^1] = [!5^1//6]`; the receiving data locations will still be blank/one.

You can use the exclamation on either side to zero-fill blanks in the receiving location(s).

Example:

```
M [31,...,36] = [!21,...,26]
```

To spread a category variable so that each response get its own column, use following type of statement. This example is the same as above, the second and fourth categories are in column five, but this time, column 21 will have a response of 2 and column 22 will have a response of 4.

Example:

```
M [21,...,26#1//6] = [5^1//6]
```

Here is the data after you modify it:

```
DISPLAY_ASCII 21.5

DISPLAY 21.5
2
123456
------
24
```

You can also use MAKE_DATA to perform simple category adding or removing operations. See *~DEFINE VARIABLE=, Category Types* for more information on category definitions.

PUNCH STRING TYPE VARIABLES

Punch string variables ([loc$P]) can be used to add or subtract punches from a column set.

Syntax: M punchvar operator "first_col_punch(es)\next_col punches\etc."

Options:

`punchvar`
Multi-location data variable, must have the same number of categories as punches you wish to add to the location.

`operator`
See operators listed on the first page of MODIFY.

`first_col_punch(es)`
Punch(es) can be any of the possible column binary punches (1234567890XY) or B for blank. By default a location by itself implies it holds a numeric value, so if you wish to move punches or text you must include $ or $P.

Examples:

`M[sex$P]="2"` Moves a 2 into a variable.

`M[10.2$P]+="1\23Y"` Adds 1 punch to column 10,

and 2,3, and Y to column 11.

`M[10.2$P]-="2"` Removes a 2 punch.

MODIFY <table>

This command is used to add tables together, combine tables, or any other operation using columns and rows of tables.

Syntax:

```
M <table region> operator <table region or expression>
```

Options:

```
<table region>
```

Tablename (column_list BY row_list) or

varname[$R <=tab> column_list BY row_list]

The complete table region specifications and examples are listed below:

Syntax:

```
varname[$R <=tab> column_list BY row_list]
```

Options:=tab

=tabname	Tabname is the name of table to use.
=?	Get column and row size from right side of the equation.
absent or =*	The table is in memory ready to print.

column_list

Defines which columns will comprise the region. Separated from row_list with BY. A list of column_list and row_list options follows.

row_list

Defines the rows that will comprise the regions. Separated from row_list with BY. A list of column_list and row_list options follows.

Row and Column list options

#	that column or row
ALL	all columns or rows
T	the Total column or row
NA	the No Answer column or row
LAST	the last column or row

Row and Column list options

LAST - #	can be followed with a dash and then a number to say the row number above LAST
D	where D is a single number
A TO B	where this represents a range for A through B
A,B,...,C	where this is a patterned ellipsis

You can join items of any type to form compound lists.

If you use the [$R=?] form, you cannot specify anything else. For all other forms, if nothing else is specified, then ALL BY ALL is assumed.

Examples:

```
T001[$R= ?] = T002
```
Makes table same as table T002

```
T002[$R= T001]
```
Same as above

```
T003[$R= *]
```
Uses the last table in memory

```
T004[$R]
```
Defines a region for later use

```
t001(1 BY 3)
```
Creates table t001 with 1 column, 3 rows

```
[$R ALL BY 1 2 5]
```
Region of all columns and rows 1,2,5

```
[$R 1 TO 4 5,7,...,13 BY ALL]
```
Columns 1-4,5,7,9,11,13 by all rows

```
[$R 1 LAST BY LAST - 2 ]
```
Column 1 and last by second row from last

operators

+, -, *, /, and special operators:

= Move cells only

+= Add cells

-= Subtract

*= Multiply

/= Divide

%= Percentage

<table region or expression>

Table region as above, or an expression that returns a number or numbers to fill the cells. Expressions referencing tables have the following functions:

Functions of expressions referencing tables

FLIP ( )	Turns columns to rows and vice versa
JOIN_COLUMNS	Extends a table region. Appends columns of one table to another.
JOIN_ROWS	Appends a table region. Appends rows of one table to another.
LOADED( )	Returns true or false depending on whether the table is currently loaded.
NUMBER_OF_COLUMNS( )	Counts numbers of columns in region.
NUMBERS_FROM_TABLE( ) (VEC, COLS, ROWS)	Takes data from table and moves to data.
NUMBER_OF_ROWS( )	Counts number of rows in a region.
REPLICATE( )	to have a smaller number of columns or rows act on a large number of columns or rows (must be evenly divisible)
TABLE_FROM_NUMBERS( ) (vec, cols, rows)	Takes data from a vector and fills the table.

See Also *Mentor Volume I: "9.3.2 FUNCTIONS": Table Related Functions*.

MODIFY_ASCII (MA)

Modifies the contents of a data location or variable in ASCII format.

Syntax:

```
MA column.width
```

MA column-column
MA datavar

When you use MODIFY_ASCII, a template prints above the columns to be modified and the prompt displays MA-->. If you press **Enter**, nothing is changed. If you enter fewer characters than the field specified, only the characters you type are changed. If you enter too many characters you get an error message and are prompted to try again. You can modify a maximum of 70 columns.

Related Commands:

DISPLAY_ASCII, DISPLAY_BINARY, DISPLAY_COLUMN, DISPLAY_TEXT, MODIFY_BINARY, MODIFY_COLUMN, MODIFY_TEXT , PUT_ID, and SHOW *.

See Also:The CLEANIT utility in the *Utilities* manual.

MODIFY_BINARY (MB)

Modifies the contents of a data location or variable in binary format.

Syntax:

```
MB column.wid
MB column-column
MB datavar
```

MB displays each column and its current data, and prompts for new data:

Example:

```
MB 5.2

Column  ASCII  Binary
1/5       2      2              MB -->+1
1/6       *      1,3,9          MB -->2,5,9
```

You are prompted to enter data for each column of the variable or column set specified. Possible codes are punches 1 - 9, and 10 (or 0), 11 (or X), and 12 (Y).

You can specify multiple punches combined with a period (i.e., 1.3 for punches 1, 2, and 3), or specify separate punches separated by commas (1,3,5).

Just specifying punches replaces the data with those punches. A plus sign (+) before the punches adds those punches without affecting other punches, while a minus sign (-) removes just those punches. In the example above, the +1 will add a 1 punch to column 5, which now will have a 1 and a 2 punch.

You can continue entering codes for a column on the next line by specifying an ampersand (&) at the end of the current line.

B (or b) will blank the column. If you type !, no data is changed, and you will stop modifying and be returned to the ~CLEANER prompt. If you press **Enter**, nothing is changed in that column, and you continue modifying the next column.

Related Commands:

DISPLAY_ASCII, DISPLAY_BINARY, DISPLAY_COLUMN, DISPLAY_TEXT, MODIFY_ASCII, MODIFY_COLUMN, MODIFY_TEXT, and SHOW *B

See Also:The CLEANIT utility in the *Utilities* manual.

MODIFY_COLUMN (MC)

Modifies the contents of a data location or variable in column format.

Syntax:

```
MC column.width or variable
```

MODIFY_COLUMN works the same as MODIFY_BINARY, except that you enter the punches as numeric codes relative to the first column of a multiple-

column field. There are 12 possible punches per column (1 - 9, 0, X, and Y), so the 5 punch of the second column is referenced as punch number 17. You are prompted once only, and can modify a maximum of 70 columns at a time:

Example:

```
MC 5.2

Columns  Codes
1/5.2:   2,13,15,21
MC -->   3,24
```

This replaces the punches in columns 5 and 6 with a 3 punch in column 5 and a Y punch in column 6.

Related Commands:

DISPLAY_ASCII, DISPLAY_BINARY, DISPLAY_COLUMN, DISPLAY_TEXT, MODIFY_ASCII, MODIFY_BINARY, and MODIFY_TEXT

MODIFY_TEXT (MT)

Modifies the contents of a text type variable (usually collected in Survent with a TEXT type question). Text data is stored in a 'text area', usually at the end of the data case. It is used to store open-ended responses that can be very long or short. A “pointer” is created in the data which points to the beginning of the text response; in this way you can have variable positions for text data from case to case depending on the size of prior text responses, and thus, store much more information in the same space.

Syntax:

```
MT text pointer column or text variable or [loc$T]
```

MODIFY_TEXT calls the CfMC internal editor. This is a full screen editor on personal computers, or a line editor on line mode terminals. Refer to your *Survent* manual, section *4.2.1: RESPONDING TO DIFFERENT QUESTION TYPES,* for instructions in using the editor for your operating system.

If you open a questionnaire file with ~QFF_FILE <filename>, Mentor will know where the

text is; otherwise, you must specify the start of the text area on either the ~CLEANER FILE or the ~INPUT statement with TEXT_LOCATION=col<.wid>. Col.wid refers to the starting location of the text data. If you do not specify a width then the program assumes you mean starting here through the end of the case.

By default the Survent program sets the text location to the first column of the next record after the last data column through the end of the case, unless specified otherwise in the PREPARE specifications with the TEXT_LOCATION= header option.

Related Commands:

DISPLAY_ASCII, DISPLAY_BINARY, DISPLAY_COLUMN, DISPLAY_TEXT, MODIFY_ASCII, MODIFY_BINARY, and MODIFY_COLUMN

NEW_CASE (NEW)

Positions the user at the end of the data file and opens an empty data case to work with. You must have a data file open with the ALLOW_NEW option specified on the ~INPUT or FILE command or have created a new input file using ~INPUT CREATE.

Syntax:

```
NEW "caseID"
```

Case ID is optional; if included, the new case will be given that ID.

Related Commands:

~INPUT ALLOW_NEW, ~INPUT CREATE

NEXT (N)

Positions the user on a data case and displays its case identifier (ID). You can then DISPLAY or MODIFY the data in that case. The default is to move to the next

case in the file. You can also indicate a particular case ID, or a number of cases forward.

Syntax:

```
NEXT #, +#, string
```

Examples:

N Moves to the next case in the file

N"0023" Moves to case ID 0023

N +10 Moves forward 10 cases

N 20 Moves to the 20th case in the file

N FIRST Moves to the first case in the file

N LAST Moves to the last case in the file

When referencing the case ID, you must use the exact string of the ID; for example, "23" is not the same as "023". Case IDs can include any ASCII character, not just numbers. Use ~CLEAN PUT_ID to initially assign or change a case ID. Case IDs may not be in order in the file, so you cannot assume moving forward a relative number of positions in the file will put you on a particular case. For example, moving forward ten cases from case ID 5 (e.g. N +10) does not guarantee you will be on case ID 15.

NEXT displays "(DELETED)" on cases deleted with ~CLEAN ASSIGN_DELETE_FLAG.

Use FILE SELECT to limit your search to particular cases. Use FIND or HUNT to find the next occurrence of a case that fits a particular condition.

Related Commands:

BACKUP, FILE, FIND, HUNT, ~SET SEQUENTIAL_READ_

NEXT_REDO (NR)

Moves to the next case (see NEXT) and executes the previous command.

Syntax:

```
NR
```

Example:

```
DB 1.5; MB 1.5; DB1.5
NR
```

This would display the five columns, bring up the data modifier prompt, and redisplay the modified columns on the next case.

NO_UPDATE

Prevents Mentor from updating the input file. Even if you have made changes to the case, you could drop them with NO_UPDATE. You would use this command in a procedure to make changes to one or more cases, perform some operation with the changed case, but not write the changes back out to the input file.

Syntax:

```
NOUPDATE
```

Related Commands:

YES_UPDATE and ~INPUT DROP_CHANGES

NULL

A blank statement that does nothing, but often serves as a placeholder on a label to GOTO.

Syntax:

```
NULL
```

Example:

```
GOTO NEXTITEM
```

```
...
NEXTITEM:  NULL
...
```

OK_COLUMNS (OKCOL)

Means that the data in the field(s) specified is good and doesn't need to be checked. It also means that the columns should subsequently be ignored by CHECK_COLUMNS. It is usually used on locations such as the case ID field and locations holding data generations, or SURVENT TEX question data.

Syntax:

```
OKCOL [col.wid]
```

Example:

```
OKCOL [1.10]
```

This example says what is in columns 1-10 is always good, no matter what is in the columns.

You can also check multiple column sets in this statement.

Example:

```
OKCOL [1.10] [2/5.3] [3/40,...,4/40]
```

PAUSE

Pauses the program and displays the message "Press any key to continue" at the console.

Syntax:

```
PAUSE
```

~CLN PRINT_LINES

PRINT_LINES (PRT)

Prints the text specified, and the data or text associated with the variables named, to the print file (if one is specified), to the list file, and/or to the terminal. This is generally used to produce case-by-case reports on a data file rather than standard cross-tabulations. See *"9.1 GENERATING SPECIALIZED REPORTS"* for examples, in the *Mentor Volume I.*

Syntax:

```
PRT <#|num var|string var> "format line" datavar1
datavarn
```

Options:

#|num var|string var
The number of the print file specified on the >PRINT_FILE meta command when the particular file was opened; # can be any number 1-20. This is only specified for printing to multiple print files, and it is not required. Num var or string var can also be used to specify the number or the name of the print file to print to. The default is the currently opened print file.

"format line"
Consists of text and \codes, and must be specified inside quotes ("). \codes refer to any of the special print controls you can specify here. \codes are listed below, see *"Printing the Contents of Data or Variables."*

datavars
Any data variable. The text that prints depends on the type of variable and whether you are printing the title, response text, response code, response number, or just the data. The types of variables are listed below; see *"Types of Data Variables."*

Example:

```
PRT "This is the response \S" QN1
```

The PRT command specification can be continued with an ampersand (&):

Example:

```
PRT #1 &
"This is a print statement for person \S at \S" Person &
DATE_TIME
```

One format line can be up to 140 characters long and can be continued by using two ampersands:

Example:

```
PRT "This is a long text info line"&&
    " that is being specified across two lines"
```

Output goes to the print file (if one is opened), otherwise to the list file. Note that the list file will also include your specification lines and any program messages. Send program messages to the terminal as well by using the ECHO option on either the >PRINT_FILE meta command or on the LISTFILE command.

Any printed line must fit in the parameters of the PAGE_WIDTH specified either on the >PRINT_FILE or LISTFILE. If you are using >PRINT_FILE with the LASER_CONTROL=<file name> option, then printing is also controlled by the print strings specified there. For instance, you can specify the print string to use for \B (bold) and \U (underline), as well as the page size, initial and final print strings. Refer to *Appendix D: CFMC CONVENTIONS*, *Command Line Keywords, LISTFILE* in the *Utilities* manual.

Related Commands:

See also PRINT_TO_DATA and TERMINAL_PRINT, which have a similar format and use the same \codes to control formatting. TERMINAL_PRINT prints to the screen only. PRINT_TO_DATA prints specially formatted lines to the data. PRINT_TO_DATA allows you to zero-fill and put decimals in numeric data, or write out response codes and subscripts for categorical variables.

Printing The Contents Of Data Or Variables

\ print control items are filled with the corresponding data variable in the data variables at the end of the format line. The format for these print controls is:

Syntax:

`\<modifier(s)><width><.numdecs><|maxitems><code>`

Options:

modifiers

Control how multiple items print, justification, numeric format, and number of lines to print. See "More on Modifiers" below.

width

The number of characters to use when printing. If there is no width, the width of the variable is used, or the width of the string to be printed (this can be of variable length).

.**numdecs**

The number of decimals to print when printing numeric variables.

|maxitems

The maximum number of response texts, codes, or category numbers to print from a multiple response categorical variable, multiple field datavar, or a vector.

<code>

* Prints response codes from a category variable. Response codes available from variables built with ~DEFINE PREPARE= or from Survent specifications in the ~PREPARE module.

\S Prints response text from a data location or category variable; trail-

ing blanks are dropped. Multiple responses are separated by commas.

\# Prints the subscript position of the category of the variable(s). The first category in the variable is assigned the numeric subscript of 1, the second category is assigned 2, etc.

\V# Prints the following (must be used with \S):

V1	Variable's name. [Q1]
V2	Variable's title text. [What's your age?]
V3	Variable's location. [1/5]
V4	Variable's question number. [0.10]
V5	User text. [This is a comment about the variable.]

Example:

```
PRINT_LINES "Variable's name: \V1s \V1s" Q1 Q2
```

This will print the Variable names for questions one and two.

Example:

```
PRINT_LINES "For: \V2 \NAnswer Was: \S" AGE AGE
```

This will print the title and the response to the variable AGE.

Types of Data Variables

TypeExampleText that prints

Category `[10^One:1/2/Last:3-Y]` Text of response
[23#"Less than 10"0-9/10-99]

Numeric `[10.5]` Number (right justified)

Punch `[10.5$P]` Punches in the format

"13.5/26/B/0XY/7" with

slashes between columns.

String `[10.5$]` Prints the string left- justified, strips trailing blanks.

Location `[10.5$L] or name`Prints exactly what is in the data, including blanks.

Text `[10.1$T]` Prints the text in a text variable (collected with Survent).

Expression `([10.5]*5)/NUMITEMS([20^1//50);` Prints the resulting number. You must use a semi-colon(;) to end the expression if more datavars will be specified after it.

More on Modifiers

The \<modifiers> modify the printing of the strings from data variables. Use an underscore (_) or space ( ) after the modifier to separate it from other parts of the \code printing.

The main special character modifiers are (the rest of the special character modifiers are used for printing special characters, see “Printing Special Characters” below):

\<, \=, \>

To left-justify, center, or right-justify the text printed, respectively. This requires a width in which to justify the item. By default, strings are left-justified and numbers are right-justified.

Example:

PRT "'\>20S'" Name

will print:

```
' Fred Smith'
```

\+, \-

These control what to do if there are control codes in the text of the title or response(s) being printed. The default for titles is \+ meaning execute the control codes; for instance, if there is a \N for new line, start printing on the next line. \- says not to execute the control codes in the text. The default for data is \-, meaning a backslash in the data will print as a backslash.

Example:

```
PRT "\{$,}10.2S" Income
```

will print:

```
$1,150,123.00
```

This prints the number with two decimals, commas in the thousands place, and a dollar sign ($) in the thousands place.

Letter Modifiers

Modifiers can also start with a letter. The basic letter modifiers are A, J, K, M, O, Q, R, Y, and Z. There are also letter modifiers that are print control codes not associated with any data variable: G, N, X, and T (and the numbers 1 and 2), see *"Other Standard Print Control Codes"* below. The rest of the letter modifiers (B, E, F, I, U, W, and C) are for use with laser printers, see *"Print Control for Laser Printers"* below.

\A#

Controls wrapping of strings to 2nd+ lines. If a width is not specified, items wrap at the page width. All other printing continues on the last line.

A# options:

0 - Wrap at width on words or according to the variable (for new lines). Text questions wrap at their line breaks.

1 - Wrap within width on words only, unless there is a hard-coded line break (\n) (DEFAULT). Text questions show no line breaks.

2 - Wrap like \A1, but indent the 2nd+ line to the second word of the first line.

3 - Wrap like \A1, but print line breaks (<CR>) in text questions as ;<space> instead of just a <space>.

Example:

```
PRT "\A3_20S" Reason
```

This would print 20 characters wide, and print a semi-colon "; " between each line of text on a text question.

\J#

Controls what to print between multiple sub-items.

J# options:

1 - Comma <space>: ', ' (default)

2 - Comma only: ','

3 - <Space> only: ' '

4 - Nothing: ''

Example:

```
PRT "\J2_20#" Cardtype
```

will print:

```
01,03,17,23,32
```

This would print a 20 characters wide string of the category numbers for the variable Cardtype with a comma only between the items.

\K

Forces each sub-item to a new line before printing. All other controls stay in effect within sub-items.

Example:

```
PRT "\K_18S" Cardtype
```
will print:

```
Visa Gold card
American Express card
```

This would cause the second sub-item (American Express card) to start on a new line.

\M#

Controls truncation of strings.

M# options:

1 - Print "..." at the end and warn

2 - Print nothing extra, but warn

3 - Print "..." at the end and do not warn

4 - Print nothing extra and do not warn

5 - Print "..." at the end and print an error message

6 - Print nothing extra and print an error message

7 - Wrap at the <width> specified. (DEFAULT)

Example:

```
PRT "\M1_20S" Other_hosp
```
will print:

```
St. Michael's hosp...
```

This would print a 20 character string, and if the string to be printed was wider than 20, it would truncate the printing with '...' and print a warning in the list file.

\O#

Controls the number of characters to indent the 2nd+ lines when wrapping, where #=number of characters. It is overridden by \A2.

Example:

```
PRT "\O3_18S" Othrcard
```
will print:

```
American Express Gold card
```

This would cause the second line to print 3 characters to the right of the start of the first line.

\Q#

Says how many lines to print (maximum) for this print item, where #=number of lines to print. The default is unlimited.

Example:

```
PRT "\Q2_18S" Cardtype
```
will print:

```
Visa Gold card
American Express
```

This will not print the last line “card” because it is restricted it to printing only two lines.

\R#

Controls which sub-item to start printing at for multiple sub-items, where #=sub-item to start at. The default is the first sub-item. This is useful in that you can

specifically place each sub-item on the line wherever you want, without using the defaults for spacing and continuation characters.

Example:

```
PRT "\R1_16S   \R2_16S" Othrcard Othrcard
```

will print:

```
Visa Gold Card     American Express
```

This would cause “Visa Gold Card” to print first if it is the first code chosen, and “American Express” to print if it is the second code.

\Y#;\Y%#;\Y$#

In addition to controlling items and sub-items, you can get combinations of things printed for each **\S**, *****, or **\#** code, such that you can see all the information about the item in a formatted output using **\Y#**, where # is one of the numbers below. A different thing prints for **\S**, *****, and **\#** such that all possible combinations of response code, number, and text are printable.

\Y%# - removes the space between items. Use \Y$# to say print all categories in the format specified and flag the answered ones. This option allows you to generate a hard copy of the answered questionnaire.

\Y# with **\S** always prints the text last; with ***** it always prints the response code last; and with **\#** it always prints the response number last. Here are all the combinations of numbers and \codes, where 'response' is the response code, 'number' is the response number, and 'text' is the response text:

For **\S**, the **\Y** sub-codes return:

1: response text

2: (response) text

3: number text

4: #number text

5: zero-filled-number text

6: number response text

7: #number response text

8: zero-filled-number response text

9: number (response) text

10: #number (response) text

11: zero-filled-number (response) text

12: response number text

13: response #number text

14: response zero-filled-number text

15: (response) number text

16: (response) #number text

17: (response) zero-filled-number text

For *, the \Y sub-codes return:

1: (response)

2: number response

3: #number response

4: zero-filled-number response

5: number (response)

6: #number (response)

7: zero-filled-number (response)

For \#, the \Y sub-codes return:

1: #number

2: zero-filled-number

3: response number

4: response #number

5: response zero-filled-number

6: (response) number

7: (response) #number

8: (response) zero-filled number

Example:

```
PRT "\Y2S" Othrcard
```
will print:
```
(01) Visa Gold Card
(04) American Express Card
```

This would cause “Visa Gold Card” to print first if it is the first code chosen, and “American Express” to print if it is the second code.

\Z#

Controls zero-fill printing of numbers or category numbers (\#).

Z# options:

1 - Zero-filled, fixed length

2 - Fixed length, but not zero-filled

3 - Variable length and not zero-filled (default)

Example:

```
PRT "\Z2_20#" Cardtype
```
will print:
```
 1, 3, 17, 23, 32
```

This would print a 20 character wide string of the fixed length category numbers for the variable”'Cardtype”. Control what prints between the numbers with \ J.

With options 1 and 2, you can also control how wide to make each item with a colon and a number indicating the width. `\Z1:2` would print each item zero-filled with a fixed-width of two. `\Z2:3` would print each item not zero-filled, with a fixed width of three. See ~CLEAN PRINT_TO_DATA for an example of using this width option.

Other Standard Print Control Codes

These controls allow you to format the text printed; they are not associated with any data variable.

Certain controls are for positioning the following text; they use a number or numbers to specify where to go or what to do. They are as follows, where number (#) is any number that would fit within the page width and length:

\#G - Goes to this print position on the current line (a number (#) is required).

\#N - Prints new line character(s); the default is to start a new line for every new "format line", plus any \N's specified. Use ~SET -AUTOMATIC_NEW_LINE to force the program to stay on the same line by default.

\#T - Generates this number (#) of tab characters.

\#X - Skips this number (#) of spaces.

\(#1,#2) - Goes to print position #2 on line #1 of the page. This must be to the right of the current line position or on the next line or below. The line number must be <= the page length. The line number defaults to the current line (if left blank), and the print position defaults to one.

Use the options BOTTOM/TOP_MARGIN and PAGE_LENGTH/WIDTH on either the >PRINT_FILE meta command or the ~SET command to control the page size for printing. Use WHEN TOP/WHEN BOTTOM to print at the top or bottom of the page. Use SKIP_LINES to skip lines independent of the format in the PRINT_LINES.

Controlling the Language to Print

\Lxx only prints the following text if language "xx" is specified in the >Language "speaking" value or is the first language in the language set. This can be specified by saying:

>language speaking=xx or >language set=(xx yy). It will also print if the “set” or “speaking” value is “**” for “print everything.”

Printing Special Characters

Other codes are used to print characters that are difficult to print or control when the line is printed. They are:

Codes used to print special characters

“”	Prints a quote (“)
****	Prints a backslash (\)
\'	Prints an apostrophe (‘)
\[	Says start ignoring all backslash codes (just print them) except \]
\]	Says stop ignoring all backslash codes
\P	Prints a page kick and clears the screen if output is to the terminal
\!	Like \N, but prints the line now and causes the line pointer not to go to the next line for the next case (this is most often used for printing multiple items across cases on the same line)
\?	Prints the line now and pauses (when you are working interactively)
\^##	Allows you to print any printable characters from your printer’s ASCII code sequencechart, where ## are two hexadecimal digits *
\^A_	Prints a control sequence, where A is any letter (e.g. \^G for a bell). As an option: This can be followed by an underscore (_) so you can put other characters right next to the control sequence

* See *Appendix G: GRAPHIC CHARACTERS* in your *UTILITIES* manual for a listing of the codes in the standard ASCII code sequence chart.

To print a quote, use two quotes in a row.

Example:

```
PRT "Here is a ""quoted"" string."
```

will print:

```
Here is a "quoted" string.
```

Using \! you can print information from more than one respondent on a single printed line. You must turn off the program's automatic new line default. Here are some sample specifications and output.

Example:

```
~SET -AUTOMATIC_NEW_LINE
  ~INPUT data1
  ~DEFINE
   PROCEDURE={proc4:
    PRT "First: \S\20g" Case[1.4$]
    NEXT
    PRT "Second:\S\40g" Case
    NEXT
    PRT "Third:\S\60g\!" Case
   }
  ~EXECUTE PROCEDURE=proc4
  ~END
```

will print::

```
First: 0001        Second:0002         Third:0003
First: 0004        Second:0005         Third:0006
```

Text from variables containing either \N or \P will print with new lines and page breaks, unless \+ or \- was specified before the \S or \V to print the text.

Example:

```
var1[$T="Title of var1\Nsecond title line\N &&third
title line after the backslash N" &
1/10^"category title with a \Nline break":1]

PRT "Title: \V \Nresponse: \S" var1 var1
```

will print:

```
Title: Title of var1
second title line
third title line after the backslash N
response: category title with a
line break
```

Print Control for Laser Printers

In addition to the other print controls, certain \codes can be used to insert printer control strings into the text. These will only be used if you have specified LASER_CONTROL=filename on the >PRINT_FILE meta command. The laser control contains the replacement print strings, otherwise all \codes will be stripped from the text. You can provide replacement strings for a specific word processor or printer. See *Appendix D: CFMC CONVENTIONS*, *Command Line Keywords*, LISTFILE LASER_CONTROL= for a complete listing of laser control keywords and examples.

Control codes:

\B - Starts *bold* character attribute. **\-B** turns it off

\E - Ends all of whichever of the above five enhancements are on.

\F - Starts *flashing* character attribute. **\-F** turns it off

\I - Starts *inverse* character attribute. **\-I** turns it off

\U - Starts *underline* character attribute. **\-U** turns it off

\W - Starts *wide* character attribute. **\-W** turns it off

You can set color attributes on and off as follows:

\C+fb_ Sets the foreground/background color, where “f” and “b” can be any of:

Z:Black**W**:White

B:Blue**G**:Green

C:Cyan**Y**:Yellow

M:Magenta**R**:Red

An underscore (_) optionally can be used to separate the code from the text; if the text after the "b" character is not a "c" it is considered text. The program will also read \DC+fb and \DDC+fb since they are used in the Survent program for default colors.

E Resets color back to printer default color.

You can also use your own print control strings to do whatever you want. These strings are defined in the LASER_CONTROL file specified on the >PRINT_FILE statement:

\~<letter> Where letter can be any letter A to Z, this turns on a user-specified print control string.

\~-<letter> Turns off the corresponding user-specified print control string.

These controls can be passed automatically from Survent PREPARE specifications to the text and/or response of variables to be printed in Mentor by using the ~SPEC_RULES option USE_PRINT_ENHANCEMENTS when the questionnaire is compiled.

PRINT_TABLES (PRTTAB)

This is the same as the ~EXECUTE PRINT_TABLE option, except that the table does not need to be loaded into memory first. The rules for table elements in the ~EXECUTE block apply, i.e., if an EDIT statement is active in the ~EXECUTE block when a table is printed with PRINT_TABLE, then the table will be printed with that EDIT statement.

Syntax:

```
PRTTAB <tablename>
```

You can specify more than one table on the PRTTAB statement. The line can be continued with an ampersand (&).

Example:

```
PRTTAB T001 T004 T023 T957 ...
```

PRINT_TO_DATA (PRTDATA)

Has the same syntax as the PRINT_LINES command, except that the results are "printed" into the data instead of to the screen or a print file. Used to create special data formats. For instance, it can be used to zero-fill or put decimals in numbers in the data, or to recode data into numeric codes.

Syntax:

```
PRTDATA  datavar repeat=# "format items"  variables
```

Options:

datavar

Any single or multiple location data variable

repeat=#

The number of times to re-execute the PRTDATA statement for multiple variables. The number must match the number of variables.

"format"

Any print format from PRINT_LINES

variables

Are optional and fill the "format" data elements. There must be the same number of variables as "format" data elements

Example:

```
PRTDATA [10-45] "Name: \S" Respname
```

This puts “Name: Fred Smith” in the data columns 10 - 45.

You can do any math equation on the PRINT_TO_DATA line to put the result in the data as a decimal or zero-filled number.

Example:

```
PRTDATA  [10.4] "\Z0_3.25" [10.4] +1
```

This puts the sum of location [10.4] and 1 back in 10.4, and zero-fills the result, as a three-digit number with two decimals.

Example:

```
PRTDATA [5/15.5] "\5.2S"  [10-13] / 5.1
```

This puts the result of location 10-13 divided by the number 5.1 in location record 5, columns 15-19 as a decimal number with 2 decimals, i.e., 23.95.

Example:

```
PRT_DATA [11.48] "\Z1:2J4_4#" [1.2^1//24]
```

This copies the multi-punched data from columns one and two ([1.2^1//24]) as zero-filled (Z1) two-column (:2) subscript positions (\#) with no comma separator (J4) to columns 11 to 59 ([11.48]). If the punches were 3, 5, and 9, the data would look like “030509.” This is an example of converting multi-punched data to zero-filled ASCII data (this is often referred to as “spreading the data.”)

PUT_ID

Puts the data from the location specified into the case ID field of the case header. The data variable must be a string type variable (i.e., [col.wid$]). This is the only way you can change the case ID. Remember, the data you display for the case ID

columns cannot be the actual case ID in the case header, and simply modifying the data will not change the case ID unless you also use PUTID. Display the actual case ID with the CASE_ID system constant (e.g., SAY CASE_ID).

Syntax:

```
PUTID string variable or [col.wid$]
```

Example:

```
PUTID [1/5.4$]
```

Places the contents of columns five through eight in the case ID field.

REDO

Executes the previous command.

Related Commands:

FIND_FLAGGED_REDO; FIND_REDO; NEXT_REDO

RESTORE

Restores the current data case to when it was last written to disk. If ~INPUT ALLOW_UPDATE is set and you have moved to a new case, you cannot RESTORE the case to its original state. It is best to back up your data file before making modifications or write changes to a new file (see WRITE_CASE and ~OUTPUT).

Syntax:

```
RESTORE
```

SAY (S)

Will print strings and/or variables to the list file or print file. The format of the line is dependent on the type of variable being shown. Blank spaces are removed at the

beginning and end of strings. One blank space prints between each element. To print the quote character, use "". To print on the next line, use \N at the beginning of a string. System constants can be used to annotate the display (see CASE_ID below).

Syntax:

```
S datavar "string1" variable1 "string2" variable2 etc.
```

Example:

```
S  "For"  CASE_ID  "respondent's gender is" "\N" GENDER
```

This will print:

```
For 0001 respondent's gender is (Gender: 2: MALE)
```

Notice that named variables, when referenced, print the variable name, the response code, and the text of the variable in parentheses.

Related Commands:

See PRINT_LINES to print lines with more specific format control. See *8.3.1 SYSTEM CONSTANTS* in your *MENTOR* manual.

SET

Sets any options you can set in the ~SET block. See the ~SET command for more information. Each line of SET commands needs to start with SET.

Syntax:

```
SET option1, option2
SET option3, option4, option5
```

One special SET option affects ~CLEANER data modification:

TESTING_MODE does not allow any changes to the data file unless the file is opened with ALLOW_UPDATE. (TEST)

These ~SET options are particularly useful in the ~CLEANER block:

```
CLEANER_DEFINITION=
LOGGING
TESTING_MODE
PRODUCTION_MODE (default for ~CLN FILE)
```

Related Command:

~SET

SHOW

Lists the structure and content of a variable without having to go to the ~SHOW block. See ~SHOW for details.

Syntax: SHOW options

Options:

*** -** Displays the entire case in ASCII card image format.

***B -** Display the entire case in column binary (punch) format.

Related Commands:

~SHOW

SHOW_TABLES (SHOWTAB)

Lists the tables/regions in memory. Tables with an asterisk (*) next to them have been modified but not stored yet.

Syntax:

SHOWTAB

SKIP_TO (SKIP)

Tells the program the number of lines to skip before printing again.

Syntax:

```
SKIP #
```

Option:

- The number of lines to skip from current position. If you use a negative number, it skips forward to a position # lines above the bottom of the page.

Example:

```
SKIP 5
```

This will skip down five lines from the current position.

Example:

```
SKIP -4
```

This will skip to four lines from the bottom of the page.

Skips will occur in the default print file, unless you indicate otherwise. You can indicate a specific print file by referencing it by name or number.

Example:

```
SKIP #5; -3
```

This will skip to three lines from the bottom of the page in printfile #5. (Without the semi-colon, (`SKIP #5 -3`) Mentor will handle this an arithmetic expression and skip two lines.)

Example:

```
SKIP #"guidos"; -3
```

This will skip to three lines from the bottom of the page in printfile named guidos.

Related Commands:

PRINT_LINES and WHEN TOP/WHEN BOTTOM

STORE_TABLES (STORE)

Stores the table(s) specified into a DB file. The DB file must open with write access. See >CREATE_DB, >USE_DB, >DB_STATUS.

Syntax:

```
STORE table1 table2 tablen
```

TERMINAL_PRINT (TERMPR)

Works like PRINT_LINES for formatted display, but goes to the terminal (console) only, not to the open list file or print file. This is used in interactive applications, or to display messages to the user as a procedure is executing.

Syntax:

```
TERMPR  "format"  variables
```

Options:

"format"

\! Print the line right now

\? Print the line, and wait for **Enter** to continue

\C Clear the screen

variables

Any previously defined variable; optional.

Related Commands:

See PRINT_LINES for more detailed description of options available.

TERMINAL_SAY (TERMSAY)

Displays category text and data to the terminal (console) only and not to the open list file or print file. This command could be used in an interactive application, or to display messages while a procedure is executing. See examples under the SAY command.

Syntax:

```
TERMSAY "string1" variable1 "string2" variable2 ...etc.
```

TERMINATE (TERM)

Quit the procedure on the current case, and go to the end of the file. It will then print the associated strings and variables, and execute any command after EXECUTE_EOF commands. The print line has the same format as the SAY command.

Syntax:

```
TERM  "string1" variable1 "string2" variable2 ...etc.
```

You must specify at least one string or variable to print when TERMINATE executes.

TRANSFER (T)

Has the same syntax as MODIFY <data>, but TRANSFER verifies that the data on both sides of the equals = sign is the same type. The MODIFY command is used to convert data from one type to another, while TRANSFER is used to change the contents of a variable. For example, TRANSFER would be used to add two numbers together to get a third number variable.

Syntax:

```
T  a = b
```

Example:

```
T [5^1/2/3] = [10.3#" ed"/"bob"/"joe"]
```

```
T [5] = [6^5]
```
would return an error because category data is being moved into a numeric location.

You can zero-fill the receiving field with the variable modifier "*Z".

Example:

```
T [10.5*Z] = [20.2] * [32.2]
```

Related Commands:

MODIFY

UNDELETE

Removes the delete flag. The input file must be opened with the ~INPUT ALLOW_UPDATE option before the flag can be removed. Because the case is dropped from memory you need to reposition on the case with the NEXT command before issuing the UNDELETE command.

Syntax:

UNDELETE

Related Commands:

ASSIGN_DELETE_FLAG

UNLOAD_TABLES

Unloads tables from working memory. UNLOAD_TABLES does not save the table in a DB file.

Syntax:

```
UNLOAD table  table  table
UNLOAD *
UNLOAD !
```

Options:

table

The name of a table to be unloaded.

*

Unloads all tables not specifically LOADed or CREATEd (see LOAD_TABLES and CREATE_TABLE)

!

Unloads all tables from memory

UPDATE

Immediately writes the current case back to the input file, overwriting the original case.

Syntax:

```
UPDATE
```

Related Commands:

~UPDATE

UPSHIFT (UP)

Changes all alpha characters in the specified position to lower case. (See also *Mentor Volume I:* Chapter 3: Changing Case.)

Syntax:

```
UP datavar
```

Example:

```
UP gender
UP [1.80$]
```

Related Commands:

```
~CLN DOWNSHIFT, ~SET CASE_SENSITIVE
```

VIEW

Views the current case using a questionnaire created by ~PREPARE specifications. Requires a questionnaire file (QFF) to be opened with ~QFF_FILE as well as the matching data file with the same case length.

Alter the response to a question by typing the keyword ALTER, and then enter the new response.

Syntax:

```
VIEW
```

Related Commands:

```
~VIEW, NEXT
```

See Also: The *SURVENT* manual, Chapter 4, *4.1.4 VIEWING A PREVIOUS INTERVIEW.*

WHEN TOP/WHEN BOTTOM

Says what to do when you reach the top or bottom of the page when printing formatted lines using PRINT_LINES.

Syntax:

```
WHEN TOP/BOTTOM

PRINT_LINES commands, other ~CLEANER commands, or
conditions ...
END_WHEN
```

If WHEN TOP is specified, it dictates what to do at the top of a page. If WHEN BOTTOM is specified, it dictates what to do at the bottom of the page. See related statements ~SET (page length, width), PRINT_LINES (to print formatted lines), and >PRINT_FILE (to control the print file parameters).

Example:

```
WHEN BOT 3
SKIP_TO -3
PRINT_LINES "\NLine 1 of WHEN BOT block\N"
PRINT_LINES "Line 2 of WHEN BOT block\N"
PRINT_LINES "Line 3 of WHEN BOT block"
END_WHEN
EXECUTE_EOF
SKIP_TO -3
PRINT_LINES "\NLine 1 of WHEN BOT block\N"
PRINT_LINES "Line 2 of WHEN BOT block\N"
PRINT_LINES "Line 3 of WHEN BOT block"
```

The example above will print on the last three lines of each page. There will be no blank lines between the last regular printed line and the first line of the WHEN BOTTOM block.

If you want a blank line separating the last regular printed line from the first line of the WHEN BOTTOM block, include it inside the WHEN BOTTOM block. Be sure to count this line on your WHEN BOTTOM # and SKIP_TO -# settings. See *8.1 GENERATING SPECIALIZED REPORTS, Printing A Report Footer Using WHEN BOTTOM.*

If you use both WHEN TOP and WHEN BOTTOM together, the WHEN BOTTOM statements must precede other regular print statements.

You can also use WHEN TOP/WHEN BOTTOM to create multiple print files by using the syntax "WHEN TOP ##" where the first # is the pound sign and the second # is a number indicating a specific print file. For example, WHEN TOP #2 says execute the following commands when the second print file is at the top of the page. WHEN_BOTTOM #2 4 says execute the following commands when four lines from the bottom of the second print file. Make sure that all print commands

inside each WHEN TOP/BOTTOM block refers to only one print file. The following example creates two different print files with different headers and footers and different print file lengths.

