

# Contents

.....

<b>1 Getting Started .....</b>	<b>13</b>
Introduction.....	13
1.1 WHAT IS MENTOR?.....	13
1.2 BENEFITS OF USING MENTOR.....	13
Complex Feature Set.....	14
Powerful Command Language.....	15
Advanced Database capabilities.....	16
Integration of Survent And Mentor.....	17
Ease of Use.....	18
1.3 THE BASIC STEPS OF THE REPORT/TABBING PROCESS.....	19
1.5 SUMMARY OF THE CHAPTERS.....	22
1.6 GETTING TECHNICAL SUPPORT.....	24
<b>2 Preparing Your Data.....</b>	<b>27</b>
Introduction.....	27
2.1 OVERVIEW OF THE CLEANING PROCESS.....	27
Cleaning Steps.....	27
2.2 CLEANING DATA.....	28
2.2.1 Other Types of Data Files.....	29
2.2.2 Why Clean the Data?.....	29
2.2.3 Understanding The Questionnaire.....	30
2.3 CLEANING SPECIFICATIONS.....	32
Generating A List Of Error Messages.....	33
Sample Error Listing.....	37
Program-Generated Error Messages.....	40
2.3.1 Cleaning Examples.....	41
Cleaning Punch Data.....	43
2.3.2 Correcting Errors.....	66
Manually Cleaning The DATA.....	66
Auto-fixing The Data.....	71

# Contents

2.3.3 Subsequent Cleaning Runs . . . . .	76
2.4 CLEANING WITH SURVENT VARIABLES . . . . .	77
A Sample Survent Questionnaire . . . . .	79
Cleaning Specifications Generated By A Compile . . . . .	80
Alternate CLN File . . . . .	82
Custom Cleaning Specifications . . . . .	82
Condition And Branching Statements . . . . .	83
Variable Modifiers . . . . .	85
Generating A List Of Error Messages . . . . .	87
Program-Generated Error Messages for Survent questions . . . . .	89
2.4.1 Correcting Errors . . . . .	91
Using Survent-type Cleaning Screens . . . . .	91
Modifying TEX Question Responses . . . . .	93
Auto-fixing The Data . . . . .	94
2.4.2 Subsequent Cleaning Runs . . . . .	97
2.5 REFERENCE . . . . .	98
2.5.1 Quick Reference: Cleaning Commands And Examples . . . . .	98
. . . . .	104
Example CHECK Statements . . . . .	104
2.5.2 Sending Error Messages To A Print File . . . . .	107
2.5.3 Specifying More Than One Command Per Line . . . . .	109
2.5.4 Additional Commands . . . . .	109
<b>3 Reformatting Your Data . . . . .</b>	<b>111</b>
Introduction . . . . .	111
3.1 WHY REFORMAT DATA? . . . . .	111
The Overall Structure . . . . .	111
Rules For Manipulating Data . . . . .	112
3.1.2 Blanking Data . . . . .	114
3.1.3 Printing Text and Data Fields . . . . .	114
3.1.4 Data Manipulation for Punch, String, and Numeric Variables . . . . .	118
Direct Data Moves . . . . .	118
3.1.5 Data Manipulation for Predefined Variables . . . . .	133
3.1.6 Relational Operators . . . . .	146
3.1.7 Formatting Data Elements . . . . .	151
Zero-Filling Data . . . . .	151
Decimal Points in Data . . . . .	153

Spreading Multi-Punched Data . . . . .	154
Transforming Numbers Into Strings . . . . .	156
Recoding 10-Point Scales . . . . .	156
Recoding To Exclude Selected Responses . . . . .	157
Recoding To Reverse A Scale Question . . . . .	157
3.1.8 Data Manipulation in the ~CLEANER Block . . . . .	158
3.2 Creating Subsets of Data Files . . . . .	159
HOLD_OUTPUT_UNTIL_SUBSET . . . . .	159
~EXeCUTE Do_subset . . . . .	160
Sampling=#n and sampling=.n . . . . .	160
Try_for_sampling=#n and try_for_sampling=.n . . . . .	161
Select= . . . . .	161
Casewritten . . . . .	161
Combining options . . . . .	162
Num_sample_cases= . . . . .	164
Repeatable Subset Results . . . . .	165
3.3 Mentor EQUIVALENTS TO SPL . . . . .	165
<b>4 Basic Tables . . . . .</b>	<b>169</b>
Introduction . . . . .	169
4.1 PARTS OF A TABLE . . . . .	169
4.2 TABLE BUILDING BASICS . . . . .	172
4.3 DEFINING TABLE ELEMENTS . . . . .	174
4.3.1 Assigning Variable Names . . . . .	178
Default Varname Generation . . . . .	180
4.3.2 Changing Table Element Defaults (The DEFINE EDIT Statement) . . . . .	181
The Three Levels of EDIT . . . . .	184
4.3.3 Changing Table Processing Defaults (The SET Statement) . . . . .	185
4.4 TABLE BUILDING (The INPUT and EXECUTE statements) . . . . .	188
Printing Individual Tables (Using TABLE_SET or TABLE=) . . . . .	189
Storing Tabsets in the DB file (Using STORE_TABLES) . . . . .	190
Making Several Tables (Using MAKE_TABLES) . . . . .	193
4.5 META COMMANDS . . . . .	195
The DB File . . . . .	196
4.6 DEFINING DATA . . . . .	197
Data Types . . . . .	203
Using Punctuation to Create Categories . . . . .	205

# Contents

Joiners .....	207
4.6.1 Summary of Rules for Defining Data .....	208
Samples of Data Field Locations. ....	208
Punctuation Used In Referencing Data Field Locations .....	210
Category Definitions Using Caret (^) For Punch Data .....	210
Punctuation Used In Defining Punch Data .....	212
Category Definitions Using Pound Sign (#) For ASCII And Numeric Data .....	213
Punctuation Used In Defining ASCII And Numeric Data .....	214
4.7 DEFINING THE BANNER .....	215
4.8 FORMATTING BANNER TEXT .....	218
Sample Specifications And Table .....	220
The STUB TABLE_SET. ....	221
Formatting A Banner Wider Than 80 Columns .....	223
Editing the Banner .....	227
4.9 GENERATING BANNER SPECS. ....	227
How to create a banner using make_banner format .....	237
4.10 DEFINING INDIVIDUAL TABLES .....	245
How To Add Ranking To A Table .....	246
How To Add A Base To A Table .....	247
How To Add Summary Statistics To A Table. ....	249
4.11 SAMPLE SPECIFICATION FILES. ....	251
Using the DB File .....	251
Putting It All Together. ....	254
Sample Table .....	264
Defining a Procedure for Complex Banners .....	266
4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES .....	268
CAT Question .....	273
FLD Question .....	273
Survent specs .....	274
Mentor tabset .....	274
Mentor tabset .....	275
NUM Question .....	276
Survent specs .....	277
Mentor tabset .....	278
Survent specs .....	278
Mentor tabset .....	279
Changing What Appears in the DEF File .....	279
!MISC option. ....	280

NUM_EXCEPTIONS=subtitle,subtitle .....	281
USING E-TABS .....	305
More E-Tab Options .....	308
<b>5 Intermediate Tables .....</b>	<b>309</b>
Introduction .....	309
5.1 Expressions and Joiners .....	309
Logical Joiners .....	310
Vector Joiners .....	310
Math Joiners .....	311
5.1.1 Logical Joiners .....	312
5.1.2 Vector Joiners .....	315
WITH .....	315
BY .....	315
WHEN .....	316
INTERSECT .....	317
NET .....	317
OTHERWISE .....	318
JOIN .....	319
5.1.3 Mathematical Joiners And Operators .....	325
5.2 Axis Commands/Cross-Case Operations .....	326
5.3 Changing Table Specifications .....	330
Global Print Options .....	331
Column Print Options .....	332
Row Print Options .....	332
Sample Table Printed With Default Options .....	333
Print Options .....	335
Changing Percent Base Within A Stub .....	360
5.4 Printing Multiple Banners For Each Table Row .....	363
5.5 TABLE NAMES .....	366
Printing Leading Alpha Character .....	367
Specify Starting Name .....	368
Printing Name With Prefix Or Suffix .....	369
Replacing "Table" .....	369
Specifying Unique Table Names .....	370
Printing Different Table Names .....	372
5.6 Reprinting Tables .....	372

# Contents

Accessing The DB File . . . . .	374
Defining A New Edit Statement And Table Header . . . . .	374
Reprinting The Tables . . . . .	375
Adding Statistics Rows To Finished Tables . . . . .	377
<b>6 Advanced Tables . . . . .</b>	<b>381</b>
Introduction . . . . .	381
6.1 TOP BOX/BOTTOM BOX SUMMARY TABLES . . . . .	381
6.1.1 Top Box Tables with Constant Percentage Base . . . . .	382
6.1.2 Top Box Tables with a Changing Percentage Base . . . . .	386
6.1.3 Ranking of Top Box Tables . . . . .	389
6.2 SUMMARY STATISTICS (MEANS) . . . . .	392
6.2.1 Means on Rating Scales Using the Variable Definition . . . . .	393
With No Recoding Needed . . . . .	393
With A Numeric Don't Know Excluded . . . . .	398
With The Scale Reversed . . . . .	401
With The Scale Reversed And DK/NA Coded As Numeric . . . . .	403
With 10 Coded As A Zero, An X, Or Y . . . . .	404
6.2.2 Means For Range Type Variables . . . . .	407
Interpolated Medians . . . . .	408
6.2.3 Means For Numeric Data . . . . .	412
With No Recoding Necessary . . . . .	412
With Don't Know Coded As A Number . . . . .	415
With A Numeric Value Coded As A Non-Numeric . . . . .	416
6.2.4 Summary Statistics in the Column Variable . . . . .	419
Summary Statistics In Both The Column And The Row . . . . .	421
6.2.5 Means And Medians Using The EDIT Options . . . . .	424
Means On A Rating Scale . . . . .	425
Means On A Rating Scale With Rows In The Middle That Need To Be Excluded . . . . .	427
Means On A Range Variable . . . . .	429
Changing The Default Print Options . . . . .	430
Column Medians . . . . .	433
Percentiles . . . . .	443
6.2.6 Mean Summary Tables . . . . .	446
Rating Scales With No Recoding . . . . .	446
Rating Scales With Recoding Needed . . . . .	449
Rating Scales With The Don't Know Coded As A Numeric . . . . .	449

Rating Scales With The Scale Reversed . . . . .	451
Rating Scales With The Scale Reversed and Don't Know Coded As A Numeric . . . . .	452
Rating Scales With 10 Coded As A Zero (0) . . . . .	453
Range Variables. . . . .	454
Numeric Data With The Don't Know Coded As A Non-Numeric . . . . .	455
Numeric Data With The Don't Know Coded As Numeric . . . . .	457
Numeric Data With A Numeric Value Coded As A Non-Numeric Code. . . . .	457
Using the "BY" joiner. . . . .	459
6.2.7 Means Scattered Throughout The Table . . . . .	461
Mean/Frequency Summary Table . . . . .	465
6.2.8 Summary Statistics with Arithmetic . . . . .	467
6.3 WEIGHTED TABLES. . . . .	470
6.3.1 Weighting with Weight Value already Stored in the Data. . . . .	472
6.3.2 Weighting using the SELECT Function . . . . .	476
6.3.3 Printing Both a Weighted and an Unweighted Total Row . . . . .	476
6.3.4 Storing the Weight in the Data. . . . .	479
6.3.5 Assigning Different Weights to Different Banner Points . . . . .	481
6.3.6 Printing Both a Weighted and an Unweighted Total Column . . . . .	484
6.3.7 Assigning Different Weights To Different Rows . . . . .	487
6.3.8 WEIGHTING USING MULTIPLE FACTORS . . . . .	490
6.4 SUMMARY TABLES (MARKET SHARE) . . . . .	490
6.5 HOLECOUNT AND BREAK TABLES. . . . .	493
6.5.1 Holecourt Table with Different Brands (Locations) in the Banner. . . . .	495
6.5.2 Holecourt Table with Rating Scales (Different Values) in Banner. . . . .	498
6.5.3 Holecourt Table with a Varying Percentage Base. . . . .	501
6.5.4 Break Table with a Multi-level Banner. . . . .	504
6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES) . . . . .	507
6.6.1 Simple Multiple Location Tables . . . . .	509
6.6.2 Tables With Both the Row and the Base Overlaid. . . . .	517
6.6.3 Overlay Tables With Summary Statistics (Means). . . . .	520
6.6.4 Tables with the Banner and the Row Overlaid. . . . .	523
6.7 LONG BRAND LISTS. . . . .	526
6.7.1 Producing Net Categories . . . . .	526
6.7.2 Ranking With Nets And Sub-Nets. . . . .	529
6.7.3 Suppressing Blank Rows in a Large List. . . . .	538
6.7.4 Collapsing Low Mentions into another Category. . . . .	544
6.7.5 Printing Subtotal Rows . . . . .	549

# Contents

6.8 MASTER-TRAILER PROCESSING .....	554
<b>7 Tables Viewable Through a Browser .....</b>	<b>565</b>
Introduction .....	565
WebTables Overview .....	565
Browser Compatibility .....	566
Example Mentor Files Available Online .....	566
7.1 Basic WebTables .....	566
Writing Specs .....	567
Managing Your Specs .....	567
Using Set Commands in tabs.spx .....	567
Global Edit Statements in the tabs.spx .....	568
Files Needed to Create WebTables .....	570
Files Produced by the Mentor Run .....	571
Note for Windows/DOS clients .....	571
When manually testing and preparing to Go Live .....	571
Basic steps to Running a Live Mentor 8.1 Job .....	572
Setting up Directories and Running WebTables .....	572
Automating or Running Tables on a Set Schedule .....	572
7.2 Complex WebTables .....	573
Basic Colors .....	573
Helpful Internet Websites for Choosing Colors .....	574
Sources for CSS Help .....	574
Validation Websites .....	574
Basic Commands with and without Cascading Style Sheet .....	575
Example Files and How They Work .....	576
The tabs.spx file .....	576
The banner_a.def File .....	580
The rrnr_x.def file .....	581
The index.html file .....	584
.....	586
Example Output of a WebTable .....	587
Custom CfMC Scripts .....	587
THE WebPass SCRIPT .....	587
7.3 ON-DEMAND TABLES .....	589
7.3.1 Installation Requirements .....	589
7.3.2 Installation requirements .....	590



.....	590
Setup .....	590
7.3.3 Editing .....	596
WEBTAB.ADD.....	597
7.3.4 Adding a Base or Weight .....	600
Adding a File on the Command Line .....	603
Using the Define Box .....	604
edit options .....	606
.....	608
Files Created .....	608
7.3.5 Creating On-Demand Tables .....	609
On-Demand Selection Screen .....	609
Final Output of Tables.....	612
Preparing Mentor Output Files For Post Processing.....	612
Augmenting Prepare Specs to Enhance Tables.....	625
<b>8 Statistics (Significance Testing) .....</b>	<b>627</b>
Introduction.....	627
8.1 SIGNIFICANCE TESTING TO MARK CELLS .....	628
8.1.1 The STATISTICS Statement .....	628
8.1.2 Independent, Dependent, Inclusive, and Printable Tests .....	630
new protection value in significance testing .....	630
8.1.3 Setting the Confidence Level .....	631
8.1.4 Standard Significance Testing.....	633
8.1.5 Changing the Confidence Level .....	637
8.1.6 Bi-Level Testing (Testing at Two Different Confidence Levels).....	640
8.1.7 Using Nonstandard Confidence Levels.....	642
8.1.8 Inclusive T Tests .....	643
8.2 Changing the Statistical Base .....	644
8.2.1 Changing to the Any Response Row.....	644
8.2.2 Changing to Any Row in the Table.....	649
8.2.3 Changing in the Middle of a Table .....	651
8.3 Changing the Statistical Tests .....	654
8.3.1 The All Possible Pairs Test.....	654
8.3.2 The Newman-Keuls Test Procedure .....	655
8.3.3 Other Testing Procedures .....	660

# Contents

Repeated Measures Option . . . . .	662
Example One - using the All Possible Pairs test . . . . .	664
Example Two - using the anova_scan and repeated measures . . . . .	670
Example Three - using the Fisher test and Repeated Measures . . . . .	677
8.3.4 Changing the Variance . . . . .	683
8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES . . . . .	684
8.4.1 Weighted Tables with Different Weights. . . . .	691
8.5 PRINT PHASE STATISTICAL TESTING. . . . .	695
8.5.1 EDIT Options . . . . .	696
8.5.2 Changing the Confidence Level and the Type of Test . . . . .	700
8.5.3 Changing the Type of Test by Row . . . . .	701
8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING . . . . .	705
8.6.1 Testing Mean Rows Only. . . . .	706
8.6.2 Excluding any Row from Statistical Testing. . . . .	709
8.6.3 Excluding Columns with Low Bases from Statistical Testing . . . . .	713
8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES. . . . .	719
8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING). . . . .	726
8.8.1 Direct Comparison Testing . . . . .	727
8.8.2 Distributed Preference Testing . . . . .	732
8.9 CHI-SQUARE AND ANOVA TESTS. . . . .	735
Discussion Of Output . . . . .	750
Other ANOVA And Chi-square Options . . . . .	753
8.10 NOTES ON SIGNIFICANCE TESTING . . . . .	756
8.10.1 What Can and Cannot Be Tested . . . . .	756
8.10.2 Degrees of Freedom . . . . .	758
8.10.3 Verifying Statistical Tests . . . . .	759
8.10.4 Error and Warning Messages . . . . .	761
8.10.5 Commands Summary . . . . .	764
<b>9 Specialized Functions . . . . .</b>	<b>767</b>
Introduction. . . . .	767
9.1 GENERATING SPECIALIZED REPORTS . . . . .	767
The PRINT_LINES Command . . . . .	772
Line Printing Control Codes . . . . .	772
Codes Used To Print Information From The Data Or A Variable . . . . .	773
Codes To Print Specific Characters. . . . .	775

Variable References . . . . .	777
9.1.1 Printing a Report Footer using WHEN BOTTOM . . . . .	789
9.2 TABLE MANIPULATION . . . . .	791
9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS . . . . .	822
9.3.1 System Constants . . . . .	823
Variable Constants . . . . .	825
Case Reading Constants . . . . .	828
System Information Constants . . . . .	832
9.3.2 Functions . . . . .	836
ARITHMETIC FUNCTIONS . . . . .	837
VECTOR FUNCTIONS . . . . .	840
NUMBER RETURNING FUNCTIONS . . . . .	843
LOGICAL FUNCTIONS . . . . .	850
TABLE RELATED FUNCTIONS . . . . .	852
INTEGER FUNCTIONS . . . . .	855
STRING FUNCTIONS . . . . .	858
9.4 PARTITIONING DATA FILES . . . . .	862
Preliminaries . . . . .	877
Basic Sums and Statistics . . . . .	877
Sets of Variables: Newman-Keuls Preliminaries . . . . .	878
9.4.1 The Newman-Keuls Procedure . . . . .	882
9.4.2 Statistical Testing In Mentor . . . . .	883
9.4.3 TABLE-BUILDING PHASE . . . . .	884
Notations And Miscellaneous Facts . . . . .	884
Estimate For Means, Standard Deviations, Standard Errors and Correlations . . . . .	886
More Estimates . . . . .	887
FORMULAS FOR STATISTICS CREATED DURING TABLE BUILDING (ROW=) . . . . .	888
. . . . .	890
FORMULAS FOR STATISTICS CREATED DURING THE PRINT PHASE (EDIT=) . . . . .	890
T and Z Tests . . . . .	891
WALKER T-Test . . . . .	892
. . . . .	892
RANK SUM/WILCOXEN TEST . . . . .	892
More references: . . . . .	893

# Contents

# GETTING STARTED

## INTRODUCTION

This chapter provides an introduction to the benefits and features of Mentor, and a summary of each chapter in this manual. It also provides an overview of the process of producing tables from data. For more information go to “1.3 THE BASIC STEPS OF THE REPORT/TABBING PROCESS”.

If you need a definition of any terms or concepts, a glossary of terms is included in the *Utilities* manual. Users of Survent, CfMC’s interviewing package, will be familiar with many of the concepts and terms used in this manual.

### 1.1 WHAT IS MENTOR?

Mentor is a reporting and cross-tabulation program designed for companies in market analysis, telemarketing, market research and related fields. In addition, Mentor comes with various supporting utilities.

### 1.2 BENEFITS OF USING MENTOR

There are many advantages to using Mentor, including:

- Complex feature set
- Powerful command language
- Advanced database capabilities
- Integration with Survent
- Ease of Use

Each of these benefits is explained in the following sections.

**COMPLEX FEATURE SET**

Mentor's primary function is to create tables, but it also does the following:

- clean data
- create variables
- create a database, which includes:
  - ♦ variables
  - ♦ procedures
  - ♦ table elements
  - ♦ tables
- specify bases and filters
- weight the data in tables
- use IF-THEN-ELSE logic
- run statistics, such as:
  - ♦ Mean
  - ♦ Standard Deviation
  - ♦ Standard Error
  - ♦ Newman-Keuls
  - ♦ T-tests
  - ♦ Chi-squares
  - ♦ All-possible-pairs significance testing
- list open-ends
- show frequency distributions
- input and output multiple data types, such as:
  - ♦ CfMC System files
  - ♦ ASCII
  - ♦ Binary
  - ♦ swapped-binary

- ◆ hex
- sort data files
- concatenate data files
- merge data files
- create subsets of data files
- print tables in various formats

This is only a partial list of features, and new features are being added with each release. See the section “1.6 GETTING TECHNICAL SUPPORT” at the end of this chapter to see how to get the latest information about Mentor and how to submit a request to the CfMC programming staff.

## POWERFUL COMMAND LANGUAGE

Mentor is a full-featured, cross-tabulation and reporting system. It is a program that gives users direct command-line capabilities with which to process research information. This complex program is efficient for the highly skilled spec writer who wishes to use the spec language directly. (New users may want to generate their tables using SCAN, a utility program described in the *Utilities* manual.)

While simple, straightforward jobs are easy to produce, Mentor also allows you to handle large, complicated jobs as well. You can create boiler-plate specification files and then tailor them for specific jobs. You can also create libraries of questions or specifications for specific clients or jobs.

One powerful feature of the Mentor command language is the ability to join data locations in a variety of ways. Data locations can be combined using a variety of joiners (or connectors) including arithmetic joiners (add, subtract, multiply, divide), vector joiners (WITH, BY), logical joiners (AND, OR, NOT), relational joiners (>, <, =, and <>), and combinations of these. Joining variables allows you to create complex expressions that can be used for table banners, for example. This combining of simple variables into more complex expressions is efficient because

## **GETTING STARTED**

### *1.2 BENEFITS OF USING MENTOR*

the building blocks, once defined as variables, can be reused repeatedly, simply by referring to them by variable name, once they are in the database file.

You can create new categories or manipulate the data using Survent screens or at the spec line. You can combine existing variables to create complex combinations of categories, and give these definitions their own variable names as well. Your definitions can include bases, weights, statistics and significance testing. You also have several different formatting and printing options. You can print a table to your screen, revise it as needed, and then print it again, saving it this time to a print file or sending it to a printer.

The utilities are user-friendly menus that prompt a beginning user through building a simple cross-tab or defining a questionnaire. You can use Survent and Mentor to create your own utilities or interfaces that are tailored to the specific needs of your own shop or clients.

## **ADVANCED DATABASE CAPABILITIES**

Mentor creates and uses a database contained in a DB file. The DB file acts as a library, maintaining definitions of all the various elements processed throughout a complete job. The Mentor database significantly reduces processing time and makes the administration of large, repetitive jobs (such as tracking studies) much easier.

DB files can be used by more than one job. For example, you may have the same exact demographic questions for each study. Once those question variables exist in a DB file, you can reference those definitions and use them as they are, or modify them (i.e., perhaps the column location has changed) and optionally save the changed variables in a new DB file. Up to ten user-defined DB files can be referenced at the same time during a single job.

The advantages of Mentor DB files are:



- anything created by the program (variables, table elements, tables) can be saved into a DB file for future use;
- items stored in a DB file can be retrieved quickly and easily;
- tables stored in a DB file can be recalled and given minor modifications without having to reprocess them;
- items stored in a DB file take up less disk storage space;
- since complex elements are stored by a simple name, it is easier to generate ad hoc tables;
- experienced users can generate large numbers of historical tables efficiently.

## INTEGRATION OF SURVENT AND MENTOR

A principal feature of this software is the extensive integration of Survent and Mentor. For example, you can write a questionnaire to create your own user interface for Mentor. This means that it is possible to have a questionnaire, written for Survent, which asks a person what variables to tabulate, and then automatically produces the tabulation. (CfMC's utility SCAN is an example of this.) Many other scenarios are possible, including retab systems, and specifications that interact with a specific database. You can design client-specific questionnaires with Survent, or CfMC can create questionnaires for your company on a contractual basis.

Because you can mix the modules of Survent and Mentor commands, complex, hierarchical studies are much easier to manage (this also called master-trailer processing). It is possible to write specifications in which there are master records (i.e., families), first-level detail information (i.e., family members), second-level detail information (i.e., trips made by family members), and so on. Each level of information has its own questionnaire, and during the interviewing process, Survent passes information to and from Mentor as necessary.

Any process that you need to use to capture data can be a Survent-Mentor application, including those not traditionally thought of as market research. Many companies can take advantage of the ability to do high-quality interviewing. For example, the software could be used with a list of clients, with each client having an

## GETTING STARTED

### 1.2 BENEFITS OF USING MENTOR

array of tests to be done (some more than once). You could use CfMC software for the workers who need to review and manage the testing process. You can use Survent to control the content and location of informational text on the screen. Users see text written in language they understand, and you can reduce errors by controlling responses. Mentor manages the data. We call this ZTCS, Zero Training Custom Software, and it provides limitless possibilities.

## EASE OF USE

CfMC software allows you to create readable, accurate, and fully labeled tables easily from a data file. Using the utilities, you can produce a wide range of tables just by responding to a set of menus. To indicate what information you want on your tables, you can use either data locations or names of variables from the study. You can also use the utilities for generating and manipulating data (including sorting, merging, and creating subsets from data files) and writing reports.

If you have Survent, it will take text and data locations from the original questionnaire and generate specifications that include all the basics of a table (the title, banner, stub labels) that you can use as is or modify. This way, you can produce a wide range of tables just by knowing the names of the questions to tabulate. For example, you can tell Mentor you want the answers from the question "OWN" cross-tabulated with the answers from the question "YESNO." Here is the table Mentor would create:

TABLE 001

BANNER:4. Do you currently own or rent your home?

STUB:1. Do you, or any member of your household, have credit cards?

	TOTAL	OWN	RENT	DK/NA
TOTAL	201	107	91	3
	100%	100%	100%	100%

Yes	150 75%	88 82%	61 67%	1 33%
No	47 23%	16 15%	29 32%	2 67%
DK/NA	4 2%	3 3%	1 1%	-

Mentor is an extremely flexible tool. The menu-driven interfaces of the utilities allow its use by minimally trained staff, enabling the advanced spec writer time to develop more complex processes needed for specific applications.

### 1.3 THE BASIC STEPS OF THE REPORT/TABBING PROCESS

Processing a tabulation job typically consists of several key steps:

- 1 Setting up the data file**
- 2 Defining the data descriptions**
- 3 Cleaning the data**
- 4 Generating new data from existing data**
- 5 Building and printing the report-ready tables**

These steps are detailed in the following sections.

#### **6 Setting up the Data File**

Data files need to be in the CfMC System file format for Mentor to access it. Your data file can come from Survent or other interviewing or data entry software. Survent data files are already in the CfMC System file format. You can use the utilities to convert other types of files (ASCII, binary) to a System file.

When converting a data file, you can also sort it on very simple or complex criteria. You can also check for and make corrections for duplicate case IDs, duplicate data, or missing data. If you need to create a subset of the data (often used to test cleaning or table specifications), you can also select a specific portion of a larger data file (this is similar to using a base, but more efficient in terms of processing time).

## **7 Defining Data Descriptions**

Data definition can be done in two different, but not exclusive, ways: use Survent's DB and/or DEF files, or use command language to create specification files. The definitions consist of a variable name or label, question and response text, data location and width, and type of question. These elements are then used by the program when performing the other phases of the job.

Mentor is very efficient. Once data has been defined, the remaining phases - cleaning, generating, and table building - are able to use these data definitions. The DEF and DB files can be preserved, modified, and/or added to, allowing quick and easy processing.

## **8 Cleaning the Data**

At the start of the cleaning process, you probably want to generate a holecount (marginal) to find out what is in your data file and get an idea about how much cleaning it requires. You can generate a holecount and other reports about the data with the utilities, see the *Utilities* manual.

Data cleaning involves systematically examining the data along with some logic specifications. With Mentor, you can clean your data interactively or in batch mode. (Data collected by Survent would typically need little, if any, cleaning.)

Specifications tell Mentor which data is valid and which have errors for each case. You can develop cleaning instructions dealing with linked responses (skip patterns, conditionals) or non-linked responses. Both will check for valid responses, check that no more than one of the responses was entered (for single-response questions), or might check to be certain that combinations of answers make sense. You can

choose to have the program automatically “fix” the data for you all at once, or get a report of errors and fix them interactively.

Mentor’s cleaning commands are comprehensive. You can use IF-THEN-ELSE structures, and you can nest these structures. You can use GOTO statements for skip patterns or branching. You can also provide your own text for error messages to make error messages easier to understand.

## **9 Generating New Data**

Once the data is clean, you may want to combine or otherwise manipulate your data to create new categories (this is commonly called recoding). Mentor provides a number of ways for combining data, performing arithmetic calculations and other data manipulation. You move data around to provide consistency between cases in different data files (i.e., across different waves of a tracking study). You can also combine data to form new items. For example, you might, for efficiency, want to create a single category for females over 35 who drive sports cars.

You can create new data by doing calculations on existing data items. For example, you may have numeric data on how many miles a car has been driven and gallons of gas it used. You can combine these figures to compute a miles-per-gallon figure. You can also add entirely new items to the data file, such as weighting factors or you can remove the rotation from data for series of questions that were rotated and stored in different fields.

## **10 Building and Printing Tables**

Mentor produces tables using already defined question variables located in the DEF and DB files and, if necessary, more complex variables. These complex definitions can consist of any combination of other variables. You can use text-type variables to specify the format of items, such as titles, headings or complex banners. You can also use variables to dictate the printing options, such as whether to include vertical or horizontal percentages.

Variables are the fundamental units of processing with Mentor. The simplest variable consists of all of the valid answers to a single question from a questionnaire. You may already have variables from Survent. You can define your

own or other variables, such as a base that includes only males, or a variable for specific numeric ranges.

Changing how the data appears on your tables is easy. Your options include whether or not to include frequencies, percents or statistics. Percentages can be based on a specific row, or you can have the percent base change mid-table. Percents can print with from zero to two decimal places of significance. Frequencies can print as whole numbers, or with one or two decimal places (for weighted data, for instance). You can also have your choice of different levels of decimal significance with generated statistics. You can have each row in the table print in a different format; for example, you can have some rows that have frequency and percents, but also have other rows that just print as frequencies (statistics, for example).

You can edit any text on your tables and reprint your tables without having to re-process the data in the tables. You can also use a variety of options to tailor the overall page format your tables. Again, these parameters can be changed between tables (i.e., do it one way for the first ten tables, do it another way for the next ten.). Options include the length of the page (number of lines per page), the width of columns in the banner, the width of row labels, the spacing between rows, whether or not to print summary rows and columns (Total, No Answer), and page numbering.

## **1.5 SUMMARY OF THE CHAPTERS**

The chapters in this manual help you with the standard steps of a reporting or tabulation project. The exception to this is the supporting utility programs that are covered in a separate volume. We recommend that you read all chapters of this manual, even if you feel that you are already familiar with some of it or don't need those particular features for your operation. Each chapter builds upon previous chapters, especially the table building chapters (Chapters 4, 5, and 6). Here is a short description of what is covered in each chapter:

***Chapter One:*** Getting Started. This chapter provides a basic overview of Mentor and a summary of the reporting process.



**Chapter Two:** Cleaning the Data. Using Survent or your own variables, you can check and correct your data.

**Chapter Three:** Reformatting your Data. From your existing data, you can use the format of your data and create new categories of data for processing purposes.

**Chapter Four:** Basic Tables. This chapter teaches you how to produce simple tables. It also includes an explanation of the table defaults.

**Chapter Five:** Intermediate Tables. This chapter teaches you how to make changes to the default settings for tables, including printing settings.

**Chapter Six:** Complex Tables. This chapter explains how to add statistics (mean, standard deviation, and standard error) to your table, how to create top-box/bottom-box tables, how to weight your tables, how to use loop variables, and how to create break tables. It includes many examples of specification files and completed tables.

**Chapter Seven:** Customizing specifications for viewing tables via a browser. This chapter describes Mentor programs such as WebTables, On-Demand tables, etc.

**Chapter Eight:** Statistics. This chapter explains how to add T-tests, chi square, and ANOVA tests to your tables.

**Chapter Nine:** Specialized Functions. This chapter explains how to generate specialized reports and manipulate tables with System constants and special functions.

**Appendices:** The appendices are in a separate volume and cover several topics. Appendix A shows the formulas Mentor uses in statistical testing. Appendix B is the largest appendix, and it is a listing of all the tilde commands (tilde command begin with a “~”) including the syntax and a list of options for the command.

## 1.6 GETTING TECHNICAL SUPPORT

We hope you find this manual useful. CfMC is always looking for feedback about our software, our manuals, and our technical support. Below is a list of ways to contact us, depending on what your needs are.

<u>If you want to:</u>	<u>Contact:</u>	<u>Voice:</u>	<u>Email:</u>
get general information	Receptionist	(415)777-0470	info@cfmc.com
lease CfMC software			
add more users			
get an upgrade			
(East region)	Marketing	(212)777-5120	joycer@cfmc.com
(West region)	Marketing	(415)777-0470	sales@cfmc.com
get help with using the software			
have a suggestion for the software			
want a new feature added to the software			
want to report a bug			
(East region)	NY Tech Support	(212)777-5120	denised@cfmc.com
(Midwest)	Texas Tech Support	(409)775-7732	mcblair@cfmc.com
(West region)	SF Tech Support	(415)777-2922	support@cfmc.com
get training	Training	(415)777-0470	train@cfmc.com
get additional copies of the manuals			
have corrections for the documentation			
	Documentation	(415)777-0470	doc@cfmc.com



get the latest CfMC news

CfMC Newsletter	(415)777-0470	
CfMC-SF web site		<a href="http://www.cfmc.com">http://www.cfmc.com</a>

talk to other users of CfMC software

Spec-talk discussion group		<a href="mailto:spec-talk@cfmc.com">spec-talk@cfmc.com</a>
----------------------------	--	--

pick up files from CfMC

send files to CfMC

Bulletin Board System	(415)896-2362	<a href="mailto:bbs@cfmc.com">bbs@cfmc.com</a>
	(modem)	
or CfMC FTP site		<a href="ftp://ftp.cfmc.com">ftp://ftp.cfmc.com</a>

get a quote from the San Francisco service bureau

San Francisco office	(415)777-0470	<a href="mailto:sf_sb_mgr@cfmc.com">sf_sb_mgr@cfmc.com</a>
or CfMC-SF web site		<a href="http://www.cfmc.com">http://www.cfmc.com</a>

get a quote from the Denver service bureau

Denver office	(303)860-1811	<a href="mailto:denver@cfmc.com">denver@cfmc.com</a>
or CfMC-Denver web site		<a href="http://www.cfmc.com/denver">http://www.cfmc.com/denver</a>

get a quote from the New York service bureau

New York office	(212)777-5120	<a href="mailto:denised@cfmc.com">denised@cfmc.com</a>
-----------------	---------------	--

**Our mail address is:**

Computers for Marketing Corporation


547 Howard St.

San Francisco, CA 94105

Company: 415-777-0470

Fax: 415-777-3128

Tech Support: 415-777-2922



**GETTING STARTED**

*1.6 GETTING TECHNICAL SUPPORT*

# PREPARING YOUR DATA

## INTRODUCTION

This chapter provides information on how to prepare and clean a data file. It is important to become familiar with your data in order to write procedures that tell Mentor what the data should look like. Mentor compares the existing data format to your revision of the data format, identifies errors, and provides several ways to correct these errors. You can check and correct the data interactively, one case at a time, or you can have Mentor automatically check and modify all the cases at once.

This chapter also describes how to correct data by modifying it, transferring it to a new location, or combining data elements in the file.

## 2.1 OVERVIEW OF THE CLEANING PROCESS

If you don't have any cleaning experience, and you just need to make a few changes to a data file, use the CLEANIT utility, which is described in the *Utilities* manual. If you already have a list of data corrections to make and you have some experience with the Mentor ~CLEANER block, you can skip to “2.5.1 Quick Reference: Cleaning Commands And Examples”.

### CLEANING STEPS

<u>Task</u>	<u>Program</u>
1) Make a backup copy of the data file	DOS or UNIX copy command
2) Translate raw data file into a CfMC System file	COPYFILE MAKECASE
3) Generate marginals	HOLE FREQ
4) Write and run cleaning specs	Your word processor Mentor
5) Take error listing and change data	Mentor
6) Repeat steps 3 through 5 until data conforms to cleaning specs	

## 2.2 CLEANING DATA

Once you have collected your data, you should check it for errors. This section summarizes the steps necessary to clean data which are described in detail in the rest of this chapter. If you are using Survent to collect your data, Survent can also automatically generate cleaning specifications for you. See section “2.4 CLEANING WITH SURVENT VARIABLES”.

- 1 Make a backup copy of data file. Always have a copy of the data in its original form before you start cleaning.
- 2 Translate your raw data file into a file that Mentor can read. Mentor data files are referred to as System files, and have an extension of ".tr". There are two menu-driven CfMC utility programs you can use, MAKECASE and COPYFILE, to translate raw data files. They are described in the *Utilities* manual. The MAKECASE utility converts 80 column ASCII or binary card image files into Mentor System files. The COPYFILE utility converts files with records longer than 80 columns and other file types. You can also use these utilities to sort the file and check for duplicate case IDs.
- 3 Generate reports that provide an overview of the data (these reports are commonly called "marginals"). Use the HOLE utility to generate a holecount, a count of the punches in the data for each column. This can help you spot obvious problems with the data. Use the FREQ utility to generate frequency distributions, or a list of which ASCII characters are in specified locations in the data. This is useful to look at short, multiple column data, such as zip codes.
- 4 Write and test a procedure(usually referred to as "cleaning specs") that will check the data. To test the procedure, you can tell Mentor to only look at a certain number of cases, and see what errors it produces. You may need to modify your cleaning specs based on these results. Running the procedure on all of the cases in the data file will then generate a list of errors, or locations in the data that need to be checked.
- 5 Clean the data, based on the errors generated by the cleaning specifications. When data looks incorrect, you can choose to delete it, change it by referring back to the original survey, or change it based on guidelines that you have established.
- 6 Repeat steps three through five until you consider the data clean. (You can also use a final holecount to check the tables you generate from your data file.)

### 2.2.1 Other Types of Data Files

While we recommend that you do so, you do not have to convert your data file to a CfMC System file. You can write a procedure to have Mentor read and generate a data error report on a different type of data file. Mentor can read ASCII, binary, swapped binary or DTA files directly. The disadvantages to working with a data file that has not been converted are:

- Processing will be slower because Mentor must convert each case before reading it.
- Cases cannot be flagged for errors or deletion.
- You cannot modify the data interactively using Mentor.

If, however, you have a small ASCII file, it may be easier to write a Mentor procedure to read and check the data file and then edit it in a word processing program. Your procedure can also include some data checking and modification. The basic specifications would look something like this:

```
~DEFINE PROCEDURE= {name:
  data checking or modification commands
}
~INPUT file, ASCII=length      (or other file type option)
~OUTPUT newfile, ASCII=length  (or other file type option)
~EXECUTE PROCEDURE= name
~END
```

See “2.3 CLEANING SPECIFICATIONS”, and “2.3.2 Correcting Errors” for examples of procedures that read and modify raw data. *CORRECTING ERRORS*, *Auto-Fixing The Data* contains two example procedures that modify data in batch mode. You can also refer to *Appendix B: TILDE COMMANDS* under the specific command for information.

### 2.2.2 Why Clean the Data?

The cleaning process is one of the most important steps in data tabulation. Human error occurs during data collection, especially in a paper and pencil study that has been manually keypunched. There is considerable advantage in using a CRT-type interviewing package such as Survent since it makes and checks its own data file. No matter what measures you take to prevent errors, some will occur. Serious

problems can jeopardize the validity of a study. If the logic of the questionnaire is flawed (e.g., causing multiple responses to single-response question), you may have to do replacement interviews or do the entire study over. Let's assume that your questionnaire has good logic and your data collection methods are conservative.

You must assume any data file you have not already looked at is dirty. A dirty data file is one that may have errors in it. These errors may be isolated to one column or question (more than one answer in a single answer question, number answers outside of the defined range, etc.) or across several columns or questions (skip patterns not followed, respondent asked about brands that they have never heard of, open-ends coded incorrectly, keypunch error, etc.). Data received from the field must be put through a cleaning process to increase its accuracy.

To adequately clean a data file, you must be able to change the information in the file. You need the ability to add or remove characters from columns, move data from one location to another, or blank columns altogether. You may choose to manually change one case at a time (interactive cleaning), or have the computer modify all of the cases automatically ("auto-fixing"). Interactive cleaning is the safest way to clean data, because you refer back to the original survey to make corrections. Auto-fixing is faster, but you must be careful that your modifications do not compound your errors. A combination of interactive cleaning and auto-fixing can give you both accuracy and speed. You can choose to have Mentor auto-fix simple problems and correct complex errors manually.

When you are through with the cleaning process, your data is it is not 100% accurate, it merely conforms to the cleaning procedures you have written. There is no such thing as data that is completely accurate. The cleaning process is merely an attempt to minimize errors, and increase the accuracy of your data.

#### 2.2.3 Understanding The Questionnaire

Before you do any cleaning, take the time to read through your study to make sure you understand its framework and specifics. Once you have written your cleaning specifications, run them on part of the data file to list out all errors before the data is altered. This will expose any questionnaire execution problems or specification errors. If the same error occurs frequently, make sure that it is actually an error in the data (by examining a few cases) and not a specification error. If you confirm

the error, your decision will be whether to discard these questions or questionnaires, or to salvage them. Generating the initial error listing is a key step that can save you time.

Below is a sample questionnaire. Make sure you understand the questionnaire's logic by asking yourself two questions.

- 1 **Who** should answer each question?
- 2 **How** should each question be answered?

The questions below are meant to appear as they would on a self-administered questionnaire, collected with paper and pencil.

**Example:**

Q 1.PLEASE ENTER YOUR NAME.

(5-14) \_\_\_\_\_

Q 2.WHAT DAY OF THE WEEK IS TODAY?

(15-17)      MON  
                  TUE  
                  WED  
                  THU  
                  FRI  
                  SAT  
                  SUN

Q 3.DO YOU HAVE ANY SIBLINGS (BROTHERS OR SISTERS)?

(18)      1    YES  
                  2    NO (SKIP TO Q5)

Q 4. HOW MANY SIBLINGS DO YOU HAVE?

(19-20) \_\_\_\_\_

Q 5. WHAT OTHER MEMBERS ARE IN YOUR IMMEDIATE FAMILY?  
CHOOSE ALL OF THE FOLLOWING THAT APPLY:

(21)      1    GRANDMOTHERS  
                  2    GRANDFATHERS  
                  3    GRANDCHILDREN  
                  4    COUSINS

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

- 5 IN-LAWS
- 6 NEPHEWS
- 7 NIECES
- 8 NONE OF THE ABOVE

**WHO** should answer each question?

The respondents who should answer a question are sometimes called the base of the question. During the course of a questionnaire, respondents may be excluded from answering a question and skipped to another section of the questionnaire. The remaining respondents make up the base of the question. In our sample questionnaire, respondents who do not have siblings should not answer the question about the number of siblings.

**HOW** should each question be answered?

Responses for questions have limits. Word responses must be recoded into a finite number of numeric or letter codes either by Mentor or manually. Numeric responses usually have some sorts of boundaries. Answers outside of these boundaries are either considered errors or the bounds of the questions are expanded to allow these answers. In the sample questionnaire above, question Q 4. has valid answers of 1-10. The cleaning procedure will report errors on any case containing numbers or characters outside of these bounds. The same is true for question Q 5.; valid punches are 1-8.

**NOTE:** No more than 7 answers may be recorded and 8 (none of the above) cannot be given with any other answer. All of these parameters must be checked in cleaning for consistency throughout the data file.

## 2.3 CLEANING SPECIFICATIONS

**Before beginning the cleaning process, make a copy of your CfMC data file!**

Always have a copy of the data in its original form, i.e., before you started cleaning. In fact, if you find you are making changes after each subsequent run of your cleaning specifications, it is good practice to make interim copies of your data file. Then if one set of changes must be undone you may only have to go back one or two copies to get to a place before the last set of data modifications. You can use your list file and ~SET LOGGING to keep track of the changes you have made.



## GENERATING A LIST OF ERROR MESSAGES

Cleaning specifications are a set of instructions (called a procedure in Mentor) that checks the data for valid responses based on your description of the data. These data descriptions are called variables. If you do not know how to define variables refer to “2.3.1 *Cleaning Examples*”. In the cleaning procedure you will write statements that tell Mentor which columns to check and to print an error message if the data does not match your description. For example, you can include statements to indicate where a skip pattern should have been followed, what the number of responses should be or how they should be ranked. You will use the list of data errors to either clean interactively on a case-by-case basis or to modify your procedure to correct some errors in batch mode.

Here is a cleaning specification file based on the sample questionnaire in section 2.2.1. Refer to “4.2 *TABLE BUILDING BASICS*” for an explanation of other useful spec file commands and “4.5 *META COMMANDS*” for an introduction to meta (>) commands. Explanations in this section will be limited to those commands pertaining specifically to cleaning.

**NOTE:** Two single apostrophes (") indicate either a comment, or a command that is optional or not always be needed. Mentor does not process anything preceded by these marks, which is not a quote (“), rather two single apostrophes.

```
>CREATE_DB procs

''>USE_DB procs

~DEFINE
PROCEDURE={showerr:
  OK_COLUMNS [1.14]
  CHECK [15.3#MON/TUE/WED/THU/FRI/SAT/SUN] "Not a valid code"1
  CHECK [18^1/2] "Should be single 1,2"
  IF [18^1] THEN
    CHECK [19.2*Z#1//10] "Should be a number 1-10"
  ELSE
    CHECK [19.2^B] "If 18 not 1,19.2, Should be blank"2
```

---

1. Mentor will print its own error message if you do not provide one. See under the heading Program-Generated Error Messages later in this section for examples.

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

```
ENDIF
CHECK [21*P^1//7/(-)8] "Should be multi 1-7, or single 8"
CHECK_COLUMNS
}

~INPUT sampl <options>

''~SET CLEAN_ALLOW_BLANKS

~EXECUTE PROCEDURE=showerr

~END
```

This specification file will only report where the data does not match the description. No data will be changed. Columns that do not need to be checked are specified first.

It then checks that the day of the week in columns 15-17 is one of the seven codes listed. Question Q 2. in column 18 must be a code 1 or 2 and not both, designated by the / separating the punch values. If the respondent has siblings, column 18 = code 1, then question Q 3. in columns 19-20 must be in the range of 1- 10. Values outside of the range are not acceptable. If the respondent did not have any siblings, column 18=2, then we expect columns 19-20 to be blank. We expect question Q 5. to have any number of responses between 1 and 7, but if it has an 8 (none) then no other answers should appear. The designation of punches 1-7 means multiple codes are acceptable, the / designation separating the 1-7 from the 8 means that the 8 is an exclusive code.

Finally, we check all the remaining columns in the case (22-80 in our example) to make sure they are blank. See *Appendix B: TILDE COMMANDS*, *~CLEANER* for more examples of the CHECK command.

Here is an explanation of the commands used in the example specifications above.

- 
2. You could also say [19.2#" "] to mean columns 19 and 20 are an empty string (blank). You must be sure that the number of spaces inside the quote marks is the same width as the field. This syntax is especially useful when saying an ASCII variable can also be blank., e.g., [19.2\*Z#1//10" "].

### **>CREATE\_DB procs**

Tells Mentor to create a DB file called PROCS to store the cleaning procedure. This is optional, but useful during the actual cleaning process if we need to re-execute the procedure on each case. The procedure can be accessed in this file by the name assigned to it.

### **>USE\_DB procs**

Opens the DB file containing the cleaning procedure. Again this is optional, but useful if you need to use the procedure again either to actually clean cases or to recheck the data after cleaning. By storing your procedure in a DB file it is not necessary to re-define it each time. The entire ~DEFINE section could be eliminated in future runs of these specifications.

If you use the meta command >CREATE\_DB, you do not need the meta command >USE\_DB in the same run. Once you have created a DB, get items from it in another run with >USE\_DB.

### **~DEFINE PROCEDURE=**

The ~DEFINE keyword that tells Mentor that you will be specifying a group of commands to be executed at some later point either on a single case or on the entire data set. There are many commands that could be specified inside the procedure structure. Those used in this example are some that you will use frequently. Other useful commands are listed at the end of this chapter.

### **OK\_COLUMNS**

Tells Mentor that any data in these columns is valid. CHECK\_COLUMNS will not look at these columns. These are usually the case ID columns or non-coded open-ended responses such as name and address that will be listed out separately.

### **CHECK**

Compares the data to the description of the data given here and prints an error message when the data does not match the description. Mentor will print default error messages or you can provide your own inside of "quotes" as shown in the example.

**IF-THEN-ELSE-ENDIF**

Specifies some condition that must be met for the following commands to be executed. See *2.3.1 CLEANING EXAMPLES*.

**CHECK\_COLUMNS**

Tells Mentor to check every column not specifically examined with a cleaning statement or marked as OK by OK\_COLUMNS to make sure it is blank. It checks that columns are blank when conditionals are not met. This command is especially useful when you have many skip patterns in your questionnaire. By using this command it is not necessary to write an additional statement to check for blank columns for each conditional not met, e.g., ELSE CHECK [19.2^B] in our example. You cannot define your own error message and the command will not display the actual data, but it will list the columns you should look at for extraneous data. Refer to the sample error listings later in this section for an example.

CHECK\_COLUMNS should usually be the last command in your procedure.

**~INPUT *sample***

Opens the data file. There are many options available with this command. STOP\_AFTER= is used to read only the number of cases specified. It is very useful for testing your procedure for syntax errors or errors in data descriptions such as referencing the wrong columns. This option will cause Mentor to stop reading the data file after the number of cases specified. If you have a very large sample or a complex set of cleaning conditions it will save time to test your procedure on a small number of cases first. For instance, you could run your specifications on a 50 case sample and then review the error summary. If a given data error appeared more than a few times, you would want to double check your cleaning specifications for errors perhaps in logic or an incorrect data location. The option ASCII= opens an ASCII data file. After the equal sign you must specify the length (number of columns) of the longest case in your file. Refer to *Appendix B: TILDE COMMANDS* under ~INPUT for other file types and options allowed on this command.

**~SET CLEAN\_ALLOW\_BLANKS**

Tells Mentor that data fields specified in either CHECK or CLEAN statements can be blank if they do not otherwise fit the description. This is especially useful for self-administered questionnaires where it is unlikely that every question is answered. If a particular field cannot be blank the cleaning procedure can include a

check for that field so that a blank still produces an error. In our example question Q 4. in columns 19 and 20 should be blank (skipped) if the respondent answered No to question Q 3.

**~EXECUTE PROCEDURE=showerr**

Tells Mentor to execute the procedure on the cases in the specified ~INPUT file.

**~END**

Exits Mentor, closing all files opened during the run.

Here are two commands that will help you debug your procedures:

**>QUIT ERRORS=#**

Stops executing after this number (#) of syntax errors is reached. This means all syntax errors reported by Mentor, not just those found in your procedure.

**~SET PROCEDURE\_DUMP**

Echoes Mentor's internal process as it executes a procedure, to help determine the source of the error message.

Run your specification file (i.e., CLEAN.SPX) through the Mentor program by typing the following statement from the command line of your operating system:

```
Mentor CLEAN.SPX CLEAN.LFL                                DOS/UNIX
RUN Mentor.CGO.CFMC;INFO="CLEANSPX CLEANLFL" (MPE XL)
```

The results of the run will go to the list file (CLEAN.LFL or CLEANLFL). This is your error listing. Errors will be listed out case by case. An error summary prints at the end. The error summary is a count of the occurrence of each error message across all cases.

**SAMPLE ERROR LISTING**

For the purpose of explaining error messages, we will refer to the data listing printed below. See "9.1 GENERATING SPECIALIZED REPORTS" for the specifications that produced this type of formatted listing.

ID	Q 1	Q 2	Q 3	Q 4	Q 5
--	-----	---	---	---	---

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

0001	MICKEY	MON	1	12	1
0002	OLIVER	TUE		0	2
0003	PURDY	WED	3	1	1, 2, 3
0004		THU	2	?	8
0005	POLLY		1	1	4, 5
0006	FLYNN	FRI		?	4, 5
0007	SAMANTHA	SAT	1	?	8
0008	PERCY	SUN	1	7	B
0009	WOLFIE	MON	1	3	2, 8
0010	SWEETPEA	WEE	1	2	6

A blank indicates blank punch data (no punch present) and ? indicates missing numeric data. These are defaults for a formatted listing from Mentor.

The list file CLEAN.LFL will contain an error listing similar to the one below.

```
ID: 0001
  error 3: [19.2#] Should be a number 1-10: a valid
           answer is required [19.2#]="12"
  error 14: [22] field should be blank [22]="1"
ID: 0002:
  error 2: [18^] Should be a single 1,2: a valid answer
           is required [18^]=" "
  error 4: [19.2#] If 18 not 1,19.2, Should be blank:
           extra punches [19.2#]="0"
ID: 0003:
  error 2: [18^] Should be a single 1,2: a valid answer
           is required [18^]="3"
  error 4: [19.2#] If 18 not 1,19.2, Should be blank:
           extra punches [19.2#]="1"
ID: 0005:
  error 1: [15.3#] not a valid code: a valid answer is
           required [15.3#]=" "
ID: 0006:
  error 2: [18^] Should be a single 1,2: a valid answer
           is required [18^]=" "
ID: 0007:
```

```
error 3: [19.2#] Should be a number 1-10: a valid
answer is required [19.2#]=" "
ID: 0008:
error 5: [21^] Should be multi 1-7, or single 8: a
valid answer is required [21^]=" "
ID: 0010:
error 1: [15.3#] not a valid code: a valid answer is
required [15.3#]="WEE"
error 14: [22] field should be blank [22]="1"
```

12 errors in 10 cases

```
error 1: 2 [15.3#] Not a valid code
error 2: 3 [18^] Should be a single 1,2
error 3: 2 [19.2#] Should be a number 1-10
error 4: 2 [20] If 18 not 1, 19.2 should be blank
error 5: 1 [22^] Should be multi 1-7, or single 8
error 14: 2 $check_columns
```

The case ID is followed by a line that has an error number, the data location, your error message, the program-generated error message, and the data location with the actual data in quotes.

In this example, there are case ID 0001 has an answer of 12 in columns 19.2, for question Q4, regarding the number of siblings, and only answers from 1 to 10 are valid. Case ID 0002 has an error for columns 19.2 because column 18 is not a 1. In this case, you might want to go to original survey to see if question Q4 was actually blank, or if the answer was just not entered into the data. Case ID 0003 has an invalid answer (3) in column 18, and this causes a second error for columns 19.2.

If the cleaning specifications did not include the lines

ELSE

```
CHECK [19.2^B] "If not 1, 19.2 Should be blank"
the error listing would be different because CHECK_COLUMNS would print an
error for each column that should be blank, such as cases 0002 and 0003.
```

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

```
ID: 0001
  error 3: [19.2#] Should be a number 1-10: a valid answer
           is required [19.2#]="12"
  error 14: [22^] field should be blank [22^]="1"
ID: 0002:
  error 2: [18^] Should be a single 1,2: a valid answer is
           required [18^]=" "
  error 14: [19.1] field should be blank: [19.1]="0"
ID: 0003:
  error 2: [18^] Should be a single 1,2: a valid answer is
           required [18^]="3"
  error 14: [20.1] field should be blank: [20.1]="1"
ID: 0005:
  error 1: [15.3#] not a valid code: a valid answer is
           required [15.3#]=" "
ID: 0006:
  error 2: [18^] Should be a single 1,2: a valid answer is
           required [18^]=" "
ID: 0007:
  error 3: [19.2#] Should be a number 1-10: a valid answer
           is required [19.2#]=" "
ID: 0008:
  error 4: [21^] Should be multi 1-7, or single 8: a valid
           answer is required [21^]=" "
ID: 0010:
  error 1: [15.3#] not a valid code: a valid answer is
           required [15.3#]="WEE"
  error 14: [22] field should be blank [22]="1"
```

12 errors in 10 cases

```
error 1:      2 [15.3#] Not a valid code
error 2:      3 [18^] Should be a single 1,2
error 3:      2 [19.2#] Should be a number 1-10
error 4:      1 [22^] Should be multi 1-7, or single 8
error 14:     4 $check_columns
```

## PROGRAM-GENERATED ERROR MESSAGES

Here are a list of the standard types of cleaning errors and the error messages Mentor generates:

<b>Error</b>	<b>Error Message</b>
A question is blank when an answer should be present.	a valid answer is required



Single response question with more than one response.	too many answers
Punch question with invalid punches in the column(s).	extra punches
Single response question with an invalid punch code or number out of range.	a valid answer is required
Question with an invalid punch.	too many answers
Multi-response question where there is an exclusive response with another punch(es) or code(s).	exclusive code violation
Multi-response question with duplicate ASCII codes.	duplicate codes
Multi-response question with invalid ASCII codes, leading or embedding blanks.	invalid code or blank fields

### 2.3.1 Cleaning Examples

This section provides examples for cleaning simple types of questions. Many common cleaning situations are covered here. In later sections you will find examples of more complex cleaning situations.

The terms *answer*, *mention*, and *response* are used interchangeably throughout the rest of this chapter. The terms *field* and *data location* each refer to the column or set of columns that should contain a valid response.

In order for the Mentor program to check the validity of your data, each question must be defined as a variable. Mentor then compares it to the actual data for errors. A data variable is something defined within square brackets ([ ]). Within those brackets you provide the data's location, a data modifier, the data type, and finally the data categories. For more information on defining variables, especially as they relate to data tabulation, see sections "4.6 DEFINING DATA", "5.1 Expressions

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

*and Joiners*” and “5.2 Axis Commands/Cross-Case Operations”. “4.6.1 Summary of Rules for Defining Data” provides example sets that summarize the rules for defining ASCII, punch, and numeric data.

#### Example:

```
[2/5 .2 *F #1//17]  
[42 *P ^1//5/(-)Y]
```

Data locations can be defined in one of two formats: either record number/column (2/5), where each record is 80 columns, or absolute column(85). You may then specify a width using a period and the width (85.2), or specify a range of columns using a dash (85-86).

Variable modifiers used in cleaning are:

- \*D#** the field contains the number of decimal places specified.
- \*F** checks unique mentions (i.e., no duplicates allowed) across multiple locations. Leading or embedded blanks in the field are an error.
- \*F#** this numeric field has <#> decimal places.
- \*L** checks for all possible mentions across multiple locations. Leading or embedded blanks in the field are an error.
- \*P=#** specifies the maximum number of answers allowed for a punch variable. The default is \*P=1 meaning only one answer is allowed. Just \*P means that there can be as many answers as there are categories defined. # may be any number 1-255.
- \*S** the field is multi-punched, but only one of the punches will be retained.
- \*Z** this field contains leading zeros (0).

Data type is referenced by a caret (^) for punch categories, and a pound sign (#) for either numeric or ASCII categories. Categories are made up of either the string or number in the data, or the punch or punch number. Each category is separated by a single slash (/). A double slash (//) means a set of categories from first to last category (e.g., 1//5 to mean 1/2/3/4/5). A category can be marked exclusive of all others with a minus sign (-) enclosed in parentheses. This means that only this answer should be present in the data, and if it appears with any other it is an error.

You can combine modifiers, but you only need one asterisk (\*). Spaces are optional between variable items.

## CLEANING PUNCH DATA

### *Single Column/Single Punch*

2B. How often do you use your BANK CARD--  
at least once a month, at least every 3 months, a  
few times each year, less than that, or never?

- |      |   |                      |
|------|---|----------------------|
| (10) | 1 | AT LEAST MONTHLY     |
|      | 2 | EVERY 3 MONTHS       |
|      | 3 | A FEW TIMES PER YEAR |
|      | 4 | LESS                 |
|      | 5 | NEVER                |

This is an example of a single column, single response question. The cleaning statement will check that only one of the allowed punches appears in the data. Because this is the default we do not need to include the modifier \*P=1 in the variable definition.

```
CHECK [10^1//5] "Should be single 1-5"
```

This means that in column 10 there can be one of five possible punches 1, 2, 3, 4, or 5. If more than one of the allowed punches is found in column 10, the program will print the error message defined inside the double quotes. It is also an error if the response is something other than one of these punches or is blank.

This is how the statement would be rewritten to allow a blank response. A single slash is used to separate two single categories.

```
CHECK [10^1//5/B] "Should be single 1-5 or blank"
```

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

Now this statement says that column 10 may be blank (no punches present) or may contain one of the other allowed punches one through five. You can also use N to mean "not these punches", or use N with B to mean "not blank"

You can use the command `~SET CLEAN_ALLOW_BLANKS` to globally allow blanks on any question that you verify with a CHECK statement.

#### *Single Column/Multiple Punches*

2A. Which of the following types of credit cards do you, or your household members have?

- (8) 1 GENERAL PURPOSE CARDS
- 2 BANK CARDS
- 3 RETAIL STORE
- 4 GAS/OIL COMPANY CARDS
- 5 CAR RENTAL/AIRLINE
- 6 OTHER

This question allows up to six responses. All of the punches will be stored in a single column. Punch data stores up to 12 punches per column. They can be referred to by their punch position from the starting column (1-12) or by the actual punch (1-9, 0, X, Y), where 0, X, and Y mean punch positions 10, 11, and 12 respectively.

This statement will check for any of the allowed punches in column eight. The \*P modifier by itself means that column eight may contain any or all of the punches one through six.

```
CHECK [8*P^1//6] "Should be multi 1-6"
```

Again blank will be an error unless it is specifically allowed.

```
CHECK [8*P^1//6/B] "Should be multi 1-6 or blank"
```



**Multiple Columns/Single Punch**

- Q 1. Where did you acquire this product?
- (15) Received as a gift.....1
    - Appliance store.....2
    - Department store.....3
    - Furniture store.....4
    - Catalog showroom.....5
    - Discount store.....6
    - Hardware store.....7
    - Kitchen specialty store.....8
    - Building supply store.....9
    - Pre-installed.....0
    - Mail order.....X
    - Kitchen remodeler.....Y
  - (16) Plumber.....1
    - Other.....2

In this example we need to check for one punch, but over multiple columns (e.g., a long list of choices where you are looking for the first mention). This question has 14 possible responses. Since punch data stores up to 12 punches per column, this question requires two columns.

The cleaning statement checks columns 15 and 16 for a punch. If more than one punch is found the error message is printed.

CHECK [15.2^1//14] "Should be single 1-14"

15.2 tells the program that the field starts in column 15 and is two columns wide, or columns 15 and 16. The punches are referenced by their position to the starting column. Hence a two punch in column 16 is in punch position 14, starting from column 15.

You can also check a set of questions in a single CHECK statement, for instance a series of rating scales.

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

CHECK [51,...,55^1//5]

This statement will check each column in the set: 51; 52; 53; 54; and 55, for any valid response one through five. The ellipsis (...) is used after the first column number to abbreviate this list of consecutive data locations. The columns do not need to be consecutive, but you will need to specify each one, e.g., [50,51,2/64,5/26^1//5].

#### *Multiple Columns/Multiple Punches*

Q 8. From your own experience and knowledge, what do you especially like about this restaurant? (Check all that apply)

- (15)
- 1 Good service/prompt service
  - 2 Dependable/continuous service
  - 3 The courteous employees they have/helpful
  - 4 I like the food/good food
  - 5 Food selection
  - 6 Good prices
  - 7 Computerized/accurate billing
  - 8 Helpful in explaining billing questions
  - 9 Good entertainment
  - 0 Variety of entertainment
  - X Nice family atmosphere
  - Y Established place/has been around for awhile
- (16)
- 1 Accessible/Available/They're everywhere
  - 2 Like everything/good place
  - 8 Other
  - 0 Don't know/No answer
  - X Nothing

This question allows more than one response, but if either of the last two responses is chosen then no others should be present.

The statement below checks for any of the allowed punches in columns 15 and 16. It also checks that if either Don't Know or Nothing was chosen then no other response is allowed.

```
CHECK [15.2*P^1//14/20/(-)22/(-)23] "Should be multi
1-14,20 or single 22 or 23"
```

### ***Multiple Columns (non-contiguous)***

This cleaning situation would usually occur where you have a question with an Other/specify and no columns or not enough columns were available in the data file to code the 'Other' responses in columns consecutive to the original question. You still need to write separate CHECK statements that check for inconsistencies such as an exclusive punch violation code or more than one response when only one is allowed, in addition to valid punches or blank.

### ***Complex Single Punch***

#### **Example:**

```
IF NUMBER_OF_ITEMS ([1/10^1//10] WITH [2/10^1//5]) <> 1
    ERROR "Should only be one answer in cols 1/10 or 2/10
    combined"
ENDIF
```

This example uses a more complex cleaning statement than you have seen in previous examples. There are two data locations and a possible 15 categories, but only one can be present. We need a way to treat the two data locations and all of their categories as one unit in order to count the total number of categories present. The keyword WITH is a vector joiner that connects the categories in record one column 10 with those in record two column 10. This forms a single expression including all the categories. NUMBER\_OF\_ITEMS is a Mentor function (meaning it acts on the expression given inside the parentheses) that counts the number of categories that are true for the current case. If the result is not equal ( $\neq$ ) to one, then we instruct the program to flag the case and print the error message specified. To allow No Answer you would say  $> 1$ .

### ***Multiple Punches With An Exclusive Response***

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

#### **Example:**

```
IF ([1/10^1//10] OR [2/10^5] AND [2/10^5]
    ERROR "Can be multi in 1/10,2/10 or 2/10^5"
ENDIF
```

This example is similar to the one described above, but more than one response can be present in either location. In addition, one response is considered exclusive, meaning if it is present then no other response is allowed.

This example uses two other joiners, OR and AND. This IF statement says that if you have any of the allowed categories turned on anywhere in either record one column 10 OR record two column 10, AND record two column 10 is a five punch (and exclusive), then this is an error.

Remember, neither of these examples checks the validity of the actual data in these columns. They only count categories present in the expression formed by the joiner(s). Prior to one of these IF blocks, you would check the data with individual CHECK statements for each column.

Refer to *Appendix B: TILDE COMMANDS* under ~DEFINE VARIABLE= for information on other joiners and functions. Joiners are also used in data tabulation to form complex banners and to base the table (see “5.1 Expressions and Joiners”).

#### ***Cleaning ASCII Data***

ASCII data can be collected either as a number or a string. To specify ASCII data, use a pound sign(#). Categories of responses are specified in the same way as for punch data with single (/) or double slash (//). ASCII data can also be checked for zero-filled columns. Numeric data can be checked for literals like DK indicating a response such as Don't Know or Refused.

#### **Numeric Data**

Q 3. How many years have you lived at this address? \_\_\_\_\_



(Enter RF for refused)

This question collects a numeric response and includes a literal if the respondent does not provide an answer.

CHECK [7.2#1//20/RF] "Should be a number 1-20 or RF"

The cleaning statement will check for a range of numbers or the literal RF. An error will occur if what is found in the data does not match your description (number out of range, blank field, different literal, etc.).

For this example we have assumed that the range can be 1-20 years. That requires a two column field. If the range were 1-5 years we would still be checking a two column field since the literal RF will be coded into two columns.

### **Zero-Filled Numeric Data**

You may be post-processing the data file in a software package that requires a number in every column. It is easy to modify the previous cleaning statement to check for leading zeros rather than blanks. In addition, it is often easier to examine your data if all of the columns in a numeric field contain a number.

Here is the same CHECK statement modified to verify that where the response is less than 10, the leading columns are zero-filled.

CHECK [7.2\*Z#1//20/RF] "Should be 01-20 or RF"

\*Z says that leading columns for numeric responses must contain a zero or else it is an error. The default is to allow blanks.

### ***Single Response String Data***

Responses coded as literals are referred to as string data. Strings can be alphabetic or numeric and unlike punch data there is only one code per column. Typical examples would be state or zip codes.

Q 11. Please enter the appropriate 4-character code for your department. \_\_\_\_

In this question the respondent must fill in a code designating their department within a company. To clean this question you would need to check the field for each legitimate code (presumably from a list of all the possible department codes):  
CHECK [5/5.4#A111/P222/M333/S444/T555] "Not a valid dept. code"

Notice how the columns are referenced in the statement above. Record/column is the program default for all column references. Absolute column references are also allowed. We could have written the data column location as 325.4.

If we wanted to allow a blank for this field then the CHECK statement would look like this:

```
CHECK [5/5.4#A111/P222/M333/S444/T555/"      "]
```

The set of double quotes acts like the B used in punch data examples. We have now defined a variable to say 5/5.4 may be any one of these responses or blank.

### *Multiple Response String Data*

#### **Coded Open Ends**

You may have an open-end on a survey coded to allow for multiple answers of two or more digits, as in the following example:

**Example:** Q 6. WHAT WAS THE MAIN IDEA OF THE AD?

(19-20) \_\_\_\_\_  
(21-22) \_\_\_\_\_  
(23-24) \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Imagine that valid answers are coded from 01 through 17. Code 95 stands for Other and code 99 stands for an exclusive code, such as Don't Know. No code should appear more than once, there can be no leading or embedded blanks, and the exclusive codes cannot appear with any other code. The CHECK statement to clean this question for all of the parameters given would look like this:

**Example:** CHECK [19.2,21,23 \*ZF #1//17/95/(-)99] "Can be 1-17,95 or 99"

All three fields are specified in the same variable. We could have used the ellipsis, e.g., [19.2, . . . ,23, to say the same thing. Also note that fields do not have to be contiguous,

e.g., [1/23.2,1/25,1/41,2/43 \*ZF #1//17/95/(-)99].

\*Z says that the first column for each of the codes one through nine must contain a zero (0), e.g., 02, and if not it is an error.

\*F will check all three locations for the responses defined in the variable. An error is generated if an invalid code is found; if any code appears more than once; if the exclusive code appears with another code; or if leading blanks or embedded blanks are found (due possibly to keypunch error).

### **Allowing Duplicate Codes**

You might have a situation where duplicate codes in the data are not an error. In this case you would need to use the \*L modifier.

**Example:** CHECK [19.2,21,23 \*ZL #1//17/95/(-)99]

\*L will still print an error for an invalid code, an exclusive code violation, or if leading or embedded blanks are found, but duplicate valid codes will not be an error.

### ***Multiple Punches in a Single Response Field***

You can clean a location if you have the situation of having multiple punches in what should have been a single-punch field. Use the \*S modifier to determine which punch to keep by the order of 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.

***Checking Multiple Locations For A Valid Response***

To simply check multiple locations for a valid response do not use either the \*F or the \*L modifier. There is also no need to specify any exclusive responses since only one valid response can appear in each set of columns. There is no relationship amongst the responses in these columns. The only errors we care about are either an invalid code or a single-digit code without a leading zero (0):

**Example:** CHECK [19.2,21,23 \*Z #1//17/95//99]

This statement is the same as writing three separate CHECK statements. It checks each of 19.2, 21.2, and 23.2 for any response 01-17 or 95-99.

***Cleaning for Skips or Bases***

A common situation you will clean for is whether skip patterns were executed correctly in the questionnaire. This is accomplished by writing an IF-THEN-ELSE-ENDIF block. In our sample questionnaire question 4 should have a valid response only if the response to question 3 was Yes (respondent has siblings).

**Example:**

```
IF [18^1] THEN
    CHECK [19.2*Z#1//10] "Should be a number 1-10"
ELSE
    CHECK [19.2^B] "Should be blank"
ENDIF
```

This example says if column 18 contains a 1 punch (answered Yes to question 3) then check columns 19 and 20 (how many siblings) for any valid response 1 through 10, otherwise (ELSE) those columns should be blank (meaning the respondent answered No to question 3). The word THEN is not required. If you use the CHECK\_COLUMNS command in your cleaning procedure, the ELSE clause in this IF statement is not needed unless you want to specify your own error message. CHECK\_COLUMNS will report any columns that are not blank. Commands are indented and on separate lines for readability only.

You can have unlimited levels of IF blocks within IF blocks. You can use the GOTO command to branch into or out of an IF block.

### ***Branching in the Cleaning Procedure***

The GOTO command allows you to branch anywhere (forward only) in your cleaning procedure including into or out of IF blocks. GOTO is especially useful to skip large blocks of questions where a conditional statement would contain many statements, or to exit a complex IF block. GOTO's are often used at the beginning of a section such as check only if male or female. Most of the time you will control spec file execution with IF blocks.

**NOTE:** Test carefully if you skip into an IF block. You could get unexpected results. An ELSE coming after the GOTO label will be ignored. Below is an example to illustrate this point.

```
~DEFINE
PROCEDURE=TEST:
  GOTO XX
  IF [5^1] THEN
    PRINT "after the if/before XX"
    XX: PRINT "at XX/before else" <-- Only this line is
executed
  ELSE
    PRINT "after XX"
  ENDIF
}
```

In our example paper and pencil questionnaire let's say that a No response to question 3, "Do you have any siblings?," skips to question 11. The IF statement checks column 18 for a 2 punch (respondent does not have siblings), the GOTO command then branches to the label Q11 skipping all of the cleaning statements for questions five through ten.

**Example:** IF [18^2] THEN  
GOTO Q11

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

```
ENDIF  
cleaning statements for questions 5-10  
Q11: cleaning statement for question 11
```

Without the GOTO in the previous example, you would have to include the cleaning statements for questions five through ten in one or more IF blocks to control when they would execute.

If your procedure includes a CHECK\_COLUMNS command then the program will print an error if any of the columns used by questions five through ten are not blank. See the example cleaning procedure in section 2.3 for the sample paper and pencil questionnaire.

A GOTO statement can have multiple conditions and labels. If the question you are checking includes more than one category, such as male/female respondent, you can write a GOTO with a label for each one.

```
Example: CHECK [18^1/2] "Should be single 1,2"  
            GOTO (*,Q11) [18^1/2]  
cleaning statements for questions 5-10  
Q11: cleaning statement for question 11
```

The CHECK statement checks column 18 for a valid response. The GOTO says if the response is a one continue processing (\*), and if the response is a two branch to the label Q11. If response one also branched to another point in the cleaning procedure, then we could have provided a label for it, e.g., GOTO (Q20,Q11) [18^1/2]. The comma delimiter is optional, but use at least one space between labels if you do not use a comma.

#### *Complex Cleaning*

Mentor provides many features to help you check the validity of your data. You have seen just a few of the most commonly used commands in our example cleaning procedure. In addition to other cleaning commands there are also functions that allow you to count responses, add fields to check a sum, modifiers to

treat the data found in a field differently than the program default, or keywords called system constants that allow you to quickly get information about your data. We will cover a few of the more common examples here, but refer to sections “9.3.1 System Constants” and “9.3.2 Functions” for information on other useful features.

### ***Counting Number of Responses***

In a previous example under *Multiple Columns (non-contiguous)* we used the NUMBER\_OF\_ITEMS function to count categories present across a range of columns. The same function can also count categories in a single column.