Example:

```
~define

PROC={proc1:
          WHEN TOP #1
PRINT #1 "Top of page file one."
          ENDWHEN
          WHEN TOP #2
PRINT #2 "Top of page file two."
          ENDWHEN
          WHEN BOT #1
PRINT #1 "Bottom of page file one."
          ENDWHEN
          WHEN BOT #2
PRINT #2 "Bottom of page file two."
          ENDWHEN
PRINT #1 "Printfile 1 gets one line per trip through the
proc."
PRINT #2 "Printfile 2 gets two lines per trip through
the proc."
PRINT #2 "Printfile 2 gets two lines per trip through
the proc."
          }

'' Using bank data causes the proc to run 100 times.
          ~INPUT bank
          >PRINTFILE aaa #1 page_length=50
          >PRINTFILE bbb #2 page_length=40
          ~EXC PROC=proc1
          ~END
```

WRITE_CASE (WRITE)

Writes the case to the output file. You must have an ~OUTPUT file open.

Syntax:

```
WRITE #file [datavar] [idvar]
```

Options:

#file

Specifies which file to write to when multiple data files are open.

[var]

The location of data to move (optional).

[idvar]

The location of the new case ID (optional).

WRITE with no options writes the whole case to the output file.

Example:

```
WRITE [1.240] [1.4]
```

The data file is 240 columns with the case ID in 1.4

If you have multiple ~OUTPUT files open, your WRITE_CASE command should specify which file to write to. Indicate the file by #1, #2, etc.

Example:

```
WRITE #1
```

WRITE_QSP

Writes out an ASCII copy of a table to the program-generated spec file (QSP) opened with ~SPEC_FILES. This is the specification file version of the table, not the print version. Use this command to transfer db files (in which you have stored finished tables) between platforms rather than recompiling the specifications on the new platform or to keep a record of the table and its elements. (WRITEQSP)

For example, where the sample size is very large the basic tables could be run on a mini because of its speed and disk capacity, but table manipulation work could be done on a PC by making ASCII versions of the tables needed and transferring them to the PC.

Syntax:

```
WRITEQSP <tablename>
```

Example:

```
>USE_DB samp1 ''Where previous tables run stored
~SPEC_FILES samp1
~CLEANER
>REPEAT $A=T001,...T020  ''e.g, T001 etc. are the
"table names in the DB file.
 WRITEQSP $A
>END_REPEAT
~END
```

This is a sample of the specifications written out to SAMP1.QSP:

```
table=  T001: { 2,11,  2,11
  header=  tabtop_h
  footer=  tabtop_fo
  title=  qn1_t
  column=  tabtop_c
  row=  qn1_r
  banner=  tabtop_bn
  stub=  qn1_s
  main_edit=  tabtop_e
 r=-1 25. - 25. 5. 5. 10. 1. 11. 8. 5. 15.
 r=0 - - - - - - - - - - -
 r=1 7. - 7. 2. 1. 1. - 4. 1. - 4.
 r=2 5. - 5. - 1. 4. - 3. 2. 1. 4.
 r=3 6. - 6. 2. 1. 2. - 2. 1. 2. 4.
 r=4 2. - 2. - 1. 1. - 1. 1. 1. 1.
 r=5 3. - 3. - 1. 1. 1. 1. 1. - 1.
 r=6 2. - 2. 1. - 1. - - 2. 1. 1.
r=7 23. - 3.47826 4. 3. 3.33333 1. 3.72727 3.16667 3.
3.64286
```

```
r=8 23. - 1.3774 1.1547 1.58114 1.22474 ? 1.3484 1.47196
0.8165 1.21574
r=9 23. - 0.28721 0.57735 0.70711 0.40825 ? 0.40656
0.60093 0.40825 0.32492
 }
```

In SAMP1.QSP:

T001 is the name of the table (this may be different from the name printed on the table).

2,11, 2,11 indicates that of the total number of rows (11), two are system columns and rows, the rest are user-defined rows.

r= indicates a table row beginning with system Total row or the first user-defined row. The values for each table cell follow.

Related Commands:

`~WRITE_SPECS, ~DEFINE TABLE=`

YES_UPDATE

Forces an update of the input file even if a case is not changed. This command would be used in a procedure to control when and if a case is updated.

Syntax:

`YESUPDATE`

Related Commands:

`NO_UPDATE, ~INPUT DROP_CHANGES`

~COMMENT (COM)

This command reads all lines after it (except meta commands starting in column one) as comments until the next tilde command. A minus sign (-) before a tilde command has the same affect as making it a ~COMMENT block.

Syntax:

```
~COM
```

~COPY

This command was once used for copying data files, but it is now no longer needed. Use your operating system COPY command, or "~INPUT filename options" and "~OUTPUT filename options WRITENOW" to copy files or change the format of files.

You can use the COPYFILE utility to copy, combine, and create subsets of data files, the RAWCOPY utility to recover corrupted data files, and LZW to compress data files and move them across platforms. See the *Utilities* manual for descriptions of COPYFILE, RAWCOPY and LZW.

~DEFINE (DEF)

This command is used to define data, tables and text. Once defined, an item can be accessed anywhere in Mentor by name. In addition, items can be defined just before they are used; that is, in procedures, interactive cleaning runs or when executing tables.

Use ~SHOW to show the contents of the defined items in more detail. Defined items need a keyword and a name, as follows:

Syntax:

~DEF keyword=name: <contents of item>

The default keyword is VARIABLE=. VARIABLE= and AXIS= do not have to be specified. Mentor can tell the differentiate between the two.

Text and keyword-driven items must end with a closing brace (}). Variable descriptions must have an ampersand (&) at the end of the line to continue to the next line.

Below is a list of and short description of each keyword. The syntax and options are detailed further in the following pages. Because VARIABLE is most often used, it is mentioned first in this manual. The rest are listed in alphabetical order.

KEYWORD	DESCRIPTION
AXIS=	Defines axes for table column and row definition. This has the same syntax as VARIABLE=, except that it allows cross-case operations. You can specifically define columns and rows in a TABLE_SET.
BANNER=	Defines the banner to print over the columns of the table. This can also be defined as a TABLE_SET.
EDIT=	Defines options to control the way the tables print. These options also can be defined in a TABLE_SET.
LINES=	Defines lines of text to be printed on tables. Can also be defined in a TABLE_SET, using TITLE=, FOOTER=, HEADER=, etc.
PREPARE=	Makes simple variables using PREPARE questionnaire writing syntax.
PROCEDURE=	Defines programs to be used for case-by-case data table reporting or modification.
STATISTICS=	Defines the groups of columns or rows on which to do statistical testing (multiple T-tests).
STUB=	Defines the text and printing controls for rows of a table. Can also be defined as TABLE_SET.
TABLE=	Reads tables directly in ASCII format; used to create a filled table, or to import cell values from other systems (Lotus, QuattroPro, etc.)
TABLE_SET=	Defines groups of table elements to be used together. Any table element used in the ~EXECUTE block may be defined.
TABLE_SPECS=	Similar to TABLE_SET, but allows more than one occurrence of a table-building option.

KEYWORD	DESCRIPTION
VARIABLE=	Defines data for use in procedures of tables. This is the default ~DEFINE item type.

~DEF VARIABLE= (VAR)

This is the default keyword for ~DEFINE. Therefore, you don't need to include VARIABLE= before the definition. Variables have data elements (datavars) enclosed in brackets ([ ]); you can combine variables with joiners to form more complex expressions. If you want to continue the description on the next line. use an ampersand (&) at the end of the line.

Syntax:

VAR= varname: [$T="text" (#loops, incr) dataloc mod type categories]

VAR=

keyword, this is optional

varname

Name of the variable. Variable names may be up to 14 characters, starting with a letter and including letters, numbers, periods and/or underscores (_).

:

A colon is required if the variable has more than one datavar; that is, if you use any kind of joiner or function. The colon must follow the varname.

Examples:

Age [10#1//5] - colon not required

Group [11^1//3] - colon not required

Totlike: SUM([15.2,17,19]) - colon required (function)

Age_by_group: Age by [11^1//3] - colon required (two datavars)

Young.group_1: [10^1] and [11^1.3] - colon required (two datavars)

[ ]

Brackets enclose all new data variables. More than one [ ] item may be included in a variable. If so, it is connected by logical or vector joiners. If you don't use a colon, there cannot be a space between the varname and the left bracket.

$T="text"

Title of the variable; it prints when the variable is referenced in tables or procedures. If not specified, the variable name is printed as the title (or you can specify your own titles for tables).

To print a quote, use two quote marks "". Use \N to force a new line in the printed string. You can extend the text on a variable to the next line by using a dot and ampersand (.&) at the end of the first line, and continuing on the next line inside a new set of quotation marks. Also, you can say &&.

Example:

```
[$T="here is some text that needs to be extended" &
" to the next line like this" 10.5 # a/b/c/d/e]
```

(#loops,incr)

Used for LOOP type variables where data being referenced is over many columns with the same data structures. Defines the number of times to loop, and the data increment (distance) between iterations of the loop. Similar to overlay tables.

Example:

```
RATE[(5,4)14^1//5]
```

This would make a loop variable of 5 categories that would be summed from columns 14, 18, 22, 26 and 30.

Syntax:

```
VAR= varname:[$T="text"(#loops,incr)dataloc mod type&
categories]
```

dataloc

Can be any of the following:

col.wid or col1-col2

The column starting location and number of columns used, or starting column-ending column. Can have multiple column locations, but all must be the same length. Use the ellipsis to continue adding columns (i.e., [1.2,3,...,9]) would reference columns 1.2, 3.2, 5.2, 7.2, and 9.2.

label

The name of a pre-defined variable, either from a prior VARIABLE= definition or from a converted ~PREPARE variable. You can get a subset of the labeled item by using (col.wid) after the name; i.e., to get the month out of a date variable, you could use [date(3.2)]; this gets the third position in the date variable for a length of 2.

mod

An optional variable modifier; use them to determine how to combine categories in variables referencing multiple columns, or to modify numeric data references. Use *F and *L for multiple column references and use *P for numeric references.

***F**

Nets counts per category across columns (e.g., counts only the first mention).

Example:

```
CHECK [19.2,...,25*ZF#1//45/(-)99])
```

- duplicate responses are an error
- leading or embedded blanks are an error

***L**

Sums counts per category across columns (e.g., sum all mentions). Use it for cleaning.

- duplicate responses are an error
- leading or embedded blanks are an error

***P**

Makes separate categories for all locations (default). You can use this for cleaning data.

Example:

```
CHECK [2/12*P=3^1//5(-)Y])
```

*P=n Maximum number of responses allowed (1-255) for punch variables. *P=1 is the default. Minimum number of columns that cannot be blank (can be 1-127) for string variables (usually collected in Survent with a VAR type question), for example:

```
[10.15*P=5$].
```

These are the modifiers you can use in combination with *P:

! Returns a zero (0) when the location is blank.

***D#** Says the variable contains decimals.

***F#** Says this variable has implied decimals.

***RANGES=#-#,#,a,b,c**

Specifies the numeric range, exception number, and exception codes of the variable. Exception codes can be assigned a value to be used in statistical calculations, e.g., a=99. *RANGES can be abbreviated to *R=.

Syntax:

```
*RANGES= #-#, #, a, b, c
```

Options:

#-# Minimum to maximum value included in evaluation (REQUIRED)

Numeric exception value outside the range, e.g., `[6*R=1-5,9]` where 1 is the minimum value, 5 is the maximum, and 9 is the additional valid number.

a Exception code1 to be excluded from evaluation, but OK (i.e., it will not produce a cleaning error)

b Exception code2 to be excluded from evaluation, but OK (i.e., it will not produce a cleaning error)

c Exception code3 to be excluded from evaluation, but OK (will not give a cleaning

error)

Ranges Example:

```
AXIS= $[MEAN,STD] [5.2,6,...,12*RANGES=1-5,,DK=99]
```

This would exclude 6, 7, 8, 9 and 0 codes from the mean and standard deviation evaluation of the columns referenced, but it will include the exception code DK recoded as 99 for calculation of the statistics.

*RANGES is useful when doing statistical testing for tables (see AXIS=).

***S**

Keeps only one punch in a multi-punch field. You specify whichpunch to retain by the order of the categories in your statement.

Example:

```
MODIFY [10^6/5/4/3/2/1] = [9*S^6/5/4/3/2/1]
```

In this example, if the field contains the punches 6, 4 and 2, only the 6 will be retained. If the field contains punches 4 and 2, only the 4 will be retained.

***Z**

Fills the field with leading zeros (0).

Combining Variables

Where it makes sense you can specify more than one variable modifier, but use only one asterisk (*) in front of the first modifier listed.

Example:

```
CHECK  [5.3,...,15*ZF#1//17/(-)99]
```

Says this multi-location variable must be zero-filed (*Z) and must not contain duplicates, leading or embedded blanks (*F).

Example:

`[5.4*ZD2]`

Indicates that this variable contains leading zeros and two decimals.

type

The type of variable. These include:

None	If no type is specified, the default is numeric. Data must be a right-justified number in the field to be valid.
#	Numeric ranges and ASCII code combination categories
^	Punch codes categories
$	ASCII string; strips trailing blanks. (default)
$L	ASCII "long" string; trailing blanks are not stripped.
$P	Column binary (punch) format.
$T	Text pointer (for compressed TEXT variables collected in Survent.
$R	Region (columns by rows) of a table. See the ~CLEANER MODIFY <table> command for more information.

Categories

The codes used to describe categories within a datavar. Categories will be created for each data location specified, according to the variable modifiers above. Categories can include single elements, ranges (#-#) or multiple data elements separated by commas (,).

Category types

- Represents ASCII characters in a field; they can be numbers, numeric ranges (10-23), letters (AB), ASCII ranges (a-z), or special character strings ("*!"). You can combine any of these elements into each category. Maximum width of 20 for numbers (with 14 digits of significance) and 8 for alpha.

^ - Represents binary punch data. Use the relative number of the punch from the first punch of the first column. There are 12 punches per column, 1-9, 0, X, and Y; so the 3 punch of the second column is referenced with 15. You can use N to mean "not these punches", B for "blank" (no punches), or A to mean "has all these punches". Punch references may be grouped with commas or dashes to indicate ranges (1-10,23,25). You may reference the 10th through 12th punches as 0, X, and Y, or 10, 11, and 12. There is no maximum width.

Multiple categories for table definitions or functions are separated by a slash (/). Two slashes says continue making categories from the last category seen if the two categories are single values; i.e., 1//5 is valid, but 1-3//9 is not.

Statistics Categories

Control when statistical (specified with ~DEFINE STATISTICS=) tests are done on a particular category with a plus sign (+) or the option STATS before the category. This can come before or after any category text, and must be enclosed in parentheses ( ).

Example:

```
RATING1: [1.5^TOP:(+)1,2/1//6/BOTTOM:(+)5,6]
```

In this example, statistical tests will be done on the "top box" row (netted categories one and two) and the "bottom box" row (netted categories five and six) only.

You can also specify this option after a double slash (//), but before the category definition (e.g., [5^1//(+)5]). If you specify the option before a double slash then all the categories coming after it are affected (e.g., [5^(+)1//5]), not just the first category.

Exclusive Categories

For data cleaning purposes you can mark a category as exclusive with a minus sign (-) or the option EXCLUSIVE. Use this for data that would otherwise contain multiple responses (punch or ASCII). This can come before or after any category text, and must be enclosed in parentheses ( ).

Example:

```
[CHECK [2/12*P^1//5(-)/Y]
```

In this example, the data in record two, column 12 can contain one or more of punches one through five, but if a Y punch (12) is found then only it should be present.

Each exclusive category must be indicated separately with a minus sign.

Example:

```
CHECK [5^(-)1/(-)2/(-)3/(-)4/(-)5]
```

This treats 1,2,3,4, and 5 as exclusive categories. You can put a minus sign before a double slash, for example [5^(-)1//5], but this only makes the first category, 1, exclusive.

Example:

```
[CHECK [10.2,...,20*F#1//17/(-)96/(-)97/(-)98/(-)99]
```

In this example, responses 96, 97, 98, and 99 are considered exclusive of all other allowed responses.

ASSIGNING TEXT TO CATEGORIES

Assign text to a category by specifying it before the category followed by a colon. The category text will print whenever the category is printed on a table or by a procedure. For instance, if our example variable were assigned as the table column, then the text would print above its respective column category to create a simple banner.

Example:

```
[2/12^MALE:1/FEMALE:2]
```

Text must be enclosed in quotes if it contains spaces or special characters.

Example:

```
[2/12^MALE:1/"ALL FEMALES":2]
```

DEFINE VARIABLE, MISC.

EXPRESSIONS AND JOINERS

Complex variables or expressions require a colon immediately after the varname, and include logical and/ vector joiners, and mathematical operators and functions. Parentheses are used to separate parts of complex expressions.

LOGICAL JOINERS

AND	Both sides of joiner must be true
OR	Either side is true
NOT	Reverse the truth of the following statement

VECTOR JOINERS

WITH	Extends the list of categories
BY	Crosses each category
OTHERWISE	Uses the first category sets, if present; otherwise, the second category set
NET	Nets the corresponding categories
INTERSECT	If categories in both groups are true, combined category is true
WHEN	Like BY, but only one category returned from the right side of the item

STRING JOINERS

JOIN	Appends either two text or string variables

MATH JOINERS

RELATIONAL MATH JOINERS

EQ or =	Equal
NE or <>	Not equal
LT or <	Less than
GT or >	Greater than
LE or <=	Less than or equal to
GE or >=	Greater than or equal to

MATH OPERATORS

*	Multiply
/	Divide
+	Add
-	Subtract
%	Percentage
++	Add even if some element is missing
*=,/=,+=,-=,%=	Perform operation to item on left of =

Example:

```
a: (b WITH c) BY ([5^N1] AND d)
```

JOINERS DEFINED

Logical Joiners

These take expressions and first convert them from multi-category to single category, then combine them to check whether the item is true. The result of a

logical expression is always either TRUE or FALSE. Use parentheses to separate disparate parts of logical expressions.

Example:

```
IF (a OR b) AND NOT(c OR d)
SAY "This is a logic test"
ENDIF
```

AND - Combines two expressions and is true if some category of both of its components are true, false if either of its components are false.

OR - Combines expressions and is true if any of its components are true, false if and only if all of its components are false.

NOT - Returns a category with the opposite truth value of the expression stated.

Vector Joiners

These combine expressions to alter their meaning or form other expressions.

BY - Creates categories for every combination of its component expressions. Crosses all categories on the left of the BY with the first category on the right, then all categories on the left with the second category on the right, and so on until all combinations have been created.

Example:

```
[5#a/b/c] BY [6#x/y/z]
```

This make the combinations ax/bx/cx/ay/by/cy/az/bz/cz.

INTERSECT

Combines the categories of the vector on the left and the vector on the right. If both categories are true, the category is set to true. It must have the same number of categories on each side of the option.

Example:

```
[1^1/2] INTERSECT [3^4/5]
```

If column 1 has a 1 punch and 3 has a 4 punch, the first category is true. Likewise, for the second category if column 1 has a 2 punch and column 3 has a 5 punch.

NET

Combines the categories of the vector on the left and the vector on the right. If either category is true, category is set to true. Must have the same number of categories on each side of the option.

Example:

```
[1^1/2] NET [3^4/5]
```

If column 1 has a 1 punch or 3 has a 4 punch, the first category is true. The same is applies for the second category, if column 1 has a 2 punch or column 3 has a 5 punch.

OTHERWISE

Checks for some response in the first expression specified. If yes, responses from the first expression are used. If no, categories in A are true, then the next expression specified (B) is used.

Example:

```
[5^1//5] OTHERWISE [6^3//7]
```

WHEN

Works like BY, but categories on the right are combined to make one condition.

WITH

Connects two or more expressions to form one expression with all categories. Used most often to continue banner specifications.

Example:

```
sex WITH age WITH [10#1/2/3]
```

STRING JOINERS

JOINJoins either two text ($T) or string ($ or $L) type variables.

Example:

```
TOTNAME: FIRST_NAME$ JOIN LAST_NAME$
```

This example combines the string variables FIRST_NAME and LAST_NAME to form TOTNAME.

MATHEMATICAL JOINERS AND OPERATORS

Relational Joiners

Expressions can be formed by defining a comparison of variables and/or numbers using:

LT or < less than

LE or <=less than or equal to

GT or > greater than

GE , >=greater than or equal to

EQ, = equal

NE or <>not equal

Here are some useful examples for logical comparisons to check for subsets among categorical variables:

var1 > var2

Every category in var2 is must be in var1 but var1 may have some category that var2 doesn't; can also use LT (<) to reverse the comparison.

var1 = var2

Var1 and var2 have exactly the same categories, including possibly none at all.

var1 <> var2

Each has some category the other doesn't.

OPERATORS

The operators +, -, /, *, and % are used to define the addition, subtraction, division, multiplication, and percentage of variables and/or constants. There is also a ++ operator, which does addition but also treats missing elements as 0s, unless all elements are missing:

Example: `A ++ B`

This is A + B if both are present, A if B is missing, B if A is missing, and missing if both A and B are missing.

For vectors, ++ also works on a category-by-category basis (like NET).

An equal sign (=) after the operator means to do the operation directly to the element on the left of the equation:

Examples: expression result

a += b a + b

a /= b a / b

a -= b a - b

a *= ba * b

a %= b percent a is of b

You can have single categories work on vectors; each category of the vector will be affected by the single category item; vector ++ 0 will fill zero into empty categories.

The * is often used as a vector multiplier. A * B will tell you if there are any crossed categories, like A INTERSECT B.

Example:

```
IF [33^1//5] * [34^1//5]  SAY "share a category" ENDIF
```

This determines whether there are shared codes in columns 33 and 34, codes 1 - 5. If there are, it prints the message.

FUNCTIONS DEFINED

Functions can be used to modify the meaning of variables and expressions. These are used for mathematical computation, table enhancement, or in the building of tables in procedures. Here is a short description of all functions in alphabetical order. See *"9.3.2 Functions"* and *~CLEANER MODIFY <table>* for detailed information.

Syntax:

```
function(argument1,argument2,argumentn)
```

The function name must be immediately preceding the opening parenthesis. A comma is required between arguments, and a closing parenthesis must follow the arguments.

ABSOLUTE_VALUE(vector) (ABS)

Returns the positive values of the numbers in the vector.

AVERAGE(vector1, vector2,...) (AVG)

Returns the average of the numbers present.

BALANCE(vector) (BAL)

Returns the vector, followed the NO ANSWER category for the vector.

CASCADE(vector)

Returns TRUE if categories are all true starting at the first category until no more are true.

CATEGORY_FUNCTION(-#,vector) (CFUNC)

Returns the vector with different combinations of Total, No Answer, and Any Response before and/or after the vector.

-32	=	T	before
-16	=	NA	before
-8	=	NET	before
-4	=	T	after
-2	=	NA	after
-1	=	NET	after

Add together elements to get combinations. If you want T and NET before, and NA after, use -32 -8 -2 = -42

COMPLETE(vector)

Returns TRUE if all categories are true.

DOWNSHIFT(datavar) (DOWN)

Changes all alpha characters in the specified position to lower case. See also *Mentor Volume I: Chapter 3: Changing Case.*

EXPONENT(vector) (EXP)
Returns the exponent of the numbers in the vector.

FILE_COMPARE("file1", "file2",STOP_AFTER n errors, filter options, # lines until resynch) (FILECOMP)

Compares two files believed to be identical. File 1 is the master file and file2 is the compare file.

FIND_STRING(string_variable1, string_variable2)

Reports the number of times string_variable1 appears in string_variable2. This is a string search, text can be inside other words. The search is not case sensitive. For example:

Example:`FIND_STRING("IF","ENDIF")`

will return a 1. You can use variables in your search. FIND_STRING is like MATCHES(), but MATCHES only find exact matches, so `MATCHES("IF","ENDIF")` will return a 0. Typically, you would use FIND_STRING to find text in a text location, for example:

Example:`IF FIND_STRING("COLA",[1.10$]) > 0`
THEN ...

This would match "cola", "cocacola" and "Pensicola".

FIRST_SUBSCRIPT(cat vector) (FIRSTSUB)
Returns the subscript (number) of the first category seen in the vector.

FIRST_VALUE(vector) (FIRSTVAL)
Returns the first numeric value present in the vector.

FLIP (table region)
Flips the columns and rows in a region of a table.

FSIG(df1,df2,f)

Returns the level of significance for a one-tailed test base on df1, the degrees of freedom in the numerator, and df2, the degrees of freedom in the denominator, and f, the f-ratio. See TSIG.

JOIN_COLUMNS(table region 1, table region 2) (JOINCOL)
Extends a table by columns. It is used to set two tables or table regions side-by-side on a page; i.e., first wave vs. second wave numbers.

JOIN_ROWS(table region 1, table region 2) (JOINROW)
Extends tables by rows. It is used to set two tables or table regions above the other on a page; i.e., to combine two product lists.

LAST_SUBSCRIPT(cat vector) (LASTSUB)
Like FIRST_SUBSCRIPT, except it returns the subscript (number) of the last category seen in the vector.

LAST_VALUE(vector) (LASTVAL)
Like FIRST_VALUE, except it returns the last value present in the vector.

LOADED(table region) (LOAD)
Returns TRUE or FALSE depending on whether the table region is currently loaded in core.

LOGARITHM(vector) (LOG)
Returns the natural logs (e sub n) of the numbers in the vector.

MAKE_BOOLEAN(function) (MKBOOL)
Treats the result of another function as boolean.

MAKE_NUMBER(function) (MKN)
Treats the result of another function as a number.

MAXIMUM_VALUE(vector1,vector2,...) (MAX)
Returns the highest number present in the vectors.

MAXIMUM_VALUE_SUBSCRIPT(vector1, vector2, …) (MAXSUB)
Returns the subscript of the item with the highest value in the vector.

MINIMUM_VALUE(vector1,vector2,...) (MIN)
Returns the smallest number present in the list.

MINIMUM_VALUE_SUBSCRIPT(vector1,vector2,…) (MINSUB)
Returns the subscript of the item with the lowest number seen in the vector.
NET(vector)
Returns the NET of the vector followed by the vector.

NUMBERS_FROM_TABLE(#,table) (NUMFRTAB)
Make a vector with # numbers and load the numbers from the table there.

NUMBER_FUNCTION(1,vector) (NFUNC)
Returns the subscript of the category found. Used to assign ordered values. If more than one value found, returns MISSING.

NUMBER_OF_COLUMNS(table region) (NUMCOL)
Counts the number of columns in a region. It is used to calculate the number of columns to use in future tables.

NUMBER_OF_ITEMS(vector1,vector2,...) (NUMITEMS)
Returns the total number of categories present. If there is a data location ([col.wid]), returns the number of punches.

NUMBER_OF_ROWS(table region) (NUMROW)
Counts the number of rows in a region. It is used to calculate the number of rows to use in future tables.

RANDOM_CATEGORY(cat vector) (RANDOMCAT)
Picks a random category that is true from a categorical data variable. See also the RANDOM system variable.

RANDOM_SEQUENCE(vector variable(,seed)) (RANDOMSEQ)
Used to obtain specific random values, it generates two category vectors: the next random number and the resulting seed for the next call to the program for a random number.

RANK_TABLE_COLUMNS(high/low,low_ties/median_ties,<region>) (RANKTAB-COL)
Reads a region of a table and returns the rank value of each item in a column.

REPLICATE(table region, # of col reps, # of row reps) (REP)
This is used to have a smaller number of columns or rows act on a larger number of columns or rows (must be evenly divisible).

SELECT_VALUE(vector,<vector or VALUES(#,...,#)>) (SELECT)
Returns a number which is the number in the second vector or values list corresponding to the category seen in the first vector.

SQUARE_ROOT(vector1) (SQRT)
Returns the square roots of the numbers in the vector.

STRING_FROM_NUMBER(num, wid, dec)

Converts a numeric argument into a string. Num is the numeric argument, wid is the string width, and dec is the number of decimals. You can use a negative number for wid to zero fill the string. This function is useful in combination with the PUTID command to assign case IDs. Num, wid and dec can be either numbers or variables.

STRING_FROM_NUMBER can use the system constant CASE_NUMBER for the numeric argument, for example:

Example: `PUTID STRING_FROM_NUM(CASE_NUMBER, -4, 0)`

STRING_LENGTH(<"string",$,$T, or $P string variable>) (STRLEN)
Returns the number of characters in the string variable, starting at the first position and going to the last non-blank character.

STRIP(string variable)
Strips leading and/or trailing blanks from a string variable or data location containing string information.

SUBSCRIPT(vector)
Returns the subscript (position of the category) of the category found.

SUBSTITUTE(string variable)
Allows you to substitute one string of characters for another in a string variable or data location containing string information.

SUM(vector1, vector2,...)
Returns the sum of the numbers and categories present.

TABLE_FROM_NUMBERS(vector, # columns, #rows) (TABFRNUM)
Takes data from the vector and fills a region of a table.

TSIG(df,t)

Returns the level of significance for a two-tailed test base on df, the degrees of freedom and t, the calculated t-value. TSIG(df,t) = FSIG(1,df,t*t). See FSIG.

UPSHIFT(datavar) (UP)

Changes all alpha character in the specified position to upper case. See also *Mentor Volume I: Chapter 3: Reformatting Your Data*

VARIABLE_EXISTS(string variable) (EXIST)
Evaluates the string variable as a variable name and checks for its existence in any open data base (DB) files.

VECTOR_FUNCTION(#,vector) (VFUNC)
Returns the first or last item present in the vector.

4 Returns the first item
5 Returns the last item

WORD_MATCHES(string variable,string variable) (MATCH)
Returns the number of ASCII character matches in the string variable to the string variable.

WORD_STARTS(vector,"string" variable) (START)
Returns the number of times the second string starts a word in the first string. Non case-sensitive search.

X(vector or math equation)
Returns a 0 for all items that are blank (so can get math done when some items are missing).

VARIABLE OFFSETS

Data offsets allow you to reference data locations relative to an initial variable data location. This would most likely be used in a procedure where you need to reference multiple locations in some iterative way. The offset can be changed by updating the offset variable and a new location will be referenced.

Syntax: `[OFFSET=varname VARIABLE=varname col.wid]`

OFFSET=varname
Specifies any variable or expression which results in a number. This says the columns used will be determined by adding the number to the original source columns of the variable.

VARIABLE=varname
Allows you to apply an offset to an existing variable. Since offsets are general variables they may in turn contain variables which are offset,and so on.

Example:

item[12.3] wid[15.3]

kick: wid * (item - 1)

offa[OFFSET=kick 21.2] offb[OFFSET=kick 23.3]

offc[OFFSET=; VARIABLE=offa]

offd[OFFSET=kick VARIABLE=offa]

x[offa] "will carry the OFFSET specified on offa

xx[offa,offa]"will carry the OFFSET specified on offa

DATA ASKING VARIABLES

This feature is usually used with a ~GO_TO or ~CLEANER GOTO command to control execution of a batch job or procedure. Data asking variables are filled by requesting input from the console. The text of the variable is displayed, and depending on the variable type, the response is limited to valid answers. Wherever the variable appears in a procedure, the program will stop and prompt for the data.

Data asking variables are ignored if the program is not being run interactively, i.e., Mentor &specfile.

Syntax:`name[$T="text to prompt with -->" $B/$N/$S=ASK]`

Options:

$B	is for boolean; expects Yes, No, Y, or N as a response.
$N	is numeric; expects a numeric response (1, 23, etc.)
$S	is ASCII; expects any ASCII response (Recode this case?, etc.)

The following example illustrates a typical data recoding procedure.

Example:

```
~DEFINE
Continue[$T="Recode this case? (Yes/No)   "$B=ASK]
Endit[$T="Terminate the session? (Yes/No) "$B=ASK]
PROCEDURE={Intvcase:
PRINT_LINES "\NAbout to RECODE Case id \S " CASE_ID
GOTO (*;NO) Continue
INTERVIEW
YES_UPDATE
GOTO OUT
NO:   PRINT_LINES "\nDid not recode Case id \S" CASE_ID
GOTO (*;OUT) Endit
TERMINATE "\N... End of automatic RECODE on datafile"
OUT:  NULL }
~EXECUTE PROCEDURE=Intvcase
```

Related Commands: TERMINAL_SAY

Illegal Variable Names

System constants and the function NOT( ) are illegal variable names. Both the full option and its abbreviation are disallowed. You will get a spec error indicating that these are reserved words.

RESERVED WORDS

ALTER_FLAG	ALTFLAG
CASE_ID	CASEID
CASE_NUMBER	CASENUMBER
CASE_WRITTEN	CASEWRITTEN
CATEGORIES	CATS
DATE_TIME	DATETIME
DELETE_FLAG	DELFLAG
DUD	
EOF_DATA	EOFDATA
ERROR_FLAG	ERRFLAG
FALSE	
FIRST_CASE	FIRSTCASE
JULIAN_DATE	JULIANDATE
LAST_CASE	
LINE_NUMBER	LNNUM
MATH_VALUES	MATHVALS

RESERVED WORDS

MISSING	
NOT(	
PAGE_NUMBER	PGNUM
RANDOM_VALUE	RANDOMVAL
TABLE_NAME	TABNAME
TOTAL	
TRUE	
VALUES	VALS

~DEF AXIS=

AXIS= is like VARIABLE= except that cross-case functions are allowed. You can use cross-case functions to calculate statistical computations or make special types of tables. Use an axis to define tables; it cannot be used in a procedure. Like VARIABLE=, you do not need to include AXIS=.

Syntax:

```
$[options] expression $[options] expression $[]
expression
```

$[options]

The $[options] are listed below; separate multiple items, with commas or spaces. Items will print in the order specified. Use <-option> to turn specific elements off.

$[ ]

Turns all options off and end the cross-case function block; this returns you to regular variable definition mode.

Example:

```
[5^1//3] $[MEAN,STD] [5] $[ ] [6^1//3] $[MEAN,STD] [6]
```

This expression returns three categories for column five, computes the mean and standard deviation for column five, then turns them off, returns three categories for column six, and finally computes the mean and standard deviation for column six.

$Options:

$[BASE]

Returns two values for each category specified, its count and its Effective Sample Size and treats it as the statistical base for succeeding rows. It produces two lines, the weighted N and the effective N. There should be only one category defined after a $[BASE], then the following $[ ] element(s) should be defined.

- When the table is unweighted, weighted N = effective N.
- The weighted N is typically used for the percent base on the rows that follow.

It tells you everyone that is in the test (i.e., it gives you the denominator for the statistics for that column). It does not filter the data that follows.

You would use the $[BASE] when the Total or Any Response rows are not satisfactory bases. For Example:

- if you wanted to show the total row on the table and you wanted the statistics to be calculated on a sub-sample of the total.
- if you wanted to have the base change from row to row or block to block.

- if you wanted to show the effective n (statistical base) on a weighted table. You do not need a $[BASE] for means; the mean calculation does not use the $[BASE] to determine the denominator for the statistics.

Multiple bases on one table are allowed.

Example:

$[BASE] [5^1] $[ ] [6^1//5] $[BASE] [7^1] $[ ] [8^1//5]

Statistical bases stay in effect for all categories except $[MEAN]'s until another $[BASE].

$[BREAK] (BRK)
"Breaks" the expression so that multiple columns or rows are created by this one expression.

$[BREAK=#]
Like $[BREAK], but will break the expression every # categories, so only needs to be specified once.

$[BREAK_CONTROL=name] (BRKCTRL)
Filters cases from BREAK columns or rows by the variable used. The variable is interrogated before doing the break and calculating. This cuts down run time and sets the proper total for the column or row; otherwise the total will be the same for each of the breaks.

Example:

```
var: $[BRKCTRL=wave] qq8_pre $[BRK] qq8_post
```

The variable WAVE must have as many categories as the number of breaks. It can be redefined as WAVE(1/2) if other categories exist. If the first category of WAVE is true, then qq8_PRE is calculated; the same rule applies to qq8_POST depending on the second category of WAVE.

$[EFFECTIVE_N] (EFFN)

Returns the effective N for T-tests defined with STATISTICS=. This will be the same as the base for non-weighted tables, and a smaller number for weighted tables. Used to calculate the T statistic.

$[FILTER]

Defines which cases will be allowed into the next part of the table. Unlike "By" Or "When", the filter stays in effect across other axis elements, such as $[mean]. The filter will stay in effect until another $[Filter] description is seen, or until you specify $[-Filter].

Example:

Row=: TOTAL $[Filter] [5#1] $[] [6#1//5] $[Filter] [5#2] $[ ] [6#1//5]

This would create a table with a total and two rating scale listings each with a different Filter.

NOTE: $[Filter] does not make a category, it just limits the cases that get into the following categories.

$[FILTER_BASE]

A combination of the $[BASE] and $[FILTER] commands, which creates a set of "Base" rows and at the same time applies a filter to the succeeding rows. See $[BASE] for a description of the base row(s) created. The filter stays in effect across $ [ ]axes until another $[Filter_base] or $[Filter] is seen.

The reason for this feature is that the $[BASE] only provides a statistical base and does not apply a "filter" to the data, whereas FILTER_BASE will do both.

Example:

```
$[FILTER_BASE] [72^1.3] $[] [5^0/1/2/3/4] $[MEAN,STD] [5]
```

$[DO_STATISTICS] (STAT)
Turn on STATISTICS= statistical testing for this part of the table, the default. Turn off with -STAT.

$[FREQUENCY] (FREQ)
Returns the frequency counts for the expression; usually used in addition to the SUM, MEAN, or PERCENTILE to print the number of responses included in them.

$[INTERPOLATED_MEDIAN]
Produces a median value that is the mid-point between the value between the true median and the next value beyond it. Thus, if your data contained 1, 3, 5,7, the interpolated median would be 4. (See *Mentor, Volume I: 6.2.2 MEANS FOR RANGE TYPE VARIABLES*.)

Syntax:

```
$[INTERPOLATED_MEDIAN(#)]
```

Use # to indicate the size of the array used to calculate INTERPOLATED_MEDIAN. If Mentor cannot calculate the median, you may need to increase the number over the default of 50. (See *Mentor, Volume I: 6.2.5 Lost Medians*.)

$[MAXIMUM] (MAX)
Returns the maximum value found in all of the cases read.

Example: `$[MAX] [10.5]`

$[MEAN]
Calculates the mean of the following variables. You may get means on as many numeric variables as you like by joining them with the WITH joiner, or referencing them within the [ ]s of a data variable.

You can use #-[datavar] to reverse the values before doing the mean; use a number one higher than the highest number (i.e., for a 3 point rating scale, use 3+1=4).

Example:

```
$[MEAN]  4 - [11]
```

This example reverses the values. 1 becomes 3 (4-1), 2 remains 2 (4-2), 3 becomes 1 (4-3).

You can write one statement to get means across many columns and exclude values from the calculation for all of the columns. You can also compute an overall mean category for same column set.

Example:

$[MEAN] [1/23,...,1/35*R=1-5] WITH [(13,1)1/23*R=1-5]

This statement first computes the means of columns 23 through 35, using only the values 1-5 (6-9 and 0 are excluded), then an overall mean category of the same columns using the loop variable structure. This example computes an overall mean starting at column 23 for 13 iterations with a gap of one column between iterations (i.e., column 23 gets incremented by one for each iteration of the loop up to column 35). *R= is the abbreviation for *RANGES=.

$[MEAN_FREQUENCY] (MFREQ)
Calculates the mean and frequency, creating two rows or columns. The first is the frequency and the second is the mean.

$[MEDIAN] (MED)
Returns the median value found in all of the data cases. This is the same as saying $[PTILE=.50].

Syntax: `$[MED(#)]`

Use # to indicate the number of buckets the program uses to hold numbers while calculating the median. To speed processing you can reduce this number if you know you have fewer than 25 values.

$[MINIMUM] (MIN)

Returns the minimum value found in all of the cases read.

Syntax: `$[MIN] [loc.len]`

Example: `$[MIN(50)] [10.5]`

$[NET_OVERLAY] (NOLAY)
Takes two expressions (of the same size) and nets together the cells of their categories. You can have as many NET_OVERLAYS as you like, but all expressions must match for size and type. You may have $[MEAN,STD,SE] in the overlay.

$[OVERLAY] (OLAY)
Takes two expressions (of the same size) and SUMS the cells of their categories. You can have as many OVERLAYs as you like, but all expressions must match for size and type. You may have $[MEAN,STD,SE] in the overlay.

$[PERCENTILE=#]

Returns the specified percentile. $[PTILE=.5] returns the median. You can do large variables by increasing the bucket size from 25, or speed up processing by reducing the buckets. You can get as many $[PTILE=#] items on a table as you like.

Syntax:

```
$[PTILE=x(y)]
```

x

The percentile (a number from .01 to .99)

y

The number of buckets. The default is 25.

Example:

```
abc: $[PTILE=.50] [12.3]
```

This generates a median on the numeric data in location 12 - 14.

Under some circumstances a message may print saying that buckets have overflowed when computing the $[PTILE=] on a large number of cases. To fix this, increase the number of buckets (i.e., to 50-100).

$[RAW_COUNT]

Returns the unweighted value(s) of the following variables. This is especially useful on weighted tables.

$[SE]

Returns the standard error of sample from the mean of the variable(s).

$[STD]

Returns the standard deviation of the sample from the mean of the variable(s).

$[SUM]

Returns the sum of each variable.

$[VARIANCE] (VAR)
Returns the variance of the items specified.

$[WEIGHT] (WT)
Prints the weighted total of the variables specified.

~DEF BANNER=

This defines the text which prints above the columns of a table. The text starts directly above the first data column, and after the row stub width (see ~DEFINE EDIT=). Banners should be lined up with the columns by specifying the COLUMN_WIDTH on the EDIT statement; if the widths change from column to column, use the EDIT=COLUMN_INFO option.