You may need to verify that a specific number of responses is present in a particular field. If we define the range of valid responses as separate categories, then NUMBER\_OF\_ITEMS can count the categories that have a response. The result can be compared to the number of responses that must be present.

Here is an example where respondents are asked to circle their three most common sources of news information on a precoded list. The cleaning statements follow.

```
(5)  1    LOCAL TV NEWS
      2    NATIONAL TV NEWS
      3    CNN
      4    LOCAL NEWSPAPER
      5    OUT OF STATE NEWSPAPER (e.g., WALL STREET
      JOURNAL)
      6    RADIO NEWS
      7    NPR (e.g., MORNING EDITION)
      8    OTHER
```

```
CHECK [5*P^1//8] "Can be multiple 1-8"
IF NUMBER_OF_ITEMS([5^1//8]) > 3 THEN
    ERROR "Has more than 3 punches" [5.1$P]
ENDIF
```

The first statement checks the field for valid punches, any response one through eight. The NUMBER\_OF\_ITEMS function counts the number of categories (5^1, 5^2, 5^3, 5^4, 5^5, 5^6, 5^7, and 5^8) present in the current case. If the result is greater than three then the case is flagged with an error and the message is printed.

Use the ~CLEANER command ERROR whenever you are not using a CHECK command, most often you will use ERROR inside an IF condition structure as in the example above. ERROR prints the ID of the current case and any text enclosed in "quotes" following the ERROR command. It also flags the case where an error has been found. You can locate flagged cases in the open data file with the FIND\_FLAGGED command. Optionally, you can print the contents of a field as part of the ERROR statement: as a string or literal (\$); as punches (\$P); or as a number [location]. In our example the contents of column five will be printed as a punch. Refer to Chapter 3: "Reformatting Your Data" and Appendix B: TILDE COMMANDS under ~CLEANER PRINT\_LINES for other examples.

There are two other command used to generate text: SAY and PRINT\_LINES. These commands are best used for trouble-shooting because they are not included as "hits" in the cleaning summary. SAY does not print the case ID, does not set the error flag on the case, and prints only to the list file, not to the print file.

Here is the previous ERROR statement rewritten with the SAY command. The System constant CASE\_ID causes the case id to print before the "text":

```
SAY CASE_ID "Has more than 3 punches" [5.1$P]
```

If you need to send this message to the print file, use the PRINT\_LINES command:

```
PRINT_LINES "/s has more than 3 punches /s" CASE_ID  
[5.1$P]
```

### ***Checking For A Constant Sum***



An example of a constant sum would be where you have asked respondents what percent of the time they spend on three activities. Your cleaning statement might need to check that the sum of the responses adds up to 100%.

The SUM function returns the sum of values of the numbers that exist in the data. Missing values are ignored, and if all values are missing then the result of SUM is missing.

**Example:**

```
IF SUM([5.2,7,9]) < 100
    ERROR "Responses do not add up to 100%"
ENDIF
```

In this example, the three data locations have the same length, so it can be expressed as one data reference with multiple locations, and the length only needs to be specified once. A length of one is the default; specifying it is optional. When the lengths are different each location and length must be specified separately, separated by commas:

**Example:** `SUM([5.2],[7.3],[2/10.5]) < 100`

You may specify any of the following inside the parentheses ( ):

- a data location
- an absolute number
- a number returning function such as SQUARE\_ROOT or NUMBER\_OF\_ITEMS, refer to “9.3.2 Functions” for other functions that return numbers
- a math statement such as  $2 * 4$
- the name of a previously defined numeric variable

**Example:**

```
SUM([5.2],123.5,NUMBER_OF_ITEMS([5^1//8]),8*3,AGE)
```

***Checking an Aided/Unaided Awareness Grid***

Next is an example of an aided/unaided awareness grid that asks about brand recognition and usage. In the cleaning procedure (in addition to checking for valid responses) we want to be sure that the first mention is unique and that the respondent did not mention advertising for brands not mentioned previously. Then we want to check that an aided mention was not previously mentioned (unaided).

Next we check that the aided advertising question was asked, and if so that those mentions appear somewhere previously. Finally the usage question is checked making sure that those mentions also appear in one of the previous mention questions.

Q.1a First Unaided Mention	columns 21-22
Q.1b All Other Unaided Mentions	columns 23-24
Q.1c Advertising Unaided Mentions	columns 25-26
Q.2a Aided Mentions (key brands)	columns 27-28
Q.2b Aided Advertising Mentions	columns 29-30
Q.2c Aided Usage	columns 31-32

SOFT DRINK MAJOR BRANDS LIST

	Q.1a	Q.1b	Q.1c	Q.2a	Q.2b
Q.2c					
COKE . . . . .	21 1	23 1	25 1	27 1	29 1 31 1
DR. PEPPER . . . . .	2	2	2		
GATORADE . . . . .	3	3	3	3	3 3
MELLO YELLOW . . . . .	4	4	4	4	4 4

MOUNTAIN DEW . . . . .	5	5	5	5	5	5
-----						
MUG ROOT BEER . . . . .	6	6	6			
-----						
MUG CREAM . . . . .	7	7	7			
-----						
PEPSI-COLA . . . . .	8	8	8	8	8	8
-----						
RC COLA . . . . .	9	9	9	9	9	9
-----						
7-UP . . . . .	0	0	0			
-----						
SLICE . . . . .	X	X	X	X	X	X
-----						
SPRITE . . . . .	Y	Y	Y			
-----						
SUNDROP . . . . .	<sup>22</sup> 1	<sup>24</sup> 1	<sup>26</sup> 1	<sup>28</sup> 1	<sup>30</sup> 1	<sup>32</sup> 1
-----						
OTHER (SPECIFY: ) _____	8	8	8			
-----						
NONE/NO MORE . . . . .	X		X	X	X	X
-----						
DON'T KNOW . . . . .	Y		Y	Y	Y	Y
-----						

Here is the cleaning procedure for this grid. The numbers in the left margin correspond to the explanation that follows.

~DEFINE

PROCEDURE= {GRID:

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

```
      ''Unaided First Mention(Q.1a)
1      CHECK [21.2^1//13/20/(-)23(-)24]
2      IF [21.2*P^1//13/20] THEN
''All other Unaided Mentions(Q1.b)
2a      CHECK [23.2*P^1//13/20/23]
2b      IF [21.2^1//13] INTERSECT [23.2^1//13] THEN
2c      ERROR "23.2:" [23.2$P] "HAS SOMETHING ALSO IN
21.2:" [21.2$P]
      ENDIF
''Unaided Advertising Mentions (Q1.c)
3      CHECK [25.2*P^1//13/20/23/24]
3a      IF [21.2,23.2*F^1//13/20] >= [25.2^1//13/20] ELSE
3b      ERROR "25.2:" [25.2$P] "HAS SOMETHING NOT IN 21.2:"
[21.2$P] "OR 23.2" [23.2$P]
      ENDIF
ENDIF
''All Aided Mentions (key brands only)(Q.2a)
4      IF NUMITEMS([21.2,23*F^1/3//5/8/9/11/13]) < 8 THEN
4a      CHECK [27.2*P^1/3//5/8/9/11/13/23/24]
4b      IF [27.2^1/3//5/8/9/11/13] INTERSECT
[21.2,23.2*F^1/3//5/8/9/11/13] THEN
4c      ERROR "27.2:" [27.2$P] "HAS SOMETHING ALSO IN
21.2:" [21.2$P] "OR 23.2" [23.2$P]
      ENDIF
      ENDIF
''Aided Advertising Mentions (key brands)(Q.2b)
5      IF NUMITEMS([21.2,23,27*F^1/3//5/8/9/11/13]) >=
NUMITEMS([25.2^1/3//5/8/9/11/13]) & THEN
5a      CHECK [29.2*P^1/3//5/8/9/11/13/23/24]
5b      IF [21.2,23,27*F^1/3//5/8/9/11/13] >=
[29.2^1/3//5/8/9/11/13] ELSE
5c      ERROR "29.2:" [29.2$P] "HAS SOMETHING NOT IN 21.2:"
[21.2$P] "23.2:" [23.2$P] &
```

```
"OR 27.2:" [27.2$P]
      ENDIF
5d     IF [25.2^1/3//5/8/9/11/13] INTERSECT
[29.2^1/3//5/8/9/11/13] THEN
5e     ERROR "29.2:" [29.2$P] "HAS SOMETHING ALSO IN
25.2:" [25.2$P]
      ENDIF
      ENDIF
''Aided Usage Mentions(Q.2c)
6     IF NUMITEMS([21.2,23,27*F^1/3//5/8/9/11/13]) > 0 THEN
6a     CHECK [31.2*P^1/3//5/8/9/11/13/23/24]
6b     IF [21.2,23,27*F^1/3//5/8/9/11/13] >=
[31.2^1/3//5/8/9/11/13] ELSE
6c     ERROR "31.2:" [31.2$P] "HAS SOMETHING NOT IN 21.2:"
[21.2$P] "23.2:" [23.2$P] &
"OR 27.2:" [27.2$P]
      ENDIF
      ENDIF
7     CHECK_COLUMNS
```

Indenting embedded IF statements is the recommended style for specifications. This allows you to keep track of which IF's are still in effect.

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

- 1 CHECK first mention for a valid single response. Responses 23 and 24 are exclusive.
- 2 IF a brand was mentioned
- 2a THEN CHECK other mentions for a valid response: can be one or more of (1-13,OTHER) or DON'T KNOW. NONE/NO MORE would be an invalid second mention, so code 24 is not included here.
- 2b IF the first mention appears with other mentions, it must be a duplicate mention.

**NOTE:** OTHER is the exception.

- 2c THEN print an error and the contents of the columns as punches (\$P).
- 3 CHECK unaided advertising for a valid response, can be multi (1-13,OTHER) or single NONE, DON'T KNOW.
- 3a IF the response is in unaided advertising (25.2) then it must also appear in either first mention (21.2) or other mentions (23.2)
- 3b Otherwise (ELSE) it is an error. Print "error message" and print the contents of the columns as punches (e.g., [25.2\$P]).
- 4 IF all the "key brands" (i.e., the eight brands on the aided list) were mentioned then there are no brands to ask about. This statement counts key brands mentions unaided. If it is less than eight THEN it will check for aided mentions.
- 4a CHECK unaided mentions for valid codes.
- 4b IF a key brand appears in either first mention or other mentions, and also in aided mentions
- 4c THEN print an error and the contents of the columns as punches (\$P).
- 5 IF the number of key brands aware of through advertising is greater than or equal to the number of total key brands aware of, then there were some brands to ask about in aided advertising mentions.
- 5a THEN check aided advertising mentions for valid responses. Those can be any of the key brands (multi-punched) or single punched NONE, DON'T KNOW.

- 5b IF there is a mention in aided advertising (29.2) then it must be mentioned previously in aided or unaided (21.2, 23.2, or 27.2)
- 5c OTHERWISE (ELSE) print an error and the contents of the columns as punches (\$P).
- 5d IF aided and unaided advertising mentions share any responses
- 5e THEN print an error and the contents of the columns as punches (\$P).
- 6 IF the count of unaided and aided key brands mentions is greater than 0
- 6a THEN CHECK Aided Usage Mentions for valid responses.
- 6b IF there is a mention in usage (31.2) then it must be mentioned previously in unaided or aided mentions (21.2, 23.2, or 27.2)
- 6c OTHERWISE (ELSE) print an error and the contents of the columns as punches (\$P).
- 7 Report all columns that should be blank.

***Checking Ranked Responses***

There are two Mentor functions to help you clean for ranked responses. The one you use depends on what you expect the data to look like: either that all of the ranks present are consecutive starting with the first rank (less restrictive), or that every one of the ranks required is present (more restrictive)

**Example:**

Q1. RANK YOUR TOP 5 CHOICES (1 - FIRST CHOICE, 5 - FIFTH CHOICE)

MCDONALD ' S	-----	(10)
WENDY ' S	-----	(11)
BURGER KING	-----	(12)
TACO BELL	-----	(13)
PIZZA HUT	-----	(14)
JACK IN THE BOX	-----	(15)
CARL ' S JR.	-----	(16)
KENTUCKY FRIED CHICKEN	-----	(17)
ROUND TABLE PIZZA	-----	(18)

```
CAESAR'S PIZZA          -----      (19)
```

First you would check for valid response across the columns (including no response). Next you would write an IF statement with either the CASCADE or the COMPLETE function depending on how you want to check the ranking of those responses.

**Example:** CHECK [10, ..., 19^1//5/B]

*Using the CASCADE Function*

The CASCADE function checks that ranking starts with the first value and without breaks through the end of the list.

```
Example: IF CASCADE ([10, ..., 19*F^1//5]) ELSE
              ERROR "Q1 DOES NOT CASCADE"
              ENDIF
```

CASCADE requires that the starting point for ranking matches the variable definition. In this example, the respondent must have ranked one of the restaurants as one since the first category is one. Not all five rankings need to be present in the data, but they must be consecutive. For example, if a restaurant is ranked as fourth, then there must be restaurants ranked as first, second and third. However, just a one ranking or no ranking at all would not be considered an error.

In earlier examples we used the \*F modifier to verify data using a CHECK statement. \*F actually nets together occurrences of the same mention across the columns specified. In a CHECK statement if responses can be netted then duplicates exist and this is treated as an error. In checking that ranked responses cascade, we are only concerned that the rankings follow the categories defined in the variable.

In the variable defined above, \*F creates 5 categories: 10, ..., 19^1; 10, ..., 19^2; 10, ..., 19^3; ...; etc.



Duplicate rankings are netted together. For instance, if columns 10 through 19 contain two number three rankings, they are treated as one number three ranking. CASCADE then checks that the final set responses starts with the one response and continues or cascades through the end of the list or response five.

### ***Using The COMPLETE Function***

Unlike CASCADE, the COMPLETE function requires that all the categories defined be present in the data. In our example, COMPLETE says that only the set one through five is correct and anything else, including no ranking, is an error. Remember that COMPLETE could return many errors for a self-administered questionnaire since it is unlikely that all the required answers will be present.

#### **Example:**

Ranking must be consecutive, and must contain one through five (duplicate rankings are allowed):

```
IF COMPLETE ( [10,...,19*F^1//5] ) ELSE  
    ERROR "Q1 NOT COMPLETE"  
ENDIF
```

If you need to check for duplicate rankings, you will need to add this condition along with either CASCADE or COMPLETE:

```
IF NUMBER_OF_ITEMS([10,...,19^1//5]) > &  
NUMBER_OF_ITEMS([10,...,19*F^1//5])  
    ERROR "Q1 HAS DUPLICATES" [10.10$]  
ENDIF
```

The function NUMBER\_OF\_ITEMS counts all occurrences of the rankings including duplicates and then compares this to the net count. If all occurrences is greater than the net count, you have duplicates. The error message prints the data in all 10 columns so that you can see the actual duplicates.

### 2.3.2 Correcting Errors

Once you have your error listing, you should examine it to decide if any errors can be cleaned in batch mode. That will mean writing a procedure using commands that will cause the program to alter the data. Under *Auto-fixing The Data* you will find two examples of batch mode cleaning. This section will explain how to clean your data from the listing on a case by case basis using the procedure that generated the error listing. By using the procedure to clean your data, you will also find any new errors that could be introduced as a result of altering a field.

#### MANUALLY CLEANING THE DATA

Start the cleaning section of Mentor by entering:

```
CLEANIT
```

This utility starts Mentor, starts the cleaner block, and opens a log file called clean.log. CLEANIT records your commands in the log file so you can have a record of the commands you issued. If a file called clean.log already exists, CLEANIT will append to the existing file. If you do not want to append to an existing log file, you will have to rename or remove clean.log before starting CLEANIT again.

Section “2.5.1 Quick Reference: Cleaning Commands And Examples” has a list of commonly used ~CLEANER commands and a brief explanation of each one. For a complete list of ~CLEANER commands, see *Mentor, Volume 2, Appendix B*. For a description of the CLEANIT utility, see the *Utilities* manual.

Now, open the DB file for access to SHOWERR (the cleaning procedure defined in “2.3 CLEANING SPECIFICATIONS”):

```
CleaNer--> >USE_DB procS
```

Load the data file with FILE command:

```
CleaNer--> FILE sampl
```

Start the procedure SHOWERR with the HUNT command:

```
CleaNer--> HUNT showerr
```



HUNT starts the beginning of the data file, stops on the first case with an error, and displays all the errors in that case. For example:

```
ID0001, error 3: [19.2#] Should be a number 1-10: a valid answer is required
    [19.2#]="12"
```

Now you can modify the data with the MODIFY\_ASCII command:

```
CleaNer--> MA 19.2
```

Mentor will display a the two columns and their contents (vertically), and give you the MODIFY\_ASCII (MA) prompt:

```
ID: 0001, #1 1/19.2
    12
    90
    --
    12
MA->
```

This is where you will enter the correct data:

```
MA ->08
```

After modifying the data, you can check to see if the data is corrected with the DISPLAY\_ASCII command:

```
CleaNer--> DA 19.2
```

Mentor will now display:

```
ID: 0001, #1 1/19.2
    12
    90
    --
    08
```

The rest of interactive cleaning is just repeating these steps for each case with an error in it. You can use HUNT (start at the beginning of the file) or FIND (move forward from the current case) to find the next case with an error. HUNT is a good way to re-check the cases you have already corrected. If you choose NOT to correct a case, and want to move on the next case with an error, you can move forward one case with the NEXT command, and then enter

```
"FIND showerr."
```

### *Interactive Cleaning Tips*

- You can execute a procedure on *only* the current case by using an exclamation point, for example:

```
!showerr
```

- You can specify more than one command on a line by separating them with semicolons, for example:

```
NEXT; DA 19.2
```

- You can use the REDO command to repeat the previous command.
- You can define a data location that you will be modifying in several cases with a name that you can reference with another command, for example:

```
DEFINE spot[19.2]
```

```
    MA spot
```

- You can automate repetitive commands by writing a small procedure and then executing it with an exclamation point, for example:

```
DEFINE PROCEDURE= {chkit: IF [3^1]; DA 19.2; MA 08; ENDIF}
```

```
    !chkit
```

- If you have a long or complicated command that you want to edit, you can use the >EDIT\_PREVIOUS command. This will bring up the previous command in the on-line editor. Correct the command, and when you press ESC, you will exit the editor, return to the CLEANER block, and the command will be re-executed on the current case.
- You can combine commands into a file and instruct Mentor to read the file, for example:

- 1 Create the file CLNIT.SPX with the following lines:

```
~CLEANER
>USE_DB procs
SET LOGGING
FILE sampl
FIND showerr
```

- 2 Execute Mentor, using an ampersand to tell Mentor to expect input from the keyboard once the spec file has been read, and "echo" to send all program messages to the screen:

```
MENTOR &clnit.spx clnit.lfl,echo (DOS/UNIX)
RUN MENTOR.CGO.CFMC;INFO="&CLNITSPX CLNITLFL,ECHO" (MPE XL)
```

### ***Clean.log***

Below is a sample of a clean.log file. In this example, the user opens the Roadrunner data file and changes the first four columns of the first case from "0001" to "abcd". What the user typed is included in the log file; those lines start with "con:" for console.

```
con: file rrunr
con: ma 1.4
ID: 0001 (study code=RRUN, int_id=intv):1.4
Display 1/1.4:
  0
  1234
  ----
  0001
con: abcd
con: da 1.4
ID: 0001 (study code=RRUN, int_id=intv):1.4
Display 1/1.4:
  0
```

```
1234
----
abcd
con: ~end
Enter (Y)es or (N)o please-->
con: y
```

### *Using Cleaning Screens*

Mentor can also display a cleaning screen when it finds an error, and you can modify data from this screen. You can set up a controlled cleaning run for someone else to execute. Creating screens for others to use has several advantages:

- they will only be able to modify the data columns you specify.
- they don't need to know data modification commands.
- error checking is built in.

Referring to our example cleaning specifications in section 2.3, you would substitute the EDIT command for CHECK. Mentor will first list an error, and then present a cleaning screens.

**Example:** EDIT [18^1/2] "Should be single 1,2"

You can add explanatory text to the data description to display on the cleaning screen.

**Example:** EDIT [\$T="Has Siblings"18^Yes:1/No:2] "Should be single 1,2"

Mentor displays a screen similar to this:

```
Has Siblings
001 Yes
002 No
-->
```

enter the new values or RES/BLK/TERM ([ 1/18.1^B ])

Mentor automatically generates a three digit zero (0) filled number for each category defined in the variable when it is displayed in a question structure (similar to Survent) as above.

Refer to “2.4.1 Correcting Errors” for an explanation of this screen and a sample specification file.

There are limitations to using this method. You could write a valid CHECK statement that could not then be used with the EDIT command. The variable you define must represent a single question in order for Mentor to present it with a cleaning screen. For instance the statement CHECK [10, . . . , 15^1//5] will check columns 10, 11, 12, 13, 14, and 15 separately for a one through five punch. In effect this is six separate questions. In this case you could not substitute EDIT for CHECK. You would need to write separate EDIT statements for each of the six columns.

### **AUTO-FIXING THE DATA**

Auto-fixing is essentially a set of rules for cleaning the data without examining each case for why the error occurred. An example might be an unverified data file where the data is off by a column due to a keypunch error. Cleaning with this method will mean that your data set is not as reliable, but the cleaning process will be less labor intensive, especially as the number of cases increases.

We recommend that you always save the auto-fixed data to a new file, either using a copy of your original System file or by writing the modified cases to a new file. Make periodic copies of the data file especially if you find yourself modifying the procedure in subsequent cleaning runs. In this way, you would only have to go back one or two copies to undo the last set of changes made to the data file.

Another option is to create a new file for the "clean" data. ~OUTPUT creates the new data file and WRITE\_CASE writes the case to the new file. Use WRITE\_CASE as the last command in your cleaning procedure.

## PREPARING YOUR DATA

### 2.3 CLEANING SPECIFICATIONS

What follows is an example of auto-fixing using the sample paper and pencil questionnaire from section 2.2.1. This cleaning procedure uses a new ~CLEANER command, CLEAN. **USE THIS COMMAND WITH CAUTION; IT BLANKS THE DATA LOCATION WHEN IT DOES NOT MATCH YOUR DATA DESCRIPTION.** Like the CHECK command, you can define your own error message in addition to the one Mentor prints for each case it changes. This message is especially useful in the error summary by telling you how many times a variable was changed to a blank.

An explanation of how each command in the procedure will affect the data follows this example.

#### Example:

```
~DEFINE
PROCEDURE={AUTOFIX:
  CLEAN [15.3#MON/TUE/WED/THU/FRI/SAT/SUN] "Not a valid code"
  CLEAN [18^1/2] "Should be single 1,2"
  IF [19.2#10//99] THEN
    TRANSFER [19.2] = 10
  ENDIF
  CLEAN [19.2*Z#1//10" "] "Should be a number 1-10 or blank"
  IF [19.2^B] THEN
    MAKE_DATA [18^2]
  ELSE
    MAKE_DATA [18^1]
  ENDIF
  IF [21^8] AND [21*P^1//7] THEN
    MAKE_DATA -[21^8]
  ENDIF
  CLEAN [21*P^1//7/(-)8] "Should be multi 1-7 or single 8"
  BLANK [22-80]
  WRITE_CASE
```



```
}  
~INPUT SAMPL  
~OUTPUT SAMPL2  
~EXECUTE PROCEDURE=AUTOFIX  
~END
```

**CLEAN [15.3#MON/TUE/WED/THU/FRI/SAT/SUN] "Not a valid code"**

Blanks the data associated with the *day of the week question* when it does not match one of these responses.

**CLEAN [18^1/2] "Should be single response 1,2"**

Blanks the data in the columns of the *has siblings question* if the response is not a single response of one or two.

```
IF [19.2#10//99] THEN  
  TRANSFER [19.2] = 10  
ENDIF
```

Says if the answer to the *number of siblings question* is more than 10, recode it to be 10.

The TRANSFER command can modify numeric, string\$, or punch\$P data (though punch data is more often modified with the MAKE\_DATA command). In this example we have a very simple modification. You can also modify data based on the result of an expression such as multiplying one location times another.

TRANSFER also verifies the data on both sides of the equal sign (=) and will return an error if the data types do not match. A dollar sign (\$) is required to modify string data such as the codes for day of the week in our example procedure. Refer to the related command, MODIFY, under ~CLEANER in *Appendix B: TILDE COMMANDS* for more information and examples.

**CLEAN [19.2\*Z#1//10/" "] "Should be a number 1-10 or blank"**

Blanks the data for the *number of siblings question* if it is not a valid response 1-10 or blank. Given the skip pattern in this questionnaire we know it is possible for this field to be blank. The empty quotes " " response in the data definition

[19.2#1//10/" "] instructs Mentor that blank is also a valid response.

**IF [19.2^B] THEN**

**MAKE\_DATA [18^2]**

**ELSE**

**MAKE\_DATA [18^1]**

**ENDIF**

Says if the *number of siblings question* is blank, generate a response of two (No) for the *has siblings question*, and if it is not blank generate a response of one (Yes).

The MAKE\_DATA command replaces punch type data; it blanks the location first. It is the recommended way to simply add or remove punches from a data location. The columns to be modified are defined like any other punch variable by specifying the location and the new punch value inside square brackets ([ ]).

**IF [21^8] AND [21\*P^1//7] THEN**

**MAKE\_DATA -[21^8]**

**ENDIF**

Says if the *other family members question* has a none response (code eight) with any of the other valid responses (codes one through seven), then remove the eight response.

In this example, we have used one of two modifiers available on the MAKE\_DATA command. These are useful when you do not want to blank the data location. Minus (-) removes a punch. In this procedure we only want to remove the eight code, but leave any others in tact. A plus sign (+) would add a punch to the location.

**CLEAN [21\*P^1//7/(-)8] "Should be multi response 1-7 or single 8"**

Blanks the *other family members question* if it is not a valid code of 1-7 (any combination) or exclusive code 8.

**BLANK [22-80]**

Blanks the remaining columns in the case. The BLANK command unconditionally blanks the field specified.

**WRITE\_CASE**

Writes the case to the output file SAMPL2.

**Auto-fixing Case By Case**

You could correct cleaning errors by writing an instruction to modify the data on a case by case basis correcting those columns that are in error. In the following example, we have written a procedure to correct only the cases with reported errors. The case to correct is identified by its ID number located in columns one through four.

**Example:**   ~DEFINE  
                  PROCEDURE={CORRECT:  
                  IF [1.4#0001] THEN  
                    TRANSFER [19.2] = 10  
                  ENDIF  
                  IF [1.4#0002] THEN  
                    MAKE\_DATA [18^2]  
                    BLANK [19.2]  
                  ENDIF  
                  IF [1.4#0003] THEN  
                    MAKE\_DATA [18^1]  
                  ENDIF  
                  IF [1.4#0005] THEN  
                    TRANSFER [15.3\$] = "THU"

```

ENDIF
IF [1.4#0006] THEN
    MAKE_DATA [18^2]
ENDIF
IF [1.4#0007] THEN
    TRANSFER [19.2] = 5
ENDIF
IF [1.4#0008] THEN
    MAKE_DATA [21^1/2]
ENDIF
IF [1.4#0009] THEN
    MAKE_DATA [21^8]
ENDIF
IF [1.4#0010] THEN
    TRANSFER [15.3$] = "WED"
ENDIF
}
~INPUT SAMPL,ALLOW_UPDATE
~EXECUTE PROCEDURE=CORRECT
~END
    
```

### 2.3.3 Subsequent Cleaning Runs

Once the errors have been corrected in the data, the cleaning procedure that produced the original error listing should be rerun to verify that no errors remain. In the following example, we will use the clean data file, SAMPL2, as the input file. If additional errors are found, SAMPL2 can be edited either interactively on a case-by-case basis, or by modifying the auto-fixing procedure and rerunning it.

**Example:** ~INPUT SAMPL2  
 ~DEFINE  
 PROCEDURE={showerr:

```
OK_COLUMNS [1.14]
CHECK [15.3#MON/TUE/WED/THU/FRI/SAT/SUN] "Not a
valid code"
CHECK [18^1/2] "Should be single 1,2"
IF [18^1] THEN
    CHECK [19.2*Z#1//10] "Should be a number 1-10"
ELSE
    CHECK [19.2^B] "Should be blank"
ENDIF
CHECK [21*P^1//7/(-)8] "Should be multi 1-7, or single 8"
CHECK_COLUMNS
}
~EXECUTE PROCEDURE=showerr
~END
```

If you saved your procedure in a DB file then it is not necessary to redefine it in subsequent runs. Open the DB file with the >USE\_DB command. Delete the lines that define the procedure or comment them out by inserting the command ~COMMENT right after ~DEFINE. Specifications falling between a ~COMMENT and the next tilde command will not be processed. See “2.3 CLEANING SPECIFICATIONS”, the >CREATE\_DB command.

## 2.4 CLEANING WITH SURVENT VARIABLES

The Survent interviewing program has been designed with features that makes data errors highly unlikely, but:

- The questionnaire could be changed after the study begins. Data collected prior to the change could be invalid.
- Errors could be made when open-ends are recoded.
- Someone reviewing the data in the CLEANER utility or in ~CLEANER could make changes to the wrong columns.

## PREPARING YOUR DATA

### 2.4 CLEANING WITH SURVENT VARIABLES

- Interviewer changes (done when a respondent changes a previous response using the ALTER keyword in Survent VIEW mode) could affect skip patterns in the questionnaire.

This section will explain how to use one of Survent's automatic spec generation options to create basic cleaning specifications. Survent specifications compiled with the `~PREPARE COMPILE CLEANING_SPECS` option produce a file of cleaning statements for each CAT, FLD, NUM, VAR, and TEX question. The resulting CLN file can be incorporated into a cleaning procedure to produce an error listing. The same procedure can then be used to clean the data case-by-case.

The Survent program also includes an option to view and/or alter data on a case by basis. Refer to your Survent manual under *4.1.4 VIEWING A PREVIOUS INTERVIEW* for more information.

If you know PREPARE syntax you could reproduce your paper and pencil questionnaire, and then generate data cleaning and/or tabulation specs with one of the compile options. See *4.7 USING PREPARE TO GENERATE Mentor SPECIFICATION FILES* for details on table building spec files.

You can produce a CLN file (and any other auxiliary file) at any time from a compiled questionnaire file (QFF).

**Example:**    `~QFF_FILE <studyname>`  
                  `~PREPARE MAKE_SPEC_FILES CLEANING_SPECS`

See *Appendix B: TILDE COMMANDS, ~PREPARE* for more information.

Each cleaning statement checks data validity against the PREPARE question specification (i.e., single/multi punched, exclusive response, range of responses, skip patterns, etc.). When these statements are executed interactively, a cleaning screen similar to what the interviewer saw is presented each time an error is found. You do not need to know any specific data modification commands to clean the data. New data is automatically checked before the procedure continues to the next

error. It is possible to suppress the cleaning screens, but then you will need to know basic data modification commands.

### A SAMPLE SURVENT QUESTIONNAIRE

When you compile Survent specifications in PREPARE, each question can be saved as a Mentor variable. These variables are stored in a CfMC DB file under the Survent label name. Just as your data file contains punches or characters in a certain column order, the DB file variables contain information that describe the contents of these columns. Each one includes the question description (text, question label, question type, column location, who should have answered this question) and answer descriptions (valid punches, ASCII responses, numbers, exception codes). The program-generated CLN file references these variables to check your data for errors.

Here is our sample paper and pencil questionnaire rewritten as PREPARE specifications. We have added a TEX type question to the original example in section 2.2.1.

**Example:** { NAME: 5.10  
NAME OF RESPONDENT:  
!VAR,,10,1 }  
{ DAY: 15.3  
DAY OF THE WEEK:  
!FLD  
MON  
TUE  
WED  
THU  
FRI  
SAT  
SUN }  
{ SIBLINGS: 18.1  
WHETHER RESPONDENT HAS SIBLINGS:

## PREPARING YOUR DATA

### 2.4 CLEANING WITH SURVENT VARIABLES

!CAT,,1

1 YES

2 NO }

{ NUMSIBS: 19.2

!IF SIBLINGS(1)

TOTAL NUMBER OF SIBLINGS:

!NUM,,,1-10 }

{ OTHERS: 21.1

OTHER MEMBERS IN IMMEDIATE FAMILY:

!CAT,,7

1 GRANDMOTHERS

2 GRANDFATHERS

3 GRANDCHILDREN

4 COUSINS

5 IN-LAWS

6 NEPHEWS

7 NIECES

(-) 8 NONE OF THE ABOVE }

{ DEMOG: 22.1

INTERVIEWER: Please gather the following information from the respondent and type the answers here.

HOME ADDRESS

RESPONDENT NAME

TELEPHONE NUMBER

!TEX }

### CLEANING SPECIFICATIONS GENERATED BY A COMPILE

The cleaning specifications below were produced by compiling the sample questionnaire above with ~PREPARE COMPILE CLEANING\_SPECS. The



resulting file is called SAMPL.CLN. We will reference it in our procedure along with the DB file also created when the sample questionnaire was compiled. SAMPL.DB contains all the information about the variables NAME, DAY, SIBLINGS, NUMSIBS, OTHERS, and DEMOG necessary for cleaning.

Responses to TEX type questions such as DEMOG are usually listed out with the CfMC LIST utility. You would then work from the LIST report to edit TEX responses. Refer to “2.4.1 Correcting Errors”, *Modifying TEX Question Responses* for commands to display and edit this data outside of a cleaning procedure.

**NOTE:** We are referencing file names using the DOS/UNIX naming convention studyname.extension. In MPE, the convention is studynameextension (e.g., SAMPLCLN). In addition, MPE has an eight character file name limit. To be consistent across platforms we are using the study code of SAMPL.

**Example:** NAME :  
    edit NAME  
DAY:  
    edit DAY  
SIBLINGS:  
    edit SIBLINGS  
NUMSIBS:  
    if (SIBLINGS(1))  
        edit NUMSIBS  
    endif  
OTHERS:  
    edit OTHERS  
DEMOG:  
    edit DEMOG

The CLN file consists of a cleaning statement for each CAT, FLD, NUM, TEX, and VAR in our sample questionnaire. The EDIT command verifies that the data matches the question definition, generates an error message when it does not, and then presents a Survent-like cleaning screen. The EDIT command performs the same function as the CHECK command (see the sample cleaning procedure in “2.3 CLEANING SPECIFICATIONS”). The CHECK command does not display a cleaning screen when errors are found. This is the main difference between the two commands.

EDIT only presents a cleaning screen when an error is found. The ALTER command presents the cleaning screen unconditionally. You might want to use this command for TEX questions.

### **ALTERNATE CLN FILE**

You have the option to produce a CLN file with CHECK commands and data variables when your PREPARE specifications are compiled. Use the command `~SPEC_RULES CLN_CHECK`. Specify this command before `~PREPARE COMPILE CLEANING_SPECS`. For example, instead of `EDIT OTHERS`, the cleaning statement would be `CHECK [1/21*P=7^1//7/(-)8]`. This would be useful if you want to edit the CLN file later on to add cleaning statements, and you want the references to be to the data locations, independent of the Survent variables which may themselves contain certain logic errors.

### **CUSTOM CLEANING SPECIFICATIONS**

Include your own cleaning specifications in your questionnaire specification file with the compiler command `{!Mentor_CLN}`. When Survent specifications are compiled with the `~PREPARE COMPILE CLEANING_SPECS` option, statements specified inside this command will be passed to the CLN file. There is no syntax checking by PREPARE. This option allows an experienced spec writer to include other cleaning commands or complex cleaning instructions in the questionnaire specifications. For example, you might need to check that the responses to three questions add up to 100%. Here is what the cleaning spec might look like.

**Example:**    `{ !Mentor_CLN`  
                  `IF SUM([15.2,17,19]) <> 100`  
                  `ERROR "Responses do not add up to 100%"`

```
ENDIF }
```

## CONDITION AND BRANCHING STATEMENTS

Cleaning statements are generated for every condition and branching statement in the questionnaire (!IF, SKIPTO, or !GOTO). The labels you see before each EDIT statement (e.g., NAME:) are generated by Mentor for every question. These serve as possible markers for branching caused by a SKIPTO or a !GOTO in the Survent questionnaire. They have no effect on a cleaning run unless there is a preceding GOTO statement that branches to that marker.

A PREPARE !IF statement is converted in the CLN file to a Mentor IF-THEN-ELSE-ENDIF block. The structure closely resembles the PREPARE syntax except that IF statements in Mentor must be closed with an ENDIF statement. For example the NUMSIBS question was only asked if the response to SIBLINGS was a one (Yes). The cleaning instruction below says to check NUMSIBS for valid responses only if the data for SIBLINGS contains a one response. (THEN is implied but never needs to be specifically stated.) The outside parentheses on Mentor IF statements are added by Mentor for clarity. They are not required for a single condition.

```
Example:  if (SIBLINGS (1))  
            edit NUMSIBS  
            endif
```

Branching in a Survent questionnaire is accomplished with either a SKIPTO or a !GOTO statement. Each one generates a GOTO statement in the CLN file.

Here is the original PREPARE specification rewritten with a SKIPTO instruction on the NO response to SIBLINGS instead of the !IF statement on NUMSIBS.

```
Example:  { SIBLINGS: 18.1  
            WHETHER RESPONDENT HAS SIBLINGS:  
            !CAT,,1  
            1 YES
```

```
(SKIPTO OTHERS) 2 NO }
```

SKIPTO OTHERS generates a GOTO cleaning command. GOTO tells Mentor to move forward to the place marked in the cleaning procedure and execute the command(s) specified there. A GOTO can conditionally move to different points in the procedure depending on the responses found in the data.

**Example:**

```
SIBLINGS :
    edit SIBLINGS
    goto (*,OTHERS) SIBLINGS
NUMSIBS:
    edit NUMSIBS
OTHERS:
    edit OTHERS
```

First SIBLINGS is checked for a valid response with the EDIT command. Then based on the response found in the data, the GOTO will execute in one of two ways. A Yes response means Mentor will continue processing, indicated by the asterisk (\*). A No response will cause Mentor to go to the label OTHERS and continue processing from there.

In the next example, the same branching is accomplished with a !GOTO statement. While it is unlikely that you would write this with a !GOTO, it illustrates what the cleaning specification would look like.

**Example:**

```
{ SIBLINGS : 18.1
  WHETHER RESPONDENT HAS SIBLINGS:
  !CAT,,1
  1 YES
  2 NO }
{ !IF SIBLINGS(2)
  !GOTO OTHERS }
```



The cleaning procedure will branch to the label OTHERS when the response to SIBLINGS is two.

```
SIBLINGS:
  edit SIBLINGS
QQ000.40:
  if (SIBLINGS(2))
    goto OTHERS
  endif
NUMSIBS:
  edit NUMSIBS
OTHERS:
  edit OTHERS
```

**VARIABLE MODIFIERS**

There are three optional modifiers that allow you to control how the data is checked by the EDIT command. These modifiers are also available to the CHECK and CLEAN commands. You can edit the program-generated CLN file to add any of these before the variable name.

**modifier** is optional, but if you include it:

- a minus sign (-) allows a blank in addition to other valid answers
- asterisk (\*) can be used to indicate that the field must be blank.
- a plus sign (+) means anything in addition to the valid punches is also okay.

**description** outlines what the command will be looking for.

<b>Variable Type</b>	<b>Modifier</b>	<b>Description</b>
<b>VAR</b>	(none)	Valid ASCII at least minimum length
	-	Valid answer or blank field
	*	Must be blank

## PREPARING YOUR DATA

### 2.4 CLEANING WITH SURVENT VARIABLES

<b>FLD</b>	(none)	Location(s) has one or more of the codes on the list (checking for an exclusive code and maximum responses).
	-	Valid answer or blank field
	*	Must be blank
<b>NUM</b>	(none)	Number within the range, or the exception number, or one of the exception codes
	-	Valid answer or blank field
	*	Must be blank
<b>CAT</b>	(none)	Has valid codes (checking for exclusive item and maximum responses) and no other punches in the location
	-	Valid answer or blank field
	*	Must be blank
<b>TEX</b>	(none)	Anything in addition is okay. Internal text pointers are okay
	*	Must be blank (i.e., not asked)
	-	Valid answer (good text pointer) or blank

As an example we have rewritten the statements that check the NUMSIBS question and the DEMOG question.

**Example:** NUMSIBS :  
    if (SIBLINGS(2)  
        edit \*NUMSIBS

```
endif  
DEMOG:  
edit -DEMOG
```

For NUMSIBS the IF condition was changed to say if the response to the SIBLINGS question was No (2) then the NUMSIBS question must be blank. The asterisk (\*) preceding the question label NUMSIBS means the data for this question must be blank. For DEMOG we added the minus sign (-) modifier to allow this question to be blank in addition to a valid response.

### GENERATING A LIST OF ERROR MESSAGES

Now that we have a CLN file we need to incorporate it into a cleaning procedure and then into a larger specification file to generate an error listing.

The commands used in the example below were covered in some detail in section “2.3 CLEANING SPECIFICATIONS”. We will only cover commands or concepts not previously discussed. Please refer back to that section for more information.

**Example:**   ~DEFINE  
                  PROCEDURE={CLEANIT:  
                    OK\_COLUMNS [1.4]  
                    &SAMPL^CLN<sup>1</sup>  
                    CHECK\_COLUMNS  
                    WRITE\_CASE  
                  }

The cleaning procedure looks very similar to the example in section 2.3. But instead of writing a CHECK statement and data description for each variable, we

---

1. The caret (^) allows any CfMC-supported operating system to read this file, i.e., as SAMPL.CLN (by DOS or UNIX) or SAMPLCLN (by MPE). Notice that &filename has been indented along with the procedure commands. Normal CfMC processing requires that the ampersand (&) be in column one of the spec file.

## PREPARING YOUR DATA

### 2.4 CLEANING WITH SURVENT VARIABLES

read in the program-generated CLN file. The ampersand (&) preceding the file name tells Mentor to read the file referenced from the current directory or group unless otherwise specified. This procedure also writes out each case to a new data file ensuring that the original data file remains intact.

Here is the above cleaning procedure incorporated into the complete specification file. The DB file must be opened first since the variables referenced in the CLN file are stored here. ~SET ERROR\_REVIEW suppresses the cleaning screens displayed by default when an error is found. See 2.3.2 *CORRECTING ERRORS, Auto-Fixing The Data* for an explanation of ~OUTPUT and WRITE\_CASE.

```
Example: >ALLOW_INDENT
           >USE_DB SAMPL
           ~DEFINE
           PROCEDURE={CLEANIT:
               OK_COLUMNS [1.4]
               &SAMPL^CLN
               CHECK_COLUMNS
               WRITE_CASE
           }
           ~INPUT SAMPL
           ~OUTPUT SAMPL2
           ~SET ERROR_REVIEW
           ~EXECUTE PROCEDURE=CLEANIT
           ~END
```

If the above file is named CLEAN.SPX you would run it by entering:

```
Mentor CLEAN.SPX -CLEAN.LFL (DOS/UNIX)
```

```
RUN Mentor.CGO.CFMC;INFO="CLEANSPX CLEANLFL" (MPE)
```



The list of errors would go to the output file, CLEAN.LFL. Below is an abbreviated error listing that could be generated by this specification file. See 2.3 CLEANING SPECIFICATIONS, Generating a List of Error Messages for more information.

ID: 0001

error 3: NUMSIBS [19.2#] has error: number in range is required  
NUMSIBS[19.2#]="12"

ID: 0002

error 2: SIBLINGS [18^] has error: a valid answer is required  
SIBLINGS[18^]=" "

ID: 0003

error 2: SIBLINGS [18^] has error: extra punches  
SIBLINGS[18^]="3"

ID 0004:

error 1: NAME [5.10\$] has error: string too short  
NAME[5.10\$]=" "

ID 0005:

error 5: DAY [15.3#] has error: a valid answer is required  
DAY[15.3#]=" "

.  
. .  
.

5 errors in 5 cases

error 1: 1 NAME [5.10\$] has error

error 2: 2 DAY [18^] has error

error 3: 1 NUMSIBS [19.2#] has error

error 5: 1 DAY [15.3#] has error

### PROGRAM-GENERATED ERROR MESSAGES FOR SURVENT QUESTIONS

Here are the automatic error messages generated by Mentor for different types of errors found.

Question Type Errors	Error Message
CAT or FLD question is blank when an answer should be present.	A valid answer is required
Single response CAT question has an invalid punch with a valid punch.	too many answers
Single response CAT question has more than one punch per column.	extra punches
Single response FLD question has an invalid response.	a valid answer is required
Multi-response FLD question has an invalid response, or leading or embedded blanks.	invalid codes or blank field
Multi-response FLD contains duplicate responses	duplicate codes
Multi-response CAT or FLD question has an exclusive response with another punch or response.	exclusive code violation
VAR question does not contain the minimum number of typed characters.	string too short
NUM question is blank when an answer should be present.	an answer is required
NUM question has an invalid answer (out of range or invalid exception code).	a valid answer is required
TEX question has a bad internal pointer or is blank when an answer is required.	an answer required
TEX question is not blank (skipped) and it should be.	must be blank

You can specify your own error message inside "quotes" after the EDIT command.

**Example:** EDIT SIBLINGS "Should be single punched 1 or  
2"

### 2.4.1 Correcting Errors

You have several choices for cleaning your data once you have an error listing.

- Clean each case with one of the data modification commands described in “2.5.1 Quick Reference: Cleaning Commands And Examples”.
- Clean using the procedure we wrote to list out the errors. There are three advantages to this approach: you do not need to know data modification commands; EDIT only allows access to columns with errors; changes to the data are checked for errors before you move to the next case.
- Clean the data with an auto-fixing procedure.

#### USING SURVENT-TYPE CLEANING SCREENS

The EDIT command in the program-generated CLN file presents a Survent-like screen for data cleaning. It is similar to the screen the interviewer would see during actual interviewing with the C-Survent software. The top of the screen displays the text of the question including any recode table. The bottom of the screen displays the question name, the data location and the current data. If you enter an invalid response an appropriate error message displays, just as it would for an interviewer.

Here is a sample spec file called SCRNEEDIT.SPX. This file contains all the commands needed to clean our sample data file using the Survent screens.

**Example:** >ALLOW\_INDENT  
>USE\_DB SAMPL  
>PRINT\_FILE LOGIT, ECHO  
~DEFINE  
    PROCEDURE={CLEANIT:  
        OK\_COLUMNS [1.4]  
        &SAMPL^CLN  
    }  
~SET LOGGING  
~INPUT SAMPL2,ALLOW\_UPDATE

## PREPARING YOUR DATA

### 2.4 CLEANING WITH SURVENT VARIABLES

~EXECUTE PROCEDURE=CLEANIT

The ~SET ERROR\_REVIEW statement was omitted. For interactive cleaning we do not want to suppress the Survent cleaning screens. The data file we wrote out with the error listing procedure is opened in ALLOW\_UPDATE mode to save all of our changes.

The CHECK\_COLUMNS command does not present a cleaning screen so it has been omitted from this procedure. The columns reported by CHECK\_COLUMNS in the error listing should be examined interactively, and either modified or blanked using the BLANK command.

~EXECUTE PROCEDURE=CLEANIT executes the procedure starting at the beginning of the data file. The error lists to the screen and then the cleaning screen for that question is presented. New responses are checked against the question's parameters just as they would be during an actual interview. A Survent error message displays when an invalid response is entered. The error summary is displayed when Mentor reaches the end of the data file (refer to the sample error list and summary earlier in this section).

There is no ~END command in this file, but you could put one in, otherwise you will provide it from the keyboard when you want to exit Mentor.

The command line to run this spec file would look like this:

```
Mentor &SCRNEDIT.SPX CON(DOS)
```

```
Mentor "&SCRNEDIT.SPX" CON(UNIX)
```

```
RUN Mentor.CGO.CFMC;INFO="&SCRNEDIT CON"(MPE XL)
```

In this sample, the first case has an error in the NUMSIBS. The procedure will stop there and display this error message:

```
ID: 0001
```

```
error 3: NUMSIBS [19.2#] has error: number in range is required
```

```
NUMSIBS[19.2#]="12"
```

```
ID: 0001
```

```
...and now you can modify NUMSIBS[19.2#]="12"
```

Press <return> to continue

This shows that case ID 0001 has an answer (12) in columns 19 and 20 which is outside of the allowed numeric range of 1-10 for this question. When you press <Return> (or Enter) a screen similar to the one below will display.

TOTAL NUMBER OF SIBLINGS:

-->

enter the new values or RES/BLK/TERM (NUMSIBS[ 1/19.2=12 ])

RES/BLK/TERM are commands you can enter at the prompt instead of new data.

### **RES**

Restores the original data and prompts the user to continue or to begin entering commands from the console. Original refers to the state of the current case when the current command accessed it. If you are using a procedure as we are here then RES will cause the procedure to go on to the next error.

### **BLK**

Blanks the field and continues to the next error.

### **TERM**

Terminates the cleaning procedure leaving you at the CLeaNer--> prompt. Type some other command or ~END to exit Mentor.

We will correct the data by entering a new value of 10 at the arrow prompt. The new value is automatically checked against the question's structure. If it invalid then an error message will print. You will not be allowed off the screen until either the correct value is entered or one of the allowed commands.

When Mentor reaches the end of the data file you will see a message similar to this:

```
found the EOF_DATA without finding CLEANIT
```

```
CLeaNer-->
```

Enter ~END to exit or another command.

## **MODIFYING TEX QUESTION RESPONSES**

As we explained earlier, you can use the EDIT cleaning command to check TEX questions, but a Survent screen is only presented when an error is found. It is more

## PREPARING YOUR DATA

### 2.4 CLEANING WITH SURVENT VARIABLES

likely that you will be correcting typographical errors from the report of verbatims generated by the LIST utility. In that case, you will want to clean TEX responses separately from your regular cleaning run.

You must indicate the start location of the text data in the data file. By default, it is the first column of the next record after the last data column used, or the column specified on the TEXT\_START= ~PREPARE header statement option. There are two ways to do this:

- 1 Open the QFF file (~QFF\_FILE name) before opening the data file. Mentor will determine the text location from the compiled questionnaire file.
- 2 Use the TEXT\_LOCATION= option on either the ~INPUT or the ~CLEANER FILE statement.

Here are the TEX question modifying command options.

#### **MODIFY\_TEXT <location> or <varname>**

Displays the current response and lets you enter a new one. Specify either the location of the TEX question or the variable name (you must open the DB file first with >USE\_DB).

#### **ERASE\_TEXT <location> or <varname>**

Erases the response and the internal text pointers for this question. Use this command instead of BLANK for TEX questions.

#### **ALTER varname**

Presents the question in a Survent cleaning screen unconditionally. Unlike EDIT, the screen is presented whether or not an error is found.

## **AUTO-FIXING THE DATA**

Refer to “2.3.2 Correcting Errors”, *Auto-Fixing The Data* for an explanation of auto-fixing and the commands used in this procedure. The following is an example of cleaning the data in batch mode.

**Example:** >USE\_DB SAMPL  
~DEFINE  
PROCEDURE={AUTOFIX:  
CLEAN DAY

```
CLEAN SIBLINGS
  IF [NUMSIBS#10//99] THEN
    TANSFER NUMSIBS= 10
  ENDIF
CLEAN -NUMSIBS
  IF [NUMSIBS#" "] THEN
    MAKE_DATA SIBLINGS(2)
  ELSE
    MAKE_DATA SIBLINGS(1)
  ENDIF
  IF OTHERS(8) AND OTHERS(1-7) THEN
    MAKE_DATA -OTHERS(8)
  ENDIF
CLEAN OTHERS
WRITE_CASE
}
~SET ERROR_REVIEW
~INPUT SAMPL
~OUTPUT SAMPL2
~EXECUTE PROCEDURE=AUTOFIX
~END
```

The CLEAN command tells Mentor to examine each case using the data descriptions generated from the PREPARE questionnaire specifications, and to blank those data columns whenever they deviate from that description. Where needed, specific conditions are given for changing the data.

The CLEAN command can have an error message associated with it. This message is helpful in the error summary. It provides a tally of how many times this error was found causing the data to be blanked.

## PREPARING YOUR DATA

### 2.4 CLEANING WITH SURVENT VARIABLES

You could correct cleaning errors by writing an instruction to modify the data on a case by case basis. This procedure corrects the NUMSIBS and DAY variables using the TRANSFER command, and the SIBLINGS and OTHERS variables using the MAKE\_DATA command. If a column should be blank and it is not, it is corrected with the BLANK command.

```
Example:  >USE_DB SAMPL
            ~DEFINE
            PROCEDURE={ CORRECT:
            IF [1.4#0001] THEN
            TRANSFER NUMSIBS = 10
            ENDIF
            IF [1.4#0002] THEN
            MAKE_DATA SIBLINGS(2)
            BLANK NUMSIBS
            ENDIF
            IF [1.4#0003] THEN
            MAKE_DATA SIBLINGS(1)
            ENDIF
            IF [1.4#0005] THEN
            TRANSFER [DAY$] = "THU"
            ENDIF
            IF [1.4#0006] THEN
            MAKE_DATA SIBLINGS(2)
            ENDIF
            IF [1.4#0007] THEN
            TRANSFER NUMSIBS = 5
            ENDIF
            IF [1.4#0008] THEN
            MAKE_DATA OTHERS(1,2)
            ENDIF
```



```
IF [1.4#0009] THEN
    MAKE_DATA -OTHERS(8)
ENDIF
IF [1.4#0010] THEN
    TRANSFER [DAY$] = "WED"
ENDIF
}
~INPUT SAMPL2, ALLOW_UPDATE
~EXECUTE PROCEDURE=CORRECT
~END
```

### 2.4.2 Subsequent Cleaning Runs

Once the errors have been corrected in the data, the cleaning procedure should be rerun to ensure that no errors remain. In the following example, we will use the corrected file, SAMPL2, as the input file. If additional errors are found, SAMPL2 can be corrected using one of the methods described in the previous section.

```
Example: >ALLOW_INDENT
>USE_DB SAMPL
~DEFINE
PROCEDURE={CLEANIT:
OK_COLUMNS [1.4]
&SAMPL^CLN
CHECK_COLUMNS
}
~SET ERROR_REVIEW
~INPUT SAMPL2
~EXECUTE PROCEDURE=CLEANIT
~END
```

## 2.5 REFERENCE

### 2.5.1 Quick Reference: Cleaning Commands And Examples

This section is a list of the more commonly used cleaning commands and a short explanation of each of them. With these commands, you can access specific cases in your data file and correct them.

**NOTE:** Before starting the cleaning process, make a backup copy of your data file. You always want to have a copy of your “untouched” data.

Start Mentor by entering:

```
CLEANIT
```

This utility starts Mentor, starts the cleaner block, and opens a log file called clean.log. CLEANIT records your commands in a log file named clean.log. If you do not want to append to an existing log file, you will have to rename clean.log before starting CLEANIT again.

Or, you can start Mentor by entering:

```
Mentor CON CON (DOS/UNIX)
```

```
RUN Mentor.CGO.CFMC;INFO="CON CON" (MPE XL)
```

CON is short for console, this tells Mentor to expect input from the keyboard, and to send messages out to the screen. You could specify a file name to have Mentor keep a record of the messages to a List file, for example:

```
Mentor CON SAVEIT.LFL,ECHO (DOS/UNIX)
```

```
RUN Mentor.CGO.CFMC;INFO="CON SAVEIT,ECHO" (MPE XL)
```

ECHO tells Mentor to send messages to the screen as well as the List file. See *Utilities, Appendix D: CfMC Conventions* for more information on List files.

After Mentor starts, you will prompted to enter a command. Type ~CLEANER to start the cleaner block. The prompt will change to "CLeaNer-->" and you can then start cleaning your data. To enter the Mentor CLEANER block, type ~CLEANER:

```
(Enter command)-->~cleaner
      CLeaNer-->
```

A list of common CLEANER commands and short description of them follows. For a complete list of ~CLEANER commands and their syntax, see *Mentor, Appendix B: Tilde Commands*.

Action	Command	Description (command abbreviation)
Help	>HELP	Display on-line help for ~CLEANER.
Open file	FILE <filename> options	Opens a CfMC System file, for modification, with other options if needed. Changes are written to the file when you move to the next case. Related Commands: ~INPUT; ~OUTPUT, ~CLEANER WRITE_CASE.
Get Variables	USE_DB <studyname>	Opens the DB file containing variables from a Survent questionnaire.
Get a case	NEXT "<caseid>"/FIRST/LAST	Gets case with ID of <caseid>. Keywords FIRST and LAST will get first case or last case, respectively. (N)
Move	NEXT #, +#	Goes to record number#, or in file forward (+) the number of cases specified. (N) Maximum -# is 9 cases.
Set command	~SET SEQUENTIAL READ	Tells Mentor to deal with cases in case order.
Find case	BACKUP #	Goes backwards the number of cases specified.

**FIND <variable description>**

Goes to the next case or procedure name that satisfies either the variable description or procedure. The FIND command by itself re-executes the last description specified. (F)

**FIND\_FLAGGED**

Finds the next case with the error flag set on dirty cases that were written (with a cleaning procedure) to a ~OUTPUT file, to the ~INPUT file opened in ALLOW\_UPDATE mode, or opened with the FILE command. (FF)

**HUNT <variable description>**

Like FIND, but starts at the top of the data file. (H)

Use FIND or HUNT to locate the case that contains data defined in the variable.

**Example:**

```
DEFINE x[1.4#0023]
HUNT x
```

This would find the case that had 0023 in columns one through four.

Action	Command	Description (command abbreviation)
Display	DA <col.wid>, <variable> or *	Displays data as ASCII text by specifying the data columns, the variable name, or * for the entire case. (D)
	DB/DC <col.wid> or <variable>	Displays data in binary or column (punch) mode as described for DA command above, but does not do *.
	DT <col>, <text variable> or *	Displays text question data (collected in Survent with a TEX type question) by referencing the text pointer column,

the Survent question label, or \* for all text questions.

**SHOW** <varname>, \*, \*B

Displays the entire question including text and response list. \* displays the entire case in card image. \*B displays the entire case in column binary (punch) format.

Refer to your UTILITIES manual under *APPENDIX C: GLOSSARY OF CFMC TERMS* for definitions of ASCII, binary, text variables, or other unfamiliar terms.

Modify	<b>BLANK</b> [col.wid] or <variable>	Blanks the data columns. ( <b>B</b> )
	<b>ERASE_TEXT</b> <col.wid> or <variable>	Erases the text area and the internal text pointers for TEX type questions collected in Survent. ( <b>ERASETEX</b> )
	<b>MA</b> <col.wid> or <variable>	Modifies data in ASCII format.
	<b>MB/MC</b> <col.wid> or <variable>	Modifies data in binary or column (punch) format.
	<b>MT</b> <col> or <text variable>	Modifies data from a text question (collected in Survent with a TEX type question) with the program's editor. You must specify the option TEXT_LOCATION= on either the FILE or ~INPUT command.

MA, MB, MC, MT are interactive data modification commands only.

Action	Command	Description (command abbreviation)
	<b>MAKE_DATA</b> +/- [loc^punch] or variable	Modifies punch data. ( <b>MKDATA</b> ) Example: MKDATA + [18^1]
	<b>RESTORE</b>	Restores the data in this case (i.e., drops current changes), but only if you have NOT moved to another case.

**TRANSFER**

Modifies data. **(T)**

**Examples:**

numeric data:           T [19.2] = 5  
 string/ASCII data:    T [15.3\$] = "THU"  
 punch data:            T [11.2\$P] = "1,2,3"  
 sum the values:        T [20.3] = SUM( [5,6,7] )  
 add numeric data:     T [10.2] = [5] + [20]

Delete a case   **ASSIGN\_DELETE\_FLAG**

Flags this case for deletion.<sup>1</sup>  
**(DELETE)**

**UNDELETE**

Removes the delete case flag.

Redo            **>EDIT\_PREVIOUS**

Displays last line typed at the program prompt in the program's editor for modification. Type ESC to exit and re-execute the command on the current case.  
**(EP)**

**FIND\_FLAG\_REDO**

Finds the next case with an error flag and re-executes the last command line. **(FFR)**

**FIND\_REDO**

Finds the next case that satisfies the variable description and re-executes the last command line.  
**(FR)**

**NEXT\_REDO**

Goes to the next case and re-executes the last command line.  
**(NR)**

---

1. Flagged cases are not read (i.e., they are ignored) by a procedure that modifies data, WRITE\_CASE, ~COPY, ~INPUT, etc. unless you instruct Mentor to use deleted cases (USE\_DELETED and the System constant DELETED\_CASE\_FLAG).

**REDO**

Re-executes the last command line.  
**(R)**

You can specify commands on the same line if you separate them with semicolons. Pressing <Enter> will execute all of the commands specified on that line. >EDIT\_PREVIOUS will display the entire line in the editor for modification. Any of the redo commands will re-execute all of the commands.

**Example:** DA 5.3;DA 10.3;DA 20.4

Action	Command	Description (command abbreviation)
Define	<b>DEFINE</b> name[data description]	Defines a data variable and assigns a name for future reference with display, find, or modify commands. <b>(DEF)</b>

You can attach a name directly to any DISPLAY, FIND, or MODIFY command and then refer to the name the next time you use the command.

**Example:**

DA x[15.3]  
DA x

Log	<b>SET LOGGING</b>	Records interaction with Mentor either from the keyboard or a &file, and Mentor's response to each action. The log goes to the list file providing a record of all data changes. If the ECHO option is specified after the list file name, then program messages also display to the screen.
Get info	<b>&gt;STATUS INPUT</b>	Displays information on the data file and the current case.

**NOTE:** Refer to your *Utilities* manual *Appendix A: META COMMANDS* for other >STATUS options.

Exit	<b>~END</b>	Quits program and saves data changes.
Move to new tilde block	<b>~command</b>	Moves to another ~tilde block.

### EXAMPLE CHECK STATEMENTS

SINGLE RESPONSE PUNCH VARIABLE (Survent CAT TYPE)

CHECK [11\*P=1^1//5/Y] (\*P=1 is the default and does not need to be specified)

CHECK [11^1//5/Y]

MULTIPLE RESPONSE PUNCH VARIABLE WITH Y EXCLUSIVE

CHECK [12\*P^1//5/(-)Y] or CHECK [12^1-5/Y]

CHECK [12\*P=3^1//5/(-)Y] (\*P=3 says this may have up to three valid responses)

SINGLE RESPONSE PUNCH ACROSS MULTIPLE COLUMNS

CHECK [13.2^1//20/24] (\*P=1 is implied)

MULTIPLE RESPONSE PUNCH VARIABLE ACROSS MULTIPLE COLUMNS

CHECK [15.2\*P^1//20/(-)24] or CHECK [15.2^1-20/24]

CHECK [15.2\*P=10^1//20/(-)24]

SINGLE RESPONSE ASCII (STRING) VARIABLE (Survent FLD TYPE)

CHECK [17.2\*Z#1//10/99] (\*Z means the field must be zero-filled)

CHECK [3/20.2#CA/RI/MA/NY/OR/CT]

SINGLE RESPONSE ASCII (STRING) VARIABLE (Survent VAR TYPE)

CHECK [5.10\*P=5\$]

\*P=n specifies the ending column to check (starting from the right-most column in the field for a non-blank character. This can be a number 1-127. From a Survent VAR question this is the minimum number of typed (i.e, non-blank, including spaces) characters requirement for that question.



In this example, Mentor checks the field starting in column 14 through column 5 for a non-blank character. Columns five through nine could be blank. If you needed to check for a character anywhere in the field then you would specify \*P=1.

#### MULTIPLE RESPONSE ASCII VARIABLE WITH 99 EXCLUSIVE

Duplicates or skipped blank fields are an error.

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

#### MULTIPLE RESPONSE ASCII VARIABLE WITH 99 EXCLUSIVE

Duplicates are allowed, but skipped blank fields are an error.

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

Multiple Location Variables Without \*F or \*L

#### SINGLE PUNCH/SINGLE COLUMN

```
CHECK [41, . . . , 47, 51, 52^1//5/Y] (*P=1 is implied)
```

The example above simplifies this syntax:

```
>REPEAT $A=41,....,47,51,52
```

```
CHECK [$A^1//5/Y]
```

```
>END_REPEAT
```

#### MULTIPLE PUNCH/SINGLE COLUMN

```
CHECK [53,56*P^1//5/(-)Y] or CHECK [53,56^1-5/Y]
```

```
CHECK [53,56*P=5^1//5/(-)Y]
```

The examples above simplify this syntax:

```
>REPEAT $A=53,56
```

```
CHECK [$A^1-5/Y]
```

```
>END_REPEAT
```

#### MULTIPLE PUNCH/MULTIPLE COLUMN

```
CHECK [61.2,63.2*P^1//20/(-)24] or CHECK [61.2,63.2^1-20/24]
```

```
CHECK [61.2,63.2*P=20^1//20/(-)24]
```

The examples above simplify this syntax:

```
>REPEAT $A=61,63
```

```
CHECK [$A.2^1-20/24]
```

## PREPARING YOUR DATA

### 2.5 REFERENCE

>END\_REPEAT

SINGLE ASCII/MULTIPLE COLUMNS (Survent FLD TYPE)

CHECK [65.2,67,69\*Z#1//10/99] or CHECK [65.2,67,69\*Z#1-10,99]

The examples above simplify this syntax:

>REPEAT \$A=65,67,69

CHECK [\$A.2\*Z#1//10/99]

>END\_REPEAT

SINGLE NUMERIC/MULTIPLE COLUMNS (Survent NUM TYPE)

CHECK [71.3,74#1//100/DK/NA/RF] or

CHECK [71.3,74#1-100,DK,NA,RF] or

CHECK [71.3,74\*Ranges=1-100,,DK,NA,RF] (~PREPARE COMPILE  
CLEANING\_SPECS)

The examples above simplify this syntax:

>REPEAT \$A=71,74

CHECK [\$A.3#1-100,DK,NA,RF]

>END\_REPEAT

Fields Must Be Blank After The 'None/No More' Code

Assume four consecutive two column fields with codes 01-17,98, and 99, where 98 is No More and 99 is Refused. First clean with a typical CHECK statement.

**Example:** CHECK [1.2, . . . , 7.2\*F#1//17/98/(-)99]

There are three possible errors that could still exist with the 98 (No More) code.

- 1 98 is in the first position. The specification to clean would look like this:  
IF [1.2#98] THEN  
    ERROR "1.2 is 1st position and has no more code"  
ENDIF
- 2 98 is in the list, but is not the last code in the field.

This specification returns an error if the number of total answers (NUMITEMS) does not equal the category position (SUBSCRIPT) of the 98 code:

```
IF NUMITEMS([1.2,...,7.2^NB]) <> SUBSCRIPT([1.2,...,7.2#98])
    THEN ERROR "98 is not last code in the field 1-8:" [1.8$]
ENDIF
```

- 3** The code list terminates with a code other than 98 or 99. This is only an error if 98 is always coded as the last mention (except for someone who has the maximum number of responses).

This specification returns an error if there is a blank and there is not a 98 or 99 code:

```
IF [1.2,...,7.2^B] AND NOT([1.2,...,7.2#98,99]) THEN
    ERROR "1-8 has a blank and not a 98 code:" [1.8$]
ENDIF
```

### 2.5.2 Sending Error Messages To A Print File

You can create a separate report of the errors in your system file by opening a print file. This is an ASCII file and it will exclude most of the processing messages generated by Mentor when a specification file is run.

**Example:**

```
>USE_DB SAMPL
>PRINT_FILE DIRTY
~INPUT SAMPL
~OUTPUT SAMPL2
~SET ERROR_REVIEW, ERRORS_TO_PRINT_FILE
~EXECUTE PROCEDURE=CLEANIT
~END
```

>PRINT\_FILE says to create a file called DIRTY (the default extension is PRT). The ~SET option ERRORS\_TO\_PRINT\_FILE says to send the error messages and the error summary to the print file. This listing will look similar to the error list

## PREPARING YOUR DATA

### 2.5 REFERENCE

shown in “2.4 CLEANING WITH SURVENT VARIABLES”, *Generating A List Of Error Messages*.

**NOTE:** You can specify your own extension for the print file, e.g.,  
>PRINT\_FILE DIRTY^LFL,USER. Refer to *Appendix A: META COMMANDS* in your *UTILITIES* manual for more information.

If the above file is named CLEAN.SPX, you would run it by entering:

```
Mentor CLEAN.SPX -CLEAN.LFL(DOS/UNIX)
RUN Mentor.CGO.CFMC;INFO="CLEANSPX CLEANLFL"(MPE)
```

The list of errors would go to the output file, CLEAN.LFL. Below is an abbreviated error listing that could be generated by this specification file. See “2.3 CLEANING SPECIFICATIONS”, *Generating a List of Error Messages* for more information.

ID: 0001

error 3:NUMSIBS [19.2#] has error: number in range is required  
NUMSIBS[19.2#]="12"

ID: 0002

error 2: SIBLINGS [18^] has error: a valid answer is required  
SIBLINGS[18^]=" "

ID: 0003

error 2: SIBLINGS [18^] has error: extra punches  
SIBLINGS[18^]="3"

ID 0004:

error 1: NAME [5.10\$] has error: string too short  
NAME[5.10\$]=" "

ID 0005:

error 5: DAY [15.3#] has error: a valid answer is required  
DAY[15.3#]=" "

.  
. .  
. .

5 errors in 5 cases

```
error      1:      1 NAME [5.10$] has error
error      2:      2 DAY [18^] has error
error      3:      1 NUMSIBS [19.2#] has error
error      5:      1 DAY [15.3#] has error
```

### 2.5.3 Specifying More Than One Command Per Line

When you write cleaning procedures it is often more convenient to specify more than one command on a line, especially conditionals. Preceding cleaning commands with a dollar sign (\$) helps Mentor to interpret a specification line when it can be ambiguous whether or not the command is followed by a variable name or another command. You do not need a \$ on the first command on the line.

**Example:** `IF [5^1] WRITE_CASE $ELSE SAY "Case not written" $ENDIF`

In the above example the commands ELSE and ENDIF could be interpreted as variable names. Remember, preceding commands with a dollar sign (\$) ensures that Mentor will always be able to distinguish between the two.

### 2.5.4 Additional Commands

The commands listed below are related to data cleaning and generation operations. Refer to *Appendix B: TILDE COMMANDS* under the tilde commands listed below for information.

**~CLEANER Commands:**

ASSIGN_DELETE_FLAG	Flags the case for deletion.
ASSIGN_ERROR_FLAG	Turns on the error flag for this case.
CLEAR_ERROR_FLAG	Clears the error flag.
COPY	Copies data from one field to another.
FIXUP	Blanks the data location when it does not fit the data description given. Mentor does not generate any messages when a data error is found.
MODIFY	Modifies the data. See also COPY and TRANSFER.

## PREPARING YOUR DATA

### 2.5 REFERENCE

TERMINAL_SAY	Prints a message to the screen only (use in a procedure).
UNDELETE	Removes the delete flag.
UPSHIFT	Changes all alpha characters in the specified location to upper case.
<b>~SET Options:</b>	
CLEANER_DEFINITION=	Prepends this string to every ~CLEANER command.
ERROR_LIMIT=	Maximum errors allowed in a cleaning run.
ERROR_STOP=	Stops compiling a procedure when this number of syntax errors is reached.
MAXIMUM_PAST_CASES	Sets the maximum number of cases that Mentor can backup to with ~CLEANER NEXT -#.
PROCEDURE_DUMP	Echoes Mentor's internal processing messages as it compiles and executes a procedure.
PRODUCTION_MODE	Updates a case without confirmation. This is the default for the ~CLEANER FILE command.
TESTING_MODE	Does not allow any changes to the data file unless it is opened with ALLOW_UPDATE. Overrides PRODUCTION_MODE on the FILE command.
TRAINING_MODE	Does not allow changes to the data. Use this option for training purposes.

Refer to “9.3.1 System Constants” and “9.3.2 Functions” for other commands that you can use.

# REFORMATTING YOUR DATA

## INTRODUCTION

This chapter describes the most commonly used data manipulation statements and shows how to use arithmetic calculations to generate numeric data for use in tables.

### 3.1 WHY REFORMAT DATA?

There are several different ways you can change how data is organized in a data file. For example, you may need data in a study that is the combination of two variables (e.g., one variable filtered by the other).

If you want to change the the data file type (such as changing a CfMC data file to an ASCII data file), use the CfMC utility COPYFILE or MAKECASE. If you want to recover a corrupted data file, use the RAWCOPY utility. See the *Utilities* manual for a complete description of COPYFILE, MAKECASE and RAWCOPY.

### THE OVERALL STRUCTURE

You can use data manipulation statements in either the ~CLEANER or ~DEFINE block of the Mentor program. In the ~CLEANER block, data modification takes place immediately and only on the case you have in hand. With ~CLEANER, you can either use each command by itself or execute a procedure. In the ~DEFINE block, you must create procedures which are then executed on all data cases when called in the ~EXECUTE block. We will be concentrating primarily on commands that you use in the ~DEFINE block. For an example of using a procedure in the ~CLEANER block, see “3.1.8 Data Manipulation in the ~CLEANER Block”.

There are five basic commands that you can use for data manipulation:

- COPY

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

- MAKE\_DATA
- PRINT\_TO\_DATA
- TRANSFER
- MODIFY

The data variables used by these commands can be defined as punches, strings (literals), or numeric. They can be variables from a DB file that was created in EZWriter/PREPARE or they can be created with the PREPARE= instruction (see section “*Creating Variables*” under “3.1.5 Data Manipulation for Predefined Variables”).

Two other useful commands are BLANK and SAY. BLANK allows you to blank the location or data variable referenced (see “3.1.2 Blanking Data”), and SAY allows you to display the data associated with any location or data variable (see “3.1.3 Printing Text and Data Fields”).

## RULES FOR MANIPULATING DATA

### *Rules Applying to Punch Type Variables*

Defining a punch type variable will allow that data location to receive any of the valid punches (1-9,0,X,Y). Because no valid punches are defined with the variable you can add, subtract, and change punches as you wish.

[19.2\$P]      where 19.2 is the actual data location and each column will allow all punches (1-9,0,X,Y).

[city\$P]      where CITY can already exist, and enclosing it in square brackets with the \$P redefines the data location associated with city as a punch type variable.

### *Rules Applying to String Type Variables*



Defining a string type variable will allow any valid ASCII string to be put into that data location. The string will be left-justified and if too long, it will be truncated.

[21.2\$] where 21.2 is the actual data location and each column accepts only a single ASCII character.

[name\$] where NAME can already exist, and enclosing it in square brackets with the \$ redefines that data location as a string type variable.

### ***Rules Applying to Numeric Type Variables***

Defining a numeric type variable means the only valid data that will be recognized in that location is a number. If some other data exists in that location, the program will return the keyword MISSING.

[23.2] where 23.2 is the actual data location and contains only valid numeric values.

[gender] where GENDER can already exist and enclosing the variable in square brackets redefines it for the program as a numeric variable.

Any variable that exists in a DB file which originated in or PREPARE variables uses the same syntax. The program inherently knows the type of variable (CAT, FLD, NUM, VAR, TEX) from its definition and expects certain parameters to be upheld.

CAT types. There are certain punch codes that have been defined and are acceptable for the data location referenced. Any attempt to add/remove a code that does not match the definition will result in an error.

FLD types. These have certain acceptable ASCII codes, just like CAT types. Data modification must be done through redefining the FLD variable as a string to put in

the new code. If the new code generated does not match any of the acceptable codes in the original variable then that code will not appear when displaying the data location in its original format.

NUM types. The numerical range and exception codes have been defined and any number or code not matching the definition will not be a recognized number in that location.

VAR types. The length of the response is defined and any generation into the referenced data location that exceeds the length will result in an error. Anything shorter than the default will be left-justified.

TEX types. You must specify the location of text data when you open the data file, e.g., ~INPUT myfile, TEXT\_LOCATION= 3/10.

### **3.1.2 Blanking Data**

The BLANK command will blank specific data locations or will blank data referenced by a pre-defined variable. There may be occasions in data manipulation where it will be desirable to blank out existing data from a specific location before beginning any new manipulations.