Syntax:

```
BAN={varname: text to appear on first line & and text
for second and more text for second line}
```

{ - Starts the banner. (optional)

& - If at the end of a line, means print what follows on the next line on the same line as this one when the table is printed.

} - Ends the banner.

| - Says the text starts here. This is how you can include blanks at the beginning of the text lines. If you omit the vertical bar on any line, printable text begins at the first nonblank character.

\k(p) - Prints to the “list file” or to the “print file” as specified by the “>print_file<name>” command.

\k(d) - Prints to a .dlm file when ~specfile <filename>and ~set delimited_tables=() are specified. This file is intended to be imported to a spreadsheet or other programs.

\k(h) - Prints to an .htm file when ~specfile <filename>and ~set web_tables=() are specified. This file is intended to be read by a browser.

Example: `BAN= BANNER1:`

```
| heading1      heading2  ...   headingN
|   more1        more2    ...    moreN
| --------      --------        --------}
```

At the front of a line on the banner, you can use a caret (^) to print text to the left of the banner (above the row stubs). In this case, do use the caret in preference to the vertical bar, and print text to the left:

Example: `BAN= BANNER1:`

```
this will print ^ heading1  heading2 ...   headingN
over the stubs  ^ more1     more2    ...    moreN
              |  --------    --------       --------}
```

If no banner is specified, a DEFAULT banner is created. If ~SPECFILE <filename> and ~SET MAKE_BANGEN_SPECS are specified, the banner will include the text for delimited files and web tables (browser-readable).

~DEF EDIT=

Defines options for printing tables, such as page size, column and row width, and what to print in the table cells. The default format includes the following:

- System Total and NA columns and rows are printed
- Column width default is 8 spaces
- Stub width default is 20 spaces
- Frequencies print with no decimal significance
- Vertical percentages print with one decimal of significance

- Title text contains the words BANNER:, STUB:, BASE:, WEIGHT:, and FILTER: preceding respective descriptions
- Page length default is 60 lines
- Page width default is 132 columns

Many options can be returned to their default value or turned off by using a minus sign before the option; i.e., -VPER would discontinue printing vertical percents.

Syntax:

```
EDIT={varname: options }
```

{ - Starts the edit (optional)

} - Ends the edit

NOTE:

EDIT options affecting the printed page size in the currently open print file (i.e., TOP/BOTTOM_MARGIN, PAGE_WIDTH/LENGTH), are identical to the same options available under ~SET except that they only affect printed tables and no other printing. These options do not reset to a default even when a new EDIT statement is specified. The last value specified on a previous EDIT statement remains in effect until specifically replaced. Please refer to these options under the meta command >PRINT_FILE in your UTILITIES manual for more information.

There are three levels of the edit statement: GLOBAL_EDIT, EDIT, and LOCAL_EDIT.

See *Mentor Volume I: Chapter 4: Table Specific Printing Controls* for details.

Options:

ALL_POSSIBLE_PAIRS_TEST (ALLPAIRTEST)

When doing T-tests of columns or rows using the DO_STATISTICS option, this will cause tests of groups to use the all-possible-pairs method. This is the default for T-tests specified with DO_STATISTICS.

ANOVA=
Causes an analysis of variance test to be performed on the tables. You may specify the regions of the table to operate on, and you may do ANOVAs on multiple regions of the table.

Syntax:

```
ANOVA=varname or [$R col_list BY row_list]
```

Column_list BY Row_list is where the column_list and row_list can include ALL or any combination of TOTAL, NA, LAST, and numbers 1,2,etc. for the data rows or columns, separated by commas, an ellipsis (...) meaning continue the pattern of the last two items, or the option TO, which means by ones until the next column or row is specified.

You can build the variable with ~DEFINE VARIABLE=, and reference it by name only on the ANOVA= command. Use the PRINTER_STATISTICS_VALUES parameter to assign values to the rows for the ANOVA test; you can also exclude rows using PRINTER_STATISTICS_VALUES, in addition to the region described above. Refer to ~CLEANER MODIFY <table> for more information on defining the region.

ANOVA_SCAN
This performs an Anova-Scan (sometimes called Anova Screening) test on the designated groups of columns or rows; the default for T-tests specified with the EDIT= DO_STATISTICS option is the All Possible Pairs test. This is an independent test of significance used on *unweighted* data only.

BOTTOM_MARGIN=# (BOTM)
Specifies a margin of # lines at the bottom of each page.

BLANK_LINES_BEFORE_T5=#

Specifies the number of lines between a table's body and the title_5 text. (Using \N to create blanks lines will create unpredictable results.)

Example:

```
EDIT={:BLANK_LINES_BEFORE_T5=5  }
TITLE_5={: This is the text }
```

```
CALL_TABLE=     (CALLTAB)
```
Specifies a string up to 16 characters to replace the text "TABLE " before the table name on the printed table.

Syntax:

```
CALLTAB=" string "
```

Use the symbol > to right-justify the table name, e.g., CALLTAB=">abcd". Use CALLTAB=; to return CALL_TABLE to the default 'TABLE '. To suppress the table name, use -CALL_TABLE or CALL_TABLE=" ".

CHI_SQUARE=
Performs a chi square test on the table. You may specify the regions of the table to be tested, and you may do multiple chi square tests per table.

Syntax:

```
CHISQ= (options) A[$R col_list BY row_list]
```

Options:

Parentheses around options are required:

SIGNIFICANCE_LEVEL (SIGLEV)
to print the significance level on the table

CHECK_CELLS (CHKCELL)
to check for the minimum potential cell value, and not execute the test if it is < 5.

Column_list BY Row_list is where the column_list and row_list can include ALL or any combination of TOTAL, NA, LAST, and numbers 1, 2, ,etc. for the data rows or columns, separated by commas, an ellipsis (...) meaning continue the pattern of the last two items, or the option TO which means by 1's until the next column or row is specified.

You can build the variable with ~DEFINE VARIABLE=, and reference it by name only on the CHI_SQUARE= command. You can use the PRINTER_STATISTIC_VALUES option or the -COLUMN_STATISTICS_VALUES STUB= option to exclude rows from the CHI_SQUARE, in addition to the region described above.

- See Also:
  MARK_CHI_SQUARE

CHI_SQUARE_ANOVA_FORMAT (CHISQANOVAFORM)
Prints the chi square results directly under the appropriate columns.

Specify -CHI_SQUARE_ANOVA_FORMAT if you are doing both chi square and ANOVA tests on the same table.

CLEAR_SUBTOTALS (CLRSUBTOT)
Clears subtotals when printed.

COLUMN_INFO=
Controls the printing of table columns on a column by column basis. Used to adjust width of individual columns, suppress columns, print with differing number of decimals, etc.

Syntax:

```
EDIT=edit1: COLINFO=(a/b/c/d/e/f)
```

Options:

a - The options for first column (i.e., System Total)

b - The options for second column, etc. (i.e., System NA).

COLINFO affects specific columns printed on the table; if the System Total is printed, it is referenced as column 1, etc. A slash separates the column referenced, if no COLUMN= is specified the next column is used. Two slashes indicate that the next column will use default values specified on the EDIT statement.

Many of the options below have the same meaning as on the EDIT or STUB statements; if one does not, it is explained.

BLANK (B)
Blanks the cells of the column, but preserves the space on the page.

COLUMN=# (C)
Columns referenced must be so in ascending order. this option can be followed with =# to mean 'the same as' another column. If =# is used, no other options may be specified for that column.

Example:

```
COLINFO=(C=3 -VERTICAL_PERCENT/C=5=3)
```

In this example, vertical percentaging is suppressed on column three; C=5=3 means that the same options apply to column five. Column five cannot have any additional options unless they are specified on column three.

-COLUMN_STATISTICS_VALUES (-COLSTAT)
Used to exclude a column from statistical testing. Treats the column as if all were missing.

CUMULATIVE_PERCENT (CP)

FREQUENCY (FREQ)

FREQUENCY_DECIMALS= (FD)

HORIZONTAL_PERCENT (HP)

MINIMUM_BASE= (MINB)

NUMBER_FORMAT= (NUMFORM)

PERCENT_DECIMALS= (PD)

PERCENT_SIGN (PS)

PUT_CHARACTERS= (PUT)

STATISTICS_COLUMN (S)
Refer to STUB=[STATISTICS_ROW]

STATISTICS_DECIMALS= (SD)
Controls decimal significance.

VERTICAL_PERCENT (VP)

WIDTH=# (W)
Width to print for this column of numbers. WIDTH=0 will suppress the numbers in that column. This does not suppress the banner text above the column. WIDTH= is optional; specifying a number by itself sets that column to that width.

Columns for which there is no information will use default values. The default values are set at print time based on EDIT and LOCAL_EDIT values.

Example:

```
COLINFO= (4/0/C=5 W=12//15 S)
```

This example gives information for seven columns. Column one prints in a width of four, column two would not print, columns three and four would use default values, column five would have a width of 12, column six would use default values, and column seven would have a width of 15 and be a statistic column. If the table has less than seven columns, an error would occur at print time; if the table has more, then columns eight and on would use default values.

COLUMN_MEAN (MEAN)
Calculates the mean, using the values assigned for each of the rows in the COLUMN_STATISTICS_VALUES= VALUES(a,b,c,...) parameter. (VALUES() may include either positive or negative values). Statistics will print in the order specified, but COLUMN_MEAN must be specified before COLUMN_STD.

COLUMN_MEDIAN= (MED)
Causes a median to be calculated based on the values specified. Values indicate start points for each of the row ranges. A missing value causes that row to be ignored when calculating the median.

Syntax:

```
MED=VALUES(value1,value2,...,valuen)
```

The median is calculated by determining which row has the median, then calculating the amount into the rows values at which the median (50th percentile for the column) resides.

Unlike medians calculated with the AXIS= $[MEDIAN] option, the COLUMN_MEDIAN is a decimal number, meaning it is calculated from the values found in the printed table cells rather than an actual value stored in the data.

Example:

```
MED=VALUES(0,11,21,...,91,100)
```
This would be a typical set of values for a 100 point rating scale.

COLUMN_NA (COLNA)
Prints the System No Answer (NA) column.

COLUMN_SE (SE)
Calculates the standard deviation using the values specified on the EDIT=option COLUMN_STATISTICS_VALUES= VALUES(a,b,c,...) parameter.

COLUMN_SIGMA (SIG)

Prints the sigma at bottom of table. The sigma is the sum of the rows and the row percentages that are printed on the table. You can exclude rows from the sigma (see STUB=).

COLUMN_STATISTICS_VALUES= (COLSTAT)
Assigns the values to be used by the EDIT= printer statistics COLUMN_MEAN, COLUMN_STD, COLUMN_SE, COLUMN_VARIANCE, ANOVA, AND CHI_SQUARE.

Syntax:

```
COLSTAT=VALUES(a,b,c,...,n)
```

Values may be positive and/or negative.

The values may include an ellipsis (...) to continue with numbers with the same increment. Two commas in a row mean that the corresponding row will not be used when calculating the statistics requested.

If not specified, then VALUES defaults to VALUES(1,2,3,4,... to last row of table).

An example for COLUMN_STATISTICS_VALUES follows.

Example:

```
EDIT= {edit1:
MEAN,COLSTAT=VALUES(,5,...,1,,),STD,SE }
```

This would exclude the first row, assign the second row a value of 5, third row a value of 4, etc. and exclude the last and later rows.

COLUMN_STD (STD)
Calculates the standard column deviation using the values from the EDIT= option COLUMN_STATISTICS_VALUES if present, otherwise uses values 1 for the first row, 2 for second, etc. to the last numbered row of the table.

COLUMN_TNA (COLTNA)
Prints the System Total and No Answer (NA) columns.

COLUMN_VARIANCE (VAR)
Calculates the variance using the values specified on EDIT= COLUMN_STATISTICS_VALUES=VALUES(a,b,c,…).

COLUMN_WIDTH=# (CWID)
Specifies the width or number (#) of spaces for each column of data. The default is eight spaces.

CONTINUED_LOCATION= (CONT=)
Specifies where to put "(continued)" on tables that span more than one page. All of the TOP options print on the continued pages (page two to the last page), while all of the BOTTOM options print on the pages that are going to be continued (page one to the second to last page).

Syntax:

```
CONTINUED_LOCATION= option
```

Options:

TOP - Prints (continued) next to the table number in the top left corner. This is the default
TOP_CENTER (TOPC) - Prints (continued) centered at the top of the page.
TOP_RIGHT (TOPR) -Prints (continued) in the top right corner of the table.
BOTTOM (BOT) - Prints (continued) on the bottom left corner of the table.

BOTTOM_CENTER (BOTC) - Prints (continued) centered on the bottom of the page.

BOTTOM_RIGHT (BOTR) - Prints (continued) in the bottom right corner of the table.

NONE - Does not print (continued**).**

Example:

```
CONTINUED_LOCATION=TOPC
```

CONTINUED_NUMBER= (CONTNUM) - Adds a number suffix to the table name of tables that span more than one page. This can be used in combination with CONTINUED_LOCATION.

Syntax:

CONTINUED_NUMBER=option

Options:

NONE
Table name has no suffix, TABLE 001 (continued)

AFTER_CONTINUE
Suffix is after continued, in this format:
Table 001 (continued)-1
Table 001 (continued)-2, etc.
AFTER_TABLE_NAME
Adds suffix after table name, in this format:
Table 001 (continued)-1
Table 001 (continued)-2, etc.

Example:

```
CONTINUED=NONE,CONTINUED_NUMBER=AFTER_TABLE_NAME
```

This example would result in the following format:

```
Table 001 (continued)-1
Table 001 (continued)-2, etc.
```

CUMULATIVE_PERCENT (CPER)

Prints the cumulative vertical percent, that is, the sum of the vertical percents of the table up to that row, in the cells of the table. The base for cumulative percentaging is always the same as that for vertical percentaging. System rows are not included.

DATA_INDENT=# (DATAIND)
Specifies the number of spaces to indent the data columns.

DO_PRINTER_STATISTICS (PRTSTAT)
Causes Mentor to do statistical T-testing on all row stubs of the printed table. You must also specify EDIT=DO_STATISTICS_TESTS=<columns tested>, where the <columns tested> are letters A - Z for columns 1- 26. The program will mark a column that is significantly higher than another with the letter of the corresponding column.

Syntax:
```
EDIT=E:PRTSTAT DO_STATISTICS_TESTS=AB,CDE,FGH }
```

Using EDIT=NEWMAN_KEULS_TEST will cause the program to do Newman Keuls T-tests instead of using the default All Possible Pairs test. You may also modify which test is done for a particular stub by specifying [DO_STATISTICS=<stat test>] as a STUB= option for a particular row.

Printed table T-testing will be done on the printed mean if it is used, but cannot be done on a $[MEAN] defined on the variable. The KRUSKAL_WALLIS_TEST is only valid on the printed mean. It is assumed that all columns are independent when statistical T-testing of the printed table is done.

DO_STATISTICS
Instructs the program to do the T-test calculations and print letters marking significant differences when printing the table. Tables must have been created with ~EXECUTE STATISTICS=varname (see ~DEFINE STATISTICS=). Tests will use ALL_POSSIBLE_PAIRS_TEST, ANOVA_SCAN, FISHER, or the NEWMAN_ KEULS_TEST testing for groups as specified. DO_STATISTICS without a significance level will perform the ALL_POSSIBLE_PAIRS_TEST test at the 95% significance level.

Syntax:STAT=#

Options:.

80

significance level of .20, often called 80% level. This option ***cannot*** be used in conjunction with the Newman-Keuls testing

.90

significance level of .10, often called 90% level.

.95

for a significance level of .05, often called 95% level (default)..99

significance level of .01, often called 99% level.

APPROXIMATELY .nn (APPRX)

uses an approximation formula for the t-values rather than looking it up in a table. nn is any value. This option requires the results from the look-up table of values called STUDENT located in the CFMC CONTROL directory or group. This option cannot be used in conjunction with the Newman-Keuls testing.

You can test two different confidence levels by combining options with a plus sign (+). This will perform the same test on all columns, but at two different levels of significance. You can use the standard levels of .80, .90, .95, and .99, or specify any other level of significance. On the printed tables upper-case letters indicate the higher level of significance. You can perform bi-level testing only on letter- marked tests (ALL_POSSIBLE_PAIRS_TEST, ANOVA_SCAN, FISHER, NEWMAN_KEUHLS_TEST), not printer T-tests (STUB=DO_T_TEST=/DO_SIG_T=, or EDIT=CHI_SQUARE or ANOVA). *(see example)*

Example:

```
DOSTATS= APPRX .94 + .90 ALL_POSSIBLE_PAIRS_TEST
```

You can do bi-level testing both on statistics calculated during table building (~DEFINE STATISTICS=) and statistics calculated at print time (i.e., EDIT= DO_PRINTER_STATISTICS= DO_STATISTICS_TESTS=, or STUB= [DO_STATISTICS=]). At print time you can also specify the Kruskal Wallis test on a specific row.

DO_STATISTICS_TESTS= (STATTEST)
Specifies the columns to be tested for printed table statistic testing, meaning the program will use the values in the cells on the printed table. You cannot use this on a weighted table. This can be in addition to any ~DEFINE STATISTICS= item on the table. The letters A-Z are used to represent columns 1-26. This is used in conjunction with EDIT=DO_PRINTER_STATISTICS, which would cause all rows to be tested, or STUB= option [DO_STATISTICS=type] to test a particular row.

Example:

```
EDIT=Edt1: DO_PRINTER_STATISTICS STATTEST=ABC,DE,FGHI}
```

EMPTY_CELLS= (EMPTYCELL)
Specifies the character to print when there is an empty cell; i.e., on a row with percents only and 0% in the cell. The default is dash (-). You can also specify this on PUT_CHARACTERS. Use Z for zero or B for Blank.

EXTRA_ROWS_OK (XROWOK)
Says do not print an error message if there are more data rows than stub labels.

The default (-EXTRA_ROWS_OK) works as follows:

- An error message prints if the number of data rows on the table does not match the number of stub labels.
- If there are fewer stub labels than data rows, only as many rows as there are stub labels for will print.
- If there are more stub labels than data rows, all of the rows are printed but the extra labels are not.

With this option set:

- There is no change in the number of rows that are printed.
- If there are fewer stub labels than data rows, no (ERROR ####) message prints.
- If there are more stub labels than data rows, a (WARN ####) message prints.
- You cannot rank a table where EXTRA_ROWS_OK is used, and there are extra rows on the table.

EXTRA_STUBS_OK (XSTUBOK)

Reads through the end of the stub label set for possible extra stubs rather than until you run out of data rows. This option is useful if you want to print a table, do some table manipulation that creates extra rows and then print the modified table using the same stub labels. (Refer to *"9.2 TABLE MANIPULATION"* for an example)

The default (-EXTRA_STUBS_OK) works as follows:

- An error message prints if the number of data rows on the table does not match the number of stub labels.
- If there are fewer stub labels than data rows, only as many rows as there are stub labels for will print.
- If there are more stub labels than data rows, all of the rows are printed but the extra labels are not.

With this option set:

- If there are fewer stub labels than rows, an error message prints, and all of the rows are printed. Rows without labels are printed as "** no more labels at."
- If there are more stub labels than rows, NO error message is printed and all of the rows are printed, extra labels are not.
- You cannot rank a table where EXTRA_STUBS_OK is used, and there are extra stubs on the table.

FISHER

Does a Fisher test on the designated groups of columns or rows; the default for T-tests specified with the EDIT= DO_STATISTICS option is the All Possible Pairs test. This is an independent test of significance used on unweighted data only.

FLAG_MINIMUM_BASE_CELL (FLAGMINBASE)
Prints an asterisk (*) next to the values that are below the specified MINIMUM_BASE=# in statistical T- tests. The default for values that fall below the specified MINIMUM_BASE=# (when DO_STATISTICS is also specified) is to print an asterisk at the top of the column and cells that are blank.

This option may only be used if MINIMUM_BASE=# is also specified.

FREQUENCY (FREQ)
Print frequencies in the cells (default). Use -FREQ to not print frequencies.

FREQUENCY_DECIMALS=# (FDEC)
Specifies the number of decimal places to print for frequencies. The default is 0; may be a number from 0 to 7.

FREQUENCY_ONLY (FREQONLY)
Prints frequency only, no percentages.

HORIZONTAL_PERCENT
Says to print the horizontal percent, and optionally, specify the horizontal percent base. The default percentage base is the Total column.

Syntax: `HPER=value`

Options:

T for Total column

AR for Any Response column

for the number of a particular column

INDENT= (IND)

Causes the entire table to be indented. Options are CENTER or the number of spaces to indent.

INDEX
Prints the percent of a percent. The default percentage base is the Total column, so the index is the percentage of the Total column's percent that a table cell's percentage represents. For example, if the third stub of a table's total column contained 50%, and the third stub of the second banner point contained 25%, the index for that cell would be 50, since 25 is 50% of 50.

Syntax: INDEX=value

Options:

T for Total column

AR for Any Response column

for the number of a particular column

LEAVE_ENHANCEMENTS_ON
Leaves stub print enhancements on to the end of an entire the print file line. This allows you to enhance the numbers in a table as well as the text. Without this option, stubprint enhancements are automatically turned off at the end of each stub.

LEAVE_PAGE_OPEN (LVPGOPEN)
Does not print a page kick control character (^L) at the end of the table. Used to put multiple tables on a page. Does not re-print the header, banner, or footer. Use -LVPGOPEN to turn this off on the last table of the group to be printed together, or ~EXECUTE CLOSE_PAGE to immediately kick a page prior to the next table.

LEAVE_TABLE_OPEN (LVTABOPEN)
Allows you to continue tables on the same page, but forces them (unlike LEAVE_PAGE_OPEN) to have the same number of columns.

MARK_CHI_SQUARE=abcde

Marks cells as significant based on chi-square testing. This is an alternative to Neuman_Keuls, ANOVA_SCAN, and ALL_PAIRS testing. For each significant CHI_SQUARE test on the table, a formula is used to determine which cells are the most extreme. For bi-level testing, the process is repeated.

Syntax:

```
EDIT={edit1: MARK_CHI_SQUARE=abcde  }
```

a - Mark the positive adjusted residuals at the more extreme levels of significance.
b - Mark the negative.
c - Mark the positive residuals at the less extreme level of significance.
d - Mark the negative.
e - Mark the cells examined but not significant.

A letter or symbol is printed next to the cell(s), and also appears in the table's footnote. The default string is "+-*. ". Only two, four or five-digit strings are valid.

Example:

EDIT={edit2: STATS= apprx .6 + apprx .9, MARK_CHI_SQUARE=ABab }

This example would result in a table with footnote that says:

(sig= apprx 0.1 + sig= apprx 0.4) chi_square cells marked A/B for hi/lo at 0.1, a/b at 0.4

MINIMUM_BASE=# (MINBASE)

Specifies a minimum frequency for the total of a column before it will be printed. This option is used when doing statistics testing (i.e., EDIT=DO_STATISTICS is also specified and STATISTICS= has been defined), to ignore columns with a small total.

If the column is less than the value specified for MINBASE, an asterisk * will print instead of the column letter at the top of column and the cells will be blank.

Related Commands:

FLAG_MINIMUM_BASE_CELL.

MINIMUM_FOR_PRINTING (MINFORPRT)
By default Mentor will print tables even if they contain no data. MINIMUM_FOR_PRINTING allows you to set a System Total or weighted System Total threshold to prevent the printing of blank tables.

Syntax:

MINFORPRT=#

Tables with a System Total of zero are blank. To suppress the printing of blank tables, set MINIMUM_FOR_PRINTING to 0.5. This will ensure that the weighted System Total of tables that will not round up to one will not print

To print blank tables, but not print missing tables, set the MINIMUM_FOR_PRINTING to zero.

In a sample size of 1000, setting the MINIMUM_OF_PRINTING to 10 would suppress the printing of tables whose System Total was less than 1% of the total sample.

MINIMUM_FREQUENCY=# (MINFREQ)
Specifies the minimum number of the any response frequency (System Total minus System No Answer) for a row to print, otherwise it is suppressed. MINFREQ=1 will suppress any row with 0 responses. You may indicate a different column to base the suppression on by also using the SUPPRESS_ROWS_BASE= option on your EDIT statement. MINFREQ=0 or -MINFREQ will return Mentor to the default of printing all rows.

On a weighted run, the number used by MINFREQ is the unrounded weighted frequency. In some cases, you will want to set MINFREQ=0.5. This will allow frequencies that would round up to one if the row were not being suppressed to be printed.

MININDENT_LEFT=# (MININDLEFT)

Prints an error when the result of STUB_WIDTH minus (STUB_RANK_INDENT + STUB_INDENT + STUB_WRAP_INDENT) is less than MINIMUM_INDENT_LEFT. The default is 10. This prevents the printing of stubs with only a few characters per line. This happens when stub indenting options cause the stub width to become very small after indenting.

To specify a stub width to be less than 10, you must set this option to a number less than the default value of 10.

MINIMUM_PERCENT=# (MINPER)
Specifies the minimum vertical percent in the Any Response column in order to print the row. The # can be set to any value > 0. You may indicate a different column to base the suppression on by also using the SUPPRESS_ROWS_BASE= option on your EDIT statement. Turn off with -MINPER.

NEWMAN_KEULS_TEST (NKTEST)
Newman-Keuls testing of designated groups of columns or rows; the default is the All Possible Pairs test.

NUMBER_FORMAT=# (NUMFORM)
Prints the number as a plain number (#=0; default), with commas (#=1), or with commas and a dollar sign (#=2).

NUMBER_OF_CASES (NUMC)
Prints the number of cases above the table title. Default is do not print.

OVERLINE =x (OLINE)
Specifies what the overline character will be when the stub option [OVERLINE] is specified on the stub labels. It does not set overline for every stub label. The default overline character (specified as OLINE=;) is a dash (-). The overline character specified in an EDIT statement can be overridden by specifying something different on the stub label with [OVERLINE=s]. OVERLINE prints the character specified under the table cells above (overlines) the stub label it is specified on. This is often used to set off subtotals.

The number of characters in each overline is the column width minus two. The default is six (i.e., the default column width of eight minus two.). The minimum number of characters is three

Percent signs are not counted as part of the cell width.

PAGE_LENGTH=# (PGLEN)
This specifies the number of lines per page (length of the printed page). With some printers, the default page length of 60 pushes the footer onto the next page, so therefore a page length of 59 would be needed. It must be <= the page length specified on the >PRINT_FILE meta command.

PAGE_NUMBER=# (PGNUM)
Sets the page number to the value specified when the table is printed.

This option is useful when you want to reset the page number to 1 for a new banner or section of the table set, or if you want to insert tables into a set that has already been made, keeping the page numbers intact.

PAGE_NUMBER must be specified on a LOCAL_EDIT= statement with ~SET DROP_LOCAL_EDIT or turned off with LOCAL_EDIT=; If not, the same page number will print on succeeding tables.

PAGE_WIDTH=# (PGWID)
Specifies the number of columns per page (width of printed page). Must be <= the width specified on the >PRINT_FILE meta command.

Set the PAGE_WIDTH less than the print file width if you want your titles to wrap based on RUNNING_LINES=1 or 2, and you want centering to occur based on a column closer than the far right edge of the page.

PAIRED_VARIANCE (PAIREDVAR)
Uses paired variance when doing statistical testing using STATISTICS= or EDIT=DO_STATISTICS, overriding default of USUAL_VARIANCE.

PERCENT_DECIMALS=# (PDEC)
Specifies # of decimal places for vertical and horizontal percents. The default is 1; it may be a number from 0 to 7.

PERCENT_SIGN (PERSIGN)
This says to print a % sign after the percentage.

% is printed to the right of the column, causing the maximum possible table width to be one column less than the page width.

POOLED_VARIANCE (POOLEDVAR)
This uses pooled variance when doing statistical testing using STATISTICS= or EDIT=DO_STATISTICS, overriding default of USUAL_VARIANCE.

PREFIX= (PREF)
Defines the text to print before the table name on the table. It may be a string of up to 16 characters or ; to turn off. Default is no prefix or PREFIX=" ".

PRINT_ALPHA_TABLE_NAMES (PRTATABNAME)
Prints the leading letters in the table name. By default, the leading letters are stripped up to the first number when the table name prints.

PRINT _TABLE_NAME= (PRINTNAME=)
This allows you to assign table names to be printed on tables that are different from the actual table names stored in the DB file. You can assign one name for the first table and Mentor will automatically increment names on the subsequent tables.

Syntax:

```
PRINTNAME=S001
```

PRINT_BLANK_PERCENT_LINES (PRTBLKP)
Prints the blank lines for zero percent rows; one for the vertical percent and one for the horizontal percent (if requested). The default is to not print blank lines, but move the body of the table up. Use this option to maintain the same spacing on all printed tables even where some contain a zero percent row.

PUT_CHARACTERS=abcd (PUTCHAR)
Controls what to print in the table cell for 0 (whole/integer numbers), 0.0 (decimal/real numbers), missing or empty cell. The defaults are -, 0, ?, and - respectively.

Specify B to print a blank and Z to print a zero. This overrides the meta command >PUT_CHARACTERS. EDIT= COLUMN_INFO=PUT_CHARACTERS or STUB=[PUT_CHARACTERS] overrides the EDIT option. See the meta command >PUT_CHARACTERS in your *UTILITIES* manual for more information.

RANK_COLUMN_BASE= (RANKCOL)
Lets you specify the column to base table ranking on. The default is total (T), but may be the Any Response column (AR), or a column number.

RANK_IF_INDICATED (RANKIFIND)
Checks the stub set for any RANK_LEVEL commands and ranks the table only if any are found. This makes it unnecessary to turn ranking off and on with an EDIT=RANK option, but may slow down the printing somewhat.

The stub set will be read as if ranking is possible, so you will get errors if the number of stub lines is different from the number of rows.

RANK_LEVEL=# (RANK)
Specifies the rank level at which to start ranking the table.

Options:

0	no ranking
1	first level and below
2	second level and below

Ranking is done using the values in the first column of the table (Total), unless overridden with the EDIT= RANK_COLUMN_BASE=# command.

RANK_ORDER=(x)
Lets you specify ascending or descending rank for different levels of ranking. Use as many of As and Ds (separated by commas) as you wish to set for levels of ranking. The last one stays in effect for lower levels.

Options: Parentheses are required.

A	Ascending
D	Descending (default)

ROW_MEAN
Calculates the mean of data columns across a row, using the values specified in ROW_STATISTICS_VALUES.

ROW_MEDIAN (ROWMED)
Calculates the median of data columns across a row, using the values indicated. See COLUMN_MEDIAN.

ROW_NA
Prints the System No Answer (NA) row.

ROW_SE
Calculates the standard error of data columns across a row, using the values specified in ROW_STATISTICS_VALUES.

ROW_STATISTICS_VALUES= (ROWSTAT)
Assigns the values to be used on any of the "ROW_" printer statistics.

ROW_STD
Calculates the standard deviation of data columns across a row, using the values specified in ROW_STATISTICS_VALUES.

ROW_TNA
Prints the System Total and No Answer (NA) rows. This is the default.

ROW_VARIANCE (ROWVAR)
Calculates the variance of data columns across a row, using the values specified in ROW_STATISTICS_VALUES.

RUNNING_LINES=# (RUNLINE)

Controls how table text will be printed. This option affects all table text elements: TITLE=, HEADER=, FOOTER=, TITLE_2=, TITLE_4=, AND TITLE_5=. This will override the forcing of new lines from the original text definitions.

Options: 0

Prints lines as written (as defined by ~DEFINE LINES=, TITLE=, FOOTER=, etc.) This is the default.A table text variable defined with any of <, >, or = to control format forces RUNNING_LINES to 0 for that particular item. -RUNNING_LINES resets back to 0.

Leading blanks are dropped. Use a vertical bar (|) as a placeholder just as you would in a banner definition.

Example:

TITLE=:

12. Compared to other frozen dinners you now eat, would you say this one is ...?}

- Wraps the lines according to PAGE_WIDTH= and any INDENT= numbers specified.
- Prints the first line like RUNNING_LINES=1, then indents the second and subsequent lines by the length of the first word (from the first line) and any blanks immediately following, in addition to any INDENT= options specified. The maximum number of characters indented is 20.

This makes it easier to print titles in this format, where the question number is set off from the rest of the title text:

Q1a. text text text ...
next line prints here

Use \N in the text of your title variable to force a new line.

SAVE_RANK_INFO= (SAVERANK)

Saves the way this table is ranked so you can rank another table like it (for side-by-side comparisons).

Syntax:

```
SAVERANK=varname
```

The ranking information is saved in the read/write DB file under the varname specified. Use USE_RANK_INFO=varname to use on another table.

SEPARATE_VARIANCES (SEPARATEVAR)
Uses separate variances for columns when doing statistical testing using STATISTICS= or EDIT=DO_STATISTICS, overriding default of USUAL_VARIANCE.

SHOW_SIGNIFICANCE_ONLY (SHOWSIGONLY)
Prints only the significance line when printing the CHI_SQUARE and/or ANOVA in CHI_SQUARE_ANOVA_FORMAT.

This option may only be used if CHI_SQUARE_ANOVA_FORMAT is also specified.

SKIP_LINES=# (SKIP)
Skips number of lines between rows. The default is 1. Use SKIP_LINES=0 to condense the table for the maximum number of rows per page.

STAR_PERCENT=# (STARPER)
Says if the percentage is less than the value, print an asterisk (*) in the percent line of the cell. It is usually used to note numbers that print as if they are 0%, but are actually slightly above 0%, especially on a percentage-only table.

Options:

0 print the 0.
print an * if the percent in the cell is < #. Can be 0 or .001-100.

-1 is the default setting and cannot be specified as such. This means that if PERCENT_DECIMALS=0, then use .5; if =1 then use .05, etc.

A footnote that shows the STAR_PERCENT value is printed at the bottom of any table that has * in a percent cell.

STAT_MARKING
Changes the names (marking) for the columns that are tested with the STATISTICS statement. This overrides the default lettering scheme of ABCD, etc. In order to change one column, you must assign a unique letter to every column that is being tested. Use -STAT_MARKING to return to the default.

Example:

```
STATS=stat1 abcd
EDIT={edit1: STAT_MARKING="zyxw"    }
```

This will change the letters on the printed table from ABCD toZYWX.

Only letters are allowed; special characters (other than asterisk), numbers and blanks cannot be used. An asterisk can be used for a column that is not being tested. You can assign a new letter to column not being tested, it simply will not appear on the table.

Example:

```
STATS=stat2: ab,cd,fgh
EDIT={edit2: STAT_MARKING="zyxw*uts" }
```

STATISTICS_DECIMALS=#   (SDEC)
Specifies the number of decimal places for statistics. This controls printing of EDIT= printed stats (DO_MEAN, etc.), or any row or columns marked as a STAT type. The default is 1; it may be any number from 0 to 7, depending on your column width.

STATS_FOOTNOTE
The default is option is "on." Using -STATS_FOOTNOTE allows you to not include the footnotes usually printed on a table that contain statistics.

STATUS
Shows all EDIT options in effect at this time.

STUB_DEFAULT=[option(s)] (STUBDEF)
Says these will be the default STUB= [options] for rows where no STUB= [option(s)] have been specified. You may specify any STUB= [option] here. Stub labels with [option(s)] are not affected by any STUB_DEFAULT options.

System rows (e.g., Total and N/A) are affected by STUB_DEFAULT unless a STUB_PREFACE has been specified for these rows, and then only rows with [options] in the STUB_PREFACE definition will be unaffected by any STUB_DEFAULT [options]. If the STUB_PREFACE defines just the text for the System rows being printed, then they will be affected by the STUB_DEFAULT.

Empty square brackets [ ] are treated as if a stub label option had been specified; these rows are not affected by any STUB_DEFAULT=[options].

System-generated stub labels are not affected by any STUB_DEFAULT. For example, ROW=[6^1//3] without a corresponding STUB= definition for each of the three data categories would cause the program to generate the program default stub labels 6^1, 6^2, and 6^3. These labels would be affected by options specified on the EDIT= statement other than STUB_DEFAULT.

Example: `STUBDEF=[UNDERLINE]`

Says to append the STUB= option [UNDERLINE] to any stub label which has no [option] on it including the System rows.

STUB_EXTRA=[option(s)] (STUBEX)
Says to append these STUB= [options] to all of the existing STUB= [options] and to the System rows. A STUB_EXTRA option will override a conflicting STUB=[option]. These options are added to all printed rows except those with the STUB= option [-STUB_EXTRA] which says do not append the STUB_EXTRA to this row. STUB_EXTRA has the same effect as adding a STUB=[option] to

every printed row. This means that the STUB_DEFAULT option(s) have no effect on printed rows when specified in the same EDIT statement with STUB_EXTRA option(s).

STUB_INDENT=# (STUBIND)
Says how many spaces to indent the row stubs on the table.

STUB_PREFACE=x (STUBPREF)
Prepends this stubset on all tables. Useful to set options for printing of Total/NA/etc. Default is Total and N/A, with no stub controls (TNA). This can also be used as a table element in the ~EXECUTE block or in a ~DEFINE TABLE_SET. See also STUB_SUFFIX.

Options:

;	turns off a previous STUB_PREFACE.
stubset name	specifies the stubset to use.
TNA	prints Total and N/A with no stub controls. (default)
NONE	uses no STUB_PREFACE.

This option is evaluated differently by Mentor depending on whether you include it in your EDIT statement or as an EXECUTE option in a TABLE_SET=.

We recommend the latter to allow Mentor to accurately check that the number of row and stub lines match. As a TABLE_SET option, STUB_PREFACE is evaluated when the TABLE_SET is defined (see ~SET TABLE_SET_MATCH_WARN), but when it is specified as an EDIT option, the program will not generate a warning that row and stub lines might not match until the table building phase when the entire EDIT statement (including your stub preface) is processed.

STUB_RANK_INDENT=# # (STUBRIND)
is the number of spaces to indent for each rank level over level 1. The default is two spaces, but can be 0 - 10.

STUB_SUFFIX=Stubset name (STUBSUF)
Appends this stub set to the row stubs for all tables. This is used to add a standard row(s) to the bottom of all tables in a set. You can also specify the STUB_SUFFIX as a separate element in a TABLE_SET or in the ~EXECUTE block. See also STUB_PREFACE.

STUB_WIDTH=# (SWID)
Says the number of spaces for row text. Stubs longer than the STUB_WIDTH will wrap at the width, unless they have the [LONG_COMMENT] option on, or are forced to a new line with \N in the text. The default is 20.

STUB_WRAP_INDENT=option (STUBWIND)
Specifies how to indent the second and subsequent stub lines when a label continues to more than one line.

Options:

#	The number of spaces to indent the second and subsequent stub lines . The default is 0, but may be any number from 1-10.
WORD	Indents the second and subsequent lines to match up with the second word in the first line of the stub. This acts like the RUNNING_LINES=2 option.

SUBTOTAL=# # (SUBTOT)
Can be 1 to 5 for subtotaling.

SUFFIX= (SUF)
Defines the text to print after the table name on the table. It may be a string of up to 16 characters; use SUFFIX=; to return to the default of no text.

Syntax:

```
SUF="<=16 character string"
```

SUPPRESS_ROWS_BASE=# (SUPBASE)
Specifies the column to use as the basis for suppression of rows (EDIT= MINIMUM_FREQUENCY= and MINIMUM_PERCENT=). The default is to use the Any Response frequency (System Total minus No Answer), but it may be T (for System Total) or the number of a specific column. - SUPPRESS_ROWS_BASE resets to the default.

TABLE_TESTS=[$R col_list BY row_list] (TABTEST)
Specifies the region of the table for statistical tests to act on. The region variable ($R) is used to define the areas of the table to be tested. CHI_SQUARE=[region] and ANOVA=[region] will cause the appropriate test on the region specified to be calculated. Refer to EDIT= ANOVA= and ~CLEANER MODIFY <table> for information on defining a table region.

- You cannot do both a CHI-SQUARE and an ANOVA on the same table.
- When ANOVA=[region] is specified, it cannot be overridden by CHI_SQUARE.
- When CHI_SQUARE=[region] is specified, it cannot be overridden by ANOVA.
- When TABLE_TESTS=[region] is specified and both ANOVA and CHI_SQUARE are specified, Mentor will generate an error message.

This requires a two-step approach: first specifying the regions to be tested with TABTEST= and second specifying the test to be calculated. This allows you to specify all regions to be tested on the main EDIT statement and afterwards to specify the test type (CHI_SQUARE or ANOVA) on a LOCAL_EDIT statement.

If you want to use the region (or portion of a table) that is made up of the banner points two through five and rows one through 10 and you want to test for statistical significance using an ANOVA, you can add this to the LOCAL_EDIT statement in the TABLE_SET that describes the row.