The command BLANK [6.4] blanks out all data in columns 6-9. If a pre-defined variable called TIMES exists in columns 4-5, then BLANK [TIMES] would blank columns 4-5. The command BLANK [6-80] blanks all data from column 6 through 80. If CITY was a pre-defined CAT type variable with responses of 1, 2, 3, or 4, BLANK CITY would clear responses associated with the CITY variable. A punch 9 in the data location of the CITY variable would not be cleared with BLANK CITY. To clear the data location associated with CITY, use BLANK [CITY].

### **3.1.3 Printing Text and Data Fields**

The SAY command prints any text enclosed in quotes that follows the command, or prints the value of a specified variable or both. The SAY command is helpful in checking data manipulations to see the result of the manipulations. Output from the SAY command goes to the list file.

It is important to recognize the difference in results that will be printed depending on how the location is specified (string, numeric or punch).

- The location specified as [4.2] is printed as a numeric field and leading zeros will be dropped if they exist. Any value in the location that is not a good number will print as MISSING.
- The location specified as [4.2\$] is printed as a string so if leading zeros or blanks exist, they will be printed.
- The location specified as [4.2\$P] will display all the punches in each column separated by backslashes.

Contents of Columns:	SAY Command:	Results Printed:
columns 4-5=04	SAY [4.2]	4
columns 4-5=04	SAY [4.2\$]	04
columns 4-5=04, columns 6-7 are blank	SAY [4.4]	MISSING="04 "
columns 4-5=04, columns 6-7 are blank	SAY [4.4\$]	04
columns 4-5 are blank, columns 6-7=14	SAY [4.4]	14
columns 4-5 are blank, columns 6-7=14	SAY [4.4\$]	14
columns 4-7=ABCD	SAY [4.4]	MISSING="ABCD"
columns 4-7=ABCD	SAY [4.4\$]	ABCD
column 4=punches 1234, column 5=4	SAY [4.2\$P]	1234\4

To display some descriptive text followed by data, enter the text enclosed in quotes.

**Example:** SAY "ID for this case:" [1.4\$]

This will print: ID for this case: 0001

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

If you wanted to look at the result of a manipulation for a specific case, you would enter:

```
IF [1.3#020] THEN SAY [4.2$] ENDIF
```

If ID# 020 contained the number 26 in columns 4-5, you would see the result "26".

If you wanted to display exactly what was in each column, including the case ID and any multiple punches if they existed, you would enter:

**Example:** `IF [1.3#020] THEN SAY + [4.2$] ENDIF`

The "+" sign before the location allows you to see this additional information:

```
ID: 020          <-- this is the case ID number
Card 01:        <-- this is the record number
00             <-- this is a column template (columns 04 and 05)
45
26             <-- ASCII display of column contents
2              <-- binary display of column contents
6
```

If the case has multiple punches in a column range in Card 2, the command:

**Example:** `IF [1.3#021] THEN SAY + [81.2$] ENDIF`

would show the following information:

```
ID: 021          <-- this is the case ID number
Card 02:        <-- this is the record number
```



```
00          <-- this is a column template (columns 81 and 82 or
           columns 01 and 02 of card 2)
12
**          <-- ASCII display of column contents
55          <-- binary display of column contents
8
99
```

If you have pre-defined variables, the SAY command will display additional information for each CAT and FLD type question. No additional information is displayed for NUM, VAR, and TEX type questions. For a pre-defined CAT variable named GENDER:

**Example:** SAY CASE\_ID GENDER

would show the following for case 001:

```
001 gender (2:1=1=MALE)
```

The "2" tells how many possible responses there are to the GENDER question. The "1=1" says that Case 001 answered the GENDER question with response 1 and that response 1 was the first item in the response list. "MALE" indicates that the text associated with response 1 is MALE.

For a pre-defined FLD variable named DAY:

**Example:** SAY CASE\_ID DAY

would show the following for case 001:

```
001 day(7:MON=1=Monday)
```

### 3.1.4 Data Manipulation for Punch, String, and Numeric Variables

#### **DIRECT DATA MOVES**

The **COPY** command is used to copy data from one location directly to another location, whether that data is in numeric, string, or punch format. The syntax for a COPY command is:

```
COPY to_datavar = from_datavar
```

datavar is any location ([col.wid]) or any user-defined variable.

An example of a COPY command is:

**Example:** COPY [24.2] = [6.2]

If the contents of columns 6-7 was 04, then 04 is copied from columns 6-7 to columns 24-25. The original contents of columns 24-25 have been replaced with what was in columns 6-7, and columns 6-7 still have their original data.

The COPY command is the easiest way to copy the contents of a multiple punched column to another column.

The **TRANSFER** command is used to alter the data in a location. The syntax for a TRANSFER command is:

```
TRANSFER to_datavar op= from_expression
```



op refers to the different operators to add or remove data, i.e.:

- (none) Replaces data; blanks location first (only blanks valid pre-defined responses for the to\_datavar)
- + Adds data; does not blank location first
- Removes data

The **MODIFY** command is used to convert data from one type to another, while the **TRANSFER** command is used to change the contents of a variable. For example, **TRANSFER** would be used to add two number variables together to get a third number variable. **TRANSFER**, unlike **MODIFY** checks that the data is of the same type on both sides of the operator.

The recommended way to add or remove punches from a data location is the **MAKE\_DATA** command (see section “*Adding/Removing Punches*” under “*3.1.4 Data Manipulation for Punch, String, and Numeric Variables*”).

Some examples of the **TRANSFER** command follow:

**Example:** `TRANSFER [26.2 = 4`

This transfers the number 4 to columns 26-27. Since the location 26.2 is defined as numeric (no \$ or \$P in the brackets), the 4 is right-justified so column 26 is blank and column 27 = 4.

**Example:** `TRANSFER [32.2$] = "4"`

This transfers the literal 4 to columns 32-33. Since the location 32.2 is defined as a string (\$ in the bracket), the 4 is left-justified so column 32 = 4 and column 33 is blank.

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

**Example:** TRANSFER [14\$P] = "1,2,3"

This TRANSFER command says to treat column 14 like a punch type variable (\$P in the bracket) and move the punches 1, 2 and 3 to column 14.

**Example:** TRANSFER [15.2\$P] = "1,2\3,4"

This TRANSFER command says to treat both columns 15 and 16 like punch type variables (\$P in the brackets) and move punches 1 and 2 to column 15 and punches 3 and 4 to column 16. The backslash (\) says to put what's before it into the first column and what's after it into the second column. If you were interested in seeing what columns 15 and 16 looked like after this transfer you should enter:

```
DISPLAY_BINARY [15.2$P]
```

and Mentor would display:

```
          ID: 001: [ 15.2 ] =12\34
column ASCII  Binary
      15     *    1,2
      16     *    3,4
```

In the following example:

**Example:** TRANSFER S [36.4\$] = "TEST"

```
TRANSFER S = "R"
```

a new string type variable is created called S that is located in columns 36-39. The new name is specified immediately in front of the open square bracket. The letters TEST are transferred to columns 36-39. If we assume that the contents of S was TEST from our first TRANSFER command, then the second transfer command



changes the contents of the string variable S to the letter R. The letter R is left-justified so column 36 = R and columns 37-39 are blank.

An example of an operator that can be used with the TRANSFER command is the plus sign (+). When you are dealing with a string type variable, the + can be used to replace characters and not blank the receiving location first.

**Example:** TRANSFER S [36.4\$] = "TEST"  
 TRANSFER S += "R"

This first TRANSFER command will move TEST into columns 36-39. The second transfer command will move the R to column 36 and not clear the remaining columns. The result is columns 36-39 = REST.

Using the plus (+) operator with a numeric type variable results in the addition of two values. (See section “*Arithmetic Calculations*” under “*3.1.4 Data Manipulation for Punch, String, and Numeric Variables*” for more examples.)

**Example:** TRANSFER N [40.4] = 1234  
 TRANSFER N += 5

The first TRANSFER command will first blank columns 40-43, then move '1234' to the numeric variable called N that is located in columns 40-43. The second command will not clear the contents of columns 40-43 but will add the number 5 to it. The result is columns 40-43 = 1239. You can also use the TRANSFER command to copy the data from more than one location to more than one location.

**Example:** TRANSFER [44.2,46.2,48.2] = [22.2,20.2,18.2]

This command will first blank the receiving locations, then copy the contents of columns 22-23 to columns 44-45, and will copy the contents of columns 20-21 to columns 46-47 and will copy the contents of columns 18-19 to columns 48-49.

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

Since the receiving locations are defined as numeric, if the sending locations are not numeric, the receiving locations will be blanked and the non-numeric data will not be transferred.

You can also use the TRANSFER command to change the data in more than one location at a time.

**Example:** TRANSFER [44.2, 46.2, 48.2] = VALUES (1, 2, 3)

This command would transfer a 1 to column 45, a 2 to column 47 and a 3 to column 49 (with columns 44, 46, and 48 being blanked).

**Example:** TRANSFER [50.2, 52.2, 54.2] = VALUES (4, 4, 4)

This command would transfer a 4 to columns 51 and 53 and 55.

One unique use of the MODIFY command is to spread multi-punched data into multiple single-punched locations. This might be used to reformat a multi-punched question into a series of mentioned/not mentioned questions or when providing a data file for use in a program that doesn't accept multiple punches. This process would be accomplished with the following command:

**Example:** MODIFY [21, ..., 26] = [20^1//6]

This MODIFY command spreads the multi-punched data in column 20 into columns 21 to 26 with a series of ones and blanks. For instance, a 2 punch in column 20 would cause a 1 to be placed into column 22, a 6 punch in column 20 would cause a 1 to be placed into column 26 and so on.

**NOTE:** The number of single-punched columns needed equals the number of possible punches in the multi-punched column.



If zeroes are preferred over blanks for representing not mentioned, the following syntax will recode blanks into zeroes (0) and put it back into the same field:

**Example:** TRANSFER [21, . . . , 26] = [!21, . . . , 26]

In this example the exclamation point (!) means return a zero when the location is blank.

### *Adding/Removing Punches*

The MAKE\_DATA command is one way to add or remove data in punch type variables. The syntax for the MAKE\_DATA command is:

```
MAKE_DATA op [datavar]
```

datavar can be any punch type variable

Here are some examples for using the MAKE\_DATA command to add punches:

**Example:** MAKE\_DATA [8^1 . 5]

This blanks column 8 and then adds to column 8 the punches 1,2,3,4 and 5.

**Example:** MAKE\_DATA [9^1, 5]

This blanks column 9 and then adds to column 9 the punches 1 and 5.

**Example:** MAKE\_DATA + [10^X, Y]

This doesn't blank column 10 but adds the punches X and Y.

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

Here are some examples for using the MAKE\_DATA command to remove punches from punch variables:

**Example:** MAKE\_DATA - [7^1, 0]

This removes only the punches 1 and 0 from column 7. Any other punches in column 7 remain unchanged.

**Example:** MAKE\_DATA - [11^1.5]

This removes only the punches 1,2,3,4 and 5 from column 11. Any other punches in column 11 remain unchanged.

#### *Arithmetic Calculations*

In arithmetic calculations, the following arithmetic operators are available:

+	addition
-	subtraction
*	multiplication
/	division
++	when adding, substitute zero for missing elements (if both/all are missing, then the result will be missing)

For the purpose of the following example we will use the TRANSFER command to copy the number 6 to columns 4-5 and the number 4 to each pair of columns starting with 6-7 and ending with 20-21. This TRANSFER command looks like:

**Example:**

```
TRANSFER [4.2, . . . , 20.2] =VALUES (6, 4, 4, 4, 4, 4, 4, 4, 4, 4)
```



With these values assigned, the following TRANSFER and SAY commands can be used:

<b>Commands:</b>	<b>Results:</b>
TRANSFER [6.2] += 2	SAY [6.2] 6
TRANSFER [8.2] -= 2	SAY [8.2] 2
TRANSFER [10.2] /= 2	SAY [10.2] 2
TRANSFER [12.2] *= 2	SAY [12.2] 8
TRANSFER [14.2] += [4.2]	SAY [14.2] 10
TRANSFER [16.2] -= [4.2]	SAY [16.2] -2
TRANSFER [18.2] /= [4.2]	SAY [18.2] 1
TRANSFER [20.2] *= [4.2]	SAY [20.2] 24

Since we said nothing about decimal significance, the result of 4 / 6 (the TRANSFER [18.2] /= [4.2] command above) was rounded up to 1.

Continuing with the above example, if we know that columns 22-28 are blank then:

**Example:**

TRANSFER [26.2] = [22.2] ++ [4.2]	SAY [26.2]	6
TRANSFER [28.2] = [22.2] ++ [24.2]	SAY [28.2]	MISSING

If the result will not fit in the receiving location, that location will be filled with asterisks (\*)

Using SAY for a numeric field displays without leading blanks or zeroes so you're seeing the numeric value displayed left justified, regardless of the field width. If a view of the complete location width is desired, use SAY [loc\$], where the dollar

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

sign (\$) forces the number to be displayed as a string with all leading blanks and zeroes.

We could use the arithmetic operator "++" to sum the number of household members in different age groups to make a total of household members:

**Example:**   TRANSFER [40.6] = [20.2] ++ [22.2] ++ [24.2]

This example would clear the receiving location (40.6), and then place into it the sum of the three locations 20.2, 22.2, 24.2. The "++" would cause any missing value to be treated as zero. If all values were missing the result would be missing.

In addition to the above arithmetic operators, the following arithmetic functions are also available:

ABSOLUTE\_VALUE

AVERAGE

EXPONENT

LOGARITHM

SQUARE\_ROOT

SUM

X

The following examples illustrate some uses of these arithmetic functions (in all examples below, the PRINT\_TO\_DATA command blanks the receiving location first). (See "3.1.7 Formatting Data Elements" for more information on the PRINT\_TO\_DATA command.)

**Example:**

```
PRINT_TO_DATA [30.8] "%Z1_8.2S" ABSOLUTE_VALUE ([20.8])
```

This example of the ABSOLUTE\_VALUE function will put the absolute value of the number in columns 20-27 into columns 30-37. The "Z1" will zero-fill, the underscore separates the "Z1" from the location, the "8.2" specifies an 8 column field with 2 decimal places and the "S" says to print a string of characters.

**Example:**

```
PRINT_TO_DATA [30.6] "\Z1_6.3S" AVERAGE ([20.2,22.2,24.2])
```

This example of the AVERAGE function will put the average of the 3 numbers in columns 20-21, 22-23, and 24-25 into columns 30-35. The "Z1" will zero-fill, the underscore separates the "Z1" from the location, the "6.3" specifies a 6 column field with 3 decimal places and the "S" says to print a string of characters.

**Example:** `PRINT_TO_DATA [32.6] "\Z1_6.3S" EXPONENT (3)`

The example above will return the product of Euler's Constant ( $e=2.71828$ ) raised to the power of 3 (the number in the vector). Therefore this command will place 20.086 into columns 32-37, formatted as in the prior example.

**Example:**

```
PRINT_TO_DATA [52.6] "\Z1_6.3S" LOGARITHM(2.71828)
```

This example of the LOGARITHM function will return the natural log ( $e$  sub  $n$ ) of 2.71828 (the number in the vector). In this instance, 01.000 will be placed into columns 52-57.

**Example:**

```
PRINT_TO_DATA [40.6] "\Z1_6.3S" SQUARE_ROOT(26)
```

The example of the SQUARE\_ROOT function above will return the square root of 26 (the number in the vector). In this instance, 05.099 will be placed into columns 40-45.

**Example:**

```
PRINT_TO_DATA [40.6] "\Z1_6.3S" SUM([20.2,...,24.2])
```

This example of the SUM function will return the sum of the numbers in columns 20.2, 22.2 and 24.2 (the locations in the vector). The result of the SUM will be placed into columns 40-45 and will be right justified, zero-filled and will have three decimal places of significance. Any missing value will be treated as zero. If all values are missing, the result will be missing.

**Example:**

```
PRINT_TO_DATA [40.6] "\Z1_6.3S" X([20.2]) + X([22.2]) + X([24.2])
```

This example, using the X function, shows how to force a blank location or a location with something other than a valid number to be returned as a zero in an equation or numeric variable. If any (or all) of the locations were blank (or MISSING) or had alpha characters, adding the locations would normally return MISSING. The X function causes the problem location to be treated as a zero.

***Netting Punches***

The goal of the following setup is to create a new multi-punched variable (in columns 68-70) from data collected in 11 fields of 2 columns each (columns 4-25). The answers collected in the 11 fields are codes 01-26. The ~INPUT command opens a data file called DATAGENS.TR in UPDATE mode. The BLANK command will blank columns 68-70. The TRANSFER command will net the answers of 01-12 from any of the 11 fields into column 68 as punches 1-12. The net of the answers 13-24 will be in column 69 as punches 1-12 and the net of the answers 25 and 26 will be in column 70 as punches 1 and 2. The += means adds punches and do not blank the receiving location first.





**Example:**

```
~INPUT DATAGENS, ALLOW_UPDATE

~DEFINE
PROCEDURE={GENS:
BLANK [68.3]

>REPEAT $COL=04,06,...,24
TRANSFER [68.3^01//26] += [$COL.2#01//26]
>END_REPEAT

SAY CASE_ID [4.22$] [68.3$P]
}

~EXECUTE PROCEDURE=GENS

~END
```

The SAY command will show the case ID, the contents of the original 11 fields and the resulting netted punches in columns 68-70. An example of the output from this SAY command follows:

**Example:**

```
001 210102081718 128\569\
002 021721 2\59\
003 19152023 \378X\
004 1522 \30\
```

```
005 1807 7\6\
```

### ***Storing Weights in the Data***

By storing weights in the data we can speed up subsequent runs since the program will not have to recalculate the weights. To get the weights into the data, define a procedure that uses either the MODIFY, TRANSFER or PRINT\_TO\_DATA command to insert the weights into an unused location.

Use the MODIFY or TRANSFER command if the weights are comprised of integers or if you plan to store the weight as an integer but later reference the weight with decimal significance.

#### **Example:**

```
~DEFINE
PROCEDURE={GENWTMOD:
MODIFY [12.4] = SELECT ([26^1//3], VALUES (86,66,139))
}
```

or

#### **Example:**

```
~DEFINE
PROCEDURE={GENWTTRAN:
TRANSFER [12.4] = SELECT ([26^1//3], VALUES (86,66,139))
}
```

The two procedures above would store 86, 66 and 139 into columns 12-15 based respectively on the punches 1, 2 and 3 in column 26. The weights stored in columns 12-15 would be right justified and blank filled.

To reference later with decimal significance (2 decimal places) use:

```
WEIGHT= : [12.4 * F2]
```

This would change the way the program references the values inserted into the data by the two procedures above to .86, .66, and 1.39.

Use the PRINT\_TO\_DATA command to insert data with decimal significance directly into the location. The PRINT\_TO\_DATA command blanks the receiving location first.

```
Example: PRINT_TO_DATA [12.4] "\Z1_4.2S" &
SELECT ( [26^1//3] , VALUES ( .86 , .66 , 1.39 ) )
```

This command would put the weights 0.86, 0.66 and 1.39 into columns 12-15 based respectively on the punches 1, 2 and 3 in column 26.

### ***Randomly Selecting Respondents***

Defining a procedure that randomly selects respondents is another use of the TRANSFER command.

The following run would create a list of six randomly selected respondents. (See section on “9.3.1 System Constants” under “9.3.1 System Constants” for further information on the RANDOM\_VALUE constant.)

**Example:**

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

```
>DEFINE @HOWMANY 6
>PRINT_FILE RANDOM

~INPUT DATACLN
~OUTPUT RAND LENGTH=80

~DEFINE
  PROCEDURE={ASSIGN:
    BLANK [11-80]
    TRANSFER RANDVAR[11.5] = RANDOM_VALUE * 10000
    WRITE_CASE
  }
~EXECUTE PROCEDURE=ASSIGN

~INPUT RAND
~OUTPUT SORT
~SORT RANDVAR

~INPUT SORT, STOP_AFTER=@HOWMANY
~DEFINE
  PROCEDURE={WHO:
    SAY CASE_ID
    PRINT_LINES "\S" CASE_ID
  }
~EXECUTE PROCEDURE=WHO
~INPUT
~END
```

This run inserts a random number into an arbitrary location ([11.5]) of an intermediate data file copy (leaving the original data file unchanged), sorts the data

on that random number into a second intermediate data file, prints the ID numbers of the first 6 (specified by the >DEFINE @HOWMANY) respondents into a print file called RANDOM.PRT, then deletes the disposable intermediate data files.

### 3.1.5 Data Manipulation for Predefined Variables

For the following sections on data manipulation using pre-defined variables, the list of variables below will be used as examples.

```
image.chk
Data area ends at 160, Text area starts at 161
col.wid  label      QQ#  qtype  sub  other use
   4.2    TIMES     3.00 [NUM ]  Z
   6.2    OFTEN     4.00 [NUM ]  Z
GAP of 47 columns
   55.1   DISEASE   12.00 [CAT ]
   56.1   ASTHMA    13.00 [CAT ]
GAP of 9 columns
   66.1   CITY      19.00 [CAT ]
   67.1   GENDER    20.00 [CAT ]
GAP of 1 columns
   69.3   DAY        21.00 [FLD ]
   72.9   NAME       22.00 [VAR ]
   81.1   VACATION  23.00 [TEXT]  B
Text starts at 161
```

#### ***Direct Data Moves***

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

The **COPY** command is used to copy data from one location directly to another location, whether that data is in numeric, string, or punch format. The syntax for a COPY command is:

```
COPY to_datavar = from_datavar
```

where datavar is any variable which references some part of the data. It can be a label or a question number (QQ#) defined using PREPARE, or it can be any location ([col.wid]).

For this example we will evaluate five pre-defined variables using the SAY command:

<b>Commands</b>	<b>Results</b>
SAY CITY	CITY (4 : 2=2=DENVER)
SAY DAY	DAY (7 : MON=1=MONDAY)
SAY TIMES	21
SAY OFTEN	1
SAY ASTHMA	ASTHMA (3 : 1=1=ASTHMA, 2=2=EMPHYSEMA )

The following examples of the COPY command use the above pre-defined variables:

<b>Commands</b>	<b>Results</b>
COPY [8] = CITY	column 8 = 2
COPY [9.3] = DAY	columns 9-11 = MON
COPY [12.2] = TIMES	columns 12-13 = 21
COPY OFTEN = TIMES	OFTEN = 21

```
COPY      [18]      = ASTHMA    column 18 = punches 1 and 2
```

Receiving locations can be defined as any variable type, however the sending variable must be one that was defined in EZWriter/PREPARE or a PREPARE= variable. COPY does a direct copy of the sending location or variable (right-hand side of operation) to the receiving location or variable (left-hand side). If the length of the variable in the sending location is larger than the receiving location, an error message will print. If the length of the variable in the sending location is shorter than the receiving location, the copy will occur but the results will be left justified in the receiving location. This would not be desirable in copying NUM type variables.

COPY is the easiest way to move a multiple response question from one location to another.

The **TRANSFER** command is used to alter the data in a location.

The syntax for a TRANSFER command is:

```
TRANSFER to_datavar op= from_datavar
```

Operators (op) that can be used with the TRANSFER command are:

- (none)** Replaces data; blanks location first (only blanks valid pre-defined responses for the to\_datavar)
- +** Adds data; does not blank location first
- Removes data

The to\_datavar can be any of the previously described variable types or an unspecified location. The from\_datavar is the actual data to be transferred to the receiving location. The from\_datavar be a number, location, expression, string, or pre-defined variable.

The **MODIFY** command is used to convert data from one type to another, while the **TRANSFER** command is used to change the contents of a variable. For example, **TRANSFER** would be used to add two number variables together to get a third number variable. **TRANSFER**, unlike, **MODIFY** checks that the data is of the same type on both sides of the operator.

The recommended way to add or remove punches from a data location is the **MAKE\_DATA** command (see section on “*Adding/Removing Responses*” under “*3.1.5 Data Manipulation for Predefined Variables*”).

An example of a **TRANSFER** command with the **to\_datavar** as an unspecified location and the **from\_datavar** as a number would be:

**Example:**    **TRANSFER [14] = 3**

This would blank column 14 first, then put the number 3 in column 14.

An example of a **TRANSFER** command with the **to\_datavar** as a pre-defined **NUM** type variable and the **from\_datavar** as an unspecified location would be:

**Example:**    **TRANSFER TIMES = [14]**

This would put the contents of column 14 in the **TIMES** variable.

An example of a **TRANSFER** command with the **to\_datavar** as a pre-defined **NUM** type variable and the **from\_datavar** as an expression would be:

**Example:**    **TRANSFER TIMES = [14]+10**

This would put the contents of column 14, plus 10, in the **TIMES** variable.



When using TRANSFER to put a string into a FLD type variable, we must enclose the variable in brackets and add a dollar sign (\$) to keep both sides of the equation equal. An example of a TRANSFER command with the to\_datavar as a pre-defined FLD type variable and the from\_datavar as a string would be:

**Example:** TRANSFER [DAY\$] = "WED"

This would put the string "WED" into the DAY variable.

Our pre-defined variable DAY is a FLD type question and was constructed with the following answers: MON, TUE, WED, THU, FRI, SAT, and SUN. These are the only valid answers to this question. If the data contained the answer "WEE", this would not be recognized as a valid answer and would produce an error if we cleaned the DAY variable using the EDIT command. (See "2.4 CLEANING WITH SURVENT VARIABLES", "Generating A List Of Error Messages" for examples of cleaning pre-defined variables.) We can transfer an answer to the DAY variable that is not a valid answer, but it would not be recognized as an acceptable answer. If we were in a ~CLEANER block evaluating a respondent's answers, and the case we were looking at had MON as the answer to the DAY variable and we entered:

SAY DAY

Mentor displays:

DAY (7:MON=1=Monday)

If we then were to enter the following TRANSFER and SAY commands:

TRANSFER [DAY\$]="WEE"

SAY DAY

Mentor displays:

```
DAY (7 :)
```

because WEE is not a recognized answer to the DAY question.

The SAY command only shows valid answers to a question. If we wanted to see this respondent's answer to the DAY question, even if it was not a valid answer, we could treat the DAY question as a string type variable and enter:

```
SAY [DAY$]
```

Mentor displays:

```
WEE
```

Here are some examples using the TRANSFER command and pre-defined variables:

**Example:** TRANSFER [DAY\$] = "TUE"

where DAY becomes TUE and if that is an acceptable code for the original DAY question it will be recognized when the DAY question is referenced.

**Example:** TRANSFER TIMES = 1

where TIMES becomes 1 if it is within the range that TIMES was originally defined.

**Example:** TRANSFER NAME = "MARTIN"

where NAME will contain the string MARTIN as long as the string will fit NAME's original definition length. Note that because the variable NAME is a

VAR type variable, we can use the variable name without surrounding brackets and following dollar sign (\$).

**Example:** TRANSFER VACATION = "EATING"

where EATING will replace any text that currently exists in VACATION's data location.

A new name can be given to any pre-defined variable. The new name is specified immediately in front of the open square bracket (i.e., NEWNAME[INCOME\$]). The data in the location of INCOME can be referenced as a punch type variable which is called NEWNAME. INCOME remains intact in its original definition.

Some additional TRANSFER examples are:

**Example:** TRANSFER NAME2 [8.9\$] = NAME

where NAME2 is defined to be columns 8 through 16 and is a string type variable. NAME2 can now be used in place of [8.9\$] when it is necessary to refer to that location in that format. NAME's data will be put into NAME2.

**Example:** TRANSFER NAME2 = "SAM"

where the columns 8-10 will now contain the ASCII characters SAM and the remaining columns 11-16 are blank.

**Example:** TRANSFER [OFTEN\$] = "RF"

where OFTEN was originally a NUM type question with a width of 2. Here the location OFTEN is redefined as a string variable for this transfer, and the ASCII characters RF are put into the location. Since the characters RF were not part of the pre-defined variable OFTEN, SAY OFTEN will show MISSING. SAY [OFTEN\$] will show RF.

**Example:**   TRANSFER [GENDER] = 3

where GENDER is redefined as numeric for this transfer and the data location GENDER references receives a 3. This would be one way to transfer a response to the data location GENDER that is not one of the pre-defined responses. Since the number 3 was not a pre-defined response for the GENDER variable, SAY GENDER will show 'gender(2:)' . SAY [GENDER\$] will show 3.

**Example:**   TRANSFER GENDER = CATS (1)

will add the pre-defined response 1 to the GENDER variable and will not clear out the existing 3 since the 3 is not defined as a valid GENDER response.

A subsequent command:

TRANSFER GENDER = CATS(2)

would clear out the existing valid GENDER variable response 1 and insert the pre-defined response 2 to the GENDER variable but again the existing 3 would not be cleared out since it is not defined as a valid GENDER response.

### ***Adding/Removing Responses***

The best way to add or remove punches from CAT type questions is to specify which categories are to be added or removed. See “3.1.4 Data Manipulation for Punch, String, and Numeric Variables”, “Adding/Removing Punches” for details on doing this to data locations, not pre-defined variables. The syntax for using the MAKE\_DATA command with pre-defined variables is:

MAKE\_DATA op CATtypevar(response code(s))



(For a discussion of operators, see 3.1.5 *DATA MANIPULATION FOR PRE-DEFINED VARIABLES, Direct Data Moves*)

**Example:** MAKE\_DATA GENDER (2)

where GENDER is a single response CAT type and the data location will now hold a response 2.

**Example:** MAKE\_DATA ASTHMA (1, 2)

where ASTHMA is a multi-response CAT type and the data location now holds responses 1 AND 2.

All of the above examples can be changed so that the receiving location is NOT blanked prior to the data move. To do this, simply use the plus (+) sign.

**Example:** MAKE\_DATA + ASTHMA (1)

where ASTHMA is a multi-response CAT type and the data location now holds response 1 in addition to whatever responses were previously in that location.

Under certain circumstances, you may want to be able to remove a response from a location, without affecting the other responses in that location. To do this use the minus (-) sign as in the following example:

**Example:** MAKE\_DATA - ASTHMA (1)

where ASTHMA is a multi-response CAT type and if a response 1 exists in that data location it will be removed.

**NOTE:** The only types of variables where you can remove punches are in the examples above of CAT (single and multi-response) and punch. It does not make sense to try to remove punches from other types of variables.

### *Arithmetic Calculations*

In arithmetic calculations, the following arithmetic operators are available:

+	addition
-	subtraction
*	multiplication
/	division

The TRANSFER command can be used in combination with these operators. The equal sign (=) is part of the syntax:

TRANSFER datavar op= numexpr

<b>datavar</b>	is any variable which references the data. It can be a label, a question number (QQ#) from PREPARE, or a data location ([col.wid]).
<b>op</b>	is an optional arithmetic operator (+,-,/,*)
<b>=</b>	is required syntax even if no op is specified
<b>numexpr</b>	is a constant, an arithmetic operation, a function, or some combination of these that returns a number

In the example:

```
TRANSFER TIMES = 3
```

TIMES becomes 3. If this instruction is followed by:



```
TRANSFER TIMES +=5
```

then TIMES becomes 8.

Some additional examples of syntax are:

**Example:**

```
TRANSFER [22.2] = times + often
```

```
TRANSFER [22.2] = times - often
```

```
TRANSFER [22.2] = times / 2
```

```
TRANSFER [22.2] = times * 2
```

***Creating Variables***

One use of the TRANSFER command is to put data into a new location and either blank or not blank the receiving location first. The data you may want to put into the new location can be an expression that has to be evaluated, an ASCII string, a punch or punches, or a number.

Suppose you have two variables, ASTHMA and GENDER, and you want to create a new variable, ASTHMA2, that will cross those two variables. You want to create a location in the data (we will use column 68) that contains the result of the expression, which can then be defined using the PREPARE= format to assign labeling for use as a column or row variable in a cross tabulation later on.

The setup using the PREPARE= variable would look like this:

**Example:**

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

```
~DEFINE PREPARE=&
{ASTHMA2: 68
ASTHMA BY GENDER
!CAT, , 1
1 ASTHMA MALE
2 EMPHYSEMA MALE
3 NEITHER MALE
4 ASTHMA FEMALE
5 EMPHYSEMA FEMALE
6 NEITHER FEMALE
}
```

The procedure you would create to define this new variable would look like this:

#### **Example:**

```
~DEFINE
PROCEDURE={DOIT:
TRANSFER ASTHMA2 = ASTHMA BY GENDER
WRITE_CASE
}
```

If we were interested in seeing the variables ASTHMA, GENDER and the new variable ASTHMA2 we could add the following line to our procedure:

**Example:** SAY CASE\_ID ASTHMA GENDER ASTHMA2

Our whole setup might look like:

```
>USE_DB IMAGE,READ_WRITE,DUPLICATE=WARN
```





```
~INPUT DATA CLN
~OUTPUT DATA GENS

~DEFINE PREPARE=&
{ASTHMA2: 68
ASTHMA BY GENDER
!CAT,,1
1 ASTHMA MALE
2 EMPHYSEMA MALE
3 NEITHER MALE
4 ASTHMA FEMALE
5 EMPHYSEMA FEMALE
6 NEITHER FEMALE
}

PROCEDURE={DOIT:
TRANSFER ASTHMA2 = ASTHMA BY GENDER
SAY CASE_ID ASTHMA GENDER ASTHMA2
WRITE_CASE
}

~EXECUTE PROCEDURE=DOIT
~END
```

The following are a few examples we might see from the SAY command:

```
018 ASTHMA (3:3=3=NEITHER) GENDER (2:1=1=MALE) ASTHMA2
(6:3=3=NEITHER MALE)
```

019 ASTHMA (3:2=2=EMPHYSEMA) GENDER (2:1=1=MALE) ASTHMA2  
(6:2=2=EMPHYSEMA MALE)

020 ASTHMA (3:3=3=NEITHER) GENDER (2:2=2=FEMALE) ASTHMA2  
(6:6=6=NEITHER female)

### 3.1.6 Relational Operators

Relational operators (sometimes called “relops”) allow you to compare variables to one another or to a constant. They can be used to set up conditions, to limit your base for certain functions or anything else that you want done to some but not all of your cases. There are six relational operators; each can be expressed with symbols or letters (and some with words):

TYPE	SYMBOL	LETTERS	WORD
Equal to	=	EQ	MATCHES
Not equal to	≠	NE	—
Greater than	>	GT	—
Less than	<	LT	—
Greater than or equal to	≥	GE	CONTAINS
Less than or equal to	≤	LE	—

All six can be used for arithmetic comparisons.



**Example:**

```
IF OFTEN > TIMES THEN
    SAY CASE_ID "OFTEN>TIMES"
ENDIF
IF OFTEN EQ 3 THEN
    SAY CASE_ID "OFTEN EQ 3"
ENDIF
IF OFTEN + TIMES >= 42 THEN
    SAY CASE_ID "OFTEN+TIMES>=42"
ENDIF
IF OFTEN < 50 THEN
    SAY CASE_ID "OFTEN<50"
ENDIF
IF OFTEN NE [6.2] THEN
    SAY CASE_ID "OFTEN NE 6.2"
ENDIF
IF OFTEN - TIMES LE OFTEN/TIMES THEN
    SAY CASE_ID "OFTEN-TIMES LE OFTEN/TIMES"
ENDIF
IF NAME <> "MARK      " THEN
    SAY CASE_ID NAME
ENDIF
```

A cleaning instruction might be:

```
IF [4.2] > 20 THEN
    SAY "CASE ID: " CASE_ID "COLUMNS 4.2 = " [4.2]
ENDIF
```

Some examples of comparisons follow:

If you compare two numeric fields and they are both blank, they will be equal to one another. If, for example, columns 4 and 5 are blank, and if you used the instruction:

```
IF [4] = [5] THEN  
    SAY "EQUAL"  
ENDIF
```

you would see the word EQUAL. If you compared the two columns as strings (\$), as in the following example:

```
IF [4$] = [5$] THEN  
    SAY "EQUAL"  
ENDIF
```

you would see the word EQUAL here also.

When evaluating numeric fields, preceding blanks are acceptable but trailing blanks are not recognized as part of a number. If columns 2-3 are blank and columns 4-7= 1234, and you used the instruction:

```
IF [2.6#1234] THEN  
    SAY "TRUE"  
ENDIF
```

you would see the word TRUE.

If columns 4-7= 1234 and columns 8-9 are blank, and you used the instruction:



```
IF [4.6#1234] THEN
    SAY "TRUE"
ENDIF
```

you would not see the word TRUE.

When evaluating alphabetic or string fields, preceding blanks are not acceptable while trailing blanks are ignored. If columns 2-3 are blank and columns 4-7= TEST, and you used the instruction:

```
IF [2.6#TEST] THEN
    SAY "TRUE"
ENDIF
```

you would not see the word TRUE.

If columns 4-7= TEST and columns 8-9 are blank, and you used the instruction:

```
IF [4.6#TEST] THEN
    SAY "TRUE"
ENDIF
```

you would see the word TRUE.

When evaluating alphabetic or string fields and you want to check for the presence of blanks, you would use quotes (") to surround the blanks. If columns 2-3 are blank and columns 4-7= TEST, and you used the instruction:

```
IF [2.6$]=" TEST" THEN
    SAY "TRUE"
```

```
ENDIF
```

you would see the word TRUE.

If columns 4-7= TEST and columns 8-9 are blank, and you used the instruction:

```
IF [4.6$]="TEST " THEN  
    SAY "TRUE"  
ENDIF
```

you would see the word TRUE.