Example:

```
LOCAL_EDIT={: ANOVA=[$R  2 TO 5 BY 1 TO 10] }
```

If you want to use the same banner points on a different table and use rows one through five and you want to test for statistical significance using a CHI-SQUARE, you can add this to the LOCAL_EDIT statement in the TABLE_SET that describes the row:

Example:

```
LOCAL_EDIT={: CHI_SQUARE=[$R  2 TO 5 BY 1 TO 4] }
```

If there are several regions in the banner that you want tested for statistical significance using a CHI-SQUARE, your LOCAL_EDIT statement might look like this:

Example:

```
LOCAL_EDIT={: CHI_SQUARE=[$R  2 TO 5  BY 1 TO 4]
CHI_SQUARE=[$R  6 TO 8  BY 1 TO 4]
CHI_SQUARE=[$R  9 TO 11 BY 1 TO 4]
CHI_SQUARE=[$R 12 TO 15 BY 1 TO 4] }
```

If you want to do either an ANOVA or a CHI-SQUARE on every table where one of these two tests is appropriate, you can add this to the TABLE_SET that describes the banner:

Example:

```
EDIT={: TABLE_TESTS=[$R  2 TO 5  BY 1 TO LAST]
TABLE_TESTS=[$R  6 TO 8  BY 1 TO LAST]
TABLE_TESTS=[$R  9 TO 11 BY 1 TO LAST]
TABLE_TESTS=[$R 12 TO 15 BY 1 TO LAST] }
```

Use "1 TO LAST" to describe the rows you want tested. It is not necessary to know the number of rows on each table. If specific rows need to be excluded from the statistics (i.e., DON'T KNOW), then use the STUB= option

[-COLUMN_STATISTICS_VALUES] to exclude specific rows on a question-by-question basis. If you have a job with more than one banner, the appropriate regions should be specified in TABLE_SET that defines each banner using this EDIT option.

Once the regions have been attached to the banner, then for each TABLE_SET that defines the row you can add one of the following LOCAL_EDIT statements:

Example:

```
LOCAL_EDIT={: ANOVA }
```

Example:

```
LOCAL_EDIT={: CHI_SQUARE }
```

Example:

```
LOCAL_EDIT={: -TABLE_TESTS }
```

The first example says to do an ANOVA on this table for the regions specified in the banner. The second example says do a CHI_SQUARE on this table for the regions described in the banner. The third example says to do neither an ANOVA nor CHI-SQUARE on this table even though regions are specified with the banner. See *"8.9 CHI-SQUARE AND ANOVA TESTS"* for additional examples and a discussion of when to choose ANOVA versus CHI_SQUARE.

The default format, EDIT=CHI_SQUARE_ANOVA_FORMAT, for printing the ANOVA and CHI-SQUARE statistics is at the bottom of the table with the banner categories used in the test designated by arrows.

ANOVA; F: <-- 2.94 -->
NUM,DEN: 3,146
PROB: 0.0345

It is possible with this format that there may not be enough room between the arrows to print the calculated chi-square value for the CHI-SQUARE test or the F-value for the ANOVA. Statistical tests can be summarized at the bottom of a table using the option -CHI_SQUARE_ANOVA_FORMAT on the EDIT statement and labeled using the syntax TABLE_TESTS=[$T="text" $R col_list BY row_list]. The advantage of using this method is that there is more room to print the results.

Example:

```
EDIT={: -CHI_SQUARE_ANOVA_FORMAT
TABLE_TESTS=[$T="STATS FOR RELATIONSHIP " $R  2 TO 5  BY 1 TO LAST]
TABLE_TESTS=[$T="STATS FOR FREQ OF USE  " $R  6 TO 8  BY 1 TO LAST]
TABLE_TESTS=[$T="STATS FOR POTENTIAL    " $R  9 TO 11 BY 1 TO LAST]
TABLE_TESTS=[$T="STATS FOR SALES        " $R 12 TO 15 BY 1 TO LAST]
}
```

The results for an ANOVA would be printed as:

```
STATS FOR RELATIONSHIP
anova = 2.94, df1,df2 = (3,146) prob = 0.0345
STATS FOR FREQ OF USE
anova = 0.18, df1,df2 = (2,124) prob = 0.8385
STATS FOR POTENTIAL
anova = 0.52, df1,df2 = (2,124) prob = 0.6014
STATS FOR SALES
anova = 0.38, df1,df2 = (3,125) prob = 0.7704
```

CHI_SQUARE_ANOVA_FORMAT also allows you to test non-contiguous columns and to use the same column in more than one test, which is not possible when using the default format.

Example:

```
EDIT={: -CHI_SQUARE_ANOVA_FORMAT
ANOVA=[$T="STATS FOR LOW VS MEDIUM  " $R   6,7 BY 1 TO LAST]
ANOVA=[$T="STATS FOR LOW VS HIGH    " $R   6,8 BY 1 TO LAST]
ANOVA=[$T="STATS FOR MEDIUM VS HIGH " $R   7,8 BY 1 TO LAST] }
```

TCON
Add the following tables to the table of contents.

TCON (SEPARATE_FILE)

This allows you to put the table of contents in one file and the printed tables in a separate file. You must designate the two print files using the following syntax:

print tables to this file: >printfile <table_file_name> #1

print table of contents to this file: >printfile <toc_file_name> #2

All of the following conditions will cause errors:

- designating only one print file

- print files are not marked as #1 and #2
- designating the print file marked as #2 before file marked as #1
- using both table of contents keywords first and two_files

TCON=(option)
Controls what is printed in the table of contents. Parentheses are required. Turn off any default by specifying TCON= (-option). Continue options to a new line with ampersand (&).

Options:

FIRST	Prints Table of Contents first instead of at the end
FOOTER	Prints footer in TCON (default) (FOOT)
HEADER	Prints header in TCON (default) (HEAD)
INDENT=# (IND)	Indents text in TCON Default is 0, do not indent.
PRINT_PAGE_NUMBERS (PRTPGNUM)	Prints table's page number in TCON
TABLE_NAMES (TABNAME)	Prints table's name in TCON (default)
TCON_PAGE_NUMBERS (TCONPGNUM)	Prints table of contents page numbers (default)
TITLE	Prints title in TCON (default)
TITLE_2	Prints T2 in TCON (default) (T2)
TITLE_4	Prints T4 in TCON (default) (T4)
TITLE_5	Prints T5 in TCON (T5)

TFRP
Prints system Total, N/A, and base rows with frequencies only. Prints the body of the table with percents only.

TITLE_4_FOR_BASE (T4BASE)
Creates a TITLE_4 label when a BASE= or FILTER= variable is defined. By default, Mentor will include a BASE or FILTER label on a table.

Example:

```
BASE=[1/30^1]
```

Prints the following label on the table:

```
BASE: [1/30.1:30^1]
```

To suppress this label, use -TITLE_4_FOR_BASE.

See Also: ~EXECUTE TITLE_4 for information on how to define a customized BASE or FILTER description.

TOP_MARGIN=# (TOPM)
Specifies the number (#) of lines to leave blank at the top of each page. *See important note at the beginning of this section.*

UNDERLINE_CHARACTER= (ULINE)
Specifies the character to use for underlining stubs. The default is dash (-). See STUB=[UNDERLINE].

USUAL_VARIANCE (USUALVAR)
Default variance used for STATISTICS= and EDIT=DO_STATISTICS T-tests. T-tests will use the pooled variance for NEWMAN-KEULS's tests and paired variance for multiple pairs testing or direct row testing.

USE_RANK_INFO= (USERANK)
Ranks this table in the same order as a table ranked previously that specified SAVE_RANK_INFO. Tables must have the same dimensions.

Syntax:

```
USERANK= varname
```

Varname is the DB item created by SAVE_RANK_INFO.

VERTICAL_PERCENT=x (VPER)
Causes vertical percentaging to print on the table, and specifies the vertical percent base for the table.

Options:

T	System Total row
AR	Any Response row
1-n	Data row number

You can modify the vertical percent base for any row by using the VPER= option in the stub controller for that row.

~VPER turns off vertical percentaging, but not cumulative percentaging (if specified). The percentage base would be the System TOTAL row.

~DEF LINES=

~DEF LINES=
Allows you to define text and then assign it to any of the table elements that use text, including ~EXECUTE HEADER=, FOOTER=, TITLE=, TITLE_2=, TITLE_4=, and TITLE_5=.

Syntax:

LINE={varname: text}

The varname can then be specified after the appropriate element in the ~EXECUTE table building block to cause this text to print on the table. The open brace ({) is not required.

There is no limit on the amount of text; use an ampersand (&) to continue a line, otherwise a new line is assumed for the next line of text. Use \N to force a new line anywhere in the text. See EDIT= RUNNING LINES=# to override the line format with different line break rules.

There are six text positioning operators available. You can left justify (<), center (=), and right justify (>). To affect the entire title text, the single symbol must be the first character of the LINES= statement.

Example:

```
TITLE={one: =This is centered.}
```

To combine text positioning operators on the same line, put the symbol anywhere on the line, preceded by a backslash and in tandem (two symbols).

Example:

~DEFINE

HEADER={Head1:

\==First line of Header Centered\>>Page #page_number#

\==Second line of Header Centered }

This example has the Header centered, and the page number right justified on the same line.

These operators require a PRINTFILE statement in which a LASER_CONTROL file is specified. (See UTILITIES, Appendix for information about the LASER_CONTROL file.)

Either use the text positioning operators in single OR in tandem. If you use a text positioning operator (in the singular) on the LINES statement, and then use a text positioning operator in tandem on the next line, then this will cause a print time error, and the entire title will not print on the table.

Example:

```
LINES={Example:> Right Justify This.
```

\==THIS WILL NOT APPEAR ON THE TABLE.}

Error message would look like:

(ERROR #951) Justification option not allowed in this context

The proper syntax to do this would be:

Example:

```
LINES={Example:\>> Right Justify This.
```

\==THIS WILL APPEAR PROPERLY ON THE TABLE.}

Just as in the case of other print enhancements, these operators remain in effect across title elements. Thus, if centering is turned on in the header, and never turned off with \-==, any TITLE_2 or TITLE_4 that appear on the table also will be centered. Print enhancements are turned off at the beginning and end of banner text, so TITLE_5 text and footers will not be affected by previous print enhancements. Options turned on in one table do NOT carry over to the next table.

The negative form of any text positioning operator will returns the text positioning to the default of left justified.

Example:

LINES={Example2:

\>>Right Justified.

\->>Back to Left Justified. }

See ~DEFINE TABLE_SET to specifically define headers, titles, and footers for tables without having to recall them by name in the ~EXECUTE block. For instance, you can say:

Example:

~DEFINE TABLE_SET=Q=Q1:

TITLE=: Do you like ice cream? }

STUB=:

Yes

No }

ROW=: [5^1/2]

}

~EXECUTE TABLE_SET=Q1

You can also print the page number, date, and time anywhere on a LINES= table element using the special characters #option#.

Syntax:

```
#option (start.wid)#
```

Options:

DATE (DATE)
DD MMM 19YY and is date run started.

PAGE_NUMBER (PGNUM)
This prints right-justified in the width if given, or else it prints in the field just long enough to hold the number.

TABLE_NAME (TAB)
Prints the table name anywhere in the table text.

TIME (TIME)
HH:MM AM|PM and is time run started.

VARIABLE (VAR=varname)
Prints the current contents of the variable anywhere in the table text.

start.wid

This is optional and is used to specify part of date or time or the width to print the page number in. START is not used for the page number. If the width is not specified, it defaults to the rest of the field for date/time.

Example:

```
LINES={testing:
This is run date: #DATE#
This is run time: #TIME#
This is run month: #DATE (4.3)#
This is run year: #DATE (8)#  "defaults to (8.4)
Page: #PAGE_NUMBER#
This is page: #PGNUM (.3)#
Page Number: #PGNUM#
Page Number .3: #PGNUM(.3)#
Date: #DATE#
Date: #DATE(1.6)#
Time: #TIME#
Time: #TIME(.2)# }
```

Related Commands:

System information constants: DATE_TIME; PAGE_NUMBER; JULIAN_DATE; TABLE_NAME; TEXT_AREA_STATUS see *8.3.1 SYSTEM CONSTANTS*.

~ DEF PREPARE=

~DEF PREPARE= (PREP) Defines a data element in PREPARE= syntax. This allows full screen data entry and modification, and includes text, base, category, and screen control elements. See the ***Survent*** manual for more information.

Syntax:

```
PREP={ name: # col.wid
text of question
   !question type, parameters
   [response table for a CAT or FLD type] }
```

Unlike a question defined in the ~PREPARE block, col.wid is required.

~DEF PROCEDURE= (PROC)

Allows you to define a procedure which is then later executed using ~EXECUTE PROCEDURE=name. Procedures are used for a whole variety of tasks, including displaying and/or modifying data during cleaning, combining files, and writing reports.

Syntax:

PROCEDURE={procname:

```
commands and/or conditional expressions
commands and/or conditional expressions
}
```

The opening brace ({) is optional. If you want to use more than one command on a line, start any command after the first one with a dollar sign ($). Mentor will assume any text without a dollar sign after the first command is a variable name.

Example:

PROCEDURE=look

IF {5^1] $WRITECASE $ENDIF

Conditional expressions are: IF-THEN-ELSE-ENDIF, and WHILE-ENDWHILE.

The syntax for IF-THEN statements is:

```
LABEL: IF expression

       THEN
       command or conditional expression
       ELSE
       command or conditional expression
       ENDIF [LABEL]
```

The conditional expression within your IF block can be another IF statement, and you can have unlimited levels of one IF block inside another (this is sometimes called "nested" IF blocks). Having one IF block inside another can be confusing, so use this format of indenting various levels and it will be easier to read your procedure and see that each IF has a matching ENDIF.

Labeling an IF block is optional, but if you do use labels, error messages will include references to that block by label.

You can use the GOTO command to move forward to another place in a procedure or to move in or out of an IF block (see ~CLEAN GOTO or Mentor Volume One, Chapter 2: Branching In A Cleaning Procedure).

NOTE: Commands that work both in ~DEFINEd procedures and in ~CLEANER are described in the ~CLEANER section of this manual. A list of commands that can be used in a ~DEFINEd procedures is at the beginning of the ~CLEANER section.

Two specialized conditional expressions are WHEN TOP and WHEN BOTTOM, which check to see if you are at the top or bottom of a printed page (see ~CLEANER WHENTOP/WHEN BOTTOM). These are often used to create headers or footers. See *Mentor Volume One, Chapter 9: Printing a Report Footer Using WHEN BOTTOM* for details.

The other conditional expression is WHILE-ENDWHILE.

WHILE has the option MATCHING. WHILE MATCHING allows you to take information from different data files as long as they have a matching field. The matching field must match exactly and be sorted in order.

Syntax:

WHILE MATCHING "studyname" matchfield

It is important to clean and sort the data files before using WHILE MATCHING. If the secondary file(s) are not sorted on the match field, Mentor will give and error and exit. Check your files carefully, the following problems will NOT produce an error: the primary file is not sorted on the match field; duplicate ID numbers in the secondary file(s); or records that are in one file but not in another.

You can use WHILE MATCHING, for example, to do hierarchical jobs (such as master-trailer processing) on sets of files. You can have one primary file and a series of subordinate files by nesting the WHILE MATCHING command. See *Mentor Volume I Chapter 6: ADVANCED TABLES* for an example of master-trailer processing.

Related Commands:

~EXECUTE PROCEDURE=

~DEF STATISTICS=

~DEF STATISTICS= Defines a variable which designates the columns or rows to be tested when using ~EXECUTE STAT=. T-tests are done on each of the groups specified.

Syntax:

```
STAT=varname: PRINTABLE_T abcd
I=ef,gh,T=ijkl,D=1,2,P=3,4,RM=op
```

Options:

varname

The variable name under which the statistical test set will be stored in the open READ_WRITE db file (default is the local db file).

Continue the statement to subsequent lines with an ampersand & at the end of the line.

PRINTABLE_T
Use PRINTABLE_T= when printing T values. It will check that:

- STAT= tests are only in pairs.
- no column may be the second column of pair twice (because you can only print one T value per column).

abcd
The letters represent the relative position of the column or row; A is first, B is the second, and so on. Letters may be A-Z to reference up to 26 columns. Letters are placed together in groups of two or more. If more than two items are placed together, tests will be done on all combinations within the group, and the co-variance will affect the statistics.

The letter of the column that is significantly lower than another column will print on the cell of that column. Generally all rows are tested, unless you specified $[-STAT] as part of the definition or ~SET MEAN_STATISTICS_ONLY.

I=
Use I=group to signify that the items in the group are independent from each other; none of the group members has a shared response with one of the other items in the group. This speeds up processing, but does not affect the results.

T=
Use T=<letters> to do an inclusive T-test of the first column specified (e.g., TOTAL) vs. the rest of the columns, with the following rules:

- The first column includes the rest (this is checked for during data reading), for instance, the TOTAL column.
- The tests during printing are always T_tests, and are done by adjusting the values so the actual tests are the desired ones, i.e., “TOTAL but not B” vs. B, “TOTAL but not C” vs. C, etc. for the first column against the rest.
- After the first (TOTAL) column, the remaining columns can be any mixture of independent and/or overlapped groups. This means that inclusion is accounted for only with the first column specified. If more complex inclusions exist in the banner, then it will require writing a more complicated set of tests.

Example: `STAT=Stat1: T=ABCD,T=EFGH,T=IJKLM`

This would do the inclusive test on the first four columns, the next four columns, and the next five columns. Note that the first of each of these groups must be the TOTAL for the group or greater.

`PRINTABLE_T`
Use PRINTABLE_T=<letters> to check that you are following these rules:

- That the STAT= tests are only in pairs.
- That no column may be the second column of pair twice (because you can only print one T value per column).

Printing T-Test Values

You can print the T value when testing pairs of columns in addition to marking tables for significance.

Example:

STAT= PRINTABLE_T ab,ac,ad,ae,I=af,I=ag

ROW=:[1/6.1^5//1/10] $[MEAN] [1/6]

STUB=s:

...

[STATISTICS_ROW] Mean

[DO_T_TEST=*] T test

[DO_SIG_T=*]Significances

}

With this syntax, you can also test separate pairs:

Example:`STAT=AB,CD,I=EF,I=GH`

You can also do tests on rows that are not means, test columns that are in different locations in the banner, and do more than one test per table.

Example:`STAT=AB,AC,DE,FG`

You can combine rows that are marked with stat letters with printed T values on the same table by specifying EDIT=DO_STATISTICS on the table.

For printed T values, the only rules are that the STATS testing must only have pairs, and you cannot use the same column twice for the second item of a pair (because you cannot print two T values in the same space in a column).

The T-test specification is always on the STAT= lines. STAT=ab,cd would print the T values under the "b" and "d" columns.

You must specify either STUB=[DO_T_TEST=*] or [DO_T_TEST=#] on the row where you want the T value, or [DO-T_SIG= *] or [DO_T_SIG=#] to print the T significance on the row. You can use *-# to reference a data row this number # of rows above the previous data row.

You can also say [DO_T_TEST=PRINT_MEAN] to get the T values using the Mean created by the EDIT=COLUMN_MEAN and STUB=[PRINT_ROW=MEAN].

`D= and P=`

Use D= (for direct testing) on pairs of rows against each other, or P= (for distributed preference testing). You can test as many pairs of rows as needed, but you cannot use the same row twice. For more information on row testing refer to *"8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)"*.

Rows are marked with a lower case "s" if they are significantly higher than their counter part.

Example:

STAT=S: D=1,2 D=3,4 P=5,6

For distributed preference testing, you must tell the program you are distributing the preference by defining the rows with a SELECT clause that puts the distributed values evenly into each of the two groups. This is usually used when you want the "Don't know" or "No Answer" responses distributed evenly between the groups.

Example:

```
~DEFINE
TABLE_SET=Distrib:
STAT=: P=1,2
EDIT=: DO_STATISTICS=.95 }
COLUMN=: TOTAL
ROW=: SELECT_VALUE([15^1/X,B],VALUES(1,.5))WITH &
SELECT_VALUE([15^2/X,B],VALUES(1,.5) )
}
~EXECUTE TABLE_SET=Distrib
```

Notice that the X punch and Blank (B) responses are split evenly between the two groups. On printed distributed preference tables, you will get an "s" printed for significant, "e" for equal (if they are about the same), and "ns" if not significant.

RM=

For use only with the ANOVA_SCAN or FISHER tests. Repeated Measures(RM) causes Mentor to accumulate the sum of within_person variance (and the corresponding degrees of freedom) so that they can be deducted from the total_variance (and degrees of freedom) prior to calculating the F-ratio for the analysis of variance. You can use this option if you have dependent banners or a weighted sample and want to use ANOVA_SCAN or FISHER pre-scanning with subsequent all pairs testing as a substitute for the Newman-Keuls procedure.

Example:

STAT=: RM=AB, RM=CD, RM=EFGH, RM=IJKLM, RM=MNOP

EDIT options:

Statistics testing is modified depending on EDIT= options specified.

Example:

STAT=one: ab,I=cde,fg

~DEFINE EDIT=ed: DO_STATISTICS=.90, MINIMUM_BASE=5, ALL_POSSIBLE_PAIRS_TEST }

~EXECUTE STATISTICS=one ‘‘Assigns the STATS variable ONE ‘‘to the table.

EDIT=ed, COLUMN=ban, ROW=first, TABLE=T001

This defines the stats variable ONE, then executes the table doing all possible pairs test on the CDE group, and a standard T-test on the AB group and FG group. The .10 level of significance will be used (DO_STATISTICS=.90). If a column has a total <= 5, it will not print (MINBASE=5).

~DEF STUB=

~DEF STUB= defines the text for labeling each row stub, and the options for the printing of the row. This would override or be in addition to any options specified on the EDIT= statement for the table. When applicable, options have the same syntax and meaning as their corresponding EDIT= options, but apply to their specific row only.

Syntax:

STUB= {varname:

[options] text &

}

Options:

{

Starts the STUB (optional).

varname

Assigns a name to the STUB. Varnames are limited to a maximum of 14 characters.

[options]

Must precede the text for the row. See option list below.

text

Optional; **\N** will kick a new line in the text; **&** will continue the stub to the next line. Use vertical bar (|) followed by spaces to indent the text. The maximum number of characters is 294.

You can insert optional quotes around the text. These would be required if you wanted to define another stub on the same line.

}

Ends the stub.

The following summary information will be printed to the list file for each STUB= definition: the number of data rows, stats density rows, comment rows, and print_rows.

Omitting options prints the data row according to the EDIT statement parameters. Most of these options can also be specified as EDIT= options. See that statement for more information.

Stub options may also be used to turn off EDIT= elements for a row; do this by placing a minus sign before the option

Example:

[-VPER]

Means don't print a vertical percent on this row. Options affect only the row they appear on, with the exception of VERTICAL_PERCENT, KEEP_RANK, KEEP_STUB_INDENT, and KEEP_SUBTOTAL=# which stay in effect until specifically turned off.

Options to justify text (= > <) do not work on stub text. Use the vertical bar (|) followed by spaces to indent text.

Option list:

`[BASE_ROW]    (BASE)`

This row will print as a base line for STATISTICS= multiple T-testing.

You need to define an additional stub if your row variable includes a $[BASE] statement since it produces two rows: the first being the total base and the second the effective base. The total base row is typically used for the percent base on the rows that follow. This is the row that you would want to mark as the [BASE_ROW].

The [BASE] specification will cause a percent % sign to print under the row and the (letter) of the columns tested signifying it as the percentage base for statistical testing. You should also specify VERTICAL_PERCENT=* whenever you use [BASE], since the program expects the percents printed to match the percents used in the statistical calculations. Refer to EDIT=STATISTICS_PERCENT_WARN to override proper percentage checking.

[-BASE]

suppresses the base markings on the system TOTAL during statistics testing, specifically when ~SET STATISTICS_BASE_AR is specified to use the ANY RESPONSE row as the base.

[-COLUMN_STATISTICS_VALUES] (-COLSTAT)
Excludes this row from COLUMN printer statistics tests specified on the EDIT statement (e.g., COLUMN_MEAN) Other values (e.g., frequency and percent) are printed for this row. This does not affect STATISTICS= tests.

[COMMENT] (COM)
Prints a comment label, with no corresponding row of data. The label will break just like data row text at EDIT= STUB_WIDTH.

Comment rows can be assigned a rank level.

Example:

```
STUB=:
            [RANK_LEVEL=1] Liked Product net #1
            [RANK_LEVEL=2, COM] Taste subnet #1
            [RANK_LEVEL=3] Taste item #111
            [RANK_LEVEL=3] Taste item #112
            [RANK_LEVEL=2, COM] Texture subnet #2
```

```
[RANK_LEVEL=3] Texture item #121
[RANK_LEVEL=3] Texture item #122
[RANK_LEVEL=1] Disliked Product net #2
}
```

[CUMULATIVE_PERCENT] (CPER)
Prints the cumulative percent on this row. The percentage base is the same as the vertical percent.

[-VERTICAL_PERCENT] also excludes the row from the cumulative percent.

[DATA_INDENT=#] (DATAIND)
Indent the data cells on this row this number of spaces.

[DO_SIG_T=option] (SIGT)
Prints the significance level of the T values from a ~DEFINE STATISTICS= test. See the example under [DO_T_TEST=].

Options:
#
The number of the row to do the test on (excluding system generated rows).

*
use the last row seen.

*-#
use the row # rows above the current row.

PRINT_MEAN (PRTMEAN)
print the T values from the MEAN generated by EDIT=COLUMN_MEAN.

This option works in conjunction with the ~DEFINE STATISTICS= command, and *not* the EDIT=DO_PRINTER_STATISTICS command.

[DO_STATISTICS= stattest] (STAT=)
Functions the same as EDIT=DO_STATISTICS option. It does a STATISITCS test on the designated row.

STAT= Options:

- ALL_POSSIBLE_PAIRS_TEST (ALLPAIR)

  Does an all pairs test on the designated row if EDIT=DO_STATISTICS is on.

- ANOVA_SCAN

  Does an Anova-Scan test on the columns of this row specified on the STATISTICS= command.

-

- FISHER

  Does a Fisher test on the columns of this row specified on the STATISTICS= command.

- KRUSKAL_WALLIS_TEST (KW)

  Does a Kruskal-Wallis test on the columns of this row specified on the STATISTICS= command. This only makes sense on a [PRINT_ROW=MEAN] row stub.

- NEWMAN_KEULS_TEST (NK)

  Does a Newman-Keuls test on the columns of this row specified on the STATISTICS= command.

[DO_T_TEST=option] (TTEST)
Prints the T values from a STATISTICS= test.

TTest Options: #
Indicates the number of the row to do the test on (excluding system-generated rows).

* - Says to use the last row seen.
*-# - Says use the row # rows above the current row.

PRINT_MEAN (PRTMEAN)
Print the T values from the MEAN generated by EDIT=COLUMN_MEAN. This option works in conjunction with the ~DEFINE STATISTICS= command, and *not* the EDIT=DO_PRINTER_STATISTICS command. See example below.

Example:

```
STUB=:
  (5)  Completely agree
  (4)  Somewhat agree
  (3)  Neither agree nor disagree
  (2)  Somewhat disagree
  (1)  Completely disagree Don't Know/Refused to answer
[SKIP_LINES=2 STATISTICS_ROW] Mean
[STATISTICS_ROW] Standard deviation
[STATISTICS_ROW] Standard error
[DO_T_TEST=*-2] T-test for Mean built into data
[DO_SIG_T=*-2] Significance
[PRINT_ROW=MEAN] Printed table Mean
[TTEST=PRTMEAN] T-test on printed Mean
[DO_SIG_T=PRTMEAN] Significance of printed Mean }
```

[FOR_OUT_TABLES]
Controls which lines will be passed to tables saved with the SAVE_TABLE option and which lines will appear in delimited files. The default for this option is "on"; you only need to use it in its negative form [-FOR_OUT_TABLES] to prevent certain stubs from being passed to saved or delimited tables without suppressing them in the printed tables.

[FREQUENCY] (FREQ)
Prints the frequency for the row.

[FREQUENCY_DECIMALS=#] FDEC)
Prints frequencies in this row with # decimals. May be a number 0 to 7.

[FREQUENCY_ONLY] (FREQONLY)
Prints frequency only, no percentage.

[HORIZONTAL_PERCENT] (HPER)
Prints horizontal percent.

[INDEX]
Prints the index. See ~DEFINE EDIT INDEX for an explanation of index.

[KEEP_RANK_LEVEL=#] (KR=#)
Rank this and following rows at level #. Also see RANK_LEVEL to specify level and/or force high or low.

[KEEP_STUB_INDENT=#] (KPSTBIND)
Keeps the current EDIT=STUB_INDENT= in effect for subsequent rows until either a new STUB_INDENT=# or -STUB_INDENT to return to the default.

[KEEP_SUBTOTAL_#] (KPSUBTOT)
This leaves subtotaling on for following stubs at level #. (# can be 1 to 5).

[LINES_LEFT=#] (LINELEFT)
Specifies that there must be this number of lines left on the page to print this row, otherwise this row will print on the next page. This option is used to keep groups of rows together on the same page. Use [NEW_PAGE] to cause an automatic page kick to the next page.

[LONG_COMMENT] (LCOM)
Prints a comment label with no corresponding row of data. Unlike [COMMENT], it will print as far across the table as the PAGE_WIDTH.

[MINIMUM_FREQUENCY=] (MINFREQ)
Specifies the minimum frequency this row must have (in the column specified in EDIT=MINIMUM_BASE) in order to print.

[MINIMUM_PERCENT=] (MINPER)
Specifies the minimum percents this row must have (in the column specified in EDIT=SUPPRESS_ROWS_BASE) in order to print. MINFREQ=1 would suppress all rows with 0 in the base cell.

[\N]
Skips one extra line per \N found before printing the row (additional to SKIP_LINES set on EDIT statement). See also [SKIP_LINES=#].

[NEW_PAGE] (NEWPG or \P)
Skips to a new page before printing this row.

[NUMBER_FORMAT=#] (NUMFORM)
Prints the number as a plain number (#=0; default), with commas (#=1), or with commas and a dollar sign (#=2).

[OVERLINE=x] (OLINE)
Overlines the table cells on that line with either a dash, the default, or to whatever the specified overline character is on the EDIT= option OVERLINE_CHARACTER=.

The overline character for the stub may be specified on the option and it will override whatever was specified on EDIT=OVERLINE_CHARACTER=.

For example, [OVERLINE=*] says overline the cells in this row with the asterisk (*) character. The character specified as the overline character on a particular stub label will only effect that row; it does not set the overline character for subsequent rows. The number of characters in each overline is the column width minus two. Percent signs are not counted as part of the cell width.

[PERCENT_DECIMALS=] (PDEC)
Specifies the number of decimals to print in percents. The default is to print percents with one decimal place. Specify any value 0-7.

[PERCENT_SIGN] (PERSIGN)
Prints a percent sign (%) with percents.

[PRINT_ROW=x] (PRT=)
Print system row x here.

Options:

AR	Any Response
CHI	Chi Square
M	Mean
MED	Median
NA	No Answer
SE	Standard Error
SIG	Sigma
SSIG	Super Sigma (Sigma across tables)
STD	Standard deviation
SUBTOT#	Subtotal with clearing, where # can be 1 to 5
SUBTOTNC#	Subtotal with no clearing
SUP	Suppressed (Sum of all the rows suppressed with MINIMUM_FREQUENCY or MINIMUM_PERCENT)
TOT	Total
UAR	Unweighted Any Response
UNA	Unweighted No Answer
UTOT	Unweighted Total
V	Variance
#	Print the data row number specified here.

This can be useful for printing system rows in non-standard locations, or simply to rename the default text for those items.

~SET UNWEIGHTED_TOP must be on for any unweighted row printing on a weighted table.

On ranked tables you can rank print rows (except Any Response), but you must specify the rank position as either High or Low (e.g., [PRINT_ROW=MEAN, RANK_LEVEL=1H]). Rows specified with a rank option will indent according to the value given on EDIT=RANK_INDENT=# even if the table is not ranked.

When your tables include percentaging, then percents printed for either Sigma or Super Sigma represent the sum of the percents for a particular table and not the result of a calculated percent. Likewise, the sum of the percents printed on either the Sigma or Super Sigma line will be reduced for every row where percentaging has been suppressed with -VERTICAL_PERCENT by the percentage that would have printed on that row.

[PUT_CHARACTERS=abcd] (PUTCHAR)
Allows you to specify three characters which designate how to fill a cell containing zeros (either whole or decimal) or missing data.

Options:

a
Whole/integer numbers (i.e., 0). The default is dash (-).

b
Decimal/real numbers (i.e., 0.00). The default is 0 (zero).

c
Missing numbers. The default is ?.

d
'empty cells value', see EDIT= EMPTY_CELLS

Use B for blank, 0 for zeros, otherwise the character specified is printed in place of a zero integer, a zero real number, or a missing value. See also EDIT=PUT_CHARACTER and the meta command

>PUT_CHARACTERS.

[RANK_LEVEL=#, H, or L] (R)
Level at which to rank this stub. Generally used when there are nets and subnets within categories. The stub is treated as level 1 by default; that is, the highest rank level. Ranked items always stay with the closest stub above them with a higher level when that item is ranked.

is for level, and can be 0 - 9. Items marked as level 0 stay where they are; they are not ranked. Items marked with an H are ranked highest within their level. Items marked L are ranked lowest. Items marked H or L stay in original order if several in a row are marked such. Can be #H, #L, H, L, or just the number 0-9. See also [KEEP_RANK].

This stub label will indent the row by the number of spaces specified on the ~DEFINE EDIT=STUB_RANK_INDENT= command even if you are not ranking the table (the default is two spaces). You can override this by saying EDIT=: STUB_RANK_INDENT=0.

Mentor does not allow non-sequential rank levels. You cannot specify a rank level that is greater than one rank level higher than the highest rank level used so far in a table. For example, you cannot specify rank level 2 if there has not yet been a rank level 1.

[RANK_PRE_GROUP] (RPREG)
Gives a group name for ranking. Used to group sub-groups under nets. This option does a pre-sort before ranking, so it is useful when members of a group are not together in the table. To use this option, you must also have RANK_LEVEL= and ~DEF EDIT RANK_IF_INDICATED set.

[REPRINT_BASE] (PRTBASE)
Prints this stub (and label) at the top of all continued pages of a table. Stub options inside [ ]s will not be used; when this row prints at the top of pages, the format will be controlled by the EDIT= parameters.

[-SIGMA] (-SIG)
Don't include this row in sigma or super-sigma.[SKIP_LINES=#] (SKIP)

Skips this many lines before printing this row. Default is one line skipped between rows. Set SKIP_LINES=0 for dense printing. See also [\N].

[STATISTICS_DECIMALS=#] (SDEC)
Specifies the number of decimals to print in statistics. The default is one, but may be 0-7.

[STATISTICS_ROW] (STAT)
This is a statistics row. Frequency decimals are controlled by STATISTIC_DECIMALS=, and no percent will print.

[STUB_EXTRA] (STUBX)
Says use the stub options specified on EDIT=STUB_EXTRA=[option(s)] for this row (default). Turn off EDIT=STUB_EXTRA=[option(s)] for a particular row with

[-STUB_EXTRA].

[STUB_INDENT=#] (STBIND)
Specifies number of spaces to indent this stub.

[SUBTOTAL#] (SUBTOT)

Prints the subtotal here. # can be 1 to 5. SUBTOTAL can also be a value for the [PRINT_ROW] option. For example, `[PRINT_ROW=SUBTOT3]` will print subtotal 3 and then clear subtotal 3. `[PRINT_ROW=SUBTOTNC3]` will print subtotal 3 and not clear subtotal 3.

[SUPPRESS] (SUP)

Suppresses this data row regardless of the numbers in the cells.

[SUPPRESS_IF_EDIT_FREQUENCY] (SUPFREQ)

Suppresses this row if EDIT says to print frequencies.

[SUPPRESS_IF_EDIT_NO_FREQUENCY] (SUPNOFREQ)

Suppresses this row if EDIT says not to print frequencies.

[SUPPRESS_ROWS_BASE=#] (SUPBASE)

Specifies the column to use as the basis for suppression of this row (EDIT= MINIMUM_FREQUENCY= and MINIMUM_PERCENT=). Default is AR, but may be T (Total) or the number of the column.

[UNDERLINE=a] (ULINE)

Underlines this row's stub with character indicated (a). The default is a dash (-); it can be any ASCII character.

[VERTICAL_PERCENT=T/AR/#/*/(#,#)] (VPER)

Prints the vertical percent and/or changes the vertical percentage base. If the vertical percentage base is changed, it will stay in effect for subsequent rows.

[-VPER] will turn off the vertical percent for the current row and exclude the row from the cumulative percent.

Options:

T
Use Total row as the base (default).

AR
Use Any Response row as the base.

#
Use this row as a base.

Example: `[VPER=2]`

*****
Use the current row for a base. (The vertical percent for all columns in this row will be 100%.)

Example `[VPER=*]`

(#,#)
Use this cell for a base.

Example: `[VPER=(1,5)]`

This example will use the value in column one, row five as a base for the current row and the following rows.

~DEF TABLE=

~DEF TABLE= allows you to define a filled in table with ASCII specifications. You can then modify the table using ~CLEANER MODIFY <table>, or print the table in the ~EXECUTE block.

You can also read tables in from Mentor DB files, or write tables out using ~WRITE_SPECS to be transferred to other systems (such as LOTUS).

Syntax:

```
TAB={tablename:  #,#  #,#
<keyword1=varname1>
...
<keywordn=varnamen>
R=# # # # # # # #
...
R=n # # # # # # #
}
```

Options:

tablename: #,# #,#

Describes the name and number of columns and rows in the table. The first #,# is the number of columns, with the first number being the number of system columns (usually 2), and the second number the total number of columns. The second #,# is the number of rows in the same fashion.

Keywords

Any of the table elements HEADER, TITLE_2, TITLE, TITLE_4, BANNER, STUB, TITLE_5, FOOTER, STATISTICS, COLUMN, ROW, and others explained in the ~EXECUTE block.

R=#

These lines describe the numbers in the cells of the rows. The first # is the row #, where -1 is the Total row, 0 is the No Answer row, 1 is the first row, etc. The rows must be in ascending order, and must all be described. Numbers in the cells may be integers or decimal numbers. A dash (-) signifies an empty cell. Cells are separated by blanks.

Example:

```
TAB={T001: 2,3  2,5
TITLE= T001_t
COLUMN= Totl
ROW= County
BANNER= Ban1
STUB= County_s
R=-1 325. - 325.
R=0  10  -  -
R=1  150. - 150.
R=2  100. - 100.
R=3  75.  -  75.
}
```

~ DEF TABLE_SET=

~ DEF TABLE_SET= (TABSET) defines an entire table by allowing you to combine table elements. These are then executed together by referencing the TABLE_SET variable in the ~EXECUTE block (see ~EXECUTE TABLE_SET=varname). This is the preferred way to build table specifications.

Syntax:

```
TABSET={varname:
table_element1=:
table_element2=<varname>:
..
}
```

Options:

{
Starts the TABSET (optional).

varname
Assigns a name to the TABSET. Varnames are limited to a maximum of 14 characters. You can also say newname=oldname:, meaning use all of the elements from a previously defined TABSET. This is the base name for all other keyword= definitions in this TABSET. Refer under ~EXECUTE for the list of program-generated varnames generated from a TABSET name. Program-generated varnames are limited to 10 characters.

:
Required only when you are defining a new table element.

table_element=
Required. This may be any ~EXECUTE table element command.

<varname>
Optional, but the program will generate a varname for each table element defined derived from the TABSET name. You may also refer to a previously defined variable by its varname here.

}
Ends the TABSET definition.

- You may only define one of each allowed table element, but in any order (even for SET AUTOMATIC_TABLES mode). This means you may not define an element and then turn it off in the same TABSET. If you need to define tables in this way, use TABLE SPECS=.

- All table text items must have a closing brace (}); they have the same syntax as ~DEFINE LINES=. Once defined, they can be used interchangeably (i.e., an item defined as a TITLE could later be used as a HEADER).

- You may specify any ~SET commands inside a TABSET definition, but you are limited to as many as can be typed on one line. Specify the SET commands that will effect only this table or group of tables, and all your global SET commands outside of the TABSET with ~SET anywhere in your specifications before the ~INPUT statement and the ~EXECUTE block.

- Table elements can be turned off with either -table_element or table_element=;

Example:

```
TABSET={example_1:
TITLE=title1: here is the title }
EDIT=Stdedit
STUB=:
one
two }
ROW=: [5^1/2]
}
```

The TABSET name is EXAMPLE_1. It has a title named TITLE1. It is using a previously defined EDIT called STDEDIT. The row stub and row will be assigned default names based on the TABSET name.

Define a new table from a previously defined TABSET as follows, changing or adding any new variable definitions:

Example:

TABSET={example_3=example_1:

TITLE: Sameas example_1 with a base added}

BASE=: [10^1]

TITLE_4=: BASE: The table base}

}

PROGRAM-GENERATED TAB FILE

If you use ~SPEC_FILES <filename>, a TAB file is opened with this filename. Mentor will then write TABSET=name to the TAB file for each ~DEFINE TABLE_SET= it processes. You can reference the TAB file (filename^TAB) directly or use ~EXECUTE MAKE_TABLES to do automatic table execution of all variables defined with TABSET=.

Example:

~SPEC_FILES myjob

~DEFINE TABSET=q1:

TITLE=: Age or respondent

STUB=: 18-25

26-35

36-45

46-60

61 or more }

```
    ROW=:[5^1//5]
    }

    TABSET=q2: … etc.
~SET AUTOMATIC_TABLES
~INPUT data1
~EXECUTE MAKE_TABLES
~END
```

Reads the program-generated TAB file to make tables. This would create, build, and print all the tables in the order defined.

Any item usable in the ~EXECUTE block can be defined in the TABLE_SET. You can specifically call in TABLE_SETs in the ~EXECUTE block with TABLE_SET=name.

Related Commands:TABLE_SPECS=

TABLE_SPECS=

TABLE_SPECS= (TABSPEC) defines a set of table building elements as a single item to build either one table or a series of tables (e.g., changing the base). Unlike TABLE_SET= you can include more than one of the ~EXECUTE table element assignment keywords. Refer to the syntax under TABLE_SET= above.

You can use more than one of any of the allowed ~EXECUTE keywords in a TABSPEC definition. This means you can specify an item such as BASE= and then turn it off (-BASE) in the same definition.

The order of keywords is important. The program will process the table elements in the order they are defined.

ROW= must be the last keyword in the definition if you are using ~SET AUTOMATIC_TABLES mode (to do a table at every ROW= seen) and you want all the elements of the TABSPEC to be on for that table.

TABSPEC={ oldname=newname: is not allowed, but you can reference previously defined variables on a keyword= or specific table elements from a previously defined TABSET or TABSPEC by their program-generated varnames.

~END

Exits the program. Required as the last line in all specification files. Causes all file buffers to be flushed and closed correctly. Any commands after ~END are ignored.

Syntax:

```
~END
```

~EXECUTE

~EXECUTE (EXC) is used when accessing data across all cases. Builds tables by assigning variables (see ~DEFINE) to table definition elements or executing procedures across cases.

Syntax:

```
~EXC keywords
```

The keywords can be grouped by function. Action keywords cause tables to be written, etc. Table element assignment keywords assign table elements. Here are the groupings, followed by an overall explanation of table definition elements and then the keywords' explanations in alphabetical order.

Action Commanads:

BUILD_TABLES	PROCEDURE=
CLOSE_PAGE	READ_PROCEDURE=
DROP_CHAIN	RESET
LOAD_TABLE	RUN_CHAIN
MAKE_TABLES	SET
PRINT_ALL	SHOW_CHAIN
PRINT_RUN	STATUS
PRINT_TABLE	STORE_TABLE

Table Element Assignment Commands:

BANNER=	ROW_SHORT_WEIGHT
BASE=	ROW_WEIGHT
COLUMN=	STATISTICS=
COLUMN_SHORT_WEIGHT=	STUB=
COLUMN_WEIGHT=	STUB_PREFACE=
EDIT=	STUB_SUFFIX=
FILTER=	TABLE=
FOOTER=	TABLE_SET=
HEADER=	TABLE_SPECS
LOCAL_EDIT=	TITLE=
RETAB=	TITLE_#=
ROW=	WEIGHT=

These commands assign and/or define the table elements. Table elements are generally first defined in the ~DEFINE block. Once an element is assigned, it stays

in effect until changed, unless purposely dropped due to standard EDIT= options or ~SET command.

You can define any table element from the ~EXECUTE block with the specific keyword as listed above, following the same rules for defining variables in the ~DEFINE block.

Syntax:

```
~EXC table_element= {varname: definition}
```

`Options:table_element`
Any one of the table elements from ~DEFINE TABLE_SET. Table_element= must be followed by colon (:) if the table element is defining text, EDIT, LOCAL_EDIT, or if it is referenced inside a TABLE_SET=, otherwise it is optional. Table elements stay in effect until replaced, or reset to null with either table_element=; or -table_element.

{

Optional left brace for all table text elements, EDIT and LOCAL_EDIT.

`varname`
Optional. The program generates a temporary variable name such as TE00002, but you cannot reference this name later in your spec file.

}

Right brace is required for all table text elements including EDIT and LOCAL_EDIT.

Example:

```
~EXC
HEADER={: =Automotive manufacturer study }
COLUMN= TOTAL WITH[1/23.2#1-10/11-20/21-99]
TITLE={: This is the title of the next table }
ROW= [5^1//5]
LOCAL_EDIT={e: COLUMN_WIDTH=5,PAGE_NUMBER=101 }
TABLE= *
```

```
ROW= SEX WITH AGE (where SEX and AGE are previously defined variables)
COLUMN= col1[5^1//5]
BASE= [5^1]
WEIGHT= wgt1:SELECT_VALUE([5^1/2],VALUES(1.25,.79))
TABLE_SET={ tab1:
TITLE={: Sex of respondent}
STUB={:
Male
Female
ROW=: [7#M/F]
TABLE=*
}
```

Program-Generated Variable Names For TABLE_SET=

When you define a TABLE_SET the program uses its name as the base varname for any table element specified inside the TABLE_SET unless you assign your own names. The varname consists of the TABLE_SET name plus one of the extensions listed below depending on which table element assignment command you are using:

TABLE SET VARNAMES

BANNER=	_bn
BASE=	_b
COLUMN=	_c
COLUMN_SHORT_WEIGHT=	_csw
COLUMN_WEIGHT=	_cw
EDIT=	_e
FILTER=	_f
FOOTER=	_fo
HEADER=	_h
LOCAL_EDIT=	_le
ROW=	_r
ROW_SHORT_WEIGHT	_rsw

TABLE SET VARNAMES	
ROW_WEIGHT	_rw
STATISTICS=	_st
STUB=	_s
STUB_PREFACE=	_sp
STUB_SUFFIX=	_fx
TITLE=	_t
TITLE_2=	_t2
TITLE_4=	_t4
TITLE_5=	_t5
WEIGHT=	_w

Using Previously Defined Table Elements

Table elements can be defined in advance with ~DEFINE TABLE_SET and TABLE_SPECS= and then be called with either ~EXECUTE TABLE_SET= or the TABLE_SPECS=, or they can be defined as a part of the EXECUTE block.

When a table is made, the program reads the data file, builds the table, and prints the table to the screen or to a print file, as specified by the user (see the meta command >PRINT_FILE). The table and all of its elements are stored in the db file opened ReadWrite; these can be accessed at any time. The table elements can be altered by the user (see the meta commands >EDIT and >DB_TO_FILE), and the table can be reloaded into memory and reprinted with new elements without rereading the data if only cosmetic changes are being made (see LOAD= and PRINT_TABLE).

```
Syntax:
~EXC
table_element=varname
...
TAB=T001
```

The table T001 would be built using all the keywords assigned prior to the table. Use table_element=; if you no longer want an element to stay in effect, i.e., to turn off an item (i.e., BASE=;).

Keywords:BANNER= (BAN)

Variable defining the text to print above the columns of the table. Refer to ~DEFINE BANNER= for examples of a banner definition.

BASE=

Base variable (datavar); may be defined right here. Defines which data cases will be used in the table. If the variable has multiple categories, there will be one table made for each category in the base.

In addition, a base variable definition that references multiple data locations (e.g., [27,28,29^1]) will make multiple tables. Using our example variable we would get three tables: one with a base of 27^1, one with a base of 28^1, and one with a base of 29^1.

If there is a base on a table and a TITLE_4 is not defined, “BASE: ” prints on the table with any text from the base variable printing after it. However, no system-generated base labeling will appear when the EDIT statement option -TITLE_4_FOR_BASE is in effect. If TITLE_4 is set, no (extra) indication of the base is printed.

Syntax:

```
BASE= varname or variable
```

```
Example:
~DEFINE base1: $T="males" [5^1]
~EXC BASE=base1
```

In this example, the table will print 'BASE: males' under the table title.

Related Commands:

```
FILTER=
```

BUILD_TABLES (BUILDTAB)

Builds the tables defined with either TABLE_SET= or TABLE_SPECS= in this run by reading and executing the TABSET=name commands in the program-generated TAB file (see ~SPEC_FILES).

This command also generates the LPR file that contains a statement to load and print each table stored in a DB file. This would allow you to run the tables and print them later. The file can be edited to print only certain tables.

This command generates the data in the cells of the table and associates the items described for table formatting, but it does not print the table (see PRINT_ALL or PRINT_RUN for table printing commands).

Related Commands:

```
MAKE_TABLES; ~SPEC_FILES; SET AUTOMATIC_TABLES
```

CLOSE_PAGE (CLOSEPG)

Immediately closes any open printfile page; this is usually used after a reference to an EDIT variable with the LEAVE_PAGE_OPEN parameter set; after printing some tables on the same page, this will close the page. You can optionally use -LEAVE_PAGE_OPEN to close the page after the current table prints.

COLUMN= (COL)

Specifies or defines the data categories to be used constructing the columns (datavar) or horizontal axis of the table. You may include any variable joiners, functions, or cross-case functions.

```
Syntax:
COL= varname or variable or option

Example:
~EXC COL=GENDER
```

If no banner is referenced, this creates the banner variable GENDER_b, which formats the text from the question GENDER over the columns, and prints the title from the GENDER variable in the title, prefixed by “BANNER:”. See ~DEFINE BANNER= for an example of user-defined column labels.

`Options:DENSITY`
Causes a row describing the punches of the data columns referenced in the row description to be built, as well as an additional row of numbers that refers to the number of punches in the data.

Example:
`COL=DENSITY ROW=[1]`

```
     Total  n/a  multi  T/R   A/R  1   2   3    4    5  6   7   8   9   0   X  Y
1/1   50    10    5      60    40   10  15  5    7    8  3   -   -   -   -   1  1
                                    30   8   2
```

The first row describes the actual data; how many responses have column one a one punch, one a two punch, etc. The second row says "30 respondents had one response in column one, eight had two responses, and two had three responses."

HOLECOUNT
Generates a row describing the punches of the data columns referenced in the row description. The row shows the frequency of each punch occurrence.

Example: COL=DENSITY ROW=[1]

```
     Total  n/a  multi  T/R   A/R  1   2   3    4    5  6   7   8   9   0   X  Y
1/1   50    10    5      60    40   10  15  5    7    8  3   -   -   -   -   1  1
```

The row describes the actual data; how many responses have column one a one punch, column one a two punch, etc.

Refer to your *UTILITIES* manual for information on the menu-assisted HOLE utilities.

COLUMN_SHORT_WEIGHT= (COLSHORTWT)

Works the same as COLUMN_WEIGHT, but if there are fewer weights specified than there are banner points, the last weight specified stays in effect for the remaining columns.

COLUMN_WEIGHT= (COLWT)

Applies the defined weights to the individual table columns. This may be defined right here. There must be as many weights specified as there are banner points; otherwise see COLUMN_SHORT_WEIGHT.

Syntax: COLWT= varname or variable

Example:

~DEFINE WT: [10.5*F2] WITH &

SELECT([5^1/2],VALUES(1.2,.79)

~EXC COLWT=WT

This shows a two weight variable; the first weight is taken from data columns 10 to 14 with an implied two decimal significance; this weight will be applied to the first column in the table. The second weight is based on whether column 5 has a 1 or a 2 punch; if there is a 1, the value 1.2 is used for the second column; if there is a 2, the value .79 is used.

In the above example, the column variable *must* contain only two categories.

Column summary cells are not weighted.

DROP_CHAIN (DROPCH)

Drops (or deletes) the table specifications for the tables that are currently stored in memory but have not been built.

EDIT=

Specifies either the definition or the name of the variable (previously defined with ~DEFINE EDIT=) to control the printing of tables, such as the column and stub width, number of decimals in percents, etc.

FILTER= (FIL)

Specifies or defines the table filter (datavar). Like BASE=, it defines which cases will be allowed into a table. FILTER is the same as a base except that multiple categories do not generate multiple tables; if a case passes any category of the filter it is included in the table. Note that ~SET DROP_BASE has no effect on a filter. FILTER is encouraged over the use of the SELECT option on the ~INPUT statement because it is faster.

If there is a filter on the table and a filter title is not defined, a system-generated "FILTER:[filter datavar]" prints on the table in the TITLE_4 position. No system-generated filter titling will appear when the EDIT= option TITLE_4_FOR_BASE is in effect. Also if a TITLE_4 title is specified, then no system-generated titling is printed.

Syntax: FIL= varname or variable

Related Commands: BASE=

FOOTER= (FOOT)

Text to be printed at the bottom of subsequent pages. The footer is defined with either ~DEFINE LINES= or FOOTER=.

HEADER= (HEAD)

Text to be printed at the top of subsequent pages. The header can be defined with either ~DEFINE LINES= or HEADER=.

LOAD_TABLE= (LOAD)

Loads the specified table and its elements from a db file into memory. You may change the parameters for printing the table after it is loaded (e.g., DO_STATISTICS on the EDIT statement for the final printing).

Syntax: LOAD= tablename or *

LOAD=* loads the table associated with the current table name, that is, the last table made or the table with the current ~SET TABLE_NAME=name.

LOCAL_EDIT= (LOCEDIT)

Allows you to set edit parameters to be added to or to override the EDIT= statement. This overrides any conflicting commands between the EDIT= and the LOCAL_EDIT=. Turn off with LOCAL_EDIT=;, assign a new LOCAL_EDIT variable, or have ~SET DROP_LOCAL_EDIT on. This is typically used for table-specific EDIT options such as ranking or summary statistics calculated at printing time.

MAKE_TABLES (MKTAB)

Builds and prints all of the tables defined with either TABLE_SET= or TABLE_SPECS= in this run by reading and executing the program-generated TAB and LPR files (see ~SPEC_FILES). This single command performs the function of the ~EXC commands BUILD_TABLES, RESET, and PRINT_RUN.

PRINT_ALL (PRTALL)

Print every table in an open db file from the beginning table to the current table, in ASCII name order. The beginning table can be set with ~SET BEGIN_TABLE_NAME=, and the current table can be set with ~SET TABLE_NAME=.

If you change table names so that they are not executed in ASCII order, they will not print in the order they were executed in. In this case, you should use the PRINT_RUN command, or LOAD_TABLE= and PRINT_TABLE. For example, if your tables names are T010, T010a, T010b, T011 then PRINT_ALL will print them in the order of T010, T011, T010a, T010b.

PRINT_RUN (PRTRUN)

Prints the tables made in the current program run, in the order made. It requires that the program-generated LPR file be available (see ~SPEC_FILES). PRINT_RUN executes this file to load and print each table.

This only prints tables made in the current run with STORE_TABLE=, or ~SET AUTOMATIC_TABLES and ROW=, but unlike PRINT_ALL you can use any names you like on the tables.

Use ~SET TABLE_NAME= and PRINT_ALL to print all tables in specific name order, regardless of when they were made. Use LOAD_TABLE=, then PRINT_TABLE to print specific tables.

Related Commands: MAKE_TABLES

PRINT_TABLE (PRT)

Prints the last table stored in memory by LOAD_TABLE=, STORE_TABLE=, or TABLE=. This command is included in the program-generated LPR file for each table that was stored.

Related Commands: BUILD_TABLES, PRINT_RUN, PRINT_ALL, MAKE_TABLES, ~SPEC_FILES.

PROCEDURE= (PROC)

Executes the procedure with the name specified, across the cases of the open data file. There must be an output file and a ~CLEANER WRITE_CASE command to save changes in a new file, or ~INPUT ALLOW_UPDATE and ~SET PRODUCTION_MODE to update the input file.

Syntax: PROC= procname options

Options:

ON <study code>

Executes a procedure on any open data file. The <study code> is the file name, or if you specified ~INPUT SAMPL,STUDY_CODE=mine, then "mine" is the study code for the input file.

MULTI

Allows parallel processing of multiple data files. This would be useful, for instance, if you were trying to merge data from two files into one, or act on files based on information in other files.

The 'MULTI' keyword gives you control of which file will be read next and which case in the file will be read. Once the EOF_DATA has been reached for a particular file, no more processing of that file will be done, until a NEXT "caseid" or NEXT FIRST command is seen.

An example of simple parallel processing would be where three data files, each having exactly the same number of records, are being processed simultaneously.

If you expect that one or more of the input files used will reach its end-of-file and need to be rewound to the first record, do not use the MULTI option.

When a procedure is started using the MULTI option, the first record of each open input file is read. If you do not have a CHOOSE_FILE "datafile" NEXT sequence somewhere in the procedure, no other records will be read. This is in contrast to using "PROCEDURE=procname ON datafile" where a record from the primary file is read in automatically at the start of the procedure and each time the end of the procedure is reached. Records from other open input files are read using CHOOSE_FILE "datafile" and NEXT.

When reading from more than one input file where there are there are different numbers of records in each file, the procedure needs to check for an end-of-file before attempting to execute a command that requires a case in memory. Check for end-of-file in a particular data file by using one of the following conditional statements. EOF_DATA is a CfMC System constant meaning 'at the end of the data file'. *(See the example on the next page.)*

Example:

```
IF data file^EOF_DATA THEN ...(if user DB file(s) are open)
IF data file!EOF_DATA THEN ...(if no DB file(s) are open)

IF NOT data file^EOF_DATA THEN ... (if user DB file(s) are open)
IF NOT data file!EOF_DATA THEN ... (if no DB file(s) are open)
```

Once the EOF_DATA is reached in any of the data files, the CHOOSE_FILE "datafile" NEXT sequence has no effect when using PROCEDURE=MULTI. An error is generated if the procedure attempts to execute on an input file whose EOF_DATA as been reached.

PROCEDURE=proc1 MULTI will only read one respondent from each file if there are no NEXT statements in the procedure.

Example:

```
~INPUT pmula, NUMBER_INPUT_BUFFERS=3, ALLOW_UPDATE
~INPUT pmulb, NEW_BUFFER,ALLOW_UPDATE
~INPUT pmulc, NEW_BUFFER,ALLOW_UPDATE
~DEFINE
 PROCEDURE= {proc1:
  BLANK pmula![1.5$]
  BLANK pmulb![10.5$]
  BLANK pmulc![15.5$]
 }
~EXECUTE PROCEDURE=proc1 MULTI
~END
```

Input files have exactly the same number of lines. The procedure will run on all records of all the data files.

Example:

```
~INPUT pmula,NUMBER_INPUT_BUFFERS=3,ALLOW_UPDATE
~INPUT pmulb,NEW_BUFFER,ALLOW_UPDATE
~INPUT pmulc,NEW_BUFFER,ALLOW_UPDATE
~DEFINE
 PROCEDURE= {proc1:
  BLANK pmula![1.5$]
  BLANK pmulb![10.5$]
  BLANK pmulc![15.5$]
  CHOOSE_FILE "pmula" NEXT
  CHOOSE_FILE "pmulb" NEXT
  CHOOSE_FILE "pmulc" NEXT
  }
~EXECUTE PROCEDURE=proc1 MULTI
~END
```

Input files with differing numbers of records.

Example:

```
~INPUT pmula,NUMBER_INPUT_BUFFERS=3,ALLOW_UPDATE
~INPUT pmulb,NEW_BUFFER,ALLOW_UPDATE
~INPUT pmulc,NEW_BUFFER,ALLOW_UPDATE
~DEFINE
 PROCEDURE= {proc1:
  IF NOT pmula!EOF_DATA THEN
    BLANK pmula![1.5$]
    MODIFY pmula![6]=pmula![6]+1
  ENDIF
  IF NOT pmulb!EOF_DATA THEN
    BLANK pmulb![10.5$]
    MODIFY pmulb![16]=pmulb![16]+1
  ENDIF
  IF NOT pmulc!EOF_DATA THEN
    BLANK pmulc![15.5$]
    MODIFY pmulc![20]=pmulc![20]+1
  ENDIF
  CHOOSE_FILE "pmula" NEXT
  CHOOSE_FILE "pmulb" NEXT
  CHOOSE_FILE "pmulc" NEXT
 }
~EXECUTE PROCEDURE=proc1 MULTI
~END
```

Related Commands: ~DEFINE PROCEDURE=

READ_PROCEDURE=<name>

Reads the data under the control of a procedure when building tables. This allows modifications to the data in the buffer being read without actually changing the input or output data files.

You must specify the ~CLEANER command DO_TABLES in the procedure when you want the tables to be accumulated, usually at the end of the procedure.

The example of READ_PROCEDURE below creates overlay tables. It reads data under the control of a procedure when it builds tables. This allows modifications to the data in the buffer being read without actually changing the input or output data files.

Example:

```
~EXEC READ_PROC

>DEFINE @STUDY chris
~input @STUDY

~SPEC_FILES @STUDY
>PRINT_FILE @STUDY
>USE_DB @STUDY

~DEFINE
TABSET=TOTAL:
 EDIT=: -COLUMN_TNA, -ROW_NA, PERCENT_DECIMALS=2,-
PERCENT_SIGN }
 COL=: [10^1//5]
 ROW=:[2/20.2#1//50] }

"DEFINE OUR READPROC PROCEDURE
 PROCEDURE= { READER:
  DO_TABLES  LEAVE_OPEN "do tables one time, be ready to
"accumulate at end of procedure
"leave table open

  COPY [2/20.2] = [3/20.2] "copy record three's data into
"record two (where tables are
```

```
"being calculated from)

 DO_TABLES LEAVE_OPEN

 COPY [2/20.2] = [4/20.2]  "copy record four's data into
                    "record two (where tables are
"being calculated from)
 DO_TABLES } "do tables one last time

~SET AUTOMATIC_TABLES
~EXECUTE

 READ_PROCEDURE=READER

MAKE_TABLES

~END
```

RESET

Turns all table elements off. This includes the page number. This is usually specified prior to PRINT_ALL or PRINT_RUN to ensure that no conflicting table elements are still in effect from the table building phase.

Related Commands: MAKE_TABLES

RETAB=

Gets all the elements from a table previously made and loads them for subsequent tables. This is similar to TABLE_SET=oldTABLE_SET, except it gets the elements directly from the stored table previously made, not from the original TABLE_SET elements.

Syntax: RETAB=table name

ROW=

Specifies or defines the row variable (datavar) to be used as the vertical axis of the table. If you're using ~SET AUTOMATIC_TABLES, you must have the ROW= specified as the last element of the table, as that will trigger the making of the table, not a TABLE= or STORE_TABLE= name.

Syntax: ROW= varname or variable

ROW_SHORT_WEIGHT= (ROWSHORTWT)

Defines weights to be applied to the rows of the table. If there are fewer weights specified than there are rows, the last weight specified stays in effect for subsequent rows.

Syntax: ROWSHORTWT= varname or variable

ROW_WEIGHT=

Same as ROW_SHORT_WEIGHT, but there must be a matching number of weights and rows.

Syntax: ROWWT= varname or variable

This does not weight the summary row.

RUN_CHAIN (RUNCH)

Causes any tables stored since the last pass through the data to be built (executes a table pass). This is useful if you wish to do tables immediately to leave more space available for the next table.

Use this command if you need to switch to a new input file during the table run and you are using ~SET AUTOMATIC_TABLES. This forces the program to build the tables from the last pass with the input file last opened. See the example under ~SET AUTOMATIC_TABLES.

SET

Allows you to use any ~SET parameters without leaving the ~EXECUTE block, usually to set table-building related elements. Refer to ~SET for a complete list of SET commands.

Related Commands: ~CLEANER DO_SET and SET.

SHOW_CHAIN (SHOWCH)
Displays the table specifications for the tables that are currently stored in memory but have not been built.

STATISTICS= (STAT)
Assigns the named statistics variable (defined using ~DEFINE STATISTICS=) to the table.

STATUS
Displays the variables assigned to the various table definition elements.

Related Commands: ~CLEANER STATUS,~DEFINE EDIT=STATUS,~SET STATUS

STORE_TABLE=

Stores the specifications for the table named in memory until memory is full or RUN_CHAIN or RESET are invoked. Then the data is read and the tables stored in memory are accumulated and stored in the db file. Unlike TABLE=, tables are not automatically printed when using STORE=; you must use a printing command (LOAD_TABLE= and PRINT_TABLE, PRINT_ALL, or PRINT_RUN).

Syntax: STORE= tabname or *

STORE=* will store the current table.

STORE will process a large number of tables faster, because it will process all the data in the tables first and then print them. If you use TABLE=, Mentor will process and print one table at a time.

Related Commands: BUILD_TABLES

STUB=

Text used to label rows. Must have been defined as ~DEFINE STUB=. If a STUB is not specified, a default STUB is created from the ROW= variable.

Example: ROW=INCOME

If a STUB= is not specified, this would create a stub variable INCOME_s, which contains the category text from the question INCOME. This category text would print as the stub labels for each row on the table.

If your row is defined as a data variable, the data categories would print as stub labels.

Example:`ROW=[6^5/4/3/2/1]`

would generate the stub labels:

6^5;6^4;6^3;6^2; and 6^1.

You could attach text to each category to print as the stub label. See ~DEFINE VARIABLE=.

STUB_PREFACE= (STUBPREF)

Specifies a stub defined in ~DEFINE STUB= to be used at the top of the STUB for all following tables. This can also be controlled from the ~DEFINE EDIT= statement.

Syntax: STUBPRF= name or NONE or TNA

STUB_PREFACE= TNA is the default which controls printing of the Total and No Answer rows. STUB_PREFACE=NONE forces you to control the printing of the Total and No Answer rows in your stub.

This is the preferred way to assign a STUB_PREFACE or STUB_SUFFIX in a ~DEFINE TABLE_SET definition. See the detailed explanation in the note under ~DEFINE EDIT=STUB_PREFACE=.

STUB_SUFFIX= (STUBSUF)
Specifies the name of a standard stub set to be used at the bottom of the STUB for all following tables. Can also be controlled from the ~DEFINE EDIT= statement.

Refer to the note under STUB_PREFACE.

Related Commands: STUB_PREFACE

TABLE= (TAB)
Causes the program to build a table using the current table elements, then print the table immediately. Use STORE_TABLE= to increase speed; it does multiple tables at the same time. The table is stored if a ReadWrite db file is open.

Syntax: TAB= tabname or *

TABLE=* will build a table using the incremented current table name.

TABLE_SET= (TAB_SET)
Specifies the name of an entire set of table elements as defined with ~DEFINE TABLE_SET=.

Related Commands: TABLE_SPECS=.

TABLE_SPECS= (TABSPEC=)
Specifies the name of an entire set of table elements as defined with

~DEFINE TABLE_SPECS=

Related Commands: TABLE_SET

TITLE=

Table title (text variable) which prints under any table heading specified but above the banner text. A title might explain the variables used to cross-tabulate the columns and rows. Specifying a title suppresses the BANNER: and STUB: wording on a table. TITLE=; turns off the title for subsequent tables. If you want to use a title once and then drop it, use ~SET DROP_TITLE.

Related Commands: ~SET DROP_TITLE

TITLE_2= (T2)

Additional titling. Title_2 text prints between the table name line and the text line generated by the TITLE=. To no longer include the title_2 text on subsequent tables, use `TITLE_2=;`

Related Commands: ~SET DROP_TITLE_2

TITLE_4= (T4)

Additional titling. Title_4 text prints between the text line generated by the TITLE= and the banner. To no longer include the title_4 text on subsequent tables, use `TITLE_4=;` .

Related Commands: ~DEFINE EDIT= TITLE_4_FOR_BASE, ~SET DROP_TITLE_4

TITLE_5= (T5)

Additional titling, printing once after the last line of a table (footnote). This is useful for customized footnotes, for example, to provide additional information about the statistical tests that were run on the table. To no longer include the title_5 text on subsequent tables, use `TITLE_5=;` Use BLANK_LINES_BEFORE_T5 to put blank lines between a table's body and the title_5 text.

Related Commands: ~DEFINE EDIT= BLANK_LINES_BEFORE_T5=, ~SET DROP_TITLE_5

WEIGHT= (WT)

Weight variable (datavar) to assign a global weight to the table; may be defined right here. The system columns and rows are weighted along with the body of the table. Use the ~DEFINE EDIT= option UNWEIGHTED_TOP or the STUB= option [PRINTROW=Ux] to print the unweighted Total or No Answer on a weighted table.

Syntax:WT= varname or variable

Related Commands: ~EXECUTE COLUMN_WEIGHT and ROW_WEIGHT

~FREAK

Generates multiple stratified frequency counts. It produces a standard format report that includes counts, number of cells per table, percents, cumulative percents, and

sub-totals when doing variables within other variables. ~FREAK will also do all frequency tables in one pass through the data, instead of requiring an additional pass for each count as ~FREQUENCY does. You can also use the FREQ utility to access ~FREAK, it is the "Do QUICK TABLES" option on the PRINT OPTIONS menu, see the *Utilities* manual for details.

This is a port of the utility FREAK.

Syntax: ~FREAK
TABLE <tablename>=variable <WITHIN variable><WITHIN variable>
keywords

At a minimum, you need to provide one variable name. The rest is optional.

Example:~FREAK
TABLE=[11.4$]

Here is a ~FREAK frequency count on one string variable:

first -COUNT- -% OF TOTAL- CUM. %

There are 3 cells in this table

AA	3	30.0	30.0
BB	6	60.0	90.0
EE	1	10.0	100.0
--	10	100.0	

Countwidth Option:

The ~FREAK option of countwidth allows numbers wider than five digits to print.

Example:

```
~freak
countwidth 10
table *=[1$]
```

Options:tablename

This is standard Mentor table name, 1-14 characters, starting with a letter.

variable
How the data is reported is dependent on the type of variable specified.
For VAR or TEXT type data, the report is sorted in standard ASCII sequence, and the output is truncated after 20 characters. With FLD (#) or CAT (^) data types, the output is sorted in category order, followed by any data that did not fit into the field. If the data is numeric, the output is sorted in numeric order with the exception codes listed last. See the keywords below for control of missing numeric data.

WITHIN variable
Use WITHIN to cross variables. You can combine string and numeric variables.

Example:

```
~FREAK
TABLE=SEX WITHIN AGE
```

Here is ~FREAK frequency count one two variables (the numbers that are offset are the subtotals for the category):

first	second	-COUNT-	-% OF TOTAL-	CUM. %
(6 cells in table)				
AA	BB	1	10.0	10.0
AA	CC	2	20.0	30.0
AA		-- 3	30.0	
BB	BB	2	20.0	50.0
BB	CC	2	20.0	70.0
BB	DD	2	20.0	90.0
BB		-- 6	60.0	
EE	EE	1	10.0	100.0
EE		-- 1	10.0	
		-- 10	100.0	100.0

Keywords:PRINT_MISSING_TYPE

For numeric variables, it sorts missing data into various classifications. Mutually exclusive with PRINT_MISSING_RAW_DATA.

PRINT_MISSING_RAW_DATA

For numeric variables, it prints the characters that are classified as missing. Mutually exclusive with PRINT_MISSING_TYPE.

Case Sensitivity

UPSHIFT is on by default and tells the program to treat all characters as upper case, therefore “A” and “a” are counted in the same category. If you want your count to be case sensitive, use -UPSHIFT before indicating variables. This will create separate categories for “A” and “a.”

Example: `~FREAK`

```
-UPSHIFT
TABLE=[1.2$]
```

Memory Issues

~FREAK will only run as many tables as it can fit in the available memory. To run more tables, set CORE to a higher number. See the *Utitlites manual, Appendix D* for a discussion of the command line keyword CORE.

Generating Multiple Frequency Tables from Survent Variables

Here's how to make a report for all the variables in a Survent questionnaire:

`~INPUT STUDY`	Open Input data file
`>USEDB STUDY`	Open DB file with variables
`>LISTDB ,freq^dcl,SORT=QQNUM,&`	List vars in question order
`TEMPLATE="TABLE=!"`	Create lines to make tables in list
`~FREAK`	Start the report
`&freq^dcl` lines	Read in file of TABLE=<var>
`~END`	Finish the report

You can use `SORT=LOCATION` instead of `SORT=QQNUM` to produce the tables in the order the variables were put into the DB file (this is for spec files created in the Mentor ~DEFINE block.)

There are three other features of the >LIST_DB command that you can use to generate a subset of the questions in your ~FREAK frequency report:

1 To subset the variables to do by question type, use the "TYPE=VAR"

LISTDB parameter. If you used

```
>LISTDB,freq^dcl,TYPE=VAR=5
```

in the above example, you would only get NUM questions. The default is all question types. The variable types are as follows:

TYPE=VAR=	1:	Variable questions (strings)
	4:	Field questions (code lists)
	5:	Numeric questions (numbers)
	6:	Category questions (punch code lists)
	7:	Text questions (long open-end strings)

2 To get a range of questions to do, (eg. questions between question 3 and question 10) from a Survent questionnaire, use the following syntax:

```
>LISTDB,freq^dcl,sort=qqnum(<first questn>-<last questn>)
```

Where "<first question>" is the name of the question to start at, and
"<last question>" is the name of the question to end at.

3 To get frequency tables made on variables with certain names, use the

PATTERN= feature of the LISTDB statement:

```
>LISTDB,freq^dcl,PATTERN=(oe*,age,income,comment*)
```

This would get only the variables whose labels matched one of the pattern references. "oe*" returns all variables starting with "oe", which you could use as the default name on your open-ends. "age" will only get the question "age", same for "income". "comment*" would get all variables
whose label starts with "comment".

All of these features can be combined. For instance, you can get all

VAR and FIELD variables between questions 3 and 10 whose name started with "oe" by saying:

```
>LISTDB,freq^dcl,TYPE=VAR=14,SORT=QQNUM(Q3-Q10),PATTERN=(oe*)
```

~FREQUENCY (FREQ)

Generates a report on the number of items in a specified location or for a variable. This report is referred to as a frequency count. There is menu-driven FREQ utility that you can use to generate frequency counts as well, see the *Utilities* manual.

Syntax: ~FREQ datavar =-newdatavarname $T="title"

At a minimum, you need to include a printfile name, a FREQ command(s) and the name of the data file.

Example:

```
>PRINTFILE frqrpt
~INPUT data1
~FREQ [7.7$]
~FREQ [14.3]
~END
```

Options:datavar

Variable that references data, can be a variable name, QQ# or data location. The variable can be a tabset. The tabset must have a title, row and stub.

How the data is reported is dependent on the type of variable specified. With ASCII ($ or L), text string ($T) or punch ($P) variables, FREQ generates a count of the number of good ASCII and bad ASCII items, then each item is listed separately with the number of each item.

A numeric variable will list how many of the items fall into the following categories: a good number; missing, not in RNG CHECK; valid ASCII code; blank field; and, bad number. Then it will list the number of cases, number of blanks, and number of categories (or items) for each variable.

If the datavar has categories such as FLD (#) or CAT (^) types, FREQ will generate a report on the items that do not fit into the categories; use ~EXECUTE if you want to generate counts on data that fits into the categories.

newdatavarname

Optional. This creates a variable that references a tabset that contains a row and stub matching the frequency distribution. You can use this new variable to, for example, run it against a banner. This option cannot be used with punch [$P] variables. If you put a minus sign(-) in front of datavarnew, the display of frequency count will be suppressed.

Example: `~FREQ [8$] = -state01`

$T="title"

Optional. This creates a title for the new variable.

To get a sum of counts across fields, use *L.

Example: ~FREQ [10,13*L$]

To get a net of counts across fields, use *F.

Example: ~FREQ [12,15,17*F$]

By default, FREQ is not case sensitive and will put the strings TEST, Test and test in the same category. To turn case sensitivity on, use the ~SET CASE_SENSITIVE command. You can affect how a string is displayed by using the command and the U, and D variables to upshift and downshift the string (see *Mentor Volume I: "Changing Case"*).

Example: `~FREQ [1$U]` will display `"Count for 'A' is 1"`

`~FREQ [1$D]` will display `"Count for 'a' is 1"`

~GO_TO

Moves forward to another tilde command. Combined with an IF statement, you can conditionally execute another tilde command.

Syntax:~GOTO (label1, label2,...,labeln) expression

Options:

label

has been defined on the ~command you want to go to. You can do this for any tilde command. The GOTO label must be in the format `~label:command`.

Example: ~GOTO A
. . .
~A:EXECUTE

This example will force Mentor to move to the ~EXECUTE block and continue processing from that point until it encounters the end of the spec file or another ~GOTO command.

Related Commands: ~CLEANER GOTO

~HOLD_OUTPUT_UNTIL_SUBSET

This command ensures that the ~input file is read only twice for a group of ~outputs rather then potentially twice for each ~ouput. This considerably speeds up the subsetting of a large input file into many output files. The default for this option is "on". When this option is "on" the "case_written" status of any particular record cannot be ascertained during the select scanning process. This means that a case slated to be written may later no longer pass the select being used because the case was written by an earlier output.

You can set -HOLD_OUTPUT_UNTIL_SUBSET to make the written/not written state of each case be determined in a linear fashion. Generally speaking using

more than one not (casewritten) in a subset block is likely to not generate the expected sample, and often will lead to an error.

~INPUT (IN)

Opens a data file or files. Any output files or prior input files will be closed, unless the NEW_BUFFER option is used. ~INPUT with no options will close an open data file. As the data file is read, Mentor prints a message to the screen if the file contains deleted cases.

~Input files can now point to directories with dashes in the name.

You can now use the ~input files= option to point to a directory name with a dash in it. Most other characters (except dot and underscore) are unacceptable.

Example:

~input tmp-1/mt3875.asc ascii length=80 ;

Meaning of "?" character changed when specifying file patterns. Previously, the "?" referred to letters only when doing file patterns, but the operating system used the character for both letters and digits. Now the ? works the same as the operating system, retrieving files with letters and digits in the position specified. The # still means digits only and the @ or * means any character.

~input files=f?

This was used to retrieve files named fa, fb, etc. but not f1 or f2. Now it retrieves all of these.

Syntax: ~IN filename,format,keywords

Specify keywords one after the other, separated by a comma or a space, in any order except JOIN, which must be the last option specified (see below).

Options:filename

The data file name, the TR file extension is assumed. A dollar sign($) before the file name means the TR extension is not required. (Use the meta command

>-CFMC_FILE_EXTENSIONS to make $filename the default.)

Example: `~IN $data1`

*

You can use an asterisks instead of a filename to get the name from a previous >STUDY_NAME meta command

Example: `~IN *`

You can use a dollar sign instead of a filename to open a dummy (phantom) file; the default file length is 800 columns. A phantom file appears to the program as a data file with a single blank case in it. It is useful for testing procedures (that do not require data), setups where you do not have data yet, or table manipulation runs that need db files, but not data.

Example: ~INPUT $,WORK_LENGTH=800,STUDYNAME=family

If you use NULL as the filename, an immediate end-of-file is read and the file will not be saved to disk when you write to it. The default NULL file TOTAL_LENGTH is 6250. Unlike $, NULL does not create a data case.

Example:

~IN NULL,ASCII=80,STUDY_NAME=MT0379a,NUMBER_INPUT_BUFFERS=2

[MPE Users: An asterisk (*) before the filename means substitute the name for a file equation. For more information see *Appendix D: CfMC CONVENTIONS, Command Line Keywords, LISTFILE: <listfile>, option1...* in the *Utilities* manual.]

The ~input line will allow subdirectories and use of "?", "#", and "@". You can now specify patterns with files in subdirectories such as:

Example:

~input files="test/file#" ascii=80

To find all files in the subdirectory "test" with names like file1, file2, and file3. Previously you had to give a fully qualified directory name if you wanted to access files in other directories, and you could only read files in one subdirectory at a time. The "#" means "any number", "?" means "any alphanumeric character", and "@" means the same as "*" which is "any matching pattern".

format

Optional. The following options specify the format of the input data file. Only one of those listed can be used at a time. These options are meant for quick runs without translating your data file into a CFMC System file (see the ~OUTPUT option WRITE_NOW). If you plan on using the file as input several times, it is more efficient (i.e., less CPU intensive) to translate the file and use the translated file as input. A list of formats follows.

ASCII= LENGTH: IDcol.wid

Standard text type data format. Multiple punches are not kept, only ASCII codes (A-Z, 1-9,0, and other standard characters). If you do not include the location of case IDs with LENGTH=###:#.# or ID=#.#, records will not have case Ids.

Example: `~INPUT ASCII LENGTH=132  ID=1.4`

BINARY= LENGTH: IDcol.wid (BIN)

Standard multi-punch format. Records must be fixed length with 80 columns per record. This format is used on most micro-computers. IDcol.wid is optional, and defaults to 1.10.

FONE_FORMAT (FON)

Standard CfMC FONE files from Survent.

SWAPPED_BINARY=LENGTH: IDcol.wid **(SWAPBIN)**

Binary with each 2 bytes exchanging position (swapped). This is standard IBM 360 column binary format, and is used on most mini-computers and mainframes. IDcol.wid is optional, and defaults to 1.10.

UNCOMPRESSED (UNPRESS)

Translates the file in uncompressed format.

Keywords:ALLOW_NEW (NEW)

Allows use of the ~NEW or ~CLEANER NEW command to create and then append a new case to the end of the data file.

ALLOW_UPDATE (UPDATE)

When no output file is open and data is modified, this option allows the changed case to be written to disk, overwriting the original, when the ~UPDATE or ~CLEANER YES_UPDATE command is used or a procedure is executed. Phantom files always have this option on. This option should not be used with variable length ASCII files.

Related Command:~CLEANER FILE, ~SET PRODUCTION_MODE

BACKUP (BUP)

Makes a copy of the original file, deleting any cases marked for deletion. Case IDs are listed and copied as the backup is made. The file currently opened (to be worked on) is the copy, not the original file (which now has a TRX extension).

CARDS_IMAGE (CARD)

Indicates a standard text type data format in 80 columns. This is the same as using ASCII=80.

CASE_LOCATION= col.wid (CASELOC)

Specifies the starting column location and width of case to be loaded into the case buffer. The starting column defaults to 1. This could start anywhere in the TOTAL_LENGTH area, as long as it is within the total length specified. If CASE_LOCATION is not specified, the file is set to the WORK_LENGTH or TOTAL_LENGTH previously specified.