All six relational operators can also be used for alphabetical comparisons, but in most cases it does not make sense to compare words other than using equal to (EQ,=) or not equal to (NE,<>). In determining if one word is greater than another, the program does have a ranking scheme for all ASCII characters based on the standard ASCII character set:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN OPQRSTUVWXYZ  
[ ] ^ _ `abcdefghijklmnopqrstuvwxyz{|}~
```

Therefore:

```
"+" LT "/"  
"1 "LT "9"  
">" LT "E"  
"a "LT "v"
```

**NOTE:** This ranking is case sensitive so A is less than a; this is why we do not recommend using greater than or less than relational operators with alpha characters unless you're absolutely sure of the case and it is not longer

than one character. You should also enclose the characters in quotes so the program knows you're doing an alphabetic comparison.

### *Changing Case*

Variable Types U, D, and N cause ASCII strings to be treated as upshifted, downshifted, or not shifted, respectively. When displaying a data location, \$ is the same as \$n.

<b>Contents of Columns:</b>	<b>SAY Command:</b>	<b>Results Printed:</b>
THIS IS mixed case.	[1.25\$]	THIS IS mixed case.
THIS IS mixed case.	[1.25\$U]	THIS IS MIXED CASE.
THIS IS mixed case.	[1.25\$D]	this is mixed case.
THIS IS mixed case.	[1.25\$N]	THIS IS mixed case.

By default, these variables are only effective in SAY and PRINT commands. To turn case sensitivity on for ~SORT, ~FREQ or pound sign variables, use the command ~SET CASE\_SENSITIVE (see *Mentor Volume II*). When CASE\_SENSITIVE is set, these variable types can be used to force a string to be upshifted or downshifted.

## 3.1.7 Formatting Data Elements

### ZERO-FILLING DATA

The PRINT\_TO\_DATA command prints data into a data file. This command would be useful if you wanted to zero-fill a numeric type variable or if you wanted to insert decimal points in the data. The PRINT\_TO\_DATA command blanks the receiving location first.

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

The PRINT\_TO\_DATA command is similar to the PRINT command. (See “9.1 GENERATING SPECIALIZED REPORTS” for additional information on printing options.) The syntax for the PRINT\_TO\_DATA command is:

```
PRINT_TO_DATA datavar "format items" variables
```

Let's suppose we start with a two column field, with column 4 blank and column 5 = 8. Here are some examples of zero-filling numeric data:

**Example:** PRINT\_TO\_DATA [8.2] "\Z1\_2S" [4.2]

This says to print the contents of columns 4-5 with a width of two and no decimal places into columns 8-9. The "Z1" says to zero-fill the two receiving columns. The result would be columns 8-9 = 08.

**Example:** PRINT\_TO\_DATA [10.2] "\Z1\_2S" [4.2] + 1

This says to print the results of adding columns 4-5 and the number 1 into columns 10-11 with a width of 2, zero-filled, with no decimal places. The result would be columns 10-11 = 09.

**Example:** PRINT\_TO\_DATA [12.2] "\Z1\_2S" [4.2] - [8.2]

This says to print the results of subtracting columns 8-9 from columns 4-5. Assuming we used the preceding PRINT\_TO\_DATA commands, the results will be columns 12-13 = 00.

Another option for zero-filling data locations is to use the \*Z modifier. The syntax for this modifier is as follows:

**Example:** TRANSFER [21.5\*Z] = [43] + [44] + [45]





If you wanted to zero-fill all data modifications you could use the ~SET ZERO\_FILL command (See *Appendix B: TILDE COMMANDS* for more information on the ZERO\_FILL command).

## DECIMAL POINTS IN DATA

Here are some examples of inserting decimals in the data:

**Example:** TRANSFER [5.5] = 123

This TRANSFER command moves the number 123 to columns 5-9. The number is right-justified so columns 5-6 are blank and columns 7-9 = 123.

**Example:** PRINT\_TO\_DATA [10.5] "\Z1\_5.1S" [5.5]

In the above example we are saying print the contents of columns 5-9 to columns 10-14 with a length of 5 and one decimal point of significance. The "Z1" preceding the 5.1 says to zero-fill the receiving columns (the underscore is required to separate the "Z1" from the location). The "S" following the 5.1 says to print a string of characters into the location 5.1. The result is columns 10-14 = 123.0. If we wanted to print the contents of 5-9 to another location in the data file, let's say columns 15-19, and keep the length at 5 and continue to show one decimal precision but move the decimal point one place to the left we would say:

**Example:** PRINT\_TO\_DATA [15.5] "\Z1\_5.1S" [5.5\*F1]

The result of this command would be columns 15-19 = '012.3'.

If we now wanted to print the contents of columns 15-19 into a new location and remove the preceding zeros we would change \Z1\_5.1S to \5.1S.

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

**Example:** PRINT\_TO\_DATA [25.5] "\5.1S" [15.5]

This would make columns 25-29 = ' 12.3'.

It is also possible to use the PRINT\_TO\_DATA command to format the output of arithmetic operations as in:

**Example:** PRINT\_TO\_DATA [30.5] "\5.1S" 2 \* 5.5

This would make columns 30-34 = ' 11.0'.

## SPREADING MULTI-PUNCHED DATA

We can use the TRANSFER command to spread out multi-punched data into multiple single-punched fields. This is useful if making an ASCII data file for use by another program.

Syntax for this command would be:

**Example:** TRANSFER [6.8\$] = [6\$P]

By default, Mentor spreads multi-punched data into single punches separated by commas. The SUBSTITUTE command causes the commas to be ignored during output.

**NOTE:** The receiving location needs to be equal in width to the maximum number of punches possible in the multi-punched location. If the default commas are desired in addition to the single punches themselves, the receiving location needs to be equal in width to twice the maximum number of punches possible in the multi-punched location minus one.



### *Transforming String Digits into Numbers*

Using the PRINT\_TO\_DATA command, we can transform digits left-justified or randomly located in a field into right-justified NUM type data. The following two step procedure will right-justify all the data in the location, zero-fill the location, and give all data two decimal places (i.e., for currency format):

#### **Example:**

```
~INPUT DATA CLN
~OUTPUT DATAGEN S

~DEFINE
  PROCEDURE={GENS :
    PRINT_TO_DATA [20.6] "\>6S" [20.6$]
    PRINT_TO_DATA [20.6] "\Z1_6.2S" [20.6]
    WRITE_CASE
  }
~EXECUTE PROCEDURE=GENS
~END
```

The first PRINT\_TO\_DATA command right-justifies any data in columns 20.6. The 6 following the \ in this first PRINT\_TO\_DATA command is what says this is going into a six column location and the > preceding the 6 says to right justify the data. From this point on the data is in NUM format (right-justified) and we can do modifications on the data that only apply to NUM type data. The second PRINT\_TO\_DATA command does two of these number modifications. The "\Z1\_6.2S" control item does the following:

- Z1            zero-fill
- 6             the receiving location is 6 columns wide
- .2            use two decimal places

## REFORMATTING YOUR DATA

### 3.1 WHY REFORMAT DATA?

S                    print a string of characters into the location

So if our original data had three cases with the following data in columns 20.6:

" 78 "

" 2.3 "

" 9.98"

then after running the above procedure GENS on the data it would look like:

"078.00"

"002.30"

"009.98"

## TRANSFORMING NUMBERS INTO STRINGS

The function `STRING_FROM_NUMBER` converts a numeric argument into a string. This function is useful in combination with the `PUTID` command to assign case IDs.

## RECODING 10-POINT SCALES

It is common to have 10-point scales entered into a single column location with the 10 value entered as a zero. Some statistical software packages however, cannot convert the 0 to a 10 internally. If you will be exporting CfMC data files with single column 10-point scales to a third party statistical software package, the following `PRINT_TO_DATA` command will reformat your 10-point scales:

### Example:

```
PRINT_TO_DATA [120.2] "\Z1_2S"
```

```
SELECT ( [26^1//0] , VALUES (1 , . . . , 10) )
```

This command will format the data into columns 120-121 as zero-filled ("Z1") right-justified numbers.

## RECODING TO EXCLUDE SELECTED RESPONSES

It is sometimes necessary to reformat a data location into another location and exclude some response, usually "Don't know/No response", i.e., for future input into a statistics software package.

The following example would reformat column 42 to contain only punches 1-4. The "=" sign after the location clears the location first, therefore any punch in column 42 other than 1-4 would not be present after the TRANSFER.

### Example:

```
TRANSFER [42] =SELECT ( [42^1/2/3/4] , VALUES (1 , 2 , 3 , 4) )
```

It would be preferable to INPUT your original data file and OUTPUT a second data file, so as to retain your original data file in an unaltered state, as opposed to using your original data file with ALLOW\_UPDATE set.

## RECODING TO REVERSE A SCALE QUESTION

Often it is necessary to reverse the values of a scale question, i.e., when it is more desirable to have the higher value correlate to the more positive response. We can use the TRANSFER command to accomplish this:

**Example:**   TRANSFER [43^1/2/3/4] = [43^4/3/2/1]

By reversing the figures in the sending and receiving locations for only those punches we want to change, we accomplish the reversal of the scale and leave any other punches, i.e., a "DON'T KNOW/NO RESPONSE" punch, intact.

### 3.1.8 Data Manipulation in the ~CLEANER Block

In the following example we have defined two procedures in the ~DEFINE block and we will use the ~CLEANER block to execute the procedures on a case by case basis. (See "2.3.2 Correcting Errors", "Using Cleaning Screens" for a more detailed explanation of the features used in this example). The command line used to run this example would be:

```
Mentor &STATE.SPX CON                                (DOS)
Mentor "&STATE.SPX" "CON"                            (UNIX)
RUN Mentor.CGO.CFMC;INFO="STATESPX CON"             (MPE)
```

In the spec file called STATE.SPX, the procedure we have defined as SAYSTATE will show us the contents of columns 69-70 (where the state information is located). The procedure defined as FIXSTATE will change the state to CA for those cases where we want the state changed.

```
~INPUT DATAGENS, ALLOW_UPDATE
~SET PRODUCTION_MODE
~DEFINE
    PROCEDURE={SAYSTATE: SAY CASE_ID [69.2$]}
    PROCEDURE={FIXSTATE: TRANSFER [69.2$]="CA"}
}

~CLEANER
```

The above spec file will put us in the ~CLEANER block. The screen will look like this:

CLeaNer-->

At this prompt if we type !SAYSTATE, we will see what state is coded in columns 69-70 for the first case in our data file. If we decide we want to change this case, we type !FIXSTATE at the next prompt and the contents of columns 69-70 will now be CA. We can again type !SAYSTATE at the prompt if we want to see that columns 69-70 is now CA. To move to the next case we would type NEXT at the prompt. We can execute either one or both of our procedures or move to the next case.

**NOTE:** An exclamation (!) must be entered before the procedure name.) We can also issue any other ~CLEANER command. When we are done making modifications we will type ~END to exit Mentor.

## 3.2 Creating Subsets of Data Files

### HOLD\_OUTPUT\_UNTIL\_SUBSET

This command ensures that the ~input file is read only twice for a group of ~outputs rather than potentially twice for each ~output. 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.

### *Examples of subsets of data files*

Here is a list of commands and options that are used when creating subsets of data files:

- `~execute do_subset`
- `~input/~output sampling=#n`
- `~input/~output sampling=.nnn`
- `~input/~output try_for_sampling=#n`
- `~input/~output try_for_sampling=.nnn`
- `~input/~output select=`
- `~input/~output select=casewritten/not(casewritten)`
- `~input/~output num_sample_cases=`
- `~output file_name #n`
- `>random_seed=`

**~EXECUTE DO\_SUBSET**

The `~execute` command `do_subset` is what launches the subsetting run based on the `~input/~output` commands and options you have chosen. It is similar to issuing a "write\_now" command for every `~output` file.

**SAMPLING=#N AND SAMPLING=.N**

The pound sign version of `sampling=` gives you a random sample of "n" records from the data. The `.n` form gives you a random fraction of the sample (i.e. `num_sample_cases * .n`).

**Example:**

```
~input file1 sampling=#10
~output file2
~exc do_subset
~end
```

In the example above, `file2` will be created from a random sample of ten records from `file1`. The pound sign version of `sampling=` tells Mentor how many records



you want. If instead of 10 records you wanted one tenth of the sample, then `sampling=.1` is what you would use. Also note that `sampling=` can be on either the `~input` or `~output` statement or both. (See notes on `num_sample_cases` and `sub1.spx` for examples of using `sampling=`.)

When using `sampling=.n`, the number of records written will be `.n` times the number of records available to be written (i.e. `num_sample_cases`). The exact number of cases written will be the result of this calculation rounded to the nearest whole number.

### **TRY\_FOR\_SAMPLING=#N AND TRY\_FOR\_SAMPLING=.N**

The "try\_for" sampling options work the same way that the `sampling=` options do, except that it is not an error when the number of records requested is not available in the sample. For example you might select on the males in your sample and use `tryfor=#100` to get 100 males if possible, but if there are less than that keep the output anyway. When less than the number of records tried for are available a warning is generated indicating the number that were found.

### **SELECT=**

`Select=` may now be used on either the `~input`, `~output`, or both. If the `select` appears on both the `~input` and `~output`, the one on the `~input` is executed first. No record that does not pass the `~input` selection criteria will have the opportunity to be written to the output file. Both `select=` and `sampling=` may be used at the same time.

### **CASEWRITTEN**

A special variable "casewritten" may be used as part of the `select=` condition. Most often it is used with `not()`, as in `not(casewritten)`. It takes effect when one is using multiple `~output` files in a subset run.

### **Example:**

`~input file1`

```
~output file2a #1 select=[1#1-5]
~output file2b #2 select=not(casewritten)
~exc do_subset
~end
```

**COMBINING OPTIONS**

If `sampling=`/`select=` appear on both the input and output, only those records which make it past the input phase will contribute to the sampling/selection that occurs in the output phase.

For example, note the difference between the following two subset runs:

" This run gets all of the males from the input, and then writes

" a random sample of half of them.

```
~input file1 select=Q1(M)
~output file2 sampling=.5
~exc do_subset
~end
```

" This run gets half of the respondents from the input, and then " writes out the males.

```
~input file1 sampling=.5
~output file2 select=Q1(M)
~exc do_subset
~end
```

In this example, the resulting output files are likely to be similar, but they won't be the same. (Sub3.spx contains examples of using both `sampling=` and `select=` in the same subset run.) When `select=` and `sampling=` occur on the same statement (e.g. both on the `~input`) the `select=` is honored before `sampling=` is done.

Some situations to watch out for:

Suppose you want two output files each containing a random half of your original sample. You might be inclined to do:

```
~input file1
~output file2a #1 sampling=.5
~output file2b #2 sampling=.5
~exc do_subset
~end
```

The two output files in this example will contain many (about half) of the same respondents. What you probably had in mind is:

```
~input file1
~output file2a #1 sampling=.5
~output file2b #2 select=not(casewritten)
~exc do_subset
~end
```

Having divided your sample into two random halves using the specs above, you might think that dividing the sample into equal thirds would be done with:

```
~input file1
~output file2a #1 sampling=.33333
~output file2b #2 sampling=.33333 select=not(casewritten)
~output file2c #3 select=not(casewritten)
~exc do_subset
~end
```

However the above specs won't give you equal sized samples. For example, if file1 contains 100 records, then file2a will contain 33 records (OK so far), but file2b will contain .33333 of the 66 records not written so far (i.e. 22 records), and file2c will contain the rest of the sample (i.e. 45 records).

What you actually need is:

```
~input file1
~output file2a #1 sampling=.33333
~output file2b #2 num_sample_cases=67 sampling=.5
select=not(casewritten)
~output file2c #3 select=not(casewritten)
~exc do_subset
~end
```

**NOTE:** In the above example file2a will contain 33 (.33333 \* 100) records, file2b will contain 34 (67 unwritten records, times .5, and rounded to the nearest whole number), and file2c will contain the 33 as yet unwritten records. If "num\_sample\_cases" does not appear on the second ~output statement and error will result.

See sub4.spx for an example

### **NUM\_SAMPLE\_CASES=**

In order to pull a random sample from an existing sample, the number of cases in the existing sample needs to be known. For example, if you wanted five cases out of 10,000 you would want the random cases to be pulled from random locations throughout the file, not just the beginning, middle, or end of the original file.

Sometimes it's very easy to determine the number of cases in a file. For example, CfMC system files contain the number of cases in their header, and MPE ascii files contain the number of records in their file label. To determine the number of records in a variable length ascii file on Windows or Unix, a pass must be made through the data to count the records. On relatively small input files (<10,000) this counting pass is nearly imperceptible in terms of run time, but on very large samples it may increase run times noticeably. If the input/output files contain select/sample options, this further complicates determining the number of cases available from which to sample and may show a corresponding increase in run times. Setting num\_sample\_cases= will cause the subsetting process to use this

setting as the number of cases from which to draw the sample, and cause the program to not execute passes through the data to determine the number of cases available to be sampled from. This should decrease run times for very large samples, however, if the number provided via num\_sample\_cases is not correct an error will be generated.

### REPEATABLE SUBSET RESULTS

Sampling= picks a random sample from the data. Normally, each time a subseted output file is created a somewhat different collection of records will be output. By using >random\_seed= one can force the starting point of the randomizing process, and thus make it possible to repeatedly create the same "random" sample of records. Keep in mind that adding or subtracting a step in your spec file that calls a random number will change the results of subsequent random calls even if a random seed is set at the beginning of the run.

## 3.3 Mentor EQUIVALENTS TO SPL

Mentor is currently written in the C programming language. The earlier version of Mentor was written in the SPL programming language. The commands and syntax are different between the two versions. If you are already familiar with SPL commands and syntax, here are some of those commands and the current Mentor equivalents. Mentor equivalents to SPL's ADD, ZAP, MOVE, CLEAR, ONTO, INTO, ZPUT are as follows:

**ADD** statement adds punches specified in a mask to a variable.

SPL syntax:           ADD [201] \XY\

Mentor syntax:       MAKE\_DATA + [201^X, Y]

**ZAP** statement removes punches specified in a mask from a variable.

SPL syntax:        ZAP [201] \X\

Mentor syntax:    MAKE\_DATA - [201^X]

**MOVE** statement copies data from one location into a specified receiving location after blanking the receiving location.

SPL syntax:        MOVE [201.4] = "ABCD"

Mentor syntax:    TRANSFER [201.4\$] = "ABCD"

SPL syntax:        MOVE [205.4] = "1234"

Mentor syntax:    TRANSFER [205.4] = 1234

SPL syntax:        MOVE [209.2,211.2] FROM [207.2,205.2]

Mentor syntax:    TRANSFER [209.2,211.2] = [207.2,205.2]

**CLEAR** statement blanks the column(s) specified in a location.

SPL syntax:        CLEAR [201.12]

Mentor syntax:    BLANK [201.12]

**ONTO** statement moves a value from a sending location to a receiving location, always blanking the receiving location first.

SPL syntax:        ONTO [201] PUNCH \1.4\ [43.2#11/12/21/22]

Mentor syntax: TRANSFER [201^1//4] = [43.2#11/12/21/22]

SPL syntax: ONTO [201] PUNCH \1.5\ [88, ..., 92\*F^1//5]

Mentor syntax: REPEAT \$COL=88, ..., 92  
TRANSFER [201^1//5] += [\$COL^1//5]  
>END\_REPEAT

SPL syntax: ONTO [201.2] ZPUT [57.2]

Mentor syntax: TRANSFER [201.2\*Z] = [57.2]

SPL syntax: ONTO [203.2] PUT [43] + [44] + [45]

Mentor syntax: TRANSFER [203.2] = [43] + [44] + [45]

SPL syntax: ONTO [205.2] PUT SUM([68] WITH [69])

Mentor syntax: TRANSFER [205.2] = [68] ++ [69]

(See “3.1.4 Data Manipulation for Punch, String, and Numeric Variables”,  
“Arithmetic Calculations” for additional information on the “++” syntax).

SPL syntax: ONTO [10.3] ZPUT [43] + [44] + [45]

Mentor syntax: TRANSFER [10.3\*Z] = [43] + [44] + [45]

SPL syntax: ONTO [10.6] \1\ DPUT [44] / [45]

Mentor syntax: TRANSFER [10.6\*D1] = [44] / [45]

SPL syntax:        ONTO [10.6] \1\ DZPUT [44] / [45]

Mentor syntax:    TRANSFER [10.6\*ZD1] = [44] / [45]

**INTO** statement with the keyword **SPREAD** spreads out multi-punched data into multiple single-punched fields.

SPL syntax:        INTO [6.8] SPREAD [6]

Mentor syntax:    TRANSFER [6.8\$] = [6\$P]



# BASIC TABLES

## INTRODUCTION

**T**his chapter describes how to create basic tables after you have collected and cleaned your data. It covers information on using simple table options. Chapters 5 and 6 provide more detailed information on how to use intermediate and advanced table functions.

The end of this chapter provides a section on E-Tabs. For more information, go to “USING E-TABS”.

### 4.1 PARTS OF A TABLE

Table is short for cross-tabulation. A table is a rectangular arrangement of columns and rows with values in the intersecting cells. Tables can be simple cross-tabulations of one variable against another, or can be complicated and include expressions that join several variables together and include statistical calculations.

Tables generally consist of a **header**, **title**, **banner**, **stub**, **footer**, and the “**Total**” and “**No Answer**” rows and/or **columns**. These elements are defined in the following section.

**Header:** Optional text that appears on the top of the page on all tables. By convention, it is the name of the study.

**Title:** The survey question text.

**Banner:** The headings of each of the columns in the table. These headings are referred to as “banner points.” In the simplest table, the banner could be only one column labeled “Total.” Usually, banners consist of several column headings. By convention, banner points are

**BASIC TABLES**  
**4.1 PARTS OF A TABLE**

demographics (for example, age or income) or some other characteristic that distinguish groups of people that answer a survey question.

**Stubs:** The labels for each of the rows in the table. By convention, stubs are the responses to the question in the survey.

**Total:** The "Total" row/column is the number of respondents to the survey.

**No Answer:** The "No Answer" row is the number of respondents that did not answer this particular question.

**Footer:** An optional text line at the bottom of the table.

Following is a typical table with the table elements labeled.

ROADRUNNER'S PIZZA SURVEY **Header**

TABLE 001

Q1. How much do you agree with the following statement: **Title**  
 The fast food at Road Runners is worth what I pay for it.

		<b>Total</b>					
		<b>Column</b> <-----AGE----->			<-----INCOME----->		
<b>Banner</b>		Under		Over	Under	\$15-	Over
	<b>TOTAL</b>	35	35-54	54	\$15k	\$35k	\$35k
	-----	-----	-----	-----	-----	-----	-----
Total	<b>Total Row</b>	500	141	140	143	74	215
		100%	28%	28%	29%	15%	43%

		100%	100%	100%	100%	100%	100%	100%
No Answer	<b>No Answer Row</b>	75	29	18	22	16	18	29
		100%	39%	24%	29%	21%	24%	39%
		15%	21%	13%	15%	22%	12%	13%
(5) Completely agree		88	21	29	23	10	30	36
		100%	24%	33%	26%	11%	34%	41%
		18%	15%	21%	16%	14%	20%	17%
(4) Somewhat agree	<b>Stubs</b>	92	26	27	27	14	30	35
		100%	28%	29%	29%	15%	33%	38%
		18%	18%	19%	19%	19%	20%	16%
(3) Neither agree nor disagree		86	23	26	24	13	22	45
		100%	27%	30%	28%	15%	26%	52%
		17%	16%	19%	17%	18%	15%	21%
(2) Somewhat disagree		73	13	23	21	13	21	33
		100%	18%	32%	29%	18%	29%	45%
		15%	9%	16%	15%	18%	14%	15%
(1) Completely disagree		86	29	17	26	8	27	37
		100%	34%	20%	30%	9%	31%	43%
		17%	21%	12%	18%	11%	18%	17%

## 4.2 TABLE BUILDING BASICS

In order to create a table, you must define table elements, identify where to get data from, and issue commands to build and print the table. These instructions are contained in a specification file (with the extension of spx). The main commands are:

~DEFINE    this tells Mentor what information you want in the table and how you want it formatted.

~INPUT     this tells Mentor what it needs to know about the data file.

~EXECUTE   this gives Mentor the commands to build and print the table.

As you can see, these Mentor commands are preceded by a tilde (~). Most tilde commands start command blocks; that is, once the tilde command is given, the commands following it are specific to that block.

If you have CfMC's Survent software, much of the table specification work may have already been done for you. For further information, see *"4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES"*.

Here is a specification file to build a simple table.

~DEFINE

```
TABLE_SET={example:
  COLUMN=: Total
  ROW=: [1^1/2]
  TABLE=*
}
```

~INPUT \$



```
~EXECUTE
  TABLE_SET=example

~END
```

Here is the same spec file with each line explained.

```
~DEFINE                                <--start the DEFINE block
  TABLE_SET={example:                 <--name a set of table elements "example"
  COLUMN=: Total                       <--set up a column that consists of the total
                                       number of respondents
  ROW=: [1^1/2]                        <--set up two rows from answers one and two
  TABLE=*                             <--save table information in memory
  }                                     <--end the DEFINE block

~INPUT $                               <--$ use a phantom file (with one dummy case)
                                       as input

~EXECUTE                                <--start the EXECUTE block
  TABLE_SET=example                  <--build the table with the elements defined in
                                       table set "example"

~END                                    <--exit Mentor
```

If you put these specifications into a file and execute them (`Mentor file.spx -output.lfl`), you can look at the output file to see Mentor has built a small table that looks like this:

```
TABLE 001
BANNER: TOTAL
STUB:   example_r[1]
```

## BASIC TABLES

### 4.3 DEFINING TABLE ELEMENTS

	Total	N/A	TOTAL
Total	1 100.0%	-	1 100.0%
N/A	1 100.0%	-	1 100.0%
1^1	-	-	-
1^2	-	-	-

There are two columns labeled "Total" because by default, Mentor includes a system generated Total column. Tables also have by default a Total row, a N/A (No Answer) row, and a N/A (No Answer) column. You can use the ~DEFINE EDIT keyword to remove these system generated Total columns and change things like column width. This is explained in detail in “4.3.2 Changing Table Element Defaults (The DEFINE EDIT Statement)”.

## 4.3 DEFINING TABLE ELEMENTS

You can use DEFINE to define each element of a table and then call each element separately in the EXECUTE block. Or, you can use the TABLE\_SET keyword to define major elements of a table as a group and then call the group of table elements with one EXECUTE statement.

TABLE\_SET is the basic building block for creating tables. Once an item is defined or turned on, it stays in effect until it is redefined or turned off. This allows you to define table elements globally in one TABLE\_SET, and then only define what needs to change for each particular table in subsequent TABLE\_SETs. Typically, you will use one TABLE\_SET to define the banner text, the column variable, and table printing options which will be used across all of the tables in your study, and one TABLE\_SET for each question in the study, which includes the title, stubs, and row variables.



Let's use the same specification file as before, but this time define the banner once for two tables in a banner table set. Since the TABLE\_SET definition for the banner only defines part of a table, it does not need a TABLE= command. We are also adding a the >PRINTFILE command to send the tables to a separate file (see "4.5 META COMMANDS"). If you are trying these specs, look for the final tables in the file called "mytables.prt".

## BASIC TABLES

### 4.3 DEFINING TABLE ELEMENTS

```
>PRINTFILE mytables
```

```
~DEFINE
```

```
TABLE_SET={one:
```

```
  HEADER=: These Are My Tables  }
```

```
  BANNER=:
```

```
      Total      Male      Female  
      - - - - -  - - - - -  - - - - -  }
```

```
  COLUMN=: Total WITH [2^1] WITH [2^2]
```

```
  }
```

```
TABLE_SET={two:
```

```
  TITLE={:
```

```
  How much do you like building tables?  }
```

```
  STUB={:
```

```
    3-Very Much
```

```
    2-Somewhat
```

```
    1-Not Much
```

```
    9-Don't Know  }
```

```
  ROW=: [12^3/2/1/9]
```

```
  TABLE=*
```

```
  }
```

```
TABLE_SET={three:
```

```
  TITLE={:
```

```
  How much do you like Roadrunner's Pizza?  }
```

```
  STUB={:
```

```
    3-Very Much
```

```
    2-Somewhat
```

```
    1-Not Much
```



```

          9-Don't Know  }
ROW=: [13^3/2/1/9]
TABLE=*
}

~INPUT $

~EXECUTE
TABLE_SET=one
TABLE_SET=two
TABLE_SET=three

~END

```

If you run these specifications and look at the tables, you will see the column headings (these are called banner points) do not line up exactly over the numbers in the columns. You can learn how to correct this in “4.8 *FORMATTING BANNER TEXT*”.

You can also use TABLE\_SETs to do a global screening of the data, such as filters, bases, or weights. By convention, that tabset is named global. You can also include additional rows by adding a stub prefix or stub suffix. Examples of banner table sets will follow. If you have more than one banner, you can create a separate TABLE\_SET that will have definitions for the elements that will be the same in all the tables, such as headers and footers. (See “5.4 *Printing Multiple Banners For Each Table Row*” for an example of printing all the tables by more than one banner.)

Items that are defined in a TABLE\_SET stay in effect for the following tables until they are turned off or redefined, or unless you use one of the ~SET DROP keywords that instruct Mentor to only use an element once and then discard it. See “4.3.3 *Changing Table Processing Defaults (The SET Statement)*” for a complete list of the ~SET DROP keywords.

### 4.3.1 Assigning Variable Names

Mentor allows you to assign a name to any table element: a row, a banner, or an entire table set. You can then refer to the table element (or group of elements) by just using the name. This means you can use a table element exactly as before (such as a stub label set), or the same as before with minor modifications.

Here is the syntax for the TABLE\_SET command, as you can see, you can assign variables to individual table elements, the entire tabset, or both.

```
TABLE_SET={varname=oldname:
KEYWORD=varname: definition
}
```

TABLE_SET	The keyword that tells ~DEFINE that a table set definition follows. It can be abbreviated TABSET.
{	Marks the beginning of the TABLE_SET definition, is optional.
varname	Assigns a name to the TABLE_SET. The name can be 1-14 characters long, and can include periods or underscores. The maximum recommended length is 10 characters, because Mentor will use this name and add extensions to it to create table element names automatically. See “ <i>Default Varname Generation</i> ” for details.
=oldname	Optional, means use the table elements from a previously defined table set and add any new elements in this table set.
KEYWORD=	Commands for defining table elements (such as TITLE, HEADER, STUB, or ROW). These elements stay in effect until they are redefined or turned off. You can turn off

keywords by using a minus sign in front of it, for example, -HEADER. All of the table element keywords are listed and described in Mentor, *Appendix B*, ~EXECUTE.

varname	Refers to an existing variable or assigns a variable name to the keyword (this overrides Mentor's default variable names). The colon is required to assign a variable name, for example, "HEADER=mainhead" refers to the previously defined variable mainhead, while "HEADER=mainhead:" assigns the name mainhead to the heading which is about to be defined.
definition	The definition of the keyword, such as table text or information about the data to be printed in the table cells. Keywords that define table text (such as BANNER and FOOTER) and GLOBAL_EDIT, EDIT and LOCAL_EDIT require a closing brace(}) to end the keyword definition. Keywords that describe data (such as COLUMN, ROW, and STUB) do not require a closing brace (}).
}	Ends the TABLE_SET definition.

Keywords do not need to be in any particular order, but a table set can contain only one occurrence of a keyword. For example, you cannot assign a HEADER and then turn it off with -HEADER in the same table set. Variables can be assigned outside the TABLE\_SET block, but then you must instruct Mentor to make it a part of the table in the ~EXECUTE block.

So, for the previous example, instead of repeating the stub definition in the third tabset (TABLE\_SET=three), you could have assigned a variable to the stub (STUB=mystub:) when you first defined it, and then use the variable name in the third tabset with the command STUB=mystub.

```
~DEFINE
TABLE_SET={two:
TITLE={:
```

## BASIC TABLES

### 4.3 DEFINING TABLE ELEMENTS

```
How much do you like building tables?  }
STUB={mystub:
      3-Very Much
      2-Somewhat
      1-Not Much
      9-Don't Know }
ROW=: [12^3/2/1/9]
TABLE=*
}
```

```
TABLE_SET={three:
  TITLE={:
  How much do you like Roadrunner's Pizza?  }
  STUB=mystub
  ROW=: [13^3/2/1/9]
  TABLE=*
}
```

Remember, if you follow the command with a colon ("STUB=mystub:"), Mentor assumes you are redefining the variable, rather than referring to an existing definition.

Variables (and tables) can be stored and called in later runs, see “4.5 *META COMMANDS*”.

## DEFAULT VARNAME GENERATION

If you do not assign a variable name to a keyword, Mentor will generate default variable names by adding an extension to the table set name. For example, a banner from the table set named "example" would have the variable name "example\_bn". Below is a partial list of the default variable names.

BANNER=*	_bn	ROW_SHORT_WEIGHT=	_rsw
BASE=*	_b	ROW_WEIGHT=	_rw
COLUMN=	_c	STUB=*	_s
COLUMN_SHORT_WEIGHT=	_csw	STUB_PREFACE*	_sp
COLUMN_WEIGHT=	_cw	STUB_SUFFIX*	_sx
EDIT=*	_e	STATISTICS=	_st
FILTER=*	_f	TITLE=*	_t
FOOTER=*	_fo	TITLE_2=*	_t2
HEADER=*	_h	TITLE_4=*	_t4
LOCAL_EDIT=*	_le	TITLE_5=*	_t5
ROW=	_r	WEIGHT=	_w
BANNER_TITLE=*	_bt	FILTER_TITLE=*	_ft

\* means that the keyword requires a closing right brace (}) and, except for EDIT and LOCAL\_EDIT, all of these keywords define table text. All others define the actual data (i.e., type, categories, statistical calculations) to be printed in the table cells.

**NOTE:** Not having a closing brace is common error in spec files. Be sure that your BANNER, EDIT, STUB and TITLE definitions include a closing brace.

Maximum name length is 14 characters *including* the extension. (If your tabset names are ten characters or less, you will not have a problem with the length of your table element names.) This is not an exhaustive list of all the keywords or commands that can be specified inside the TABLE\_SET structure, but these are the ones that either specify a table element or print format control.

### 4.3.2 Changing Table Element Defaults (The DEFINE EDIT Statement)

## BASIC TABLES

### 4.3 DEFINING TABLE ELEMENTS

Below is a list of Mentor defaults for printed tables. The EDIT statement is what you use to change any of these defaults. There are several options available to control what and how table elements are printed on a table. For a complete list, see “5.3 Changing Table Specifications” or *MENTOR APPENDIX B: TILDE COMMANDS, ~DEFINE EDIT*.

System Total row and column

System No Answer row and column

Column width: 8 spaces

Stub (row label) width: 20 spaces

Frequencies with no decimal places

Frequencies with a value of zero print as a dash (-) in the cell

Vertical percents off the Total row (no horizontal percents)  
to 1 decimal point. Percent sign (%) prints.

Page length = 60 lines

Page width = 132 columns

No table of contents printed

Below is an edit statement that changes some of the default values. It has been given the variable name “defedt,” so this set of options can be referenced later. A comma and/or space(s) separate the options.



**Example:**

```
EDIT= {defedt: -COLUMN_NA, -ROW_NA, COLUMN_WIDTH=7,
RUNNING_LINES=1, TCON }
```

EDIT=	The ~DEFINE keyword used to specify table and page formatting controls.
{	Left brace marks the beginning of the definition. (optional)
defedt :	The name of this definition. (optional if defined within a TABLE_SET structure).
-COLUMN_NA	Suppresses the printing of the default system-generated No Answer summary column.
-ROW_NA	Suppresses the printing of the default system-generated No Answer summary row.
COLUMN_WIDTH=7	Controls the width for all table columns; the default is eight. See also COLUMN_INFO= to control column by column and STUB_WIDTH= to control row width.
RUNNING_LINES=1	Prints the table title lines continually across the page up to the specified page width (default is 132).
TCON	Creates a table of contents that, by default, includes: the table name, all titles, the header and footer, and tcon page numbers. Refer to the keywords listed for ~DEFINE EDIT=TCON in <i>Appendix B: TILDE COMMANDS</i> to change these defaults.
}	Ends the definition.

## THE THREE LEVELS OF EDIT

Mentor provides three levels of the edit statement. They are GLOBAL\_EDIT, EDIT, and LOCAL\_EDIT. GLOBAL\_EDIT settings supersede any previous GLOBAL\_EDIT, EDIT, or LOCAL\_EDIT statements. Items are defined in an EDIT statement and stay in effect until a LOCAL\_EDIT or another EDIT statement changes or turns off the item, LOCAL\_EDIT options stay in effect for succeeding tables unless specifically overridden. If you want to use a LOCAL\_EDIT option for just one table, use the SET DROP\_LOCAL\_EDIT command (see the next section on the SET DROP options). To return all EDIT settings to the defaults, use GLOBAL\_EDIT=.

With these three levels of control, it is suggested that you do the following: use GLOBAL\_EDIT to set the conditions which are true for all the tables in a single Mentor run; use EDIT for banner tabsets; and use LOCAL\_EDIT for special features, such as statistics, ranking and zero suppression on stubs.

All three EDITs can be defined within a tabset, or they can be defined in a ~DEFINE block before a tabset. Use EDIT in the ~DEFINE block, and then call the settings in with GLOBAL\_EDIT, EDIT, or LOCAL\_EDIT in the tabset. Below is an example.