CASE_SENSITIVE= (CASESENSITIVE)

This allows the system to better handle long or case-sensitive paths and filenames. The software can read all filenames and paths regardless of the length or case (Use " "s around filenames with special characters or spaces). The program will lowercase all input and output names you specify unless you use ">CASESENSITIVE", in which case the program will look for and write names exactly as specified.

In ALL cases, lowercase file extensionsare output, unless otherwise specified. It is assumed that setenv ENVIRONMENT variables are uppercase unless you specify ">CASESENSITIVE_ENV", in which case the program pays attention to the exact case.

COMMENT= “file comment” (COM)

Allows up to a 21-character data file comment.

CREATE (CR)

Creates a data file (i.e., ~INPUT sample, CREATE would create sample.TR). You must specify a length for the new file (TOTAL_LENGTH=#). This option will give you a new case when used on the ~CLEANER FILE command.

DATA_LAYOUT_MISMATCH=ERROR/WARN/OK

When reading delimited data files, CfMC adjusts the lengths to match the longest item it finds in any field, then writes the data to the fixed format file using those lengths. It also assigns the type of variable as "string"or "numeric" depending on whether the data is all numeric.

At times you will need to read delimited files more than once to update a file. Hopefully, you will know in advance the length to assign to the fields. If you do, you can build a record that will later be discarded but that has all the lengths and types of data that you want. For instance:

Gender,Age,Income

X,999,999999

The second record reflects the maximum values you will ever need (in the case of numeric data), and uses CHARACTER data where you expect character data (eg. the X). In this way, as you add data, it will always fit into your data parameters.

Once you have built a file, you can guarantee that data will match by saving a "data map control" element. The two most common forms of mismatch are a field in the new data being longer than what's allowed in the map in which case the field will be truncated and a mismatch because of the variable type. For example, the map has a field specified as numeric and the data for that field contains some alpha entries.

If there is some data out of range or of the wrong type, and you save a delimited Map to check the data with, by default it will produce an error message if it sees an invalid entry. Otherwise, you can tell it to just truncate long entries or allow data of a type that doesn't match.

The first time you create the file, specify something like the following:

>createdb myfile.db

~input newfile.dlm delimiter=comma delimit_map save_delimit_map=mymap

~output newfile.tr writenow ~end

This will create the data file "newfile.tr" with locations as specified in the .db file entry "mymap".

The next time you go to add data to newfile.tr, you would use the saved map to make sure the new data matches the old:

```
>usedb myfile.db
~input newfile2.dlm delimiter=comma use_delimit_map=mymap
    delimited_data_layout_mismatch=error
~output newfile maybecreate writenow
~end
```

This will append the data to newfile, but ONLY if the data maps match. If you want the data to be truncated (if the field is too long in the new file) or to allow string data in a numeric field, use the WARN or OK options. When OK is set and a mismatch occurs you will still get one warning so that the problem doesn't go by completely unnoticed.

DELIMITER=comma/tab/other character

Reads delimited data (such as output from Lotus 1-2-3 or Microsoft Excel spreadsheets). Delimited input files must be ASCII with only one record per case. Only one file may be read at a time.

Example: `~INPUT filename.asc ASCII=80 delimiter=comma`
Blanks may *not* be used as a delimiter.

When a map of the data is created (see DELIMIT_MAP), fields from the data file are named Field_1, Field_2, etc., but these names cannot be used as variables to reference the fields. You must either use variable names that are in the first record of the delimited file (see DELIMIT_NAME_FIRST), or create your own variable names (see DELIMIT_VARIABLE_NAME_REFERENCE). Once you have variable names assigned, you can reference them individually, merge a group of fields in to an existing data file, or, for example, run ~FREQ on individual fields. These variables may be stored in an existing open DB file, or used in a temporary local DB file.

Related Commands:

DELIMIT_DROP_FIRST, DELIMIT_MAP,
DELIMIT_NAME_FIRST, DELIMIT_QUOTE_ALPHA,
DELIMIT_VARIABLE_NAME_PREFIX,
MAX_DELIMIT_FIELDS, SAVE_DELIMIT_NAME,
USE_DELIMIT_NAME.

DELIMIT_DROP_FIRST (DELIMDROPFIRST)

Skips the first record of a delimited data file. This is useful when the first record in the file is a list of field names, rather than data. If you wish to use these names as variables in order to reference individual fields, use DELIMIT_NAME_FIRST.

DELIMIT_MAP (DELIMMAP)

Creates a file that contains a map of the data in a delimited ASCII input file. It includes the data location of each field, the field width, type of data in each field (alpha or numeric), and statistics about numeric fields.

DELIMIT_NAME_FIRST (DELIMNAMEFIRST)

Takes the first record from a delimited ASCII input file as a list of variable names for each field. This record should be a list of legal Mentor variable names separated by the file delimiter. This option automatically skips to the second line to read data, so there is no need to use DELIMIT_DROP_FIRST with it. This option overrides DELIMIT_VARIABLE_NAME_PREFIX.

DELIMIT_QUOTE_ALPHA (DELIMQUOTEALPHA)

Treats all fields in quotes as text and prints them left justified (default). Setting - DELIMIT_QUOTE_ALPHA will allow Mentor to recognize a file composed of numbers in quoted fields and print them right justified. This is especially useful when the numeric fields include leading blanks.

DELIMIT_VARIABLE_NAME_PREFIX (DELIMVARNAMEPREFIX)

Creates variable names for fields from a delimited ASCII input file. This allows you to reference individual fields, and merge them into an existing data file, or, for example, run ~FREQ on certain fields.

Example: `DELIMIT_VARIABLE_NAME_PREFIX=var`

This would give the fields the variable names of var01, var02, etc. (or var1, var2, etc. if there are less than ten fields). These variables may be stored in an open existing DB file, or used temporary local DB file.

DOTS=# (DOT)

When reading the file, print a dot on the screen every # cases, and print a message (# cases per dot) and repeat it for each pass through the data file. The default is one dot per 100 cases.

DROP_BLANK_LINES (DROPBLK)

Drops any blank lines in an ASCII input file.

DROP_CHANGES (DROPCHANGE)

Means the CfMC System (TR) file will not be updated regardless of other settings (i.e., update mode, ~CLEANER FILE, or the status of ~CLEANER NO_UPDATE or YES_UPDATE). This allows you to manipulate the data, but safeguards against changing the original data file. An example application might be running weighted tables. You would write your procedure to weight the data for the table run, but the weighted data fields would not be written back to disk.

EXCLUSIVE (XCLUSIVE)

Opens the CfMC system (TR) file with READ_WRITE access for that user only. (default) No one else may open the file to read or write to it.

FILES="pattern"

Allows you to input multiple data files that match the pattern defined. FILES= acts like the JOIN option by appending each file to end of the one previously opened.

The FILES pattern looks for actual file names without standard CfMC extensions. For instance, you must use FILES="DATA#.TR" to get all files called DATA1.TR, DATA2.TR, etc.

The pattern must be enclosed in "quotes".

You can specify a full file name or a partial name with a wildcard character as the pattern to match.

You can use the following wildcards:

*	match anything (DOS/UNIX)
?	match letter(s) (DOS/UNIX)
#	match digit(s)

You can specify more than one full file name separated by commas.

Example:~INPUT FILES="ASCA.ASC,ASCB.ASC" ASCII=80 ID=1.6

You may not use FILES with the JOIN option.

Example:~IN FILES="ASC*" ASCII=80 ID=1.6

In this example, all files beginning with the pattern ASC will be opened with this INPUT statement. In this example, they are all 80 column ASCII files and the case ID is in columns one to six.

In this software version, the ~input" line now allows subdirectories and use of "?", "#", and "@"

You can now specify patterns with files in subdirectories such as:

Example:

~input files="test/file#" ascii=80

To find all files in the subdirectory "test" with names like file1, file2, and file3. Previously you had to give a fully qualified directory name if you wanted to access files in other directories, and you could only read files in one subdirectory at a time.

The "#" means "any number", "?" means "any alphanumeric character", and "@" means the same as "*" which is "any matching pattern".

Environment variables

Environment variables now work with files= in version 8.1

Example:

~inpute files=”!CFMC!control/filename” ascii=80

ID=

Indicates the location and width of the case ID. Without it, the case ID defaults to columns one through ten (1.10).

Example:~INPUT myfile ASCII=80 ID=1.4

You can also use shorthand to indicate case ID by using a colon after the case length. See a description of the file formats at the beginning of this section.

Example:~INPUT myfile ASCII=80:1.4

IGNORE_DIRECTORY (IGNOREDIR)

Says do not use the directory associated with the input file. This would be useful if you want to write out a file without a directory. See also NUMBER_OF_CASES=.

JOIN=(f2, f3,...)

Must be the last option specified. Lets you read multiple input files as if they were one file. Says open file f2 and append it to end of the file named at the start of the ~INPUT command. The files named in the JOIN option must have a case length less than or equal to the file named at the start of ~INPUT.

Example:~INPUT DATA1,JOIN=(DATA2)

If you use this option you may not use options BACKUP, VERIFY, ALLOW_NEW, CREATE, MAYBE_CREATE, MAYBE_BACKUP, $ as a filename, or ~CLEANER NEXT LAST.

The FILES= option does the same thing as JOIN and will accept wildcards as well.

MAX_DELIMIT_FIELDS=###

Allows you to indicate the number of fields in a delimited ASCII input file. The default is 100 fields. You only need to use this option if there are more than 100 fields in the input file.

Related Commands: DELIMITER=

MAYBE_BACKUP (MAYBEBUP)

Makes a copy (TRX file) of the original file only if a TRX file does not already exist.

MAYBE_CREATE (MAYBECR)

Creates a data file if it does not already exist (i.e., ~INPUT sample, MAYBE_CREATE would create sample.TR if it does not exist).

NEW_BUFFER (NEWBUF)

Opens this as another input file so that multiple input files can be dealt with at the same time. The file just opened is set to the current input file.

NEXT_CASE_ID= (NEXTID=)

Sets the NEXTCASEID variable so the next Survent interview will use the case ID you indicate.

Example:~INPUT myfile ALLOW_UPDATE NEXT_CASE_ID=2001
~OUTPUT myfile2 WRITE_NOW

In this example, the next Survent interview run on myfile2 will use 2001 as the case ID.

TRFILE_DIRECTORY=# (Formerly NUMBER_OF_CASES=# (NUMC)

Used in addition to the CREATE option, this makes an indexed directory of the case IDs for direct case access if you are creating a new file; this is useful if you will be making many modifications to the file or to find cases much more quickly using ~cleaner's "next" command. The default is that files are built without a directory, and are read sequentially to find cases. See also IGNORE_DIRECTORY.

This is often used if you are doing a database application, or when the file will be used by Survent with Coding mode or Phone Says Data Record mode, which update existing cases.The program makes enough directory entries for the "Number of cases specified * 3". This is because if you make modifications that make the case significantly bigger, a new case is built and a new directory entry is used. This also allows for more cases to be added later. If the number of entries is set to -1, then no directory will be made for the file.

Related Commands:~ADJUST and ~OUTPUT TRFILE_DIRECTORY=

NUMBER_INPUT_BUFFERS=# (NUMBUF)

Says the maximum number of input files open concurrently; this must be said on the first of the files to be opened, and can be 1-30; the default is 5. Subsequent files to be accessed would use the NEW_BUFFER option.

PRINT_CASEID (PRTCASEID)

Prints caseIDs to the screen as an input file is being processed instead of dots. The option overrides a DOTS= setting on the same ~INPUT line.

PROTECT=#

Prevents modification to columns and below the column specified.

Example: `PROTECT=40`

This will prevent columns one through 40 from being modified.

Violation of the protection zone produces error #0.

QUIT_ON_BLANK_LINES(QUITONBLK)

Causes the program to print a warning and quit when blank lines are encountered in an ASCII input file. Most likely this option would be used when you do not expect any blank lines in the file and blanks indicate that the program might be reading a non-ASCII file.

READ_CONTROL=name(categories) (RDCTRL)

Used with data files where you will be making many tables by the same variable, this option allows you to specify exactly which data categories the cases must contain to be included. This option speeds data processing significantly.

Options:name

Is created with the ~MAKE_READ_CONTROL command. This READ_CONTROL item is what links like categories through the data.

categories

Are defined with the ~MAKE_READ_CONTROL command. These categories refer to those associated with the READ_CONTROL item and not with the original variable used to create the READ_CONTROL item. Categories may be specified using a dash, comma, or ellipsis, e.g., name(1,3-6,9) is the same as name(1,3,4,5,6,9). Categories must be listed in ascending order.

Example: ~INPUT DATACLN RDCTRL=Q3READCTRL(1,2)

READ_FIRST_CASE (READCASE)

Tells SELECT to position on the first case (default). Set on or off (-;) so you don't start reading through the file until you are ready.

RECORDS_PER_CASE=# (RECSPERCASE)

Allows you to set the number of records per case in as ASCII file, and treat several input records as one respondent. This is usually used when the original data is in card image format, and you want to make a system file in which the "cards" appear consecutively in each data record.

Each record from the original file is attached directly to the end of the previous record. For example, if you input an 80 column ASCII file, the first column from record two will be in column 81 of the output file. If the original input file has 100 columns, the first column of the second record will appear in column 101 of the output file.

All of the respondents in the input file must have the same number of records. RECORDS_PER_CASE does not checking for card number.

Example: `~INPUT filename ASCII=80 RECORDS_PER_CASE=3`

Related Commands: CARDS_IMAGE

SALVAGE_DATA (SALVAGE)

Has Mentor attempt to read a bad data file and make a new data file with the problems repaired. (You can also try to recover a corrupted data file with the RAWCOPY utility. See the *Utilities* manual.) See example next page.

Example:

```
~INPUT badfile SALVAGE_DATA

~OUTPUT goodfile WRITE_NOW
```

SAVE_DELIMIT_NAME=

Saves the array created when Mentor processes a delimited ASCII input file. This allows Mentor to use an existing array without having to rebuild one, and process faster (see USE_DELIMIT_NAME). If anything about the delimited input file changes, a new array must be built. If you write out a new data file at the same time you read in the delimited file, there is no need to save the control array. This could be used to save the format of data read from a delimited file as a .db entry using the keyword "save_delimit_name=<name>, and reuse it later using "use_delimit_name=<name>", such that any new files read would either match the format previously used, or return an error, or the data would be truncated to match based on the setting of the "delimited_data_mismatch= error/ok/warn" keyword.

Now, you can save the map as an ASCII file, then EDIT the file, then re-read the original data with the edited values, to match any new file with larger variables. The keyword to save the map is "save_delimit_map=<name>", to use a map, use the keyword "use_delimit_map=<name>".

Related Commands: DELIMITER=

SELECT=

Sets a filter for selection of cases to be used when reading the file. For example, to select all the even-numbered cases, from the bank study, use:

Example: ~INPUT bank SELECT=[4^2,4,6,8,0]

You can specify any logical expression.

Example: SELECT= Pick: [QQ23("text label":1)]

This example says to select only those cases where QQ23 contains a 1 response; the text label will appear by default as part of the titling whenever tables are created. To turn a filter off, you must open the file (with ~CLEANER FILE or ~INPUT) without the SELECT statement.

You may not use ~INPUT options ALLOW_NEW, BACKUP, VERIFY, CREATE, MAYBE_CREATE, MAYBE_BACKUP or the phantom ($) data file with this option.

SELECT= may appear anywhere on the ~INPUT line, but it needs a semi-colon (;) to end it if any other option is on the line after it.

Example: ~IN xx, SELECT= a: b WITH c; DOT=20

SERVER_ACCESS (SERVER)

Allows write access to data files running under the Survent SERVER. Use this (with caution) to update the data file while a study is live with procedures or using ~CLEANER commands.

Syntax:`~INPUT filename SERVER_ACCESS, ALLOW_UPDATE`

Mentor supports Server_Access on ~OUTPUT statement to append records. Mentor supports the following syntax.

Syntax:`~INPUT newrecs file of records to add ~OUTPUT mystudy file to be apppended to`

If the file “mystudy” exists, the new records will be appended to the file. If the file doesn’t exist, a new file will be built. This command is for coding applications where the coding is under a different CfMC server than the interviewing. Users can extract records from the interviewing server that have yet to be coded, and append the records to the coding server to be coded.

SHARE

Allows read-only access to the input file so you can modify it. Use WRITE_SHARE to allow you to modify and others to read only.

SHORT_LINE_WARN (SHORTWARN)

Warns when any line in an ASCII input file is shorter than the case ID. These warnings will only appear if you have defined case ID columns on the INPUT statement. If you have defined case ID columns and do not want to see the warnings, you can suppress them with -SHORT_LINE_WARN. Suppressing these warnings is useful because they will appear for each blank line in an input file.

STOP_AFTER=# (STOP)

Specifies the number of data cases to be used in processing. Must be smaller than the total number of cases in the data file. This is useful for testing purposes.

STUDY_NAME=studyname (STUDY)

A 1-6 character alphanumeric name used to identify a file when using multiple input files. The study name defaults to the first 6 characters of the data file name. When referencing multiple files, you can reference the studyname on variables to determine which file will be acted on. See the example under~EXECUTE PROCEDURE=MULTI.

TEXT_LOCATION=col.wid (TEX)

Specifies where the data from text variables (either [name$T] or from a ~PREPARE COMPILE run) is stored. If a questionnaire (QFF) file is open, the text data location is taken from that file if not otherwise specified. New text data is stored within the text location specified. If you do not specify a width, it defaults to allow all columns up to the end of the case. If you specify a width, other data may be stored beyond the text data area.

TOTAL_LENGTH=# (TOTLEN)

Specifies the total number of columns to allocate for each record in the data file. Use in conjunction with WORK_LENGTH, the total length defaults to WORK_LENGTH= if you do not specify it. TOTAL_LENGTH creates a memory area that you can use to pass data from one case to another. Below is an example that use the memory area in columns 801 to 804 to create case numbers starting

with 0001, increments them by one and then makes the case number (by putting it in columns 1-4) and the case ID. (Make a backup copy of your data file before trying this example!)

Example:

```
~DEF PROC=idit:
M [801.4*Z] ++= 1
M [1.4*Z] = [801.4]
PUTID [1.4$]
}
~INPUT myfile, TOTAL_LENGTH=804, ALLOW_UPDATE
~EXC PROC=idit
~END
```

USE_DELETED (USEDEL)

Uses all cases, including those flagged as deleted, when reading the input file or making a backup file.

USE_DELIMIT_NAME

Reads a control array that is created when Mentor processes a delimited ASCII input file. The control array is saved to a DB file using SAVE_DELIMIT_NAME. Using an existing array provides faster processing.

Related Commands: DELIMITER=

VERIFY

Reads the data file and verifies that the data is good. Reports the number of cases read, and the number of cases marked for deletion.

WORK_LENGTH=# (LEN)

Specifies the number of columns, starting from column one, that will be cleared from memory before each new case is read in. If not specified, it defaults to the case width specified on the CASE_LOCATION option.

WORKLEN cannot be less than the case width, and must be less than or equal to TOTAL_LENGTH.

When TOTAL_LENGTH is specified, and it is greater than WORKLEN, the first column available for cross case operations is column number WORKLEN + 1. Columns WORKLEN + 1 through TOTAL_LENGTH are not cleared when a new case is read in, allowing you to accumulate data across cases in this space.

WRITE_SHARE

Opens the CfMC system (TR) file with READ_WRITE access for the primary user, but READ_ONLY for all others. This is the default for Survent interviewing.

~INTERVIEW (INT)

Allows the user to conduct one interview. The questionnaire (QFF) file, data file (TR) and a data case must be opened first.

Syntax: ~INPUT datafile, ~QFF_FILE filename, ~NEW, ~INT

In batch operations, responses to the interview may be read in from the spec file by listing the responses in the order of the questions in the QFF file (each answer on a separate line including blank lines for all carriage returns) after the ~INTERVIEW command.

An asterisk after the ~INTERVIEW command (e.g., ~INTERVIEW *) allows responses from the keyboard only.

Related Commands: ~CLEANER INTERVIEW, VIEW

~MAKE_ASQ (MKASQ)

Converts a db file variable into an ASCII formatted entry (spec file). Creates an ASCII db entry. Use >DB_TO_FILE to get this entry into an ASCII file.

Syntax: ~MKASQ ASCIIname datavarname

Options:

ASCIIname
Spec file name for dbentryname.

datavarname

Datavar being converted.

The format of the resulting item is exactly like that of the ~DEFINE PREPARE= question structure and can be displayed and modified with the >EDIT command.

~MAKE_READ_CONTROL (MKRDCTRL)

This allows you to define a variable or use a pre-defined variable to control the reading of a data file in future runs. It is useful when you have some data element (i.e., regions or stores) that you want to reference for future table runs. In other words, this creates "threads" through the data for quick data base access. (MKRDCTRL)

This command requires an INPUT file. The ~INPUT options JOIN= and NUMBER_OF_INPUT_BUFFERS= are not allowed on the input file with this command.

Syntax:~MKRDCTRL varname = variable

Options:

varname
Names a DB file READ_CONTROL item for future reference.

variable
Must be a category or numeric type expression that results in any case not being in more than one category. The data is read and like categories are linked for faster subsequent reads.

Example: ~MKRDCTRL Q3READCTRL = Q3RC[6^1,2/3,4]

Creates a variable named Q3RC with 2READ_CONTROL categories. The READ_CONTROL item is named Q3READCTRL.

Example: ~MKRDCTRL Q3READCTRL = Q3

Uses the pre-defined variable Q3 (i.e., Q3: [6^1/2/3/4]) to create a READ_CONTROL item named Q3READCTRL with 4 categories.

See *Mentor, Volume I, "9.4 PARTITIONING DATA FILES"* for other examples.

Related Commands: ~INPUT READ_CONTROL

~MERGE

The ~MERGE command is a user-friendly utility that offers many options. The following are requirements when using ~MERGE command options.

- You must have at least two input files and an output file before even entering the ~merge block.

~input main study=in1 number_input_buffers=2

~input openends study=in2 new_buffer

~output merged

- All legal input and output options may be used on the ~INPUT and ~OUTPUT statements. Within the ~MERGE block the following must be answered:

PRIMARY=

PRIMARY_KEY=

SECONDARY=

SECONDARY_KEY=

The values for primary and secondary should be set to the study names of two of the input files currently open. Based on the example above, these would be set to in1 and in2 respectively. The PRIMARY_KEY and SECONDARY_KEY are the fields on which one wishes to match and should be set to a value that evaluates to a string. Here are some examples:

PRIMARY_key=[1.4$]

PRIMARY_KEY=SUBSTITUTE([1.4$],"a","0")

PRIMARY_KEY=[1.2$] join [11.2$] join [21.2$]

MERGE Options

The ~MERGE options are either set to a value, as in OPTION=VALUE, or turned on or off via a minus sign, as in option, -option. Except for the APPEND_LOCATION option, the setting last seen for any option is the one that will be used.

STATUS

The status command displays all of the ~MERGE options and their current settings. Here are the results of the status command showing all of the options and their default values:

- SORT_PRIMARY

- SORT_SECONDARY
- WRITE_MATCHED
- WRITE_UNMATCHED_PRIMARY
- WRITE_UNMATCHED_SECONDARY
- WRITE_DISALLOWED_DUPlicateS
- PRINT_MATCHED
- PRINT_UNMATCHED_PRIMARY
- PRINT_UNMATCHED_SECONDARY
- PRINT_SUMMARY
- PRIMARY_DUPlicateS_ALLOWED
- SECONDARY_DUPlicateS_ALLOWED
- DISALLOWED_PRIMARY_DUPlicateS=error
- DISALLOWED_SECONDARY_DUPlicateS=error
- EXEC_MCOPY_IF_MATCH
- EXEC_MCOPY_IF_UNMATCHED_PRIMARY
- EXEC_MCOPY_IF_UNMATCHED_SECONDARY
- APPEND_LOCation=0

SORT_PRIMARY will cause the primary file to be sorted into a temporary file which will then be used to do the actual merge.

SORT_SECONDARY will cause the secondary file to be sorted into a temporary file which will then be used to do the actual merge.

WRITE_MATCHED means that if the primary and secondary key fields match, a case will be written to the output file.

WRITE_UNMATCHED_PRIMARY will cause a case to be written to the output file when there is no matching record from the secondary file.

WRITE_UNMATCHED_SECONDARY will cause a case to be written to the output file when there is no matching record from the primary file.

WRITE_DISALLOWED_DUPLICATES The default for DISALLOWED_DUPLICATES is that they are written to the output files under the same control as unmatched records. If you are writing unmatched primary or secondary records and you don't want duplicated records, use the negative (minus) form of this option.

PRINT_MATCHED means that when a match occurs, a message will be printed indicating this in either the list file or open print file.

PRINT_UNMATCHED_PRIMARY means that when no matching secondary record exists for a primary record, a message will be printed indicating this in either the list file or open print file.

PRINT_UNMATCHED_SECONDARY means that when no matching primary record exists for a secondary record, a message will be printed indicating this in either the list file or open print file.

PRINT_SUMMARY is used to print summary information about the merge into the list file or open print file. The information printed includes the number of matched and unmatched records, the number of records written, the number of duplicates detected, etc.

PRIMARY_DUPLICATES_ALLOWED is used to allow duplicate match fields to appear in the primary input file, and in the case of a match, not to move the secondary data file on to the next record until all of the primary records having that match key have been used

SECONDARY_DUPLICATES_ALLOWED is used to allow duplicate match fields to appear in the secondary input file, and in the case of a match, not to move

the primary data file on to the next record until all of the secondary records having that match key have been used.

DISALLOWED_DUPLICATES controls what to print when a DISALLOWED_DUPLICATES is encountered. It can be set to ERROR, WARN or OK (no message). This setting does not affect whether DISALLOWED_DUPLICATES are written to the output file, even if it is set to ERROR.

Writing of DISALLOWED_ DUPLICATES is controlled by the settings for the WRITE_UNMATCHED_PRIMARY, WRITE_UNMATCHED_SECONDARY and WRITE_DISALLOWED_DUPLICATES.

NOTE: Using both PRIMARY_DUPLICATES_ALLOWED and SECONDARY_DUPLICATES_ALLOWED in the same ~MERGE block will result in an error.

DISALLOWED_PRIMARY_DUPLICATES controls what to print just for duplicates in the primary file. This overrides DISALLOWED_DUPLICATES.

DISALLOWED_SECONDARY_DUPLICATES controls what to print just for duplicates in the secondary file. This overrides ~DISALLOWED_DUPLICATES.

EXEC_MCOPY_IF_MATCH is used to execute the merge_copy commands found in the ~MERGE block if a match occurs.

EXEC_MCOPY_IF_UNMATCHED_PRIMARY is used to execute the merge_copy commands found in the ~MERGE block when there is a primary record that has no corresponding secondary record. Since merge_copy always puts data from the secondary record into the primary record, and because all of the secondary record is blank in the case of an unmatched primary, setting this option causes primary data fields to be blanked when there is no match.

EXEC_MCOPY_IF_UNMATCHED_SECONDARY is used to execute the MERGE_COPY commands found in the ~MERGE block when there is a secondary record that has no corresponding primary record.

APPEND_LOCation=0 until PRIMARY= has been set. Once set, the append location defaults to the absolute column number of the WORK_LENGTH of the primary input file plus one. If you use the APPEND or APPEND_ALL command, appending will begin at APPEND_LOCATION. The value of the append location is dynamic. After each append operation the append location is increased by the number of columns appended. The append location may be set to any column within the output length and it may be reset in the course of a ~MERGE block.

Data Manipulation

MERGE_COPY

The syntax of the MERGE_COPY command is:

MERGE_COPY TARGET_LOC_IN_PRIMARY = source_loc_from_secondary

The MERGE_COPY command operates essentially the same way that ~CLEAN copy operates, with the exception of several features designed to help make merging data easier. The biggest difference between MERGE_COPY and COPY is that the MERGE_COPY command locations specified on the left side of the equal sign are automatically taken to be from the primary data file, and those on the right side are taken to be from the secondary. Thus, what would be written in a ~CLEAN block as:

"copy in1![41.10]=in2![1.10]"

can be written in a ~MERGE block as:

"mcopy [41.10]=[1.10]".

NOTE: So far, MERGE_COPY does not work with text questions.

APPEND

The append commands syntax is:

APPEND_SOURCE_LOC_from_secondary

When the append command is used, the data location specified is taken to be from the secondary data file, and this data is appended to the primary data file starting at the location specified by APPEND_LOCATION (see APPEND_LOCATION= above). The value of APPEND_LOCATION changes after each append command so that successive append commands attach the data starting at the latest LAST_USED + 1 column.

APPEND_ALL

This command attaches the entire secondary record to the primary data file beginning at APPEND_LOCATION (see APPEND_LOCATION= above). Note that in order to append data, the output file length must be sufficiently greater than the TOTAL_LENGTH of the primary data file to accommodate the data being appended.

Proc=

You can define a proc and use it in ~MERGE. The commands used in the proc should be those you would use in a ~CLEAN block (i.e. not ~MERGE commands).

MERGE_defaults

The ~SET_OPTION_MERGE_DEFAULTS= allows the user to define a string of merge commands that will be executed each time a ~MERGE block begins. The string that you set MERGE_DEFAULTS equal to is not cracked until a ~merge command block is started. The string may contain any command that would be legal inside a ~MERGE block including meta commands. The string may be continued by using "&&". Backslash-Ns that appear in the

string are recognized as a new line character. Note how the use of \n in the default string below allows one to use the >QUIT command as part of the defaults setting.

~Set merge_defaults=">quit errors=1\n primary_dups_allowed" &&

"primary_key=[1.5$] secondary_key=[1.5$]"

EMPTY_CASE

EMPTY_CASE is a special variable that only is used from within procs called by a ~merge block. It should always be prefaced with a study name, as in <studyname>EMPTY_CASE. EMPTY_CASE is true when it is inside a ~MERGE block and the record referenced by "studyname" is the nonexistent side of an unmatched pair.

MERGE APPLICATIONS

1. Append one data set to another. Real life example: Append the data collected from two waves, or from a pre and post study.
2. Import data from a master data base into each data record. RLE: Take data associated with a zip code and import into each data record.
3. Locate records not in some other file. RLE: Find the records in your raw sample file that were not previously added into your .fon file.
4. Combine two files and drop the duplicates.
5. Take data from one file and conditionally move it into the other one. You forget a Phone,G statement in your QPX, but need the information in your data file to give to your client.
6. Combine two files and use an algorithm to figure out which duplicate to drop.

7 Take a master file and a trailer file and create a single file with each trailer appended to its master along with a counter on each trailer.

8 Take data from multiple records and import it to a single other record. RLE: Coding

9 Transfer text type data from one file to another.

~NEW

Opens a data case to allow data collection in MENTOR using a compiled SURVENT questionnaire. You must use ~NEW before ~INTERVIEW. You must have a questionnaire file open with ~QFF_FILE and a data file with ~INPUT filename, MAYBE_CREATE.

Example:`~INPUT data1` Opens a data file

`~QFF_FILE <file name>` Opens filename^QFF

`~NEW "caseid"` Opens a data case with (optional) ID

`~INTERVIEW` Runs the interview using SURVENT screens

Related Commands: ~CLEANER NEW_CASE

~NEXT

Positions itself at and displays the next valid data case. Use after the ~INPUT command to bring in the case you want to work with.

Syntax: ~NEXT option

Options:"id#"

Displays the case with the specified ID#.

##

Displays the case with that record (##).

#FIRST

Displays the first data case in the file.

#LAST

Displays the last data case in the file.

+/- ##

Displays the next case + or - the number specified.

By default Mentor holds up to the last 9 cases seen in memory to allow backing up in the data file. To increase this number, you will need to specify a higher number on the ~SET MAXIMUM_PAST_CASES= option.

Examples:

~NEXT "1234"

~NEXT 250

~NEXT +10

~NEXT -5

Use ~RESET to reset to top of file, then ~NEXT to display the first case. ~NEXT at last case will cause a rewind to the top of the file.

Related Commands: ~CLEANER NEXT

~OUTPUT (OUT)

Creates a new data file. This file gets cases written to it using the WRITE_NOW option or ~CLEANER WRITE_CASE. If you use the ~COPY commands, then cases get copied into the file specified on the ~OUTPUT command.

Syntax: `~OUT filename #, format, keyword`

To copy a data file, use ~OUTPUT with the WRITE_NOW option.

Example: `~INPUT file1`

~OUTPUT file2 WRITE_NOW

Here's how to append two files together into third file:

Example: `~INPUT FILES="copy1.tr copy2.tr"`

~OUTPUT file3 WRITE_NOW

Options:

filename

The data file name, the TR file extension is assumed. A dollar sign($) before the file name means the TR extension is not required. (Use the meta command

>-CFMC_FILE_EXTENSIONS to make $filename the default.)

Example: `~OUT $data1`

If you use NULL as the filename, an immediate end-of-file is read and the file will not be saved to disk when you write to it. You can use this for testing purposes when an output file is required, but you do not want to write any cases out to it.

Example: ~OUT NULL CASE_LENGTH=80

[MPE Users: An asterisk (*) before the file name means substitute the name for a file equation. For more information see *Appendix D: CfMC CONVENTIONS, Command Line Keywords, LISTFILE: <listfile>, option1...* in the *UTILITIES* manual.]

#

Specifies the number of the file for ~CLEANER WRITE_CASE when more than one output file is open.

format

~OUTPUT writes System data files (TR) by default, you can output files of other types by specifying the format. They are:

ASCII
BINARY (BIN)
CARDS_IMAGE (CARD)
HEX
SWAPPED_BINARY (SWAPBIN)
SWAPPED_HEX (SWAPHEX)
UNCOMPRESSED (UNPRESS)

If there is no input file open, then you must specify a length on the output file.

Example:~OUTPUT newfile ASCII LENGTH=80

Related Commands: ~INPUT

Keywords:

CASE_LENGTH=# (LEN)

Specifies length of file. If no length is specified there must be an INPUT data file, and the length defaults to the same as that of the INPUT file. If no data file has been accessed with the ~INPUT command then LENGTH= must be specified in the ~OUTPUT file statement. # can be an absolute number (800) or card/column (10/80).

Example: ~INPUT myfile.asc ASCII=50
~OUTPUT newfile LEN=60 WRITE_NOW

COMMENT= "comment" (COM)

Optional; allows up to a 21 character data file comment, which will display whenever the file is opened.

LINE_ENDING=

Mentor now allows you to make line endings on ASCII output files different from the default for your system. The line_ending= options are:

Syntax:

Line_ending= MPE or none
MAC or CR (carriage return)
UNIX or LF (linefeed)
DOS or CRLF (carriage return/linefeed)

MAYBE_CREATE (MAYBECR)

Creates a data file if it does not already exist. If the file exists, it is opened and cases are appended to it, and the LENGTH specified must match the length of the file.

TRFILE_DIRECTORY=# (Formerly NUMBER_OF_CASES=# (NUMC)

Makes an indexed directory of the case IDs for direct case access if you are creating a new file; this is useful if you will be making many modifications to the file or to find cases much more quickly using ~cleaner's "next" command. The default is that files are built to match the input file, that is, if the input file has a directory so will the output file, otherwise no. Use IGNORE_DIRECTORY to not make a directory even if the input file has one.

This is often used if you are doing a database application, or when the file will be used by Survent with Coding mode or Phone Says Data Record mode, which update existing cases. The program makes enough directory entries for the "Number of cases specified * 3". This is because if you make modifications that

make the case significantly bigger, a new case is built and a new directory entry is used. This also allows for more cases to be added later. If the number of entries is set to -1, then no directory will be made for the file.

Related Commands:~ADJUST and ~OUTPUT TRFILE_DIRECTORY=

STUDY_NAME=name (STUDY)

Gives a study name for the file during the current run. See >STUDY for more information.

You can have multiple output files and control where cases are written on the WRITE_CASE statement. To do this each output file must be numbered and this number must be included on the WRITE_CASE statement. Output files do not need to be opened consecutively or sequentially. The number of possible open output files is dependent upon the operating system. Under DOS the FILES= statement in the CONFIG.SYS file determines the total number all files types (i.e., list, print, temp, spec, etc.) that can be open at any given time. On a SCO Unix system, at least 26 output files can be open at the same time.

```
Example:~DEFINE
            PROCEDURE= {write:
            WRITE_CASE
                  IF [1^1] $WRITE_CASE #1 $ENDIF
                  IF [1^2] $WRITE_CASE #2 $ENDIF
                  IF [1^3] $WRITE_CASE #3 $ENDIF
                  IF [1^1,2] $WRITE_CASE #9 $ENDIF
            }
      ~OUTPUT outx
      ~OUTPUT outa #1
      ~OUTPUT outb #2
      ~OUTPUT outd #9
      ~OUTPUT outc #3
      ~EXECUTE PROCEDURE=write
```

If an unnumbered output file (e.g., OUTUX) is opened before the numbered output files, then unnumbered WRITE_CASE statements will write to the unnumbered file (i.e., the unnumbered file is treated as a different file from file #1). Opening an unnumbered output file after a numbered output file will cause an error on any WRITE_CASE statement.

If no file number is specified on a WRITE_CASE statement, data will be written either to the unnumbered output file or to output file #1 (whichever is open). In the example above the first WRITE_CASE statement will write data to file OUTX.

TRIM_BLANKS

Creates an ASCII output file with variable length records, that is one without the trailing blanks that are in a fixed length file. (The default is to create a file with fixed length records, see CASE_LENGTH.)

WRITE_NOW

Causes an output file to be written out immediately, without calling any procedure in an ~EXECUTE block.

Syntax: `~OUTPUT fileout WRITE_NOW`

When you use WRITE_NOW option, neither the input or output files are closed until a new ~INPUT or ~OUTPUT statement appears in the run. In the example below, if the procedure named "write" contained a WRITE_CASE statement:

~INPUT filein

~OUTPUT fileout WRITE_NOW

~EXECUTE proc=write

The records written out would be appended to the open output file. If you want the records written out to a different file, use a blank ~OUTPUT or restart the ~INPUT file.

You can use WRITE_NOW instead of ~TRANSLATE.

~PRACTICE

Allows one practice Survent interview each time the command is issued; no data is saved. The QFF file must be open before using ~PRACTICE (see ~QFF_FILE).

~PREPARE (PREP)

Allows you to create a questionnaire or write variable specifications in PREPARE format. This includes the ability to create spec files to be used by Survent and other software packages.

NOTE: CfMC recommends that you use ~REFORMAT to create specs for other software packages. ~REFORMAT generally does a better job of creating specs for other programs, especially COSI, SAS, and SPSS.

Syntax:~PREP keyword option option

Commas are NOT allowed as separators between PREPARE options. Separate options with spaces.

Keyword:

COMPILE

Directs the program to compile specifications. The default is Survent_SPECS. See below for the full list of compile options.

Example: ~PREPARE COMPILE Survent_SPECS CLEANING_SPECS

Keyword:

!rft_cat_01

This keyword allows all existing jobs to continue to run without error, and at the same time, users have explicit control over each question type where 0/1 coding might be an issue.

DISK_BASED_RESPONSE_LIST (DBR)

See your *Survent* manual section *3.1.4 DISK-BASED FIELD QUESTIONS FOR LARGE RESPONSE LISTS* for details.

MAKE_SPEC_FILES (MKFILE)

This command is used to produce auxiliary files from the compiled questionnaire (QFF) file. You must load the questionnaire file first with ~QFF_FILE <study name>.

Syntax:~PREP MKFILE option option

Options:

CHECK_FILE	**(CHK)**
HARD_COPY_FILE	(HRD)
QSP_FILE	(QSP)
SUM_FILE	(SUM)

Options:

The following options can be used with COMPILE. They will create additional files. See your *Survent* manual for more details on these options and the files they create.

COSI_SPECS (COSI)

Creates data tabulation specifications for COSI.

Mentor_SPECS (CMentor)

Creates data tabulation specifications (DEF, TAB files) from PREPARE question specifications.

Survent_SPECS (CS)

Creates all the files needed for interviewing (QFF and QUO files) in addition to CHK, SUM, and QSP. This is the default.

CLEANING_SPECS (CLN)

Creates data cleaning specifications (CLN file) from PREPARE question specifications.

MAKE_VARIABLES (MKVAR)

Opens a db file to store C-Mentor variables created from your PREPARE question specifications. Mentor auxiliary files such as CLN, DEF, and TAB are not made with this option.

COSI_SPECS [replaced PERSEE_SPECS (PERSEE)]

Creates COSI specifications.

QUANTUM_SPECS (QUANTUM)

Creates QUANTUM specifications (DEF, TAB files) from PREPARE question specifications.

SAS_SPECS (SAS)

Creates data tabulation specifications for SAS.

-SPEC_FILES (-SPEC)

Compiles specifications but makes only the compiled questionnaire (QFF) and quota (QUO) files. The CHK, QSP, and SUM files are not made.

SPSS_SPECS (SPSS)

Creates SPSS specifications from PREPARE question specifications. The file created is for the Windows version of SPSS. By default, creates absolute column numbers.

SSS_XML_SPECS

Creates SSS specifications from PREPARE question specifications.

UNCLE_SPECS (UNCLE)

Creates UNCLE specifications (DEF file) from PREPARE question specifications.

~QFF_FILE (QFF)

Opens the QFF file that should correspond with the INPUT and DB files. The program issues a warning message if the file is already open.

Use this command before making a new data file with the CREATE option on the ~INPUT command. This automatically assigns the length and case ID col.wid and text location to the new data file from what is specified in the QFF file.

You must use this command before generating CfMC files with the ~PREPARE MAKE_SPEC_FILES command from a complied questionnaire file.

Syntax: ~QFF filename

~READFIRSTCASE

~READFIRSTCASE can be used as a ~Set option. READFIRSTCASE performs the same task that readfirstcase currently does on the ~input statement. With this option, you will be able to turn this option on or off as a shop-wide preference.

When you have READFIRSTCASE on, a select on ~input will read through the input file until it finds a case that passes the select. The first case that passes the select may be very deep in a large data file and thus may considerably increase run times by causing a large number of cases to be read two or more times.

Also, there is **READFIRSTCASEwarn=** If you are searching for a case, and the program has to read a large number of cases to get to the case in question, you will get a warning saying that you may overload the machine by doing this task.

~REFORMAT (RFT)

See the *Survent* manual *5.7 REFORMAT* for details on using the menu-driven utility, explanation of the map (RFL) file, and a list of REFORMAT compiler commands.

~REFORMAT reads a Survent questionnaire file (QFF) opened with the ~QFF_FILE command and a data file opened with the ~INPUT command. It produces a spread ASCII data file (RFT) and a map of that file (RFL). The format of either of these files can be controlled by one of the options listed below. In addition to the RFL and RFT files, you can generate any of the files produced by a ~PREPARE COMPILE option (such as COSI, SAS and SPSS). Data locations will match the spread data and not the original questionnaire. The maximum ASCII data record size is 20,000 columns.

Keywords:

•Multi_response_category01	This works the same that it always did.
•Multi_response_field_01	This gives multi fld the same treatment as cats.
•Single_response_category01-	This does multi_response_category01 recode to single cats.
•Single_response_field_01	Does multi_response_field_01 recode to single flds.
•Single_Cat_response	This does the same thing that it always did.
•Single_Cat_01	See example below
•Multi_Fld_01	See example below
•Single_Fld_01	See example below

The keywords Single_Cat_0, Multi_Fld_01, Single_Fld_01 control whether prepare question types !CATEGORY or !FIELD get exported to “0/1” format in the .rft data set, where 0=not a response and 1= is a response for each possible response. There are four keywords which control whether this conversion takes place for single response or multiple response questions of each tpe. By default

these questions export the codes answered from the questionnaire script, instead of a separate code of “0” or “1” for each possible code.

For instance, if there was a question like:

{
Enter sex of respondent
!FLD
M Male
F Female}

And the response was “female”, the keyword “Single_field_01” would cause reformat to export data like this:

0,1

Where “0” represents male, and “1” female

NOTE: “0/1” coding is often used for multivariate statistical testing, which is one of the main reasons this woud be used.

•Use_Response_Text

Exports the text pf !CATEGORY or !FIELD type question responses instead of the response code or other possible options. This is particularly useful for SQL databases or Excel spreadsheets because it makes the data more readable for interactive queries. Using the same question and response above, the keyword “Use_Response_Text” would export:

Female
As the response.

•Renumber_Responses	Causes the response lists for !CATEGORY or !FIELD questions to be numbered starting at "1" and going to "N" where "N" is the number of codes in the list. For the question above, "Renumber_Responses" would export: 2 Since "Female" is the second code on the list.

These new keywords allow all existing jobs to continue to run without error, and at the same time, users have explicit control over each question type where 0/1 coding might be an issue.

Use of ~Specfile names

The ~Reformat command supports the setting of ~specfile name when determining the root name for RFT and RFL files. The priority is still to use the study code from the QFF file first, but it can now be overridden by the ~specfile name.

Support for conversion of codes to text for databases

The new ~REFORMAT command "Use_Response_Text=####" causes the program to export the text of the codes instead of the codes themselves in the spread data file. This is particularly useful for conversion to databases or spreadsheets where the text is preferable.

The "####" is the maximum number of characters to export per code. If not specified, the program will export all the text up to the maximum length of text on the response list. The maximum value is 1000.

If you ask for "c_mentor" specs, and the longest text on the response list is 72 characters or less, reformat will build a FIELD variable in the exported ^def file. This allows Mentor to create crosstabs with the text, otherwise the data from this will be converted to a VAR (string) type for export to other software.

Syntax: ~RFT option,option

The following options may *not* be specified together:

APPEND_CAT_DATA
CARDS_IMAGE
-EXPAND_SINGLE_CATS
LOTUS_FORMAT
SINGLE_CAT_RESPONSE

Options:APPEND_CAT_DATA (APPEND)

Outputs an ASCII version of the data followed by expanded CAT question data.

CARD_IMAGE (CARD)

Outputs data in 80-column card image format.

CORE_BLOCK_SIZE=# (CORE)

Sets the total core used for REFORMAT. The default is 2,000,000.

DO_NON_TEXT_DATA (DONTEX)

Reformats data from non-text questions. (default)

DO_TEXT_DATA (DOTEX)

Reformats data from text questions. (default)

EXPAND_SINGLE_CATS (EXPAND)

Expands or moves single response CAT questions. Use -EXPAND_SINGLE_CATS when you don't want any single response CAT questions expanded or moved.

NOTE: This is the default that prevents ~REFORMAT from automatically making and H and T record for each case that has a text question.

LOTUS_FORMAT (LOTUS)

Outputs the spread data file with commas between data items.

MAP_FILE (MAP)

Generates a map file of the data. (default) -MAP_FILE suppresses generation of the RFL file.

MULTI_CAT_01

Expands multi-response !CAT questions into zeros and ones. For example, if you had a multi response question with five possible answers, and the answers were 1, 3 and 5, the data would be spread into “10101”. Without MULTI_CAT_01, the data from the same question would be spread into “135 “.

SINGLE_CAT_RESPONSE (SINGLE)

Expands all single response !CATs as responses, overriding the default of ones and zeros.

TEXT_HOLD_SIZE=# (TEXHOLD)

Specifies the amount of space for TEX questions. (Default is 1000, 125 questions for DOS 386 systems or higher, UNIX , and 250 TEX questions for DOS 286 systems.)

Example:~INPUT data1
~QFF_FILE samp1
~RFT APPEND DONTEX

In addition to these options, you can specify the following

~PREPARE COMPILE options to generate spec files based on the spread data:

Mentor_SPECS; Survent_SPECS; CLEANING_SPECS; COSI_SPECS; MAKE_VARIABLES; PERSEE_SPECS; QUANTUM_SPECS; SAS_SPECS; or SPSS_SPECS.

Refer to ~PREPARE for information on the files created by these options, or the Survent manual for sample output files.

If you use a ~PREPARE COMPILE option:

- All !CAT questions, single or multiple response, are reformatted using !RFT_CAT_RESPONSE.
- All !LOOP questions are reformatted with the default !RFT_UNWIND_LOOPS
- TEXT questions are not reformatted.
- You cannot get data in CARDS_IMAGE, LOTUS_FORMAT, or APPEND formats.
- The defaults for any of the ~PREPARE COMPILE options are:
- EXPAND_SINGLE_CATS
- DO_NON_TEXT_DATA
- DO_TEXT_DATA
- SINGLE_CAT_RESPONSE

REFORMAT exports "SQL readable" files

Example: The ~reformat option "sql_data=(databasename=<database name> tablename=<table name>)" will create a .def file with an SQL table definition, a .tab file with SQL insert statements and a matching data (.rft) file to be uploaded to the table in an SQL database.

Example: ~input mt3710 ;

~qfffile mt3710

~specfile mt3710a

~reformat sql_data=(databasename=cfmctest tablename=mt3710a)

~end

Options include:

databasename - name of the data base

If specified, the .def file will use the name that you provided and will begin with these two lines:

create database if not exists mydatabasenamehere;

use mydatabasenamehere;

If not specified, the .def file will begin with the above two lines commented out.

tablename - name of the table (required)

include_size_on_text - using the minus form removes the size spec from text variables in the .def file (e.g. gives you text instead of text(2000)).

- this option exists because not all data base programs allow the size spec in their syntax

- the default is "on"

num_exceptions_separate - causes NUM questions that have alpha exceptions to be saved as two variables, one with only numbers and the other with only the exception codes

- the variable for the exceptions will have _extra appended to its name, as in Q1 and Q1_extra

- without this option NUM questions with alpha in them will be type char

- with this turned on you will get a type numeric and a type char variable

- the default is "off"

Here is an example .def file:

Example:

```
create database if not exists bank;
use bank;
create table if not exists mike (
 caseid char(4),
 havecard char(1),
cardtype_1 char(1),
```

cardtype_2 char(1),

cardtype_3 char(1),

cardtype_4 char(1),

cardtype_5 char(1),

cardtype_6 char(1),

The .rft file will be a tab-delimited version of the data that can be added to a database using the “LOAD” command.

The CfMC variable types are converted to SQL types as follows:

CfMC Type	Sql Type
FLD	char
NUM (without decimal)	numeric
NUM (with decimal)	dec
EXPR (without decimal)	Numeric
EXPR (with decimal)	dec
VAR	char
TEXT	text

The default for all other variable types is “char” (!spc, !phone, etc.).

If there are more than 100 characters of text, they are concatenated using concat ("first 100 chars of text","next 100 chars","more text"). If less than 101 characters, then just do ‘here is some text’. Enclosed quotes (") are converted to \" and ' to \'.

"REFORMAT" has new "MAXIMUM_DELIMITED_FIELDS=###" option

This controls how many items to put in each output delimited file. You can now have reformat create more than one delimited file with each file having a specified number of fields. This is so that users that have databases with a maximum number of supported fields can read each of the files separately into the program; for instance, EXCEL has a maximum of 255 fields in a file. If you use this feature, REFORMAT will make as many files as necessary to create the output, with each file having a maximum of ### fields.

The minimum value is 10 fields, the maximum is 32000. If you have a question variable translating to more than the maximum_fields value, the program will generate an error and request that you break the question variable into smaller pieces and try again.

If a question variable has more fields than are available to fill the current file, all of the fields of that variable will be put in the next file, etc. The files are named as <study>_#, where # is 1-999. The first file is <study>_1, then <study>_2, etc. until all variables are placed in some file.

An exception to this feature is the !LOOP variable; if REFORMAT finds a !LOOP variable in the file, it will stop the run and deliver an error message.

~RESET

Points to the beginning of the file and get the first usable case.

Syntax: ~RESET

The ~CLEANER NEXT FIRST command does the same operation.

You can also use ~RESET -READ_CASES, which causes no case to be read on a rewind.

~RESTORE

If modifications have been made to a case reverts to its original form.

Syntax:~RESTORE

Related Command:~CLEANER RESTORE

~SET

Allows you to override Mentor's default settings. The default parameters are:

AUTOMATIC_NEW_LINE
AUTOMATIC_NEW_PAGE
CASE_SENSITIVE
COLUMN_REPEAT_OVERRIDE
CLEAN_ERROR_NUMBER
EDIT_RUN_CHAIN
ERROR_LIMIT=-1
ERROR_PROCESS

ERROR_SUMMARY

INTEGER_TABLES

REUSE_LABELS

TABLE_EFFORT=1

TABLE_NAME=T001

TABLE_SET_MATCH_WARN

TESTING_MODE

VARIABLE_NAMES

Syntax: `~SET keyword, keyword, keyword`

A minus sign (-) prior to many of the keywords will either turn that keyword off or return it to its default value.

Related Commands:~CLEANER DO_SET and SET, ~EXECUTE SET

Keywords:

The keywords can be grouped by function. Here are the groupings, followed by all of the keywords and their explanations in alphabetical order.

Global controls:

NEXT_COLUMN=

RESET

SHOW_MEMORY

STATUS

Affecting ~CLEANER data modification:

CLEAN_ERROR_NUMBER

CLEANER_DEFINITION=

CONFIRM_BLANK
LOGGING
PRODUCTION_MODE
SEQUENTIAL_READ
TESTING_MODE
TRAINING_MODE

Affecting ~DEFINE PROCEDURE= operations:

AUTOMATIC_NEW_LINE
AUTOMATIC_NEW_PAGE
AUTOMATIC_RESET
BOTTOM_MARGIN=
CASE_SENSITIVE
CLEAN_ALLOW_BLANKS
ERROR_LIMIT=
ERROR_PROCESS
ERROR_REVIEW
ERROR_STOP
ERROR_SUMMARY
MAXIMUM_PAST_CASES=
PAGE_LENGTH=
PAGE_WIDTH=
PRINT_ALL_ERRORS_STOP
PROCEDURE_DUMP
SHOW_CASE_INFO
STRICT_VARIABLE_DEFINITION
TOP_MARGIN=
VARIABLE_NAMES
ZERO_FILL

Affecting table building:

ALLOW_EDIT_CHANGE
ALLOW_MULTIPLE_WEIGHT_STATISTICS
AUTOMATIC_RESET=
AUTOMATIC_TABLES
BEGIN_TABLE_NAME=
BOTTOM_MARGIN=
COLUMN_REPEAT=
COLUMN_REPEAT_OVERRIDE
DELIMITED_TABLES
DROP_BANNER
DROP_BANNER_TITLE
DROP_BASE
DROP_COLUMN
DROP_COLUMN_SHORT_WEIGHT
DROP_COLUMN_WEIGHT
DROP_EDIT
DROP_FILTER
DROP_FILTER_TITLE
DROP_FOOTER
DROP_HEADER
DROP_LOCAL_EDIT
DROP_ROW
DROP_ROW_SHORT_WEIGHT
DROP_ROW_WEIGHT
DROP_STATS
DROP_STUB
DROP_STUB_PREFIX

DROP_STUB_SUFFIX
DROP_TABLE_SET
DROP_TITLE
DROP_TITLE_2
DROP_TITLE_4
DROP_TITLE_5
EDIT_DUMP
EDIT_RUN_CHAIN
ENHANCED_DEFAULT_TITLES
ERRORS_TO_PRINT_FILE
IGNORE_SUPPRESS
LEAVE_ROOM=
LOOPKICKOUT
MAX_SSIZE
MAX_VARIABLE SIZE=
MEAN_STATISTICS_ONLY
MEDIAN_CELLS
MULTIPLE_WEIGHT_STATISTICS
PAGE_NUMBER=
PAGE_LENGTH=
PAGE_WIDTH=
PRINT_TABLE_NAME=
PRINT_TCON
REGION_CODING_MODE=
SAVE_TABLE=
STACK_1=
STACK_2=
START_TCON
STATISTICS_BASE_AR
STATISTICS_DUMP

TABLE_DROP_MODE=
TABLE_DROP_WARN=
TABLE_DUMP
TABLE_EFFORT=
TABLE_FIELD=
TABLE_LINES=
TABLE_MISSING_MODE=
TABLE_MODIFY_MODE=
TABLE_NAME=
TABLE_PRESENT=
TABLE_SET_MATCH_ERROR
TABLE_SET_MATCH_WARN
TABLE_STORE_MODE=
TOP_MARGIN=
UNWEIGHTED_TOP

KEYWORDS:

auto_banner_heading=none

This will create a banner with only the response text and no banner heading. This is how the banner will look in the .ban file:

```
tabset= { qn1_y:
banner_title=:
  Road Runners Worth Pay}
banner=:
make_banner
 [level=1] (5) Completely agree
 [level=1] (4) Somewhat agree
 [level=1] (3) Neither agree nor disagree
 [level=1] (2) Somewhat disagree
 [level=1] (1) Completely disagree
 [level=1] Don't Know/Refused to answer
  }
"qn1(5/4/3/2/1/0)
col=: [6.1^5/4/3/2/1/10]
  }
```

auto_banner_heading=label

This will create a banner with a heading made from the ~prepare question label. This will be the new default. This is how the banner will look in the .ban file:

```
tabset= { qn1_y:
banner_title=:
  Road Runners Worth Pay}
banner=:
make_banner
```

```
 [level=2] QN1
 [level=1] (5) Completely agree
 [level=1] (4) Somewhat agree
 [level=1] (3) Neither agree nor disagree
 [level=1] (2) Somewhat disagree
 [level=1] (1) Completely disagree
 [level=1] Don't Know/Refused to answer
  }
"qn1(5/4/3/2/1/0)
col=: [6.1^5/4/3/2/1/10]
  }
```

auto_banner_heading=title

This will create a banner with a heading made from the ~prepare question title. The maximum amount of question text to be used is 30 characters. If the question title is longer than 30 characters then the first 27 characters will be used followed by 3 dots. This is how the banner will look in the .ban file:

```
tabset= { qn1_y:
banner_title=:
  Road Runners Worth Pay}
banner=:
make_banner
 [level=2] Road Runners Worth Pay
 [level=1] (5) Completely agree
 [level=1] (4) Somewhat agree
 [level=1] (3) Neither agree nor disagree
 [level=1] (2) Somewhat disagree
 [level=1] (1) Completely disagree
 [level=1] Don't Know/Refused to answer
```

```
  }
"qn1(5/4/3/2/1/0)
col=: [6.1^5/4/3/2/1/10]
  }
```

See mt2922c.spx

auto_banner_heading=both

This will create a banner with a heading made from the ~prepare question label and the question title. The question label is followed by a colon and 1 blank space. The maximum number of characters for the question label and title is 30. If there are more than 30 characters then the first 27 characters will be used followed by 3 dots. This is how the banner will look in the .ban file:

```
tabset= { qn1_y:
banner_title=:
  Road Runners Worth Pay}
banner=:
make_banner
 [level=2] QN1: Road Runners Worth Pay
 [level=1] (5) Completely agree
 [level=1] (4) Somewhat agree
 [level=1] (3) Neither agree nor disagree
 [level=1] (2) Somewhat disagree
 [level=1] (1) Completely disagree
 [level=1] Don't Know/Refused to answer
  }
"qn1(5/4/3/2/1/0)
col=: [6.1^5/4/3/2/1/10]
  }
```

Using no ~set auto_banner_heading options or using ~set -auto_banner_heading gives you the default.

ALLOW_EDIT_CHANGE (EDITCHANGE)

Says that a new EDIT statement will not change the format of the banner or stub from the ones that were loaded with the table. -ALLOW_EDIT_CHANGE turns this off.

ALLOW_MULTIPLE_WEIGHT_STATISTICS (MULTIWGT)

Allows STATISTICS= testing on tables with different weights. By default, Mentor will generate an error if you attempt statistical testing when you have COLUMN_WEIGHT, COLUMN_SHORT_WEIGHT, ROW_WEIGHT or ROW_SHORT_ WEIGHT in your specs. Statistical testing is calculated using the weighted values.

If the columns on your table are dependent, significance testing will happen only if a given respondent that appears in more than one of the columns tested is weighted the same in both columns. You cannot do significance testing on dependent columns if the same respondent has different weights. For example, you can do significance testing on a table that has both a weighted and unweighted total, but you cannot test the unweighted total against any of the weighted columns.

If your columns are independent, that is, the columns share no respondents, you can do statistical testing on columns with differing weights.

AUTOMATIC_NEW_LINE (AUTONEWLINE)

Prints a new line between statements to the open print file. This is the default.

-AUTOMATIC_NEW_LINE turns this off, allowing you to print information from more than one print statement on the same line.

-AUTOMATIC_NEW_LINE does not affect program-generated new lines, error messages, printed tables, the end of a procedure, or printing to the screen from ~CLEANER.

By default, the program goes to a new line for each case as it executes a procedure. To print on the same line in-between cases, use ~CLEANER PRINT_LINES option\!. This says print the print buffer now, and leave the cursor at the end of the line.

Example:

```
          ~DEFINE
 PROCEDURE={sample:
  PRINT_LINES "This is case \S" [1.4$]
             NEXT
             PRINT_LINES "  This is case \S" [1.4$]
             NEXT
  PRINT_LINES "  This is case \S\N" [1.4$] }
          >PRINT_FILE samp1
~SET -AUTONEWLINE
~INPUT samp1,STOP_AFTER=9
~EXECUTE PROCEDURE=sample
~END
```

AUTONEWLINE is turned off so that every three cases will print on the same line, but a new line (\N) will print in between each set of three. Here is the sample print file:

```
This is case 0001  This is case 0002  This is case 0003

This is case 0004  This is case 0005  This is case 0006

This is case 0007  This is case 0008  This is case 0009
```

AUTOMATIC_NEW_PAGE (AUTONEWPG)

Kicks to a new page in the print file between each procedure executed from a specifications file.

AUTOMATIC_RESET (AUTOSET)

Resets MAX_VARIABLE_SIZE from 60,000 to 15,000 in the ~EXECUTE block (default). This allows Mentor to process more tables in each data pass.

If your specifications return to the ~DEFINE block after a ~EXECUTE block, MAX_VARIABLE_SIZE does not reset to 60,000. In this case you would need to reset MAX_VARIABLE_SIZE back to 60,000 after each ~EXECUTE block (~SET RESET does not reset MAX_VARIABLE_SIZE). Alternatively, you can say -AUTORESET, but this could cause your run to use more passes than might otherwise be necessary in the ~EXECUTE block.

For users running the non-WATCOM version of Mentor on DOS 286 machines, AUTORESET resets MAX_VARIABLE_SIZE from 5000 to 2000. If your specifications return to the ~DEFINE block after a ~EXECUTE then you should reset MAX_VARIABLE_SIZE to 5000. -AUTORESET could result in ~SET LEAVE_ROOM being smaller than MAX_VARIABLE_SIZE possibly causing the run to fail.

Related Commands:

LEAVE_ROOM, MAX_VARIABLE_SIZE

AUTOMATIC_TABLES (AUTOTAB)

Causes a table to be executed at every new ROW= command in the ~EXECUTE block. AUTOMATIC_TABLES gives tables a name based on the current ~SET TABLE_NAME=. Tables are stored in the DB file opened with Read/Write access. You can set this on or off (-).

The input file is not read until either a ~EXECUTE PRINT_ALL command is found or the end of the table pass is reached. If you need to change data files during a tables run add a RUN_CHAIN statement at the end of each ~EXECUTE block to force those tables to be built immediately.

Example:

```
~SET AUTOTAB
~INPUT data1
~EXECUTE TABLE_SET=tab1,RUN_CHAIN
~INPUT data2
```

```
      ~EXECUTE TABLE_SET=tab2,RUN_CHAIN
      ~INPUT data3
      ~EXECUTE TABLE_SET=tab3,RUN_CHAIN
PRINT_ALL
     ~END
```

BEGIN_TABLE_NAME=<name> (BGNTAB)

Sets beginning table name (for ~EXECUTE PRINT_ALL) and tabname too if not specified.

BOTTOM_MARGIN=# (BOT)

Specifies the number of blank lines to leave at the bottom of the page in the currently open print file. This number must be less than or equal to the bottom margin specified on the meta command >PRINT_FILE. The default is three.

Related Commands:

PRINT_FILE, BOTTOM_MARGIN, ~DEFINE EDIT=BOTTOM_MARGIN

CASE_SENSITIVE

Causes ~FREQUENCY, ~SORT, and ASCII variables (also known as pound sign[#] variables) to be case sensitive when counting, sorting or matching ASCII characters in a data file.

To change how ASCII characters are displayed (but not the contents of the data file) use the $U, $D, $N variable types. To change the case of an ASCII string in a data file, use the ~CLEAN DOWNSHIFT/UPSHIFT functions. See *Mentor, Volume I, "Changing Case"* and *"String Functions." Also see Apprendix A, Meta commands in the Utilities Manual.*

In addition, the software can read all filenames and paths regardless of the length or case (Use " "s around filenames with special characters or spaces). The program will lowercase all input and output names you specify unless you use ">CASESENSITIVE", in which case the program, will look for and write names exactly as specified.

In ALL cases, lowercase file extensionsare output, unless otherwise specified. It is assumed that setenv ENVIRONMENT variables are uppercase unless you specify ">CASESENSITIVE_ENV", in which case the program pays attention to the exact case.

For example, when turned on, files referenced, such as "Study.db", will look for "Study.db" and not "study.db". By default, it will look for "studydb" or "Study.db" or "STUDY.DB".

Related Commands:

`~CLEAN UPSHIFT, ~CLEAN DOWNSHIFT, ~FREQUENCY,~SORT`

CLEAN_ALLOW_BLANKS (CLNAB)
Globally allow blanks in a procedure despite the data editing statements.

CLEAN_ERROR_NUMBER (CLNERRNUM)
The default for this option is on. Using minus (-CLEAN_ERROR_NUMBER) will remove the error numbers that are included in the error listing and error summary. These errors are generated by the CHECK and ERROR statements in a cleaning procedure.

CLEANER_DEFINITION=<string> (CLNDEF)
Prepends a string to every command you say in the ~CLEANER block not started with a $. This is useful if you are doing the same command over and over (e.g., $MODIFY_ASCII, $DISPLAY_ASCII, etc.).

Syntax: CLNDEF=<string>

Example: ~SET CLNDEF=$MODIFY
~CLEANER T001 = T005

This would cause ~CLEANER $MODIFY T001 = T005 to be executed.

COLUMN_REPEAT=# (COLREP)

Will print multiple banners consecutively for each row in a table run using STORE_TABLES, TABLE=, or ~SET AUTOMATIC_TABLES. You specify one set of table specs, then indicate the other banners, and multiple sets of tables are run with interleaved table names.

Syntax:COLREP=#

where # is the number of column sets you will be using. There is no known limit to the number you may assign here.

With COLREP set, the program assigns table names leaving room for the additional columns as it builds the tables. It does all the tables specified for the first COLUMN= specified, then remakes the tables for each COLUMN= found after that.

PRINT_RUN then prints the tables in table number order so that the same stub is seen against all banners in a row. For example, QN1 is the stub for the first three tables: the first with BAN1 as the column, the second with BAN2 as the column, and the third with BAN3 as the column.

Example: ~SET COLREP=3,AUTOMATIC_TABLES

```
~EXECUTE
        COLUMN=ban1
        ROW=qn1          "table 1
        ROW=qn2          "table 4
        ROW=qn3          "table 7
        COLUMN=ban2        "tables 2  5  8
        COLUMN=ban3        "tables 3  6  9
        RESET,PRINT_RUN
```

You can specify names for the tables, but you must leave room for the new names the program will make for the additional banners, e.g., if COLREP=2, TABLE=T001 and TABLE=T003 are ok, but not TABLE=T002 in between them.

Related Command:COLUMN_REPEAT_OVERRIDE

COLUMN_REPEAT_OVERRIDE (COLREPOVERRIDE)

Controls which table elements stay in effect for ~SET COLUMN_REPEAT. Mentor will carry over (in addition to the column variable) the base, header, footer, statistics, and weight variables specified on the first column set of tables unless specifically replaced and turned off with -COLUMN_REPEAT_OVERRIDE. By default, the program expects to process a new title, stub and row variable for each table in a COLUMN_REPEAT.

You must use -COLREPOVERRIDE in order to change any of the following options between banners:

BASE=

COLUMN_WEIGHT=

EDIT=

FOOTER=

STATISTICS=

WEIGHT=

CONFIRM_BLANKED_COLUMNS (CONFIRMBLANK)

Determines when you get a confirmation message when using OK_COLUMNS or interactively blanking columns. The default is CONFIRM_BLANKED_ COLUMNS=1; you may set this to a higher number when more than one column is being blanked. CONFIRM_BLANK=0 will turn off the confirmation messages.

DELIMITED_TABLES=(options)

Writes a delimited table to a file that can be used to import tables into spreadsheets, word processors or presentation graphics software. The file will contain the text from the table, and the numbers of the table separated by the delimiter you choose.

To create delimited tables, you must have a Mentor run which would create printed tables (see *Mentor Volume One, "Tables I")*, plus a ~SPECFILE statement and the

DELIMITED_TABLES statement. The delimited tables will go into the file named on the ~SPECFILE command with a .DLM extension.

Delimited tables usually require some work to make them correctly formatted. In the .DLM file, there is no distinction made between the banner, title, table name, footers, etc., so it up to the user to make adjustments to this text as necessary after it has been imported to the other software. Also, tabs associated with text will be lost, so they will have to be replaced in the new file.

You may change the position of the ~specfile statement to logical points in your specs. This is especially useful in runs where you want control over what goes into the print file, the delimited file and the html file.

If you've used a ~specfile with no name (in order to close out what you are doing) and then do something that requires a ~specfile to be open, you will get an error telling you this and the run will try to get back in synch when it finds a new tilde block.

Defaults for delimited tables are:

- Tab is the delimiter.
- The delimiter is applied to the body of the table only.
- Characters that appear with numbers, such as percent signs, dollar signs, commas, stat letters are omitted (it is advised to indicate that numbers are in percentages or dollars somewhere in the titling of the table).
- Cells that are blank, have a dash, or are considered missing will contain a zero.
- If a delimiter besides a tab is used, text from the table is put in quotes.
- Tabs within text elements are converted to a single space.
- Only the first line of title text is printed.
- Titles will remain centered or right-justified if the column width in the new program is the same as it was in the Mentor table.
- Only the first line of stub text is printed. (Set a large STUB_WIDTH on your table specs if you don't want to lose the text. This may require add PAGE_WIDTH= to the >PRINTFILE statement and LINE_LENGTH= to the ~SPECFILE statement; both default to a width of 132 characters.)

- Suppressed stubs, footnotes and comments are not included.

Options:BANNER (BAN)

Includes the banner in the .DLM file. Setting this option on ensures that all the lines of a multi-line banner is included. -BANNER suppresses the banner.

COLUMN_WIDTH=# (CWID)

Specifies a fixed column width. The default width is 20, but may be set to any positive number. This is used in conjunction with DELIMITER=fixed option.

DELIMITER=

Specifies the table delimiter character. The first character after the equals sign will be the delimiter, it may be set to any character, including a space. You may also set the delimiter to DELIMITER=fixed, and the tables will take their format from theSTUB_WIDTH and COLUMN_WIDTH options.

DO_TABLE_NAME (TABNAME)

Includes the table name (e.g. T0001). -DO_TABLE_NAME suppresses the table name.

NOTE: Unless some titling is passed to the delimited tables, there will be nothing to distinguish the end of one table from the start of the start of the next one.

FOOTER (FOOT)

Includes the footer. -FOOTER suppresses the footer.

HEADER (HEAD)

Includes the header. -HEADER suppresses the header.

LABELS

Includes all title text from the table. -LABELS will suppress all table titling (banner, footer, header, title, title_2, title_4, and title_5), even if they are set to "on" in the DELIMITED_TABLES statement.

STUB

Includes the stubs. -STUB suppresses the stubs. (If you want to suppress only certain stubs from the delimited file, use the ~DEFSTUB option FOR_OUT_TABLES.)

STUB_WIDTH=# (SWID)

Specifies a fixed width for stubs. The default width is 20, but may be set to any positive number. This is used in conjunction with DELIMITER=fixed option.

TABLE_TITLE

Prints the table name. -TABLE_TITLE suppresses the table name.

TITLE_2 (T2)
Prints the TITLE_2.

TITLE_4 (T4)
Prints the TITLE_4.

TITLE_5 (T5)

Prints the TITLE_5.

Tips for exporting to a word processing program (Word, WordPerfect):

- use a tab as a delimiter
- set up tab stops before bringing the in the data
- don't use proportional fonts

Tip for exporting to spreadsheets (1-2-3, Excel):

- use a comma as a delimiter

Example: `~SET DELIMITED_TABLES=(DELIMITER=, -LABELS)`

DOUBLE_SPACE_ERRORS

Prints a blank line in between cleaning errors and lines in the error summary.

NOTE: The following DROP commands can be turned off, meaning the table elements will once again be retained and used in sub-sequent tables until turned off or redefined, by using the SET command within a tableset or ~EXECUTE block and the DROP command minus sign in front of it, for example, ~SET -DROP_ BANNER.

DROP_BANNER (DROPBAN)

Drops the banner after it is used in a table. This removes only the banner. If you want to drop the data definition as well, you must use DROP_COLUMN.

DROP_BANNER_TITLE (DROPBANT)

Drops the banner title after it is used in a table. This removes only the banner title. If you want to drop the data definition as well, you must use DROP_COLUMN.

DROP_BASE (DROPB)

Drops each ~EXECUTE BASE setting after it is used in a table. This only removes the base definition, if you have used a TITLE_4 as a base label, use DROP_TITLE_4 as well.

DROP_COLUMN (DROPCOL)

Drops the column definition after it is used in a table.

DROP_COLUMN_SHORT_WEIGHT (DROPCOLSHORTWT)

Drops the short weight for a column (see ~EXECUTE for a definition of COLUMN_SHORT_WEIGHT).

DROP_COLUMN_WEIGHT (DROPCOLWT)

Drops the weight for a column (see ~EXECUTE for a definition of COLUMN_WEIGHT).

.

DROP_EDIT (DROPEDT)

Drops each ~EXECUTE EDIT= setting after it is used in a table.

DROP_FILTER (DROPFIL)

Drops the filter after it is used in a table.

DROP_FILTER_TITLE (DROPFILT)

Drops the filter title after it is used in a table.

DROP_FOOTER (DROPFOOT)

Drops the footer after it is used in a table.

DROP_HEADER (DROPHEAD)

Drops the header after it is used in a table.

DROP_LOCAL_EDIT (DROPLOCEDIT)

Drops each ~EXECUTE LOCAL_EDIT setting after it is used in a table.

DROP_ROW

Drops the row definition after it is used in a table.

DROP_ROW_SHORT_WEIGHT (DROPROWSHORTWT)

Drops the short weight for a row (see ~EXECUTE for a definition of ROW_SHORT_WEIGHT).

DROP_ROW_WEIGHT (DROPROWWT)

Drops the weight for a row (see ~EXECUTE for a definition of ROW_WEIGHT).

DROP_STATS

Drops statistics defined after they are used in a table.

DROP_STUB (DROPS)

Drops the group of stubs defined after they are used in a table. This only drops the stub text, you must use DROP_ROW to drop the data definition for the row.

DROP_STUB_PREFIX (DROPSTUBPREF)

Drops the stub prefix after it is used in a table.

DROP_STUB (DROPSTUBSUF)

Drops the stub suffix after it is used in a table.

DROP_TABLE_SET (DROPTABSET)

Drops all elements of each ~EXECUTE TABLE_SET after they are used in a table. This option will also drop any previously defined banners.

DROP_TITLE (DROPT)

Drops each title after it is used in a table.

DROP_TITLE_2 (DROPT2)

Drops each TITLE_2 after it is used in a table.

DROP_TITLE_4 (DROPT4)

Drops each TITLE_4 after it is used in a table.

DROP_TITLE_5 (DROPT5)

Drops each TITLE_5 after it is used in a table.

DROP_WEIGHT (DROPW)

Drops each ~EXECUTE WEIGHT setting after it is used in a table.

EDIT_DUMP (EDITDUMP)

Prints edit features that are on for each table.

Example:

```
Dump of edit options for banner1_e (0)  9 columns (7 bytes each), stub width = 40
Frequency_DECimals=0  Percent_DECimals=0 Statistics_DECimals=2
Vertical_PERcent=-1  FREQuency PERcent_SIGN  COLumn_NA  ROW_TNA
SKIP_lines=1  EMPTY_CELLs=- TCON  Title_4_for_BASE
RUNning_LINEs=1  PaGe_LENgth=66 PaGe_WIDth=132  TOP_Margin=3
BOTtom_Margin=3  PUT_CHARacters=-Z?  CONTinued=TOP
CALL_TABle=TABLE  SUPpress_rows_BASE=-2 INDent=0  no special column
```

control STUB_Rank_INDent=2 MINimum_INDent_LEFT=10
NUMber_FORMat=0 HEADer TITLE Title_4 PRinT_PaGe_NUMbers

EDIT_RUN_CHAIN (EDITRUNCH)

Forces a ~EXECUTE RUN_CHAIN for each change in the main EDIT=statement. This is the default. Set on or off (-).

Data tabulation jobs should have only one main EDIT statement with multiple LOCAL_EDIT statements because every new ~EXECUTE EDIT= causes a new pass at the data file and will slow processing down.

ENHANCED_DEFAULT_TITLES (ENHDEFT)

Prints a question's data location along with the actual question text as the default title of the table.

ERROR_LIMIT=# (ERRLIM)

Specifies the maximum number (#) of cleaning errors permissible in the data. The program executes the entire cleaning procedure and quits if the specified error limit has been reached; this might be after more than one case. The default is no error limit or -1. If you set this option equal to either zero (0) or 1, the program will quit after the first case with an error.

ERROR_PROCESS (ERRPRO)

Process ~CLEANER EDIT/ALTER/ENTER commands in a procedure using interactive Survent screens for data modification. The procedure must be referenced in interactive mode to do this (i.e., &filename).

ERROR_REVIEW (ERRRE)

Lists all errors for each case without presenting the Survent screen for data modification. Whether using an interactive command or not, sends a listing of all errors to the list file.

ERROR_STOP (ERRST)

Stops on the first error found in a case by the cleaning procedure and presents the Survent-like cleaning screen for data correction. The default is ERROR_PROCESS. This option is intended for interactive cleaning, i.e., &filename.

ERROR_SUMMARY (ERRSUM)

Print the number of each error message found during a procedure at the end of the procedure (default). Suppress with -ERRSUM.

ERRORS_TO_PRINT_FILE (ERRPRT)

Lists only the errors found by a procedure to the print file. All program messages go to the list file. Once specified, this option can be turned off in a run with -ERRPRT.

IGNORE_SUPPRESS

Overrides all row suppression options. Examples of row suppression include those invoked by the STUB=options [SUPPRESS] and [MINIMUM_ FREQUENCY=1]

LEAVE_ROOM=# (LVROOM)

Sets the amount of memory space to leave open at the end of each tables pass. The default size is 20000 bytes. This effectively controls the maximum table size in a multiple pass run. If LEAVE_ROOM is small, a large table could get size errors if it is the last table in a data pass.

Although making LEAVE_ROOM smaller will allow more tables per pass, a more effective way to do this is by using the CORE: command on the command line (e.g., Mentor <specfile> <listfile> CORE:1000000) to get more memory initially, thus allowing more tables per pass. The default CORE space is 2,000,000 on most machines; you can set it higher up to the actual amount of memory available on your computer (i.e. 4000000 for 4 megabytes). The higher the CORE, the more tables per pass, and faster processing. Refer to *Appendix D: CfMC CONVENTIONS, Command Line Keywords* in your *UTILITIES* manual for more information.

To get the maximum space for a table within the allocated CORE space, use the ~EXECUTE RUN_CHAIN command prior to the table (RUN_CHAIN runs prior tables and clears available memory). Next set LEAVE_ROOM=100 to save a very small amount of the available space for the next table.

LOGGING (LOG)

Logging goes to either the list file or the open print file. If both a list file and a print file are open then the logging record goes to the print file while program messages go to the list file. If only a print file is open then the logging record goes to this file and program messages print to the screen.

Example:

```
con: file sample
con: ma 1.4
ID:  (study code=    , int_id=    ):1.4
Display 1/1.4:
   0
   1234
   ----
   1111
con: abcd
con: ~end
Enter (Y)es or (N)o please-->
con: y
```

In the log file, “con” is short for console, which is anything you type at the keyboard. In this example, the log file shows that the data file SAMPLE was opened in the ~CLEANER block with the FILE command, columns one through four were displayed in MODIFY_ASCII mode and the original data 1111 was changed to abcd, the user issued an ~END command, and the program asked for verification that the case should be changed before terminating.

LOOP_KICKOUT

Stops processing loop variables, such as [(5,2) 11.2], after the first blank category. This allows runs to execute faster. [See *Mentor Volume I: Chapter 6, "Multiple Location Tables"* for a description of loop variables.]

LOOP_KICKOUT stops the loop at the first blank field, even if there is data in fields that follow the blank field.

LOOP_KICKOUT stops only because of blank fields, not missing fields. When a numeric field contains data that does not convert to a number, the field is defined as missing rather than blank, and this would not stop the run. Thus, a variable defined as var1: [(5,2) 11.2#01/02/03] will not stop processing even if the columns 11-20 contain nothing but Xs.

Mentor generates a warning the first time that LOOP_KICKOUT stops processing a variable because of a blank data field. Later variables on which LOOP_KICKOUT may have an effect do not cause a warning to be generated.

If a row variable is defined so that it contains more than one looped variable, the looping does not stop until all loop variables are blank.

When the loop variable is blank, Mentor completes work on the current trip through the loop in order to close statistics, etc. In order to create looped tables having a total answering base, you must include that base in the TABLE_SET or on the row definition. Without this, the tables that are produced will always contain at least one No Answer if the LOOP_KICKOUT takes effect.

MAX_STATISTICS_SIZE=# (MAXSSIZE)

This sets the amount of memory to use for statistical testing. The maxssize setting may be used when doing stats testing across many columns. The default is 60000. If you get an error because maxssize needs to be larger, you can increase the size to whatever value you think you need. This setting is inserted into the spec file, and it should only be used if the user gets an error saying that they should increase their maxssize.

A word of caution: Increasing this value takes away resources from other Mentor features. So, you will need to increase the total amount of memory used by the program by specifying core:####### on the Mentor command line.

Here is an example of the use of maxssize and increasing your core:

Syntax: ~SET MAXSSIZE=150000 (increase maxS 2.5 times)

In order to run with more core, enter something like this on the Mentor command line: `-->mentor xxxx.spx-xxxx.lfl core=500000`

Example:

This example was used to check stats across 54 columns.

```
~in data
~specfiles maxs
~def
tabset=ban1:
statistics=:a-z1
edit=:dostat
}
col=:[11.2#1//52]
}

tabset=row1:
row=:$[base] [13.2] $[] [3.2#1//5] $[mean] [3.2]
}

~set autotab tomanystatsletters=warn maxssize=100000
~exc
maketables
~end
```

MAX_VARIABLE_SIZE=# (MAXVSIZE)

Sets the amount of memory the program allocates for the creation of variables and axes. The default is 200,000 outside of the ~EXECUTE block. This value resets to15, 000 inside the EXECUTE block to make more room for table building.

It may be necessary to set this higher if you have a very large variable and are getting size-related error messages. Note that if you do not have a lot of memory on your computer, setting this higher may result in problems with other operations requiring memory, such as table building. But MAXVSIZE can be set and reset as many times as you like during an application. You can also adjust the amount of system memory Mentor will use with the CORE command (see the *Utilities* manual, Appendix D).

Other SET keywords that affect memory usage are ~SET -VARIABLE_NAMES and LEAVE_ROOM=#. -VARIABLE_NAMES turns off the program's generation of labels for the categories of variables created, and releases the memory it would otherwise use for that task. It would be used, for example, if you were creating a large variable and you knew that you would be providing a STUB for the table you are building rather than using the System's default category names. LEAVE_ROOM= only affects table storage.

Here is an example of a good use of MAXVSIZE.

Example:
```
~SET MAXVSIZE=300000 ''increase maxv 1.5 times
–AUTOMATIC_RESET    ''don't lower during execute
–VARIABLE_NAMES   ''don't create variable labels
~INPUT yourfile
~FREQUENCY [1.10$]=Zipcodes
~EXECUTE
COLUMN=TOTAL ROW=Zipcodes TAB=*
      ~END
```

In this case, we set MAXVSIZE high to generate the variable ZIPCODES from the data file (assuming the data file had many different zip codes in it.)

AUTOMATIC_RESET prevents MAXVSIZE from being returned to15000 to in the ~EXECUTE phase.

MAXIMUM_PAST_CASES=# (MAXPASTCASE)

Sets the maximum number of cases that can be held in memory to allow ~CLEANER NEXT -# to say back up # cases in the data file. The default is nine.

The program can only back up to a case that has been accessed. If you say ~CLEANER NEXT-3, then the program will display the third to the last case it had previously accessed.

MEAN_STATISTICS_ONLY (MEANSTATONLY)

Does ~DEFINE STATISTICS= T-tests on the mean rows only, and overrides default statistics testing on all other rows. This is useful when you only want means tested in your job.

MEDIAN_CELLS=##

Sets the size of the array for median calculations. The default is 50. (An array is sometimes described as “the number of buckets for values.”) This may be put in your mentinit file to apply to all Mentor runs. If a question mark appears where the median should be, you have a “lost median” and need to set MEDIAN_CELLS higher. See *Mentor, Volume I, Chapter 6: Lost Medians* for details.

Related Commands:

~DEF AXIS $[MEDIAN], ~DEF AXIS $[INTERPOLATED_MEDIAN], ~DEF EDIT COLUMN_MEDIAN

NEXT_COLUMN=[location] (NEXTCOL)

Allows you to specify a starting data location, which thereafter can be referenced as *, causing the location to increment by one each time it is referenced. This is especially useful in defining iterative data locations.

Syntax: NEXTCOL=location or [location]

Example:~SET NEXTCOL=[5]
~DEFINE PROCEDURE=C:
>REPEAT $A=01,...,50
CHECK [*^1//5] "error message"
>END_REPEAT }

This would check column 5 for a 1 - 5 punch, then check column 6 for any punch 1 - 5, or X, or Y, etc.

PAGE_LENGTH=# (PGLEN)

Sets the page length in the currently open print file. The default is 66. The minimum is 5. The maximum is 32000. This must be <= the length specified on the meta command >PRINT_FILE command. Maximum page length is determined by the limits of your operating system.

Related Commands:>PRINT_FILE PAGE_LENGTH; ~SET DELIMITED_TABLES; ~SPEC_FILES LINE_LENGTH=; ~DEFINE EDIT= PAGE_LENGTH=

PAGE_NUMBER=# (PGNUM)

Sets or resets the current print file page number to the number given. The default is one.

Related Commands:>PRINT_FILE PAGE_NUMBER, ~DEFINE EDIT=PAGE_NUMBER

PAGE_NUMBER_INCREMENT= (PGNUMINC)

Changes the page numbering increment from the default of one.

Example:~SET PGNUMINC=10

This will create page numbers 1, 11, 21, etc.

PAGE_WIDTH=# (PGWID)

Sets the page width in the currently open print file. Default is 132. Must be >=20 and <= the width specified on the meta command >PRINT_FILE. Maximum page width is only limited by your printer or other software.

Related Commands:

~DEFINE EDIT=PAGE_WIDTH, >PRINT_FILE PAGE_WIDTH

PREFER_FLD_TO_NUM

Affects how variables are created with ~WRITE_SPECS C_Mentor_SPECS. Forces variables that could be converted to a NUM question to be converted to a FLD question. By default, a variable such as [10.2#1//20] is converted to a NUM question, but a variable such as [10.2#1/AA/BB/CC/DD/EE] is converted to a FLD question because there are too many alpha exception codes. Using PREFER_FLD_TO_NUM, both of these variables will be converted to FLD questions.

PREVIEW_TITLES

Allows for creation of .ban and .prt files to review banner definitions for output. The .prt file will include a header, footer and other titles for printing, such as the table title and table of contents.

Example:

~SET

PREVIEW_TITLES

PRINT_ALL_ERRORS_STOP (PRTALLERRST)

Causes a table run executed with ~EXECUTE PRINT_ALL to quit on any error. This is independent of the meta command >QUIT ERRORS=.

PRINT_TABLE_NAME (PRINTNAME)

Allows you to assign table names to be printed on tables that are different from the actual table names stored in the DB file. You can assign one name for the first table and Mentor will automatically increment names on the following tables.

Syntax: PRINT_TABLE_NAME=name

See also ~DEFINE EDIT PRINT_TABLE_NAME for a full description and examples of printing different table names.

PRINT_TCON (PRTTCON)

Prints the table of contents now for the tables you have printed in the run (if you have specified TCON in an EDIT statement). In this way, you can print multiple table of contents in one table printing run.

PROCEDURE_DUMP (PROCDUMP)

Echoes Mentor's process as it compiles and executes a procedure. This is a useful procedure debugging tool, particularly to find the source of program errors that occur while the data is being processed.

Related Command:

ERROR_STOP

PRODUCTION_MODE (PROD)

Suppresses program confirmation to save changes made to a case when you move to a new case. It should only be used during interactive data cleaning or manipulation. This option is also the equivalent of saying ALLOW_UPDATE on any ~INPUT statement. An open ~OUTPUT file and a WRITE_CASE somewhere in your procedure would write changes to *both* the output file and the input file(s) with this option set. In a run using multiple input files (~INPUT NUMBER_IN_ BUFFERS=) any open input file could be updated if this option is set.

Use ~SET TESTING_MODE to return to the system default: do not allow changes to the data file unless it is open with the ALLOW_UPDATE option. The ~CLEANER FILE command opens the data file in PRODUCTION_MODE by default. You must specify PRODUCTION_MODE before an INPUT statement if the data file is opened by more than one session.

REGION_CODING_MODE=#
Modifies how a table is addressed with the region variable. One and two represents the System Total and No Answer columns and rows.

Syntax: REGIONCOD= #

Options:

0

Default; the region columns and rows are named T,NA,1,...

1

The region columns and rows are named 1,2,3,...

RESET

Sets all of the ~SET parameters back to defaults, except commands that affect core memory such as MAX_VARIABLE_SIZE.

SAVE_TABLE="extension" (SAVETAB)

Stores the PRINTED tables back to the DB file, with the name T001<extension>. These numbers (frequencies, percents, and statistics) can be recalled later for table manipulation or comparisons in wave studies. To prevent certain stubs from appearing in the saved table, use the ~DEF STUB option FOR_OUT_TABLES.

SEQUENTIAL_READ

Allows you to read through data files sequentially when using the NEXT command in a ~CLEANER block. (See *Mentor Volume One, Preparing Your Data* for a cleaning example.)

Related Commands:BACKUP, NEXT

SHOW_CASE_INFO

Adds the study name and interviewer ID when a case ID is displayed. The default is -SHOW_CASE_INFO.

Example:`ID: 0001 (study code=RRUN, INT_ID=intv)`

SHOW_MEMORY

Displays how much memory Mentor has available and how memory is allocated. These numbers will vary, depending on your operating system and configuration. For example:

Core Available= 571932

Stack_1=5000

Stack_2=2500

Max_variable_size=60000

Max_statistics_size=60000

Leave_room=20000

Work_room=40000

Stackroom=19158

The default values for memory settings should work for most runs. You can set the amount of memory allocated for Mentor by using the CORE command line keyword. See Command Line Keywords in Appendix D of the *Utilities* manual.

STACK_1=# (STK1)

Specifies the size of an internal memory area used by Mentor. The default size is 5000 on DOS WATCOM, UNIX versions of the program, and 1000 on the DOS 286 version. Users will rarely need to adjust the default setting on this option, and should do so only after consulting with a CfMC support staff member if they get a program error that stack size is too small. The source of a stack overflow is usually caused by very large variables. For example, making a row and stub for a table

based on zip code. In some samples this could lead to a variable having as many as 99,999 categories.

STACK_2=# (STK2)

See the explanation under STACK_1. The default size is 2500 on DOS WATCOM, UNIX versions of the program, and 500 on the DOS 286 version.

START_TCON (STARTTCON)

If you have multiple print files open, this command forces the TCON to be opened using whatever printfile format is currently open. (<PRINT_FILE).

STATISTICS_BASE_AR (STATBASEAR)

Forces Mentor to use the Any Response row as the default base for STATS testing. This overrides the program default, TOTAL row.

Related Commands:~DEFINE STUB=[BASE_ROW]

STATISTICS_DUMP (STATDUMP)

Prints statistics calculated in table building. This prints statistics for T-tests (i.e., All Possible Pairs, Newman Keuls, Kruskall Wallis, etc.), Anova and Chi Square tests. It will not dump stats for ~DEFINE AXIS= options such as $[MEAN, STD, SE].

STATUS

Displays the current ~SET commands in effect including the program defaults.

Here is a list of the default SET options:

PaGe_LENgth=66

PaGe_WIDth=132

TOP_margin=3

BOTtom_margin=3

PaGe_NUMber=1

working sizes=(5000,2500,60000,60000,20000,40000) coreavail=1959752

stackroom=19362

TESTing_mode

ERRor_PROcess

TABle_DRop_mode=0

TABle_STore_mode=0

TABle_MODify_mode=0

TABle_MISSing_mode=0

TABle_PRESent=0

TABle_DRop_WARN=0

REGION_CODing_mode=0

ERROR_SUMMARY

REUSE_LABels

TABle_EFFORT=1

INTeger_TABles

AUTOmatic_ReSET

TABLe_lINEs=0

VARiable_NAMEs

TABle_SET_MATCH_WARN

AUTOmatic_NEW_LINE

AUTOmatic_NEW_PaGe

COLumn_REPeat_OVERRIDE

CLeaN_ERRor_NUMber

One of the items listed for this command is WORKING SIZES. This refers to the sizes of the Mentor's various internal memory areas reserved for the manipulation of such things as variables and tables. See the ~SET keyword SHOW_MEMORY for a description of these numbers.

STRICT_VARIABLE_DEFINITION (STRICTDEF)

Disallows multiple variable specifications per line. By default, you can specify multiple variables on a line. You will get more precise error checking from the program by forcing all specifications to start on a new line.

TABLE_DROP_MODE=# (TABDR)

Determines when tables and regions are dropped from memory. Unnamed regions are always dropped immediately even though the table they reference may stay in memory.

TABLE_DROP_MODE *(cond)*

Options:

0

Explicit unload of every table/region. (default)

1

All tables/regions which were automatically loaded for an operation are dropped. If T1 and T2 were not loaded and therefore were automatically loaded to carry out this operation: CREATE T3 = T1 + T2, they would be dropped from memory directly after T3 was created.

2

Causes a region or table to be dropped from memory after every operation.

TABLE_DROP_WARN=# (TABDRWARN)

Determines how program will react when unloading tables that have been modified since last storing.

Options:

0

Return warning if created table or explicitly loaded table is unloaded. (default)

1

Return warning if anything is unloaded.

2

Return no warning regardless of what is unloaded.

3

Return warning and do not unload table unless the table is explicitly named. (not unload *,!)

TABLE_DUMP (TABDUMP)

Prints the frequencies for each row of the table.

Example:

```
(K3)QUICK_DUMP of table T001 (11,9) at
0054863a:0054b602, status=2

row=-1:  16 - 16 2 6 4 2 3 10 6 7        System Total
row=0:   - - - - - - - - - - -           System No Answer
row=1:   8 - 8 1 4 2 - 2 6 5 1           First data row, etc.
row=2:  4 - 4 - 2 1 - 2 2 3 -
row=3:  7 - 7 - 4 1 - 3 4 4 2
row=4:  7 - 7 1 3 2 - 2 4 4 1
row=5:  5 - 5 - 2 2 - 2 3 3 1
row=6:  6 - 6 1 2 3 - 1 4 2 3
```

row=7: 5 - 5 1 2 - 2 - 3 1 3

The first two numbers (e.g., 8 - 8) is the intersection of the System Total and the first row category. Thereafter, the frequencies for each cell are listed.

Related Command:~DEFINE TABLE=

TABLE_EFFORT=# (TABEFFORT)
Determines the amount of work to do when table processing is requested.

Options:

1

Read specs and data and do tables

3

Read specs and make zero-filled tables

4

Read specs but stop before reading data and making tables. This gives you all syntax errors without actually creating any tables, and a table of contents. If EDIT=TCON is specified, then just a table of contents will print to the open print file.

5

Shows all of the elements in effect in printed table format by providing a dump of the EDIT statement. This includes: default values; the column and row variables; and the category descriptions along with the stubs, banner, and data locations used to calculate statistics.

TABLE_FIELD=variable (TABFLD)
Creates a new set of tables every time the table field changes in the data file. Data should be sorted by table field element.

Refer to *Mentor Volume I"9.4 PARTITIONING DATA FILES"* for an example using this option.

TABLE_LINES=# (TABLINE)

Sets the number of blank lines to leave prior to the table title. The default is 1 blank line.

TABLE_MISSING_MODE=# (TABMISS)

Determines the error level to return when a table reference cannot be found.

Options:

0

Return an error. (default)

1

Warn that the table was not found and return an empty table of appropriate dimensions.

2

No warning returned and a table of the appropriate dimensions is printed.

TABLE_MODIFY_MODE=# (TABMOD)

Determines the error to return when regions and tables do not fit together perfectly.

TABLE_MODIFY_MODE=# *(cond)*

Options:

0

Return an error. Will not return an error when using ~CLEANER CREATE_TABLES. (default)

1

Return warning but carry out the operation.

2

Return no message and carry out the operation.

TABLE_NAME=<name> (TABNAME)

Assigns the name you want used when ~EXECUTE STORE=* or TABLE=* gets executed during table making. Numbers will increment by 1 and letters will alpha kick. The table name must start with a letter (see ~DEFINE EDIT=PRINT_ALPHA_ TABLE_NAMES to override the automatic stripping of beginning letters), and can be from 1 to 14 alphanumeric characters long. The default table name is T001.

TABLE_NAME=!Take the current table name and make it the beginning table name (for PRINT_ALL).

TABLE_NAME=*Set the table name to one past the last one seen.

Related Commands:

~DEFINE EDIT=CALL_TABLE=, SUFFIX=, PREFIX=, PRINT_TABLE_NAME=

TABLE_NUMBER_ADJUSTMENT=# (TABNUMADJ)

Adjusts numbers on tables by the value specified. Use this command to force a number like 4.4999999 to round up to 4.5 and print as “5”.

Options:

*

Get the program default (currently this is to add .0000001).

#

Any positive or negative number

If you choose to use a number other than the program default, be sure to check the numbers in your tables carefully. This command has the same syntax as the meta command >NUMBER_ADJUSTMENT= (*UTILITIES Appendix B: META COMMANDS*). It works on the table immediately after it is made by adjusting all the values except the borders. It is independent of any adjustment made in printing, i.e., you can have neither, either, or both.

TABLE_PRESENT=# (TABPRES)
Determines the error level to return when a table is not found during printing.

Options:

0	Return error
1	Return warning
2	Return no error or warning message

TABLE_SET_MATCH_ERROR (TABSETMATCHERR)
Prints an error if the number of categories in the row and stub variables (as defined with ~DEFINE TABLE_SET= do not match). This includes rows created with either STUB_PREFIX or STUB_SUFFIX in the TABLE_SET= definition.

TABLE_SET_MATCH_WARN (TABSETMATCHWARN)
Prints a warning if the number of categories in the row and stub variables (as defined with ~DEFINE TABLE_SET=) do not match. This is the default. This includes rows created with either STUB_PREFIX or STUB_SUFFIX in the TABLE_SET= definition.

TABLE_STORE_MODE=# (TABST)
Determines when and if a table will be stored into the read/write db file. The default is that tables are only stored when specifically requested with ~CLEANER STORE_TABLES.

Options:

0 Tables are stored only when explicitly stored with ~CLEANER STORE_TABLES.

1 Stores any created table after modification.

2 Stores any created and/or loaded table after modification.

3 Stores any table whatever its origin after modification.

TESTING_MODE (TEST)
Does not allow any changes to the data file, unless the file is opened with ALLOW_UPDATE.

Related Commands:PRODUCTION_MODE

TOP_MARGIN=# (TOP)
Specifies the number of blank lines to leave at the top of every page for formatted output from ~CLEANER PRINT_LINES.

Related Commands:~DEFINE EDIT= TOP_MARGIN=; > PRINT_FILE name, TOP_MARGIN=.

TRAINING_MODE (TRAIN)
Does not allow any changes to the data file.

Related Commands: PRODUCTION_MODE

UNDEFINED_TABLE_CELLS= (UNDEFTABCELL)
Indicates whether or not undefined table cells are allowed in a table.

Options:

WARN

The default; a warning is generated when cells cross that by definition do not exist (e.g., a $[MEAN] crossed with a $[MEAN]). The warning appears when Mentor builds the table and a question mark appears in the undefined cell(s).

OK

Suppresses the warnings.

WARN

Mentor will generate an error an exit before building the table.

UNWEIGHTED_TOP (UNWTTOP)

Prints two additional rows at the top of every table, unweighted Total and unweighted No Answer. This option causes these two extra rows to print whether the table is weighted or not, so that the same stub label set will work on both weighted and unweighted tables.

VARIABLE_NAMES (VARNAME)

Generates category names for variables defined under ~DEFINE. Say - VARIABLE_NAMES to define bigger stub variables more efficiently. You can set this in MENTINIT file in the CFMC CONTROL directory or group to have it in effect for all sessions.

This option might cause the program to generate an error requesting that STACK_2 be increased. This has to do with the internal program memory areas. Contact the CfMC hotline for assistance if this happens. An example might be program-generated variable names for a large list like a zip code table.

ZERO_FILL (ZF)

Zero-fills any ~CLEANER MODIFY and TRANSFER commands or procedure operations.

~SET WEB_TABLES

"TCON_ANCHOR_TEXT=" will remove the current anchor tag text of BACK TO TABLE OF CONTENTS from the html file. "TCON_ANCHOR_TEXT=" is text provided by the user that will replace the current anchor tag text of BACK TO TABLE OF CONTENTS in the html file. The default text would remain BACK TO TABLE OF CONTENTS.

~SHOW

Displays the structure and content of any item stored in a db file including: data and text variables; tables; table sets; expressions; etc. Data variables show the data

columns, any text, variable or Survent question type, and the recode table if the variable comes from a Survent CAT or FLD question. If a data file is open, ~SHOW will also display the responses of the current case to the variable.

Syntax: ~SHOW <varname or variable>

Setting the dump switch >DUMP D2 will list the categories of a variable or expression on separate lines.

Options:

*****

Display the entire case in ASCII card image format.

***B**

Display the entire case in column binary (punch) format.

varname or variable

You can reference existing variables, define them interactively, or read them from a specification file. ~SHOW will show anything that can be defined in the ~DEFINE block, as well as ASCII specification files stored in a db file.

This option is especially useful for previewing what and how many categories you will get from an expression for a table banner using a joiner such as BY or WITH, or a modifier like *F.

Example:>DUMP D2

~SHOW
BAN1: TOTAL WITH ((TOTAL WITH [3,5*F^1/2] WITH [3,5^1-2]) &
BY [9^1//3])

categories:
#1: TOTAL

#2: TOTAL & 9^1
#3: 3,5*F^1 & 9^1
#4: 3,5*F^2 & 9^1
#5: 3^1-2 & 9^1
#6: 5^1-2 & 9^1
#7: TOTAL & 9^2
#8: 3,5*F^1 & 9^2
#9: 3,5*F^2 & 9^2
#10: 3^1-2 & 9^2
#11: 5^1-2 & 9^2
#12: TOTAL & 9^3
#13: 3,5*F^1 & 9^3
#14: 3,5*F^2 & 9^3
#15: 3^1-2 & 9^3
#16: 5^1-2 & 9^3

The SHOW command reports category numbers for stats, netted, and exclusive categories.

Example:

SHOW-->a:[5^1,2/3//5/(STATS)6/(EXCLUSIVE)Y]
a(224,1)($C: 5) [1/5.1 CAT]
categories: #1: 5^1-2 #2: 5^3 #3: 5^4 #4: 5^5 #5: 5^6 #6: 5^12
Stat tested cats: 5
Multi resp cats: 1
Exclusive cats: 6

You must enclose the data category in quotes if it is not a number. If it starts with a letter, it does not require quotes.

Example:

~SHOW
a[5.2#1a]
b[5.2#"1a"]
c[5.2#RF]

Related Commands:~CLEANER SHOW

~SORT

Sorts data records in the order specified by the given sort criteria. Sorting keys can be previously defined variable names or exact data location specified in brackets. Requires an output file that the newly sorted data file will be written to. Processes most efficiently when the incoming data file is randomized.

Syntax: ~SORT option=datavar,option=datavar

Example:

~INPUT <unsorted file>

~OUTPUT <sorted file>

~SORT [5.2$], D=[10.3$]

~END

Options:

option=

Not required. If it is not present, the data is sorted on each key (datavar) in ascending order.

ASCENDING (A)

This is the default order.

DESCENDING (D)

Sort in descending order.

BREAK_VARIABLE=4 column datavar (BV)

The four-column field should be looked at as two 2-column fields. The first 2-column field expresses a comparison between the current case and the case before it. The field will show the number of the sort key that has changed since the last case. If more than one sort key changes, the first sort key that has changed will appear. If the cases match on all sort keys, the value of the number of sort keys + 1 will appear in the field. On the first case in the data file, "00" will appear in the first two columns. The second 2-column field will show the number of the sort key that will change on the next case. If more than one sort key changes, the first sort key that changes will appear in the field and if the cases match on all sort keys, the value of the number of sort keys + 1 will appear in the field. On the last case, "00" will appear in the second 2-column field.

Example:

~INPUT <unsorted file>

~OUTPUT <sorted file>

~SORT BV=[701.4],[1.4$]

~END

datavar

Is any data variable (often the case ID location). Be sure to use string data variables if you want to sort on ASCII characters; i.e. [1/1.4$]. SORT will use the sort key on valid data, and leave things outside the key in the original order at the end of the file.

The total number of sort keys you can use is determined by a 100-column limit. String and numeric variables are stored differently, so the maximum number of sort keys is dependent on the type of variable being sorted. String variable keys are stored as their actual column width, so there is no limit to the number of keys you use, as long as the column width total is less than or equal to 100.

For example, you could have 100 sort keys, each 1 column wide or 10 sort keys, each 10 columns wide, or any combination of string variables whose columns total less than or equal to 100. Numeric variable keys are stored as 9 columns regardless of the variable's actual width, so the maximum number of numeric variable sort keys is 11. If you use a combination of string and numeric variables, the total

number of string key columns plus 9 times the number of numeric variable keys must be less than or equal to 100 columns.

String variable column width total + 9 (number of numeric variables) < 100.

~SORT is not case sensitive by default. You may set case sensitivity with ~SET CASE_SENSITIVE.

The ~SORT command will not actually begin executing until another tilde command is given, allowing any number of keys to be entered on multiple lines.

See also The *UTILITIES* manual for information on the menu-assisted utility COPYFILE which has a SORT option.

~SPEC_FILES (SPEC)

Opens program-generated files such as TAB,LPR,QSP. Specifications are written to these files when a spec-generating command is given.

Syntax:~SPEC <filename>,option

Options:

filename

Can be 4-8 characters. You will overwrite existing files of the same name if >PURGE_SAME is specified.

LINE_LENGTH=# (LINELEN)

Allows you to specify a width greater than the default of 132 for specification files opened by the program. Specify a number as large as you need. Specifically, this is very useful to write out large tables for the purpose of loading them into spreadsheet programs.

~DEFINE TABLE_SET= or TABLE_SPECS= will cause specifications (e.g., tabset=varname) to be written to the TAB file in the order defined. Likewise, ~EXECUTE MAKE_TABLES or STORE_TABLES will cause specifications (e.g., load=T001 print) for table printing to be written to the LPR file in the order the tables are executed.

This also opens a QSP file which is written to by ~WRITE_SPECS, ~CLEANER WRITE_QSP, and ~SET DELIMITED_TABLES.

~SPEC_RULES

Controls how variables will be written to the DEF file (created either by ~PREPARE COMPILE Mentor or ~WRITE_SPECS Mentor). ~SPEC_RULES must be specified *before* either of these commands.

Syntax: ~SPECRULE option, option

Options:

BASE

Converts PREPARE IF statements to BASE and TITLE_4 statements in the Mentor DEF file.

IF statements are passed as written to the DEF file, meaning PREPARE question label references are not converted to data locations. This requires that you open the matching db file (>USE_DB <study_name>).

Related Commands:>USE_DB

BASE_COMMENT

Converts PREPARE IF statements to BASE and TITLE_4 statements in the DEF file, but comment them out.

CLN_CHECK (CLNCHK)

Produces a CLN file with CHECK [datavar] statements. The default is to produce EDIT varname statements which requires you to use the matching db file (generated by the compilation of the Survent specifications) in a cleaning run.

Only CAT, FLD, NUM, TEX, and VAR type questions are passed to the CLN file.

COLUMN_MEAN= (MEAN)

Customizes the default statistics written to the DEF file for NUM questions and by the !MISC rating= option used on rating scale questions.

Syntax:MEAN= "<string>" WITH <stub item>

Options:

"string"

The string to put out as the statistics for the variable in the DEF file i.e., "$[MEAN,STD,SE,MEDIAN]"

<stub item>

Stands for the name of the user-defined stub item to be appended to the corresponding stubs.

Example:~DEFINE

STUB=statstub:

[SKIP_LINES=2,STATISTICS_ROW] Mean

[STATISTICS_ROW] Standard Deviation

[STATISTICS_ROW] Median }

~SPECRULE MEAN="$[MEAN,STD,MEDIAN]" WITH statstub

~PREPARE COMPILE Mentor_SPECS

{QN1:
!MISC rating=5
Rate this service
!CAT
1 Excellent
2
3
4
5 Unsatisfactory
6 No Answer }

{QN@:
How much would you pay for this item?
!NUM,,2,1.99-3.99,,RF}
.
.
.
~END

DO_LOOPS (LOOP)

Writes out Survent LOOP questions in Mentor loop variable syntax. This is the default.

Related Commands:Refer to (# loops, INCR) under ~DEFINE VARIABLE=.

JUSTIFY=left/center/right

Justifies the titles in spec files.

NO_BASE

Does not convert PREPARE IF statements to BASE and TITLE_4 statements. This is the default, and need only be specified if you need to turn off the BASE option.

PREPARE_Mentor_USAGE (PREPMentorUSAGE)

Translates tabsets into ~PREPARE specs. This is useful if you do not have Survent and you want to use a CfMC feature that requires a QFF file, such as REFORMAT.

SPSS_MULTI_RECORD_FORMAT

Creates SPSS specs and data for 1000-column records. This is useful for SPSS versions 5.0 to 8.0. Without this option, SPSS specs are written with absolute column numbers.

STORE_TABLES (STORETAB)

Causes the default table names starting with T001 to be included in each TABLE_SET definition. If you save your tables in a DB file then the table name will be stored with the other table elements defined for that table. You can edit this item in the DEF file to change these names.

STUB_DEFAULT=[option] (STUBDEF=)

Specifies a default stub [option] for all STUB= lines written to the DEF file. See ~DEFINE EDIT= STUB_DEFAULT

TABLE_SET(TABSET)

Writes out a TAB file with references only to TABLE_SET=<name>, and writes TABLE_SETs for each of the variables in the DEF file. This is the default.

-TABSET will write out the title, stub, and run as separate variables in

the DEF file.

USE_PRINT_ENHANCEMENTS (USEPRTENHANCE)

Passes all Survent backslash commands (e.g., \B,\U, \:label) to stub labels and titles in the DEF file. Printing is then controlled by the laser control file in the CFMC CONTROL directory or group (refer to the meta command >PRINT_FILE LASER_CONTROL= in your *UTILITIES* manual).

VARS_TABSETS

Puts tabsets for VAR and TEX questions in the DEF file.

See also *Mentor Volume I: 4.7 USING PREPARE TO GENERATE Mentor SPECIFICATION FILES* for additional examples.

~STOP_WATCH

Displays timing information. The first number is the elapsed time in minutes and seconds since the start of the job. The second number is the elapsed time in minutes and seconds since the last ~STOP_WATCH or >STOP_WATCH command.

Syntax: ~SW

Related Commands:See meta command >STOP_WATCH in your *UTILITIES* manual.

~TRANSLATE

This command is being phased out and, while it is currently functional, it will issue a warning when you use it. Instead of ~TRANSLATE, use the WRITE_NOW option to ~OUTPUT or the COPYFILE utility to import and export files. COPYFILE is in the *Utilities* manual.

Translates a TR file to either ASCII or binary format (export) or the reverse, ASCII or binary file to a TR file (import).

Syntax: ~TRANS filename,action,type,option,option

Filename

May be specified as NULL or $NULL, meaning the file will not be saved to disk when you write it (no case is made), and an immediate end-of-file is read when the NULL file is read.

Actions :IN=#.#

Specifies that the file is being imported. An output file (see ~OUTPUT) must be opened. #.# is the location and length of the case ID of the new file; if not specified, it defaults to 1.10.

OUT

Specifies that the file is being exported. An input file must be open (See ~INPUT). The file will be written out according to the parameters set on the ~INPUT file as to records to write and case length.

Types:ASCII

Standard text type data format. Multiple punches are not kept, only ASCII codes (A-Z, 1-9,0, and other standard characters). This is a universal format.

BINARY (BIN)

Standard multi-punch format. Records must be fixed length with 80 columns per record. This format is used on most micro-computers.

CARDS_IMAGE (CARD)

Like ASCII, but written out as 80 byte records.

HEX

Translates the file in hexadecimal format meaning: each column converts to three ASCII characters; 0123456789abcdef are the 16 bits; bits 0, 1, 8, and 9 are not used and are checked to be off when output is created in this format; 234567abcdef are the bits used in groups of 4 bits; a hex character is made. This can be converted to any other format on unusual computers.

SWAPPED_BINARY (SWAPBIN)

Binary with each 2 bytes exchanging position (swapped). This is standard IBM 360 column binary format, and is used on most mini-computers and mainframes.

SWAPPED_HEX

Translates the file in swapped hexadecimal format, meaning it swaps every two bytes of data.

UNCOMPRESSED (UNPRESS)

Translates the file in uncompressed format.

Options:

DOTS=# (DOT)

Controls the number of cases read for each dot printing on the console when the file is read. The default is 10.

STOP_AFTER=# (STOP)

Stops the translation after reading # cases. Only used with IN= action.

Example:~TRANS NULL,IN=1.4,ASCII

This writes all the tables in a DB file out to an ASCII file named TABSOLD.QSP:

Example:~INPUT BANK
~TRANSLATE BANK^ASC,OUT,ASCII

This translates a TR file named BANK to an ASCII file called BANK.ASC (under DOS or UNIX). The length of the file BANK.ASC is the same as BANK.TR.

Example:~OUTPUT BANK,CASE_LENGTH=800
~TRANSLATE BANK^ASC,IN=1.4,ASCII

This translates an ASCII file named BANK.ASC to a TR file named BANK.TR. CASE_LENGTH= is required on the output statement. If the length specified on the output statement is shorter than the length of the file being translated, you will get an ERROR message and the file will not be translated. If the length specified on the output statement is longer than the length of the file being translated, the output file will be created and the extra columns will be blank.

~UPDATE

Writes the current case (including any modifications) back to core (the case in its original form *will* be overwritten).

Syntax: ~UPDATE

Related Commands:~CLEANER UPDATE

~VIEW

Views a case. You must have a valid QFF file and a case in hand to use this command. This command displays the case in the same way as the VIEW option in a Survent session. You must first open the matching compiled questionnaire file with a ~QFF_FILE <studyname>. Alter the response to a question by entering the keyword ALTER, and then enter the new response. Use GOTO to move around in the questionnaire and ABORT to get out. Refer to your *SURVENT* manual *4.1.3 VIEWING A PREVIOUS INTERVIEW.*

Syntax:~VIEW

Related Commands:~CLEANER VIEW

~WRITE_SPECS (WRITESPEC)

Writes out the ASCII specification in one of several formats for compiled Mentor row or text variables, or tables. Items must be available either in the current program run or in an open DB file. This command and ~CLEANER WRITE_QSP are the only ways you can either translate a Mentor variable into another format or to unload finished tables for transfer to a different CfMC-supported platform.

Before specifying the WRITESPEC command you must instruct the program to open internal files with ~SPEC_FILES <name> command. Depending on the WRITESPEC option you choose, Mentor will write ASCII specifications to one or more of several program-generated files.

Syntax:~WRITESPEC option

Options:

Mentor_SPECS= varname
Writes out a C-Mentor TABLE_SET statement that includes a title, stub labels and row definition, and a table making statement.

COSI_SPECS [Replaced PERSEE_SPECS= varname (PERSEE)]

Creates COSI specifications.

QUANTUM_SPECS= varname (QUANTUM)
Writes out data definition and table making statements used by the QUANTUM program.

SPSS_SPECS= varname (SPSS)
Writes out data definition, stub labels, table making and printing statements used by the SPSS program.

SSS_XML_SPECS
Causes REFORMAT to export a fixed ascii file with data definitions in "Triple S" XML format. Files created in this way can be imported into any program that supports this format.

Varname must reference either a simple variable (not an expression or axis such as [5^1] WITH [6^2]) or a variable generated out of a Survent question (i.e., only a type=1 db item from >LIST_DB_CONTENTS). This also means you cannot reference a row varname generated from a ~DEFINE TABLE_SET= ROW= statement since Mentor considers it an expression.

If your variable was not generated from a Survent question specification then you will not get any title text and only default stub labels for each data category defined in the variable. You could write out the text variables with the TEXT= keyword and then merge the two files. (CMentor)

Example:

```
~DEFINE a[6^1//5]
~SPEC_FILES sample
~WRITESPEC CMentor=a
```

Specification written to SAMPLE.DEF:

```
tabset= { a_z:
title=:
  }
stub=:
 6^1
 6^2
 6^3
 6^4
 6^5}
row=: [1/6.1^1/2/3/4/5]
}
```

You could define a title and stub labels as follows:

```
~DEFINE
a[$T="This is the title"6^Poor:1/OK:2/Average:3/Good:4/"Really Good":5]
```

Specification written to the DEF file:

```
tabset= { a_z:
title=:  This is the title
}
stub=:
Poor
OK
Average
Good
Really good }
row=: [1/6.1^1/2/3/4/5]
```

}

Specifications can produce up to four separate files:

Extension:	Type of item:	Used by:
DEF	Data definitions	CMentor,PERSEE, QUANTUM,SPLM,SPSS
TAB	Table creating commands	CMentor,QUANTUM, SPLM,SPSS
LAB	Table stub labels	SPLM,SPSS
LPR	Table printing commands	SPLM,SPSS

TABLE=tablename

Writes out an ASCII version of the table from a db file into the program-generated spec file (QSP) preceded by the keyword TABLE=. This can then be modified or read directly into another db file using the ~DEFINE TABLE= command along with the generated table written.

This is the same as the ~CLEANER WRITE_QSP command.

You might use TABLE= where the sample size is very large. The basic tables could be run on an HP3000/900 series or other mini because of its speed and disk capacity, but table manipulation work could be done on a PC by making ASCII versions of the tables needed and transferring them to the PC.

This example unloads the tables from a db file and writes them out as ASCII specifications to the program-generated QSP file. Note: Refer to your *UTILITIES* manual for information on the meta commands (>) used here.

Example:

```
>USE_DB DB1
 >LIST_DB ,unld,TYPE=TABLES,TEMPLATE="TABLE=!"
 ~SPEC_FILES ascii_db
```

```
~WRITESPEC
&unld.DCL
~END
```

The >LIST_DB statement used in this example writes a statement to an ASCII file (UNLD.DCL for DOS/UNIX or UNLDDCL) for each table variable in the db file DB1. Each table variable name is substituted for ! after the ~WRITE_SPECS keyword TABLE=. &UNLD.DCL instructs ~WRITESPEC to read this file and write out ASCII versions of each the table named in the file.

Here is a sample table written out as ASCII specifications to the QSP file:

```
table=  T001: { 2,11,  2,11
 header=  tabtop_h
 footer=  tabtop_fo
 title=  qn1_t
 column=  tabtop_c
 row=  qn1_r
 banner=  tabtop_bn
 stub=  qn1_s
 main_edit=  tabtop_e
r=-1 25. - 25. 5. 5. 10. 1. 11. 8. 5. 15.
r=0 - - - - - - - - - - -
r=1 7. - 7. 2. 1. 1. - 4. 1. - 4.
r=2 5. - 5. - 1. 4. - 3. 2. 1. 4.
r=3 6. - 6. 2. 1. 2. - 2. 1. 2. 4.
r=4 2. - 2. - 1. 1. - 1. 1. 1. 1.
r=5 3. - 3. - 1. 1. 1. 1. 1. - 1.
r=6 2. - 2. 1. - 1. - - 2. 1. 1.
r=7 23. - 3.47826 4. 3. 3.33333 1. 3.72727 3.16667 3. 3.64286
r=8 23. - 1.3774 1.1547 1.58114 1.22474 ? 1.3484 1.47196 0.8165 1.21574
r=9 23. - 0.28721 0.57735 0.70711 0.40825 ? 0.40656 0.60093 0.40825 0.32492
}
```

For an explanation of the above output refer to ~DEFINE TABLE=.

You could then reload an ASCII version of the table into a db file on a new platform from this file:

```
>CREATE_DB DB2
~DEFINE
&-ascii_db.QSP
~END
```

&- instructs the program not to echo the spec file to the screen.

TEXT=varname
Will write text variables from a db file into the program-generated spec file (QSP) preceded by the keyword TEXT=.

Example:

```
>USE_DB samp
        >LIST_DB ,texvars,TYPE=123,TEMPLATE="TEXT=!"
~SPEC_FILES texvars_db
~WRITESPEC
&texvars.DCL
~END
```

Below is a sample from the QSP file showing a stub label set from the variable STUB1 and the title text from the variable TITLE1.

Example:

```
db_specfile STUB1:
  18-25
  26-32
  33-39
  40-60
  [stat] Mean
  [stat] Standard deviation
  [stat] Standard error}
db_specfile TITLE1:
  age of respondent}
```

Change db_specfile to LINES= to read this back into a DB file.

Refer to the *UTILITIES* manual for information on the meta (>) commands used in these examples.

~WRITE_QUOTA

This command reads and writes quota files, such as QUOTAMOD'S "OUT" command. You can write out an ASCII file from a quota file using Mentor. This is useful if you later want to parse the ASCII output file using Mentor.

Syntax:

~writequota xxxx<,savecounts><,outnonzero><,filename>

By default it will create "xxxx.aqu" which is an ASCII copy of the quota file. If you say “~writequota xxxx,yyyy”, it will write to the file yyyy.aqu.

You can give a fully qualified filename, such as “/home/myfiles/yyy.abc” and it will write there. You can also use the option "save_counts" to save the interviewer time and other counts at the end of the file, so that when you build a new quota file it will keep those counts intact; by default they are saved but not importable unless you remove the "*" from the front of the line. The option "outnonzero" will only write quotas that have values greater than zero. For example, an .aqu file with a few quotas and the counts:

MALE = 0

FEMALE = 0

COMPLETE = 0

TERM_IN_PROG = 0

*survent_totaltime = 000000:00:00

*survent_currentime = 000000:00:00

*interviewing_totaltime = 000000:00:00

*interviewing_currentime = 000000:00:00

*unique_number = 0

*next_caseid = 1

end

You can modify this file before using quotamod's "IN" option to update the quota file's quota values or other values.