```

EDIT={globex:
    PUTCHARS=-----
    RANKD_IF_INDICATED
    PERCENT_SIGN
    PDEC=2
}

TABSET={sample:
    HEADER={:=
        Table Heading for a Great Table
    }
    FOOTER={:=
        Wonderful Client and Associates
    }
}

```



```

    }
    GLOBAL_EDIT=globex
  }

```

Here is a list of frequently used LOCAL\_EDIT options:

```

ALL_POSSIBLE_PAIRS_TEST
ANOVA
ANOVA_SCAN
CHI_SQUARE
COLUMN_MEAN
COLUMN_MEDIAN
COLUMN_SE
COLUMN_SIGMA
COLUMN_STD
COLUMN_STATISTICS_VALUES
DO_STATISTICS
FISHER
LEAVE_PAGE_OPEN
LEAVE_TABLE_OPEN
NEWMAN_KEULS_TEST
USE_RANK_INFO

```

See *Mentor, Appendix B, ~DEFINE EDIT* for detail on each of these options.

### 4.3.3 Changing Table Processing Defaults (The SET Statement)

The SET statement controls table-processing defaults for the entire run. SET statements can be located at the beginning of the specification file, within table sets, or inside the EXECUTE block. Put SET statements at the beginning of a specification file for settings you want to be in effect for an entire run. SET statements can also be specified inside the ~EXECUTE block so they can be turned off and on to control processing for individual tables. Or, SET commands can be included in the definition for a table in combination with one of the SET DROP commands to drop the corresponding element(s) after the table set is used. See below for a list of the SET drop commands.

## BASIC TABLES

### 4.3 DEFINING TABLE ELEMENTS

From the ~SET block you control Mentor's processing defaults. Default settings are listed under ~SET in *Appendix B: TILDE COMMANDS*. In general, these controls are set for the entire run, but it is possible to specify the SET commands in a TABLE\_SET or from the ~EXECUTE block. The first SET command you will probably use is AUTOMATIC\_TABLES (abbreviated SET AUTOTAB). This automatically builds a table when Mentor encounters the ROW keyword, rather than requiring a TABLE or STORE command in each tabset. For details, see “4.4 TABLE BUILDING (The INPUT and EXECUTE statements)”.

Mentor retains table elements, such as titling, as a default until they are redefined or turned off. Keywords can be turned off by using -KEYWORD or KEYWORD=;. Since table elements remain set until they are specifically turned off, this can cause unwanted elements to show up in subsequent tables. To prevent this from happening, you can use the following ~SET DROP commands, which causes Mentor to use a table element for one table only and then discard it. This can be reversed, meaning table elements, once defined, will once again be carried to subsequent tables, by using the SET command within a tabset or ~EXECUTE block with the DROP command with a minus sign in front of it, for example, SET -DROP\_BANNER.

Here is the list of the SET DROP commands:

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\_PREFACE  
DROP\_STUB\_SUFFIX  
DROP\_TABLE\_SET  
DROP\_TITLE  
DROP\_TITLE\_2  
DROP\_TITLE\_4  
DROP\_TITLE\_5  
DROP\_WEIGHT

See *Mentor, Volume II, Appendix B, ~SET* for definitions of each of the DROP commands.

~SET TABLE\_EFFORT controls what is put into the tables. It can save you time by building tables without actual data, allowing you to find processing or syntactical errors in your spec file. Here are the options for TABLE\_EFFORT:

- |                |   |
|----------------|---|
| TABLE_EFFORT=1 | Makes tables with data from the file specified in the INPUT statement (DEFAULT).  |
| TABLE_EFFORT=3 | Makes zero-filled tables, allowing you to preview tables without processing the data file. In addition to formatting problems and specification syntax errors, errors in table size will be apparent using TABLE_EFFORT=3. You might want to test run your specifications with this setting before running a large job. |
| TABLE_EFFORT=4 | Processes only the specification file and prints a table of contents. Tables are not made and the data file is not read. Use this setting to check specification syntax only.   |

## BASIC TABLES

### 4.4 TABLE BUILDING (The INPUT and EXECUTE statements)

TABLE\_EFFORT=5                      Makes zero-filled tables, and shows all of the elements in effect on the printed table for debugging purposes. This includes all default values being used, the column and row variables, the category descriptions, and the stubs, banner, and data locations used to calculate statistics.

### 4.4 TABLE BUILDING (The INPUT and EXECUTE statements)

~INPUT opens the data file. Mentor assumes the filename has an extension of TR, because that is the default extension for CfMC data files (also called a System File). You can eliminate the expectation for the TR by preceding the file name with a dollar sign (\$) or with the META command >-CFMC\_FILE\_EXTENSIONS. The INPUT command has several options, which allow you to use different data file types or use only selected cases from the data file. See *Appendix B: TILDE COMMANDS*, ~INPUT for a complete list of options. If you need to copy or reformat your data file, see the COPYFILE utility in the *Utilities* manual.

~EXECUTE is the command that triggers the construction and printing of tables. For maximum flexibility and control, Mentor provides several ways to build tables. This allows you to make table creation and printing separate procedures, for example, if you want to run the tables and print them later. See ~EXECUTE in the *Mentor Appendix B* for all of the options.

It is possible to define table elements or tabsets directly in the EXECUTE block, but available memory limits the size of some variables (such as the column or row) that you can define, and you cannot save any elements using this method. You could also define each table element in the DEFINE block, and call each item in the EXECUTE block separately. For example,

~DEFINE

    ROW=R1: [12^3/2/1/9]

    COLUMN=C1: Total WITH [2^1] WITH [2^2]

~EXECUTE

    ROW=R1

COLUMN=C1

This method, while straightforward, requires lots of typing and would result in huge specification files. It is much more efficient to define tables in tabsets and then call them in the EXECUTE block. If you are building only a few tables, you can store tabsets in memory and call them individually in the EXECUTE block (TABLE\_SET or TABLE=). It is more efficient to have all the tables built at once, and then printed, you can do this and you can also store tabsets in a db file so you can print them later (STORE\_TABLES=). Or, you can print all of the tables created in the run with one command (MAKE\_TABLES). Below are examples of each of these methods.

### PRINTING INDIVIDUAL TABLES (USING TABLE\_SET OR TABLE=)

An easy way to define and build tables is to define table elements as a group in a table set, and then store those elements in memory. In the table set, use the TABLE= command. The TABLE= command in the DEFINE block stores the table elements and data in memory, and then the TABLE\_SET command in the EXECUTE block causes the table to be built and printed. This method is best for a building a single table or a few tables because it only makes one table for each pass through the data, which would be very slow and inefficient for many tables. This is the method used in the first two examples in this chapter:

```
~DEFINE
  TABLE_SET={example:
    COLUMN=: Total
    ROW=: [1^1/2]
    TABLE=*
  }
```

```
~INPUT $
```

```
~EXECUTE
```

## BASIC TABLES

### 4.4 TABLE BUILDING (The INPUT and EXECUTE statements)

TABLE\_SET=example

You can use TABLE= in the EXECUTE block instead of TABLE\_SET. TABLE= causes Mentor to build a table using the current table elements and then print the table immediately. TABLE= can specify a specific table name or asterisk (\*) to mean use the current table name and increment by one. The first tabset has the default table name T001, and tabset two would have the name T002. (You can use the SET command TABLE\_NAME to specify the current table name.)

### STORING TABSETS IN THE DB FILE (USING STORE\_TABLES)

Instead of the TABLE= command, you can use STORE\_TABLES (abbreviated STORE). STORE\_TABLES saves table specifications in memory until memory is full and then stores tables in a DB file. This is faster than TABLE\_SET or TABLE= because it processes the data in the tables first and then prints them. Unlike TABLE, tables are not automatically printed, so you must include a printing command in the EXECUTE block.

#### Example:

```
>CREATE_DB tabstuff
```

```
>PRINTFILE mytables
```

```
~DEFINE
```

```
TABLE_SET={one:
  HEADER=: These Are My Tables  }
  BANNER=:
      Total      Male      Female
  -----      -
  COLUMN=: Total WITH [2^1] WITH [2^2]
}
```

```
TABLE_SET={two:
  TITLE=:
  How much do you like building tables?  }
  STUB={mystub:
      3-Very Much
```

```

        2-Somewhat
        1-Not Much
        9-Don't Know }
ROW=: [12^3/2/1/9]
STORE=*
}

```

```

TABLE_SET={three:
  TITLE={:
  How much do you like Roadrunner's Pizza? }
  STUB={mystub }
  ROW=: [13^3/2/1/9]
  STORE=*
}

```

```
~INPUT $
```

```
~EXECUTE
```

```

TABLE_SET=one
TABLE_SET=two
TABLE_SET=three
PRINT_ALL

```

```
~END
```

The advantage of saving tables to the db file is that you can print tables without having to construct them again. This is useful if you have to make simple changes to the table labels (see 5.6 *REPRINTING TABLES* for how you can reprint tables that have been stored in a DB file from a previous run) or for printing a single table out of a large group of tables.

With either `TABLE=` or `STORE_TABLE=`, Mentor stores table elements in memory before making a pass through the data to build the tables. The number of tables that can be made in a single pass depends on the amount of memory available. This is either the default for your machine or what you set memory to on the `CORE` option from the command line. Refer to your *Utilities manual Appendix D: CfMC CONVENTIONS, Command Line Keywords* for more information on setting core memory higher to fit more tables per data pass.

## BASIC TABLES

### 4.4 TABLE BUILDING (The INPUT and EXECUTE statements)

Instead of using the STORE=\* command in every tabset, you can use the ~SET AUTOMATIC\_TABLES command. This causes Mentor to store tables every time it encounters the ROW keyword. (Be careful, if you include both STORE\_TABLES or TABLE in a TABLE\_SET when you have also specified SET AUTOMATIC\_TABLES, two tables will be built for each ROW keyword!)

#### Example:

```
~SET AUTOMATIC_TABLES
```

```
~DEFINE
```

```
TABLE_SET={one:
```

```
Header=: These Are My Tables }
```

```
Banner=:
```

```
      Total      Male      Female  
      - - - - -      - - - - -      - - - - - }
```

```
Column=: Total WITH [2^1] WITH [2^2]
```

```
}
```

```
TABLE_SET={two:
```

```
Title={:
```

```
How much do you like building tables? }
```

```
Stub={mystub:
```

```
3-Very Much
```

```
2-Somewhat
```

```
1-Not Much
```

```
9-Don't Know }
```

```
Row=: [12^3/2/1/9]
```

```
}
```

```
TABLE_SET={three:
```



```

Title={:
How much do you like Roadrunner's Pizza?  }
Stub={mystub  }
Row=: [13^3/2/1/9]
}

~INPUT $

~EXECUTE
TABLE_SET=one
TABLE_SET=two
TABLE_SET=three

PRINT_ALL

~END

```

### MAKING SEVERAL TABLES (USING MAKE\_TABLES)

For a standard run (several tables, a single banner, and sequential table numbers) use the MAKE\_TABLES command in the EXECUTE block. MAKE\_TABLES builds and prints all of the tables defined by reading a TAB file for a list of tables and an LPR file for a list of LOAD and PRINT commands for those tables. If you use the ~SPEC\_FILES command, Mentor will generate the TAB and LPR files for you automatically. (Survent users: If you used ~PREPARE\_COMPILE Mentor\_SPECS to generate a TAB file, ~SPEC\_FILES will overwrite it. If you need to keep the Survent generated TAB file, rename it or move it to another directory.)

(MAKE\_TABLES actually calls the BUILD\_TABLES, RESET and PRINT\_RUN commands. You can use these commands separately to have more control over your run. See *Mentor, Appendix B: ~EXECUTE LOAD\_TABLE, STORE\_TABLE, and PRINT\_ALL.*)

## BASIC TABLES

### 4.4 TABLE BUILDING (The INPUT and EXECUTE statements)

Here is a typical spec files that takes advantage of both SET AUTOMATIC\_TABLES and EXECUTE MAKE\_TABLES.

```
~SPEC_FILES
~SET AUTOMATIC_TABLES

~DEFINE
TABLE_SET={one:
  Header=: These Are My Tables  }
  Banner=:
      Total      Male      Female
      -----   -
      Column=: Total WITH [2^1] WITH [2^2]
  }

TABLE_SET={two:
  Title={:
    How much do you like building tables?  }
  Stub={mystub:
    3-Very Much
    2-Somewhat
    1-Not Much
    Don't Know }
  Row=: [12^3/2/1/9]
  }

TABLE_SET={three:
  Title={:
    How much do you like Roadrunner's Pizza?  }
  Stub={mystub }
  Row=: [13^3/2/1/9]
  }

~INPUT $

~EXECUTE
  MAKE_TABLES
~END
```

## 4.5 META COMMANDS

Meta (>) indicates a command that can be invoked in most CfMC software programs and across tilde blocks. They are used for general programming controls, variable data base access, and specification file control. Here is a description of some of the meta commands you are likely to use. Refer to your *UTILITIES* manual for definitions of all meta commands.

>DEFINE @STUDY sample      >DEFINE defines a keyword that can be used as a substitution for any string. In this case, it is used to give the name of the study at the top of a run, so it can easily be referenced on any command requiring the study name. This allows you to use “@STUDY” on each command line and Mentor will automatically pull the name “sample” into the line. This format is used for the rest of the meta commands in this section.

>PRINT\_FILE @STUDY~      >PRINT\_FILE opens a file to be printed to for tables or procedure output. There are also options to print to the screen, print to multiple files, or control the page size. **NOTE:** The tilde mark (~) acts as a delimiter for user-defined variables. This means you can append letters and/or numbers to the STUDY name, for example, >PRINT\_FILE @STUDY~2 would open the print file SAMPLE2.PRT.

>ALLOW\_INDENT              Allows indentation of meta commands and &filename, otherwise, they must start in the first column of the specification file.

>PURGE\_SAME                Purges existing files with the same name as any newly created file. This is useful if you are repeating runs often and do not want several intermediate files saved. Without

>PURGE\_SAME, Mentor renames the existing files by changing the first character of the file name up one alpha character (for example drop.prt would be renamed erop.prt). **NOTE:** Use >PURGE\_SAME with caution, it deletes existing files with the same name!

## THE DB FILE

A DB file is a machine-readable file which allows you to store variables and tables for fast retrieval by Mentor. By default, Mentor stores variables in a “local” DB file, which disappears after the current run. To store items for a later run, you create the DB file in one spec file, and then make a reference to it in another spec file. Here are the DB commands:

>CREATE\_DB @STUDY~      >CREATE\_DB opens a new data base (DB) file to store any new variables or tables made by Mentor. Later you can refer to this file to reprint built tables or rebuild tables using variables that exist in the DB file. (See the note above about the tilde mark (~) under >PRINT\_FILE.)

>USE\_DB @STUDY~      >USE\_DB opens a data base (DB) file containing previously defined Mentor or Survent variables or tables. See “4.11 SAMPLE SPECIFICATION FILES”, for an example of using the DB file for table elements. Two ampersands (&&) followed by a name will store the item in a data base (DB) file and then execute it. The file or data base item can include executable program commands and syntax suitable for the current tilde block, or it may include other tilde commands or meta commands. (See the note above about the tilde mark (~) under >PRINT\_FILE.)

## 4.6 DEFINING DATA

Data definitions tell Mentor where to find the data to tabulate, what type of data it is, and how it is organized. You can use data definitions to define the table's vertical and horizontal axes, exclude respondents from a table (this is known as a base), or give more weight to some data. Use these data definitions in your COLUMN and ROW statements.

You can join two or more variables together to form expressions. Expressions create new data categories, and you can use them to: perform across-case statistical calculations; execute functions to change or manipulate the data in specific ways; or, create special kinds of tables (such as break and overlay tables).

This section covers the rules for creating simple data definitions. In “4.6.1 *Summary of Rules for Defining Data*”, there are several examples illustrating the syntax for data field locations and category definitions. The next section, “4.7 *DEFINING THE BANNER*”, has an example of a complex table banner using the WITH joiner. Joiners and expressions are covered in detail in section “5.1 *Expressions and Joiners*”.

A data definition is usually enclosed in brackets [ ]. It can be assigned a name for future reference or as you saw earlier Mentor can generate default variable names derived from the TABLE\_SET name and the table element keyword. The data definition can also include text that will print as a title any time the variable is referenced in a table or text to label each data category (i.e., stub labels). Optional data modifiers can be used to change how data categories are tabulated.

Variables are defined with the keyword VARIABLE, and can be used in data modification, data reporting, or tables. To create cross-case statistics or need special table building controls, use the AXIS commands to define the COLUMN, BASE, or ROW. AXIS commands can only be used on tables. Refer to “5.2 *Axis Commands/Cross-Case Operations*”.

**Syntax:** VARIABLE= varname: \$T="text" & [record/col.wid, mod type categories] joiner [record/col.wid, etc.]

VARIABLE= The ~DEFINE keyword to start a variable definition. It is not required since it is the default keyword used by ~DEFINE.

varname The name of the variable; this is required unless you have a simple variable that has a name on its data specification; i.e., varname[5^1]. It can be 1 to 14 alphanumeric characters starting with a letter, and may include periods (.), and underscores (\_).

:

Required only if the variable definition contains any functions or joiners to form a complex variable expression. Remember that within the TABLE\_SET structure keyword=: tells Mentor that this is a new definition and if no varname is specified then generate the default name.

\$T="text"

The title of the variable (optional). The text prints as the title of a table element whenever the variable is referenced and another title is not provided. In a TABLE\_SET, this is the same as defining a TITLE= variable. If \$T is not specified, then the varname prints as the default title if no other title is provided.

Text can be continued onto the next line by placing a double ampersand (&&) immediately after the last quote. Continue the text on the next line by placing it in quotes also.

&

Required at the end of any line within the definition which continues to the next line.

[ ]	Required around each new data element specification, but is not required if all references are to previously named data specifications.
record	The record number for the location of the data. By default CfMC programs report all data locations in the record/col.wid format. (Data locations can be referenced by absolute column as well.)
col.wid	The starting column location and number of columns. If you don't include a width, then one column is assumed. You can also specify the location as column1-column2, where column1 is the first column and column2 the last column. For a multiple column/item specification, use commands to separate column specifications, or an ellipsis (,...) to say 'columns starting here and going to there'. Multiple column specifications create a set of categories for each single column description. Non-consecutive data locations are separated by a comma (.). See "4.6.1 Summary of Rules for Defining Data".
mod	Refers to the variable modifier used to determine how to combine categories in variables referencing multiple columns, or to modify numeric data references. The default is make separate categories for all the data locations. (optional)

Multiple data column references: col1,col2,...,coln:

- \***F** or **FIRST** nets the counts per category across columns (i.e., it counts only the first mention of each category).
- \***L** or **LAST** sums the counts per category across columns (i.e., it sums all mentions for each category).

## BASIC TABLES

### 4.6 DEFINING DATA

Use \*F to count cases (respondents, how many) and use \*L to count things in the cases (family members, how much, or how often). For example, you would use \*F when you want to know how many people bought the products they were asked about, but not how much they bought. If you are asking did they buy it, use it or eat it, use \*F, if you are asking how many did they buy, how much did they eat or how often did they use it, use \*L.





*Numeric data references:*

- ! Return a zero (0) when the location is blank, e.g., [!5/10.2] says if record five columns 10 and 11 are blank then return a zero. Note that the exclamation (!) is prior to the location being referenced, unlike the other modifiers.
  
- \*D# The variable contains decimals, e.g., [1.10\*D1] says this variable has one decimal in the data. If there is not one decimal, the value will be treated as 'Missing' (ignored). The maximum number of decimal places is 14.
  
- \*F# This variable has implied decimals, e.g., [1.10\*F2] says this variable has two implied decimals. If there is an actual decimal in the data, that will be used instead of the implied decimals.
  
- \*RANGES=#-##,a=#,b,c Specifies the numeric range and exception codes of the variable. This is useful to exclude codes in the data from future calculations, such as for Means.  
Exception codes can also be assigned a value to be used in statistical calculations, otherwise they are excluded from stats.
  - #-# range of values to be included in evaluation of the data
  - # numeric value to be excluded from evaluation, but allowed
  - a=# exception code1 to be recoded to this number (#) for evaluation
  - b exception code2 to be counted, but not included in numeric evaluation
  - c exception code3 same as above
  
- \*Z The data contains leading zeros (0), e.g., [10.2\*Z#1-10]. If there is no leading zero, the location will be considered "missing", and will be ignored

**Example:** row1: [15.2#1-50/RF/DK &  
\$(MEAN,STD,SE) [15.2\*RANGES=1-50,,RF=0]

row1	The variable name
:	Required for a complex variable or axis definition
[	Defines the start of a data reference
15.2	The data location, columns 15 and 16
#	Specifies the type of data, ASCII or numeric
1-50	The range of responses allowed for this data category
RF	The literal allowed for this category
DK	Another allowed literal
&	Ampersand continues the definition to the next line

\$(MEAN, STD, SE) The statistical tests that will be performed on the data (mean, standard deviation, and standard error) For more information, see “5.2 Axis Commands/Cross-Case Operations” and Appendix B: TILDE COMMANDS, ~DEFINE AXIS=.

[15.2\*RANGES=1-50,,RF=0] Defines which columns will be tested (15.2) and which values will be included in the evaluation. A value has been assigned to the exception code RF.

Modifiers can be combined in the same variable where it makes sense to do so. Specify only one asterisk (\*) when you combine options.

**Example:** [5.4\*ZD2]

This variable contains leading zeros and two decimals.

## DATA TYPES

`type` The type of data variable, either punch or ASCII. Data type is identified by one of two symbols: caret (^) indicating punch binary codes, pound sign (#) indicating either numeric ranges or ASCII characters. If no type is specified (i.e., just a data location is given with no categories), then data type defaults to numeric and must be a valid number that is right-justified in the field.

### *Punch Data*

Punch Data (^) is stored as 12 punches per column, 1-9, 0, X, and Y. (You can substitute 10, 11, and 12 for 0, X, and Y.) Survent CAT question returns punch data. This example represents punch data found in record 1, column 5, a width of one column, punches 1 and 2:

[1/5^1/2]

Punch data can be any width. For categories wider than one column, punches are represented as their position relative to the first punch of the first column in the data field. For example, the third punch in the second column of a data field would be referenced as 15:

[1/6.2^15]

This example represents punch data found in record 1, column seven, a width of two columns, the third punch:

[1/7.2^3]

^5/4/3/2/1/10 means punches 5,4,3,2,1, and 0 (10th punch) are stored in the data as separate categories in the order shown. The slash (/) acts to separate categories.

You can precede any punch with the letter N to mean "is not these punches," the letter B to mean "is blank" (no punch), or the letter A to mean "has all these

punches." You can combine N with B to mean "is not blank," and N with A to mean "is not all of these punches."

[10^A1,2/N3,4/AN5,6]

This example creates three categories. The first will contain records which have both 1 and 2 punched. The second category will contain records which have neither 3 or 4 punched. The third category will contain records which do not have both 5 and 6 punched.

***ASCII data (#)***

Can be alphabetic, numeric, or special characters (enclosed in quotes), or some combination. The maximum data field width for numbers is 20 and 9 for alphabetic characters. Survent FLD and NUM question types return ASCII data.

[1/55.2#0-3/4-5/6-10/RF]

0-3/4-5/6-10/RF represent the numeric ranges for each data category. RF is an exception code. Codes are not case-sensitive, and quotes are not required.

Quotes are necessary when you want to match special characters (i.e., ? / \*) or blanks. Quotes are also necessary if you want to have Mentor treat numbers as ASCII characters, meaning they must be an exact match (for example, left-justified in the field).

[1/5.4#99]

In this example, 99 is treated as a number and must be right-justified in the field (columns 7 and 8) to match this definition. That means it could be blank or zero-filled in columns 5 and 6 to match ( 99) or (0099).

[1/5.4#"99"]

In the second example "99" will be treated like all other ASCII characters and must be left-justified in the field to match the definition (99 ).

## USING PUNCTUATION TO CREATE CATEGORIES

### *Categories*

The codes or values that describe categories within the data variable. The number of categories defined will determine either the number of stub rows or banner points when tables are made.

You can use a modifier to further determine how categories are represented in a table. By default, when there are multiple column references, the category list will include one category for each column referenced, such that the total number of categories will be the product of the number of column references times the number of category descriptions. If you use the \*F or \*L modifiers, the categories are grouped across the data locations specified, and you will only have one set of categories to describe all of the data locations.

Slash (/) defines categories as single elements, e.g. 1/2/3/4/5 or A10/B52/55D to describe individual punch or ASCII categories.

Dash (-) indicates either netted punch values or a range of ASCII values, e.g., 0-3/4-5/6-10. This means for a respondent to be included in the first category there must be a zero, one, two, or three (punch or value depending on data type) in the data.

Comma (,) also creates netted categories, e.g., 5,4/5/4/3/2/1 meaning values four and five are netted into the first category.

Dashes and commas may be combined within a category.

Double slash (//) means that multiple categories will be made starting from the first category through the last, e.g., 5,4/5//1. This creates six categories where: cat1 nets

## BASIC TABLES

### 4.6 DEFINING DATA

values 4 and 5; cat2 is 5; cat3 is 4; cat4 is 3; cat5 is 2; and cat6 is 1. For punch references, // is always consecutive from the first category to the last. For ASCII categories, the prior two categories to a // determine the range to be used when creating new categories. For example, 1/3//9 would create categories 1,3,5,7, and 9.

Punch data can use the letters N (“not this punch”), A (“not all of these punches”) and B (“is blank”) to create categories. See the punch data definition on a prior page for an example.

You can control when statistical (~DEFINE STATISTICS) tests are to be done on a particular category or group of categories with a plus sign (+) or the keyword (stats) before the category. It 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 specify text on a category by specifying it before the category followed by a colon, e.g., [2/12^Male:1/Female:2]. The text defined here would print whenever the category was printed in data reports, or on tables, either as row stub labels or banner labels. Text must be enclosed in quotes (") when it contains either spaces or special characters, e.g., [2/12^Male:1/"All Females":2].

```
[1/5#1//5]
```

The same as [1/5#1/2/3/4/5] or five categories which are the numbers 1, 2, 3, 4, and 5.

[1/5.2#A1/B2/C3]

Defines three categories which are the ASCII characters A1, B2, and C3.

[1/5^1,2/3,4/5]

Defines three punch categories where punches 1 or 2 are counted in the first category, 3 or 4 in the second, and punch 5 in the third.

[1/5.2^1-5,23,24]

Defines one category that could contain any punch code 1, 2, 3, 4, 5, 23, or 24.

[19.2, ..., 23\*F#1//17])

The variable modifier \*F counts only the first mention across the columns specified, in effect ignoring duplicates by counting only the data categories with a different valid mention. Categories with the same mention are thus netted together. For example, the first category is 19.2#1, 21.2#1, and 23.2#1. \*F nets these together as one mention even if all three fields contain a value of one. This would be reflected in the frequency on the table. On the other hand, the \*L modifier would sum the mentions for a total of three in this case.

## JOINERS

### *Joiner*

A command that joins two or more variables together to form an expression. Joiners can be either logical (true or false) or vector (combines variables to create new categories).

Using what we have learned about data variables we can define a column variable to present the data using two variables from the RRUNR questionnaire, respondent sex and age. For this example we will use the vector joiner WITH to append the categories for respondent sex to the categories for respondent age. This will form a single expression with the categories from each.

[1/57^1/2] WITH [1/51^1,2/3,4/5,6/7]  
Sex                      Age

**NOTE:** We have combined or netted some of the age categories into single categories by separating them with a comma. Comma means add another value to this category. Respondents who answered either 1 or 2 are counted in the first category, either 3 or 4 are counted in the second, and either 5 or 6 as the third age category, 7 being Don't know/Refused.

The joiner WITH creates a total of six categories from these two variables, two respondent sex categories and four age categories. The categories to the left of the joiner print first. You can see this WITH example and the table it creates in “4.8 *FORMATTING BANNER TEXT*”.

### 4.6.1 Summary of Rules for Defining Data

The next few pages provide several examples of how you can define data in Mentor. The first column is a sample data definition and the second and third columns show the actual locations the definition refers to.

#### SAMPLES OF DATA FIELD LOCATIONS

RECORD	COLUMN
NUMBER	NUMBER(S) REFERENCED
-----	-----

##### 1. Single Column Locations



[1/5]	1	5
[5]	(defaults to 1)	5
[7/48]	7	48

## 2. Locations Wider than 1 Column

[1/5-9]	1	5, 6, 7, 8, and 9
[5-9]	(defaults to 1)	5, 6, 7, 8, and 9
[1/2.5]	1	2, 3, 4, 5, and 6
[4/1.3]	4	1, 2, and 3

## 3. Multiple Locations

[1/8,1/9]	1	8 and 9
[4/32,4/35,4/46]	4	32, 35, and 46
[1/16,...,1/20]	1	16, 17, 18, 19, and 20
[5/32,...,5/27]	5	32, 31, 30, 29, 28, and 27
[2/17-18,...,2/25-26]	2	17 & 18; 19 & 20; 21 & 22; 23 & 24; 25 & 26 (i.e., five fields, each two columns wide)
[1/24.3,...,1/33.3]	1	24 & 25 & 26; 27 & 28 & 29; 30 & 31 & 32; 33 & 34 & 35; (i.e., four sets of three columns)

**NOTE:** The last ".3" is not required since once a width is stated, either explicitly (24.3) or by default (8 implies 8.1), it stays in effect for all locations in the same set of brackets.

### **PUNCTUATION USED IN REFERENCING DATA FIELD LOCATIONS**

<b>Single Slash (/):</b>	Used to separate the record number from column number(s).
<b>Period (.):</b>	Used after the leftmost column number of a data field, and followed by a number giving the width of the field (in number of columns).
<b>Comma (,):</b>	Used after a data field location reference to indicate that multiple column fields are being specified.
<b>Ellipsis (...):</b>	Used after a column number in a data field location to abbreviate the listing of a sequence of consecutive locations when multiple column locations are specified.
<b>Square Brackets ([ ]):</b>	Used to enclose a variable definition.

### **CATEGORY DEFINITIONS USING CARET (^) FOR PUNCH DATA**

NUMBER OF CATEGORIES TO CREATE	PUNCH CODES USING THESE PUNCHES
-----	-----

EXAMPLE SET A: For Separate Punch Position Categories

1/12	2	1 and Y
1//12	12	1,2,3,4,5,6,7,8,9,0,X, and Y
1/2/3/11	4	1,2,3, and X
12/11/10/9/8/7/6/5	8	Y,X,0,9,8,7,6, and 5
11//7	5	X,0,9,8, and 7
1/4//9/12	8	1,4,5,6,7,8,9, and Y
13//24 column of the field	12	1,2,3,4,5,6,7,8,9,0,X, and Y in the second
1,4.9/12	2	"1,4,5,6,7,8,9", and Y

EXAMPLE SET B: For Netted Punch Position Categories

1,2,3	1	"1,2,3" (A new, single netted category)
1,2/3/12	3	"1,2",3, and Y
5-12	1	"5,6,7,8,9,0,X,Y"
1/3/4.6/7//12	9	1,3,"4,5,6",7,8,9,0,X, and Y
1-12	1	"1,2,3,4,5,6,7,8,9,0,X,Y"
12-1	1	"Y,X,0,9,8,7,6,5,4,3,2,1"
1,13,25	1	A new single netted category of the 1 punches in first 3 columns of the field.

EXAMPLE SET C: For Negative Punch Position Categories

B	1	Absence of all codes 1-Y
N1	1	Not 1
N1/3/7/9	4	"Not 1",3,7, and 9
6//9/N1-5	5	6,7,8,9, and "NOT 1-5"
NB	1	"1,2,3,4,5,6,7,8,9,0,X,Y" (same as 1-Y)

EXAMPLE SET D: Specifying Locations and Categories for Punch Data

	RECORD NUMBER -----	COLUMN NUMBERS -----	NUMBER OF CATEGORIES -----	PUNCH CODES REFERENCED -----
[1/5^1/2/12]	1	5	3	1,2, and Y
[3/7^1//6]	3	7	6	1,2,3,4,5, and 6
[5^11/7/2/1]	1	5	4	X,7,2, and 1
[5/5.2^01//24]	5	5-6	24	1,2,3,4,5,6,7,8,9,0,X,Y in column 5 and 1,2,3,4,5,6,7,8,9, 0,X,Y in column 6

**PUNCTUATION USED IN DEFINING PUNCH DATA**

**Comma (,):** Used to create a single netted category of the punches' positions listed.

**Dash (-):** Used to indicate the net of all consecutive punch positions between the first punch position specified, and the last punch position specified. (same as period)

<b>Caret (^):</b>	Indicates punch type data, single or multi-column punch field. Note that you can specify the punch codes as a position relative to the first column in the field, e.g., 12 or the actual code Y.
<b>Period (.):</b>	Used to indicate the net of all consecutive punch positions between the first punch position specified, and the last punch position specified. (same as dash)
<b>Slash (/):</b>	Used to indicate separate punch code categories.
<b>Double Slash (//):</b>	Used to indicate a set of separate but consecutive punch code categories.
<b>Square Brackets ([ ]):</b>	Used to enclose a data variable definition.

### CATEGORY DEFINITIONS USING POUND SIGN (#) FOR ASCII AND NUMERIC DATA

#### EXAMPLE SET A: ASCII CATEGORY DEFINITIONS

	Record Number -----	Column Number -----	Number of Cats. -----	Values -----
[8/4.2#17/26/AA/ZZ]	8	4-5	4	17, 26, AA,ZZ
[2/2.3#ABC/999/&&&]	2	2-4	3	ABC, 999,&&&
[6/38.3#190/B27/321/55D]	6	38-40	4	190,B27,321, 55D

**EXAMPLE SET B: NUMERIC CATEGORY DEFINITIONS**

	Record Number -----	Column Number -----	Number of Cats. -----	Values -----
[3/7.3#157/305/872]	3	7-9	3	157, 305,872
[1/1.2#28/77/93,94]	1	1-2	3	28, 77, "93 or 94"
[4/27.2#1//99]	4	27-28	99	99 separate categories
[1/37.2#1-10/11-20/31-40]	1	37-38	3	The range 1-10, the range 11-20, the range 31-40.
[1/10.5#1.99-10.99/RF]	1	10-14	2	The range 1.99 - 10.99, the exception code RF
[1/5.2]	1	5-6	1	All real numbers* in this location. *Meaning any positive, negative, or decimal number.

**PUNCTUATION USED IN DEFINING ASCII AND NUMERIC DATA**

- Pound Sign (#):** Used to indicate Numeric or ASCII type data.
- Dash (-):** Used as in "n-m", to represent a range of values from n to m.
- Comma (,):** Used within a category definition to separate alternative values.



**Quotation Marks ("):** Used to set off and identify special characters such as literals, including blank spaces.

## 4.7 DEFINING THE BANNER

Printed tables have two main elements, a banner and a stub. Since a banner normally includes the variable(s) you want on all of your tables, you can make one banner definition and just change the stubs. Generally you will use one TABLE\_SET to combine the elements of each banner (including the banner text, the column variable, the table printing (EDIT) options), and one TABLE\_SET for each of the stubs (including the titles, stub labels, bases, and row variables).

The banner TABLE\_SET can also include an overall filter variable for the set of tables (BASE or FILTER), a definition to control what prints at the top or bottom of each table (EDIT STUB\_PREFACE or STUB\_SUFFIX), a weight variable to give weights to the cases included in the tables, or specifications to choose the type of statistics to do and which columns to be included in the comparisons (see *Appendix B: TILDE COMMANDS, ~DEFINE STATISTICS* and *~EXECUTE COLUMN\_WEIGHT*).

Stub TABLE\_SETs can include items to control specific table printing needs (LOCAL\_EDIT), a variable to filter respondents (BASE), base titling (TITLE\_4), and row weighting (ROW\_WEIGHT). You can use the SET command to turn on or off certain production controls to affect groups of tables, such as whether to drop items after one table, or how to assign table names.

If you have more than one banner definition, then you can use a global TABLE\_SET to define such things as the header and footer to print on the tables, global SET commands, or a main EDIT statement for table building and printing controls. Here is a typical TABLE\_SET for a banner:

```
TABLE_SET={BAN1 :
  STUB_PREFACE=ONLY_AR1
```

## BASIC TABLES

### 4.7 DEFINING THE BANNER

```
EDIT={ : -COLUMN_TNA
      COLUMN_WIDTH=6, STUB_WIDTH=21,
      COLUMN_INFO=(C=8 W=7/C=9 W=7/C=10 W=7),

VERTICAL_PERCENT=AR, PERCENT_DECIMALS=1, -PERCENT_SIGN,

STAR_PERCENT=0, STATISTICS_DECIMALS=2, PUT_CHARACTERS=----
,
      CALL_TABLE="", RUNNING_LINES=1,
      TCON=(-TABLE_NAMES, PRINT_PAGE_NUMBERS, -TITLE_2)
      }
BANNER={ :
|
|           CITY           SEX
|
|           =====
|           DEN-  DAL-           FE- =====
| TOTAL VER  LAS  MALE MALE 18-34 35+ CHOICE SELECT SOURCE
| -----
|
}
COLUMN=: TOTAL WITH &
        [5^1/2] WITH &           ''city
        [6#M/F] WITH &           ''sex
        [65^1,2/3,4] WITH &     ''age
        [16^1//3]                ''q1
}
```

1. Controls format and printing of rows at the top of the table, which are often system generated rows (i.e., Total, No Answer, Any Response). The variable ONLY\_AR was previously defined with the keyword STUB= and then assigned to this table element:

```
STUB={ ONLY_AR :
      [SUPPRESS]      TOTAL
```



```
[SUPPRESS]          NO ANSWER
[PRINT_ROW=AR] ANY RESPONSE }
```

Table edit and text elements such as BANNER= and EDIT, require a closing right brace (}). The matching open left brace ({) is optional. Refer to the list of keywords under the heading *Default Varname Generation* for a list of which elements require a closing brace.

Other EDIT options that you can use in the banner TABLE\_SET are:

```
BOTTOM_MARGIN=          PAGE_WIDTH=
CONTINUED=              PREFIX=
CUMULATIVE_PERCENT     PRINT_ALPHA_TABLE_NAMES
DATA_INDENT=           RANK_COLUMN_BASE=
EMPTY_CELLS=           RANK_IF_INDICATED
FREQUENCY              RANK_LEVEL=#
FREQUENCY_DECIMALS=    RANK_ORDER=
FREQUENCY_ONLY         SKIP_LINES=
HORIZONTAL_PERCENT=    STUB_INDENT=
INDENT=                STUB_RANK_INDENT=
MINIMUM_BASE=          STUB_WRAP_INDENT=
MINIMUM_FREQUENCY=     SUFFIX=
MINIMUM_PERCENT=       SUPPRESS_ROWS_BASE=
NUMBER_OF_CASES        TFRP
PAGE_LENGTH=           TOP_MARGIN=
```

Other EDIT options relating to statistical testing that could logically be stored with the banner TABLE\_SET include:

```
ALL_POSSIBLE_PAIRS_TEST
CHI_SQUARE_ANOVA_FORMAT
```

## BASIC TABLES

### 4.8 FORMATTING BANNER TEXT

NEWMAN\_KEULS\_TEST

TABLE\_TESTS=<region>

These EXECUTE elements are also typically associated with a banner:

COLUMN\_WEIGHT=

COLUMN\_SHORT\_WEIGHT=

STATISTICS=

All of the EDIT options are defined in *Appendix B: TILDE COMMANDS, ~DEFINE EDIT*=. See also “5.3 Changing Table Specifications”.

## 4.8 FORMATTING BANNER TEXT

To create a banner for a table you need to define a variable that specifies the exact text, spacing, underlining, headings, etc. for each data column or banner point. To print a table with a user-defined banner it must either be assigned to the table building keyword `~EXECUTE BANNER=varname` or be defined in a `~DEFINE TABLE_SET=` with the keyword `BANNER=`.

The BANNER keyword only defines text; it does not affect the data printed under the column labels.

In designing a banner you need to consider the number of banner points you want, the width of the widest label, and the width of the stub(row) labels. Remember that the wider the banner the less room will be available for the stub labels. Most wide carriage printers allow a maximum of 132 columns with a regular font and up to 250 columns with a compressed font.

This example uses the default column width of eight spaces, suppress the default Total summary and the No Answer columns, and creates its own Total summary column. The banner has seven banner points at eight columns each for a total of







```
~INPUT @STUDY~  
~EXECUTE MAKE_TABLES  
~END
```

## THE STUB TABLE\_SET

Here is the contents of the rrnr.def file:

```
TABLE_SET= { qn1_z:  
TITLE=:  
    Q1. How much do you agree with the following  
    statement: The fast food at Road Runners is worth  
    what I pay for it.}  
STUB=:  
    (5) Completely agree  
    (4) Somewhat agree  
    (3) Neither agree nor disagree  
    (2) Somewhat disagree  
    (1) Completely disagree  
    Don't Know/Refused to answer}  
ROW=: [1/6^5//1/10]  
}
```

TABLE 001

Q1. How much do you agree with the following statement: The fast food at Road Runners is worth what I pay for it.

<-----SEX-----><-----AGE----->

**BASIC TABLES**

*4.8 FORMATTING BANNER TEXT*

	Total	Male	Female	Under 35	35-54	Over 54	Don't know/ Refused
	----	----	-----	----	-----	----	-----
Total	500	263	237	141	140	143	76
	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
(5) Completely agree	88	49	39	21	29	23	15
	17.6%	18.6%	16.5%	14.9%	20.7%	16.1%	19.7%
(4) Somewhat agree	92	43	49	26	27	27	12
	18.4%	16.4%	20.7%	18.4%	19.3%	18.9%	15.8%
(3) Neither agree nor disagree	86	48	38	23	26	24	13
	17.2%	18.3%	16.0%	16.3%	18.6%	16.8%	17.1%
(2) Somewhat disagree	73	36	37	13	23	21	16
	14.6%	13.7%	15.6%	9.2%	16.4%	14.7%	21.1%
(1) Completely disagree	86	43	43	29	17	26	14
	17.2%	16.4%	18.1%	20.6%	12.1%	18.2%	18.4%
Don't Know/Refused to answer	75	44	31	29	18	22	6
	15.0%	16.7%	13.1%	20.6%	12.9%	15.4%	7.9%

When defining your own banners, we recommend you always define your own Total column in the TABLE\_SET and in the column variable expression(TOTAL WITH variable...), then suppress the System Total/No Answer columns on the

EDIT statement with `-COLUMN_TNA`. This will ensure that operations like statistical testing are calculated off of the values you have defined for the Total column and not the system generated Total column, which only reflects a count of the number of cases that satisfy conditions for inclusion in this table.

## FORMATTING A BANNER WIDER THAN 80 COLUMNS

If your editor does not allow you to type beyond column 80 then you will have to format banners wider than 80 columns a little differently.

For this example uses the vector joiner `WITH`, which appends the categories for respondent income to the categories for marital status to form a single expression.

**Example:** `TOTAL WITH [1/52^1/2, 3/4, 5/6/9] WITH [1/54^1//6]`

The income categories are 1, 2 or 3, 4 or 5, 6, and 9. Notice that categories 2 and 3, and 4 and 5 uses a comma, so respondents who answered either 2 or 3 (or 4 or 5) on the income question will be counted as a single category.

**NOTE:** The second variable uses the double slash (`//`) to say six categories which are punches 1,2,3,4,5, and 6 instead of specifying each category separated by a single slash (`/`).

The joiner `WITH` creates twelve categories, one for the total, five income categories, and six marital status categories.

The width for this table is 117: 12 eight column banner points (plus one column if you are printing the percent sign, see previous example), for a total of 97 columns in the banner and 20 spaces for the stub labels.

**Example:** 12 banner points, each 8 columns wide

## BASIC TABLES

### 4.8 FORMATTING BANNER TEXT

```
| ----- &  
| -----
```

Starting with the Total column, this guide lines up each of the banner points over the data that will print below it. The text for each label can be placed over these guides. Vertical bar (|) marks the start of text including any blank spaces. Lines will be indented depending on which column you place the vertical bar in. Ampersand (&) at end of a line means print the following line on the same line with this one. Again, you only need to do this if your editor cannot go beyond 80 columns.

#### Example:

```
BANNER= :  
|          <-----INCOME----->&  
| <-----MARITAL STATUS----->  
|  
|          $15,000 $35,000          &  
|          Single  
|          Under      to      to      $50,000      &  
|          never      Living  
| Total $15,000 $34,999 $49,999 or more Refused Married&  
| Di-      Widowed Married To-      Refused  
|          &  
| divorced          gether  
|-----&  
|-----}
```

The first two lines are headings for the two groups of banner points, then a blank line is printed before the labels, followed by labels for each data column. The last two lines will underscore each label. Labels Divorced and Together are longer than





the default column width, so are hyphenated and placed on two lines to fit the column width.

**BASIC TABLES**

*4.8 FORMATTING BANNER TEXT*

TABLE 001

Q1. How much do you agree with the following statement: The fast food at Road Runners is worth what I pay for it.

	<-----INCOME----->					<-----MARITAL STATUS----->						
	Under Total \$15,000	\$15,000 to \$34,999	\$35,000 to \$49,999	\$50,000 or more	Refused	Married	Di- vorced	Widowed	Single never MarriedTo-	Living to- gether	Refused	
Total	500 100.0%	74 100.0%	148 100.0%	145 100.0%	70 100.0%	63 100.0%	85 100.0%	83 100.0%	96 100.0%	74 100.0%	80 100.0%	82 100.0%
(5) Completely agree	88 17.6%	10 13.5%	30 20.3%	21 14.5%	15 21.4%	12 19.0%	17 20.0%	12 14.5%	17 17.7%	14 18.9%	14 17.5%	14 17.1%
(4) Somewhat agree	92 18.4%	14 18.9%	30 20.3%	28 19.3%	7 10.0%	13 20.6%	10 11.8%	16 19.3%	17 17.7%	17 23.0%	20 25.0%	12 14.6%
(3) Neither agree nor disagree	86 17.2%	13 17.6%	22 14.9%	36 24.8%	9 12.9%	6 9.5%	14 16.5%	19 22.9%	14 14.6%	17 23.0%	11 13.8%	11 13.4%
(2) Somewhat disagree	73 14.6%	13 17.6%	21 14.2%	17 11.7%	16 22.9%	6 9.5%	11 12.9%	10 12.0%	17 17.7%	9 12.2%	11 13.8%	15 18.3%
(1) Completely disagree	86 17.2%	8 10.8%	27 18.2%	23 15.9%	14 20.0%	14 22.2%	22 25.9%	16 19.3%	16 16.7%	12 16.2%	9 11.3%	11 13.4%
Don't Know/Refused to answer	75 15.0%	16 21.6%	18 12.2%	20 13.8%	9 12.9%	12 19.0%	11 12.9%	10 12.0%	15 15.6%	5 6.8%	15 18.8%	19 23.2%

## EDITING THE BANNER

The text element `banner_title` follows the same rules as the other title elements and when present appears after the title and before the `title_4`. There is a corresponding edit option, `drop_banner_title`, which follows the same rules as the other options in the “drop” family.

You can use an asterisk ( `*` ) in a `stats=:` statement in order to indicate the last column of the banner . This feature allows a user to run stats in a job that has many different banners of varying lengths. It was introduced, in part, to allow stats to be run in a job set up to allow every question of a questionnaire to be a potential banner.

### Example:

```
stats=: a-c,g-*
```

Also, since the number of banner points in WebTables is not constrained by the size of a piece of paper, this feature can ease the spec'ing of stats on extremely wide banners. A stats statement that uses “`*`” behaves no differently than one where the actual last column letter of the banner was spec'ed explicitly.

For example, if you only have three banner points and you specify `stats=: c-*` or `stats=: d-*` you will get errors.

## 4.9 GENERATING BANNER SPECS

Mentor provides an improved way of writing banners that is designed:

- 1 to simplify the process of writing banners
- 2 to consolidate everything so that you have one set of banner specs that generate a print banner, a delimited banner, and an html banner.

This new style of writing uses banner element keywords, such as the options used with the edit statement and "banner front" keywords, which is similar to the way "stub fronts" are used.

New "banner front" keywords (only level is required):

level=# - this specifies what banner level you are describing

**NOTE:** Auto\_colspan is the default keyword.

Typically text associated with level 1 would be total, male, female and level 2 items would be age, income, etc. Every level 1 banner point needs a corresponding description in the column element.

colspan=# - used to specify how many levels are "spanned" by this banner text and associated underlining characters

justify=option - used to specify the justification for this banner text

The default justification for level 1 is right and the default justification for all other levels is center. The justification only applies to the print banner and the html banner. No justification is used for the delimited banner.

underline=character - used to specify the character to use for underling this text

The default underlining character for level 1 is a dash and the default underlining character for all other levels is an equal sign. The underline character only applies to the print banner. No underlining is used in the delimited or html banners.

New banner element keywords (only make\_banner is required):



`make_banner=option` - is used to generate banner specs for a print banner, a delimited banner, and an html banner.

where options are:

- `\k(p,h,d)` generate all 3 types of banners (this is the default)
- `\k(p)` generate only a print banner
- `\k(h)` generate only an html banner
- `\k(d)` generate only a delimited banner
- any combination of the above types

`justify_level(#)=option` - is used to specify the justification for the banner text

where # matches the "banner front" level

where options are center, right or left

The default justification for level 1 is right and the default justification for all other levels is center. The justification options only apply to the print banner and the html banner. No justification is used for the delimited banner.

`underline_level(#)=character` - is used to specify the character to use for underlining the banner text

where # matches the "banner front" level

The default underlining character for level 1 is a dash and the default underlining character for all other levels is an equal sign. The underline character only applies to the print banner. No underlining is used for the delimited or html banners.

`auto_colspan` - is used when you want the banner level text and underlining characters to "span" columns until:

- another banner front for the same level is seen
- there are no more level 1 items in the banner
- stopped by banner front option of `colspan`

`Auto_colspan` is the default.

`fill_to_level=#` - is used to have Mentor "fill-in" missing levels above level 1.

If any lower level banner item has a level above it then all items on the lower level must have a level above them. This option allows Mentor to fill-in the missing levels so the user doesn't need to provide "place holders".

`total_lines=#` - is used to specify the height of the banner in lines

If you have multiple print banners and you want each banner to be exactly the same height, you can use this option to have Mentor fill the banner with additional blank lines at the top until it reaches the number of total lines specified. This option only affects print banners.

Banner element keywords are overridden by banner front keywords.

The simplest example of using this technique in a tabset with an edit and column statement would be:

```
tabset={ban1:
  edit={:
    -coltna
  }
  banner={: make_banner
```

```

[level=1] Total
[level=1] Male
[level=1] Female
    }
col=: total with [396^1/2]
}

```

This would create a print banner using the default column width (if none was specified) and the default level 1 underlining character (dash) and default level 1 text justification (right):

```

    Total      Male   Female
-----

```

If you have a `~specfile` statement, the banner specs that are created are saved into a file named `<specfilename>.ban` and look like this:

```

banner=ban1_bn:
\k(p)
|   Total      Male   Female
| -----
\k(d)
, "Total", "Male", "Female"
\k(h)
<tr>
<td colspan="1">&nbsp;
<td colspan="1" align="right"> Total
<td colspan="1" align="right"> Male
<td colspan="1" align="right"> Female
</tr>
}

```

The default for the `make_banner` option is to make all 3 types of banners.

If you only want a print banner, you can use `make_banner=\k(p)` or any combination of `\k(h)` or `\k(d)` to get exactly what you want.

If you want to print the automatic total column and/or the automatic na column then you must provide your own banner text for those columns.

The `make_banner` option uses features (`column_width`, etc.) set in the `global_edit`, `edit`, `local_edit` and `col_info` statements.

An example of a two level print banner would be:

```

      Gender
=====
      Male  Female
-----

```

This banner was created using the default column width (if none was specified) and the default level 2 underlining character (equal sign) and default level 2 text justification (center). The specs to create this banner look like this:

```

banner={: make_banner
[level=2 colspan=2] Gender
[level=1]           Male
[level=1]           Female
}

```

The `colspan` option says this banner text (Gender) and it's associated underlining characters will span the next two level 1 items.

An example of a three level print banner would be:





```

      Denver
=====
                Age
      Gender  =====
=====
      Male  Female      Under 30 and
-----  -----  -----  -----
      30    over
  
```

And the specs to create it look like:

```

banner={: make_banner
  [level=3 colspan=4] Denver
  [level=2 colspan=2] Gender
  [level=1]           Male
  [level=1]           Female
  [level=2 colspan=2] Age
  [level=1]           Under 30
  [level=1]           30 and over
}
  
```

You could write all level 3's, followed by all the level 2's, followed by all the level 1's.

```

banner={: make_banner
  [level=3 colspan=4] Denver
  [level=2 colspan=2] Gender
  [level=2 colspan=2] Age
  [level=1]           Male
  [level=1]           Female
  [level=1]           Under 30
  [level=1]           30 and over
}
  
```

## BASIC TABLES

### 4.9 GENERATING BANNER SPECS

The level 3 banner text (Denver) spans all four level 1 items. The level 2 banner text (Gender) spans the next two level 1 items and the level 2 banner text (Age) spans the remaining two level 1 items.

Using the additional banner element keyword of `auto_colspan`, these specs would produce the same print banner:

```
banner={: make_banner auto_colspan
  [level=3] Denver
  [level=2] Gender
  [level=1] Male
  [level=1] Female
  [level=2] Age
  [level=1] Under 30
  [level=1] 30 and over
}
```

If you write all level 3's, followed by all the level 2's, followed by all the level 1's, you must use the banner front `colspan` option to specify how many levels are spanned and you can't use the `auto_colspan` option.

If you had a single item that was included in level 3 but not under level 2 as in the following example:

```

          Denver
=====
                Age
  Gender  =====  Income
=====  Under 30 and  over
  Male  Female    30   over  $50K
-----  -----  -----  -----
```

You would need to add a level 2 (without any text) in the banner specs before "Income over \$50K" as a `colspan` stopper:

```
banner={: make_banner auto_colspan
```

```
[level=3] Denver
[level=2] Gender
[level=1] Male
[level=1] Female
[level=2] Age
[level=1] Under 30
[level=1] 30 and over
[level=2] ''place holder
[level=1] Income over $50K
}
```

Or you could write this banner without using `auto_colspan` and add your own banner front colspan items as in:

```
banner={: make_banner
[level=3 colspan=5] Denver
[level=2 colspan=2] Gender
[level=1]           Male
[level=1]           Female
[level=2 colspan=2] Age
[level=1]           Under 30
[level=1]           30 and over
[level=2 colspan=1] ''place holder
[level=1]           Income over $50K
}
```

To add a total column to a banner, without any level 2 text, there is a rule for banner levels to be followed.

If any lower level banner item (Male, Female) has a level above it (Gender), then all items on the lower level (Total) must have a level above them. In other words, if there is ever a level 2, then every level 1 must have a level 2. If there is ever a level 3, then every level 1 must have a level 3, and so on.

```

                Gender
            =====
    Total      Male  Female
    -----

```

A level 2 place holder needs to be added for the total column:

```

banner={: make_banner auto_colspan
  [level=2] ''place holder
  [level=1] Total
  [level=2] Gender
  [level=1] Male
  [level=1] Female
}

```

Or, an additional banner element option of `fill_to_level=#` could be used. This says if there is ever an unspecified upper level, fill it in. The same print banner could be created by using:

```

banner={: make_banner auto_colspan fill_to_level=2
  [level=1] Total
  [level=2] Gender
  [level=1] Male
  [level=1] Female
}

```

This is an example of a three level banner with a total column:

```

                Denver
            =====
                Age
            =====
    Gender      Under 30 and   Income
    =====   30    over     over
    Total      Male  Female   $50K
    -----

```



One way to write the specs for this banner would be:

```
banner={: make_banner auto_colspan fill_to_level=3
[level=1]          Total
[level=3]          Denver
[level=2]          Gender
[level=1]          Male
[level=1]          Female
[level=2 colspan=2] Age
[level=1]          Under 30
[level=1]          30 and over
[level=1]          Income over $50K
}
```

## HOW TO CREATE A BANNER USING MAKE\_BANNER FORMAT

This section describes the steps on how you would write a banner using the `make_banner` format.

1) This is the banner request from the client:

```
Q.B  Male / Female
Q.24 25-34 / 35-54 / 55+
Q.21 High school grad or less / Some college, trade school / College grad+
Q.26 Hispanic - Yes / No
Q.22 Conservative / Middle-of-the-road / Liberal
Q.28 Party - Republican / Democrat / Independent
Q.29 Voter History - Newly registered / Repeat voter
```

2) Edit (or cut and paste) the banner request to get each item on a separate line.

```
Q.B
Male
Female
```

## BASIC TABLES

### 4.9 GENERATING BANNER SPECS

Q.24  
25-34  
35-54  
55+  
Q.21  
High school grad or less  
Some college, trade school  
College grad+  
Q.26 Hispanic  
Yes  
No  
Q.22  
Conservative  
Middle-of-the-road  
Liberal  
Q.28 Party  
Republican  
Democrat  
Independent  
Q.29 Voter History  
Newly registered  
Repeat voter

- 3) Add a total column, change question numbers into question text, add level designators. Level 2's provide the titles that span across level 1 banner points.

[level=1] Total  
[level=2] Gender  
[level=1] Male  
[level=1] Female  
[level=2] Age  
[level=1] 25-34  
[level=1] 35-54  
[level=1] 55+  
[level=2] Education  
[level=1] High school grad or less  
[level=1] Some college, trade school  
[level=1] College grad+  
[level=2] Hispanic  
[level=1] Yes

```
[level=1] No
[level=2] Think of Yourself As...
[level=1] Conservative
[level=1] Middle-of-the-road
[level=1] Liberal
[level=2] Party
[level=1] Republican
[level=1] Democrat
[level=1] Independent
[level=2] Voter History
[level=1] Newly registered
[level=1] Repeat voter
```

- 4) (Optional) Count the level 1 items. This is used to confirm the number of banner points and to determine the column\_width, stub\_width and page\_width.

```
[level=1] Total                '' 01
[level=2] Gender
[level=1] Male                  '' 02
[level=1] Female                '' 03
[level=2] Age
[level=1] 25-34                 '' 04
[level=1] 35-54                 '' 05
[level=1] 55+                   '' 06
[level=2] Education
[level=1] High school grad or less '' 07
[level=1] Some college, trade school '' 08
[level=1] College grad+         '' 09
[level=2] Hispanic
[level=1] Yes                   '' 10
[level=1] No                    '' 11
[level=2] Think of Yourself As...
[level=1] Conservative          '' 12
[level=1] Middle-of-the-road    '' 13
[level=1] Liberal                '' 14
[level=2] Party
[level=1] Republican            '' 15
[level=1] Democrat              '' 16
[level=1] Independent           '' 17
[level=2] Voter History
```

## BASIC TABLES

### 4.9 GENERATING BANNER SPECS

```
[level=1] Newly registered          '' 18
[level=1] Repeat voter              '' 19
```

- 5) Put these lines in a banner tabset adding appropriate tabset items like:

an edit statement specifying `-coltna`, a column width and a stub width

a column statement (where every level 1 item has a corresponding column and response description)

Use these banner options:

```
make_banner      (make a banner for me)
fill_to_level=2  (this is a two level banner)
auto_colspan     (span each level 1 with the specified level 2)
```

Run this setup using the `~set preview_titles` option.

- 6) Look at the print file created and make any adjustments necessary to the banner tabset specs. For example, use `\n` to force word breaks. Note that `\n` only forces word breaks in the print file and not the html file or delimited file.

```
[level=1] Total                      '' 01
[level=2] Gender
[level=1] Male                        '' 02
[level=1] Female                      '' 03
[level=2] Age
[level=1] 25-34                       '' 04
[level=1] 35-54                       '' 05
[level=1] 55+                         '' 06
[level=2] Education
[level=1] High school grad or less    '' 07
[level=1] Some col\nlege, trade school '' 08
[level=1] Col\nlege grad+             '' 09
[level=2] Hispanic
[level=1] Yes                         '' 10
[level=1] No                          '' 11
[level=2] Think of Yourself As...
```



[level=1]	Con\nserva\ntive	'' 12
[level=1]	Middle\n-of-\nthel-\nroad	'' 13
[level=1]	Lib\neral	'' 14
[level=2]	Party	
[level=1]	Repub\nlican	'' 15
[level=1]	Demo\ncrat	'' 16
[level=1]	Inde\npen\ndent	'' 17
[level=2]	Voter History	
[level=1]	Newly regis\ntered	'' 18
[level=1]	Repeat voter	'' 19

7) Look at the print file created and add dashes to hyphenate words in the banner where desired.

[level=1]	Total	'' 01
[level=2]	Gender	
[level=1]	Male	'' 02
[level=1]	Female	'' 03
[level=2]	Age	
[level=1]	25-34	'' 04
[level=1]	35-54	'' 05
[level=1]	55+	'' 06
[level=2]	Education	
[level=1]	High school grad or less	'' 07
[level=1]	Some col-\nlege, trade school	'' 08
[level=1]	Col-\nlege grad+	'' 09
[level=2]	Hispanic	
[level=1]	Yes	'' 10
[level=1]	No	'' 11
[level=2]	Think of Yourself As...	
[level=1]	Con-\nserva-\ntive	'' 12
[level=1]	Middle\n-of-\nthel-\nroad	'' 13
[level=1]	Lib-\neral	'' 14
[level=2]	Party	
[level=1]	Repub-\nlican	'' 15
[level=1]	Demo-\ncrat	'' 16
[level=1]	Inde-\npen-\ndent	'' 17
[level=2]	Voter History	
[level=1]	Newly regis-\ntered	'' 18
[level=1]	Repeat voter	'' 19

## BASIC TABLES

### 4.9 GENERATING BANNER SPECS

- 8) (Optional) Add stats letters to level 1 items. This is used to assign stats= values for stats testing.

[level=1]	Total	'' 01	A
[level=2]	Gender		
[level=1]	Male	'' 02	B
[level=1]	Female	'' 03	C
[level=2]	Age		
[level=1]	25-34	'' 04	D
[level=1]	35-54	'' 05	E
[level=1]	55+	'' 06	F
[level=2]	Education		
[level=1]	High school grad or less	'' 07	G
[level=1]	Some col-\nlege, trade school	'' 08	H
[level=1]	Col-\nlege grad+	'' 09	I
[level=2]	Hispanic		
[level=1]	Yes	'' 10	J
[level=1]	No	'' 11	K
[level=2]	Think of Yourself As...		
[level=1]	Con-\nserva-\ntive	'' 12	L
[level=1]	Middle\n-of-\nthe-\nroad	'' 13	M
[level=1]	Lib-\neral	'' 14	N
[level=2]	Party		
[level=1]	Repub-\nlican	'' 15	O
[level=1]	Demo-\nocrat	'' 16	P
[level=1]	Inde-\npen-\ndent	'' 17	Q
[level=2]	Voter History		
[level=1]	Newly regis-\ntered	'' 18	R
[level=1]	Repeat voter	'' 19	S

- 9) Add remaining banner tabset items like:

```
statistics
weights
```

- 10) (Optional) Add ~set statements to create a delimited and an html file.
- 11) If you use dashes to hyphenate words in the banner, then those dashes also go to the delimited file and the html file.

If you are creating a delimited file and an html file from these same banner specs then use \k(p) to add dashes to the print file only. Use \k(p,h,d) to resume sending characters to all 3 files.

```
[level=1]      Total                '' 01
[level=2]      Gender
[level=1]      Male                  '' 02
[level=1]      Female                '' 03
[level=2]      Age
[level=1]      25-34                 '' 04
[level=1]      35-54                 '' 05
[level=1]      55+                   '' 06
[level=2]      Education
[level=1]      High school grad or less '' 07
[level=1]      Some col\k(p)-\k(p,h,d)\nlege, trade school '' 08
[level=1]      Col\k(p)-\k(p,h,d)\nlege grad+ '' 09
[level=2]      Hispanic
[level=1]      Yes                    '' 10
[level=1]      No                     '' 11
[level=2]      Think of Yourself As...
[level=1]      Con\k(p)-\k(p,h,d)\nserva\k(p)-\k(p,h,d)\ntive '' 12
[level=1]      Middle\n-of-\nthe-\nroad '' 13
[level=1]      Lib\k(p)-\k(p,h,d)\neral '' 14
[level=2]      Party
[level=1]      Repub\k(p)-\k(p,h,d)\nlican '' 15
[level=1]      Demo\k(p)-\k(p,h,d)\nocrat '' 16
[level=1]      Inde\k(p)-\k(p,h,d)\npen\k(p)-\k(p,h,d)\ndent '' 17
[level=2]      Voter History
[level=1]      Newly regis\k(p)-\k(p,h,d)\ntered '' 18
[level=1]      Repeat voter                '' 19
```

12) (Optional) Or use >define @d \k(p)-\k(p,h,d) .

```
[level=1]      Total                '' 01
[level=2]      Gender
[level=1]      Male                  '' 02
[level=1]      Female                '' 03
[level=2]      Age
[level=1]      25-34                 '' 04
[level=1]      35-54                 '' 05
```

[level=1]	55+	'' 06
[level=2]	Education	
[level=1]	High school grad or less	'' 07
[level=1]	Some col@d\nlege, trade school	'' 08
[level=1]	Col@d\nlege grad+	'' 09
[level=2]	Hispanic	
[level=1]	Yes	'' 10
[level=1]	No	'' 11
[level=2]	Think of Yourself As...	
[level=1]	Con@d\nserva@d\ntive	'' 12
[level=1]	Middle\n-of-\nthe-\nroad	'' 13
[level=1]	Lib@d\neral	'' 14
[level=2]	Party	
[level=1]	Repub@d\nlican	'' 15
[level=1]	Demo@d\ncrat	'' 16
[level=1]	Inde@d\npen@d\ndent	'' 17
[level=2]	Voter History	
[level=1]	Newly regis@d\ntered	'' 18
[level=1]	Repeat voter	'' 19

The `make_banner` option uses features (background color, font size, font type, font color) set in a `web_format_banner` statement to affect the appearance of the html file.

The `make_banner` option uses features (background color, font size, font type, font color) set in a style sheet file and referenced by a class statement associated with the banner to affect the appearance of the html file.

- 13) (Optional) Add any extra desired banner features like:

```
underline_level(1)== (change level 1 underlining to equal signs)
underline_level(2)-- (change level 2 underlining to dashes)
justify_level(1)=left (left justify level one items)
justify_level(2)=right (right justify level two items)
```

- 14) After final review of print the banner, delimited banner and html banner, remove `~set preview_titles`. Add the name of the `.tr` and the `.def` file when available.

## 4.10 DEFINING INDIVIDUAL TABLES

Stub table sets include definitions for the table title, stub labels, data row variable, and EDIT statements. It can also include LOCAL\_EDIT statements, a base, base title, and possibly SET commands for additional control.

Here is an simple stub table set:

```
TABLE_SET= {Q1:
  TITLE={:
    Q1.  What is your favorite month of the year?
  }
  STUB={:
    January
    February
    March
    April
    May
    June
    July
    August
    September
    October
    November
    December
  }
  ROW=: [6^1//12]
}
```

Here is a more detailed explanation of the table elements defined in this example.

## BASIC TABLES

### 4.10 DEFINING INDIVIDUAL TABLES

**TITLE=** Instructs Mentor to print this text as the main table title. It prints below the table name and prior to the banner text. The **EDIT** keyword, **RUNNING\_LINES**, controls how this text will print across the top of the page. The default is to print text as written. **RUNNING\_LINES=1** will print as much text can be fit across the page before going to a new line. See 5.3 *CHANGING TABLE SPECIFICATIONS, Print Options* for more on **RUNNING\_LINES**. An example table in that section shows the print positions for other table titling.

Table titling elements follow the same rules and have the same options as **~DEFINE LINES**. Within the **TABLE\_SET** they are assigned to a specific table element with the appropriate keyword.

**STUB=** Defines the stub labels for each data row printed on your table. Excluding stubs marked as **[COMMENT]** or **[PRINT\_ROW]** labels, the number of labels must match the number of data categories defined in your row variable. (see *Appendix B: TILDE COMMANDS, ~DEFINE EDIT* options **EXTRA\_STUBS\_OK** and **EXTRA\_ROWS\_OK** and **~SET TABLE\_SET\_MATCH\_ERROR/WARN**).

**ROW=** Assigns an axis or data variable definition as the table row. See “4.6 *DEFINING DATA*” for the rules on defining data variables. These rules will apply to any data variable regardless of what table element you will assign it to, i.e., **COLUMN**, **BASE**, **FILTER**, **ROW\_WEIGHT**. See *Appendix B: TILDE COMMANDS, ~DEFINE AXIS*, for the additional rules regarding axis definition for **COLUMN=** and **ROW=** specification or axis-only definitions.

## HOW TO ADD RANKING TO A TABLE

If you want to rank a table so the stub items are ranked from high to low, then add the **KEEP\_RANK** option (abbreviated **KR**) to the **STUB** keyword, as follows:



```
TABLE_SET= {Q1:
    TITLE={:
        Q1. What is your favorite month of the year?
    }
    STUB={:
        [KR=1]  January
                February
                March
                April
                May
                June
                July
                August
                September
                October
                November
                December
    }
    ROW=: [6^1//12]
}
```

KR=1 (or KEEP\_RANK=1) means to rank this and subsequent stubs at a rank level of one (until a new rank level is indicated). You use up to ten rank levels (0-9), and the default is to rank from high to low.

To rank tables, you must also add the RANK\_IF\_INDICATED option to your EDIT statement.

## HOW TO ADD A BASE TO A TABLE

## BASIC TABLES

### 4.10 DEFINING INDIVIDUAL TABLES

A base is a way to include only a subset of a sample. For example:

```
TABLE_SET= {Q1:
  BASE=: [5^3]
  TITLE_4={: BASE: Respondents who own dogs
            }
  TITLE={:
          Q1. What is your favorite month of the year?
          }
  STUB={:
        [KR=1]  January
                February
                March
                April
                May
                June
                July
                August
                September
                October
                November
                December
        }
  ROW=: [6^1//12]
        }
```

Here are the new elements:

**BASE=** Defines a base for this table. Only those respondents meeting the data criteria defined here will be included in this table. In this



example, only those respondents who had a three punch in column five are included in the table. You can also use FILTER to describe an additional data criteria for a set of tables.

TITLE\_4= Prints below the text defined on TITLE=, is often used to describe a base or additional considerations on a table.

## HOW TO ADD SUMMARY STATISTICS TO A TABLE

Mentor makes it easy to add summary statistics, such as means, standard deviations, and standard errors to your tables. It requires the STATISTICS\_ROW option to the stub command, and the statistic defined as a part of the row variable.

Here is a simple example:

```
TABLE_SET= {Q5 :
  TITLE={ :
    Q5. ALL THINGS CONSIDERED, HOW SATISFIED OR
    DISSATISFIED WERE YOU WITH THE WAY COMMUNITY
    GENERAL HANDLED YOUR STAY THERE? WOULD YOU SAY
    YOU WERE VERY SATISFIED, SOMEWHAT SATISFIED,
    SOMEWHAT DISSATISFIED OR VERY DISSATISFIED?\N
  }
  STUB={ :
    4-VERY SATISFIED
    3-SOMEWHAT SATISFIED
    2-SOMEWHAT DISSATISFIED
    1-VERY DISSATISFIED
    DON'T KNOW/REFUSED
    [STAT] MEAN
    [STAT] STANDARD DEVIATION
```

## BASIC TABLES

### 4.10 DEFINING INDIVIDUAL TABLES

```
    }  
    ROW= : [47^4/3/2/1/0] $ [MEAN, STD] [47*RANGES=1-4]  
    }
```

STAT (or STATISTICS\_ROW) Controls printing on the row. This identifies this as a statistics row, so it will print only the frequencies and not percentages. (To control the number of decimals on the frequencies, use the EDIT option STATISTICS\_DECIMALS.) Various stub options are covered in 5.3 *CHANGING TABLE SPECIFICATIONS, Row Print Options*. A complete list can be found in *Appendix B: TILDE COMMANDS, ~DEFINE STUB=*.

\$ [MEAN, STD] The list of statistics you want to include in the row.

[47\*RANGES=1-4] If all the answers to the question were numeric, you could just put the column location [47] as the row definition. But in this case, the Don't Know/Refused answer was coded as a number, so including it will make the mean wrong (the mean will be much too high if the Don't Know/Refused was coded as a 99!). Using the RANGES modifier only includes answers one through four, and therefore eliminates the Don't Know/Refused answer from the mean.

For more information, Chapter 6 has several examples of how to add means and medians to your tables and Chapter 7 shows you how to do tests dealing with statistical significance, such as T-tests, chi square tests, and ANOVA tests.

## 4.11 SAMPLE SPECIFICATION FILES

What follows are sample specification files, first is an example of how to save elements to a DB file, then "real world" specs from the Roadrunner's Pizza study, and then, finally, specs that can make tables with complex banners process more quickly.

### USING THE DB FILE

When you have job specifications that will be used in several jobs, or a job with multiple sections that have common elements, you may wish to store the common elements in a DB file. You can then call these elements by name in a later run. For instance, if your project calls for multiple banners, the EDIT statement associated with each banner can be simplified by storing those parts of the EDIT statement that will remain constant to a DB file. You can then reference that DB file entry in each banner TABLE\_SET. In this way, EDITs can be developed for different clients or different presentation requirements, stored, and called in when needed.

When Mentor creates tables, the program automatically creates and stores elements in a local database file that is automatically removed at the end of a run. If you want to have your own permanent database file, you can write a small spec file to create DB elements, or include these commands in your larger spec file. First, you must create a DB file with the meta command >CREATE\_DB. Enter:

```
>CREATE_DB edit1
```

Use the >FILE\_TO\_DB meta command to put items from the EDIT statement into the DB file. The basic syntax is:

```
>FILE_TO_DB name #
```

"Name" can be any string up to 14 characters, including underscores ( \_ ) and periods ( . ). The pound sign ( # ) means to read all the items until Mentor encounters an >END\_OF\_FILE command.

## BASIC TABLES

### 4.11 SAMPLE SPECIFICATION FILES

#### Example:

```
>FILE_TO_DB edit_standard #
    -COLUMN_TNA
    VERTICAL_PERCENT=AR
    PERCENT_DECIMALS=1
    -PERCENT_SIGN
    STATISTICS_DECIMALS=2
    CALL_TABLE=" "
    RUNNING_LINES=1
    STAR_PERCENT=0
    TCON=( -TABLE_NAMES, PRINT_PAGE_NUMBERS,
          -TCON_PAGE_NUMBERS, -TITLE_2)
    PUT_CHARACTERS=---
>END_OF_FILE
```

Now, "edit\_standard" is a DB item stored in the DB file called edit1. You can access "edit\_standard" for any banner by first opening the DB file and then accessing the db item on the EDIT keyword in the TABLE\_SET.

```

>USE_DB edit1                                ''meta command that opens the DB
                                              ''file created earlier

TABLE_SET= {BAN1:
EDIT={:
    COLUMN_WIDTH=6
    STUB_WIDTH=21
    COLUMN_INFO=(C=8 W=7/C=9 W=7/C=10 W=7)
    &&EDITSTANDARD                          ''double ampersand (&&) gets
                                              ''item from DB file
}
BANNER={:
|          CITY          SEX
|          =====
|          DEN-  DAL-          FE- =====
| TOTAL  VER  LAS  MALE  MALE 18-34  35+ CHOICE SELECT SOURCE
| -----
}

COLUMN=: TOTAL WITH &
          [5^1/2] WITH &                    ''city
          [6#M/F] WITH &                    ''sex
          [65^1,2/3,4] WITH &              ''age
          [16^1//3]                        ''q1
}

```

**NOTE:** COLUMN\_WIDTH, STUB\_WIDTH, and COLUMN\_INFO have been specified separately in the EDIT statement since these items would typically change for each banner. Remember, the last occurrence of an option in the EDIT statement supersedes any prior occurrence of the same option.

## PUTTING IT ALL TOGETHER

So far in this chapter, we have covered the parts of table, how to define data variables, banner tabset, and stub tabsets. Now, let's put all that knowledge together. First, let's take advantage of Mentor's ability to "call in" on file from another. In the main spec file (which we'll give the extension SPX), we'll put the META commands, the SET commands, the banner tab set, the INPUT command, and EXECUTE commands, and call a separate file that contains all the individual question information (the stub tab sets). Below is a sample SPX file, a sample DEF file, and the tables they create.

To have Mentor call in the specs from another file, put an ampersand in front of the file name. To read a file that has the DEF extension, you could put the line

```
&filename.DEF
```

If your DEF file is the name of your study and you have defined the study name with the >DEFINE command, then you can use:

```
&@STUDY~^DEF
```

This reads in a file, in this case the file containing your table definitions. If you have Survent, the DEF file has already been generated for you based on your Survent questionnaire specifications.

**NOTE:** The use of a caret (^) before the file extension; Mentor reads this as a period (.) on DOS or UNIX platforms, and as nothing in MPE, so you could use your specification file on DOS, UNIX or MPE without modifying it. The tilde mark (~) acts as a delimiter for user-defined variables. This means you can append letters and/or numbers to the STUDY name, for example, with the STUDY variable defined as "sample", &@STUDY~2^DEF would open the file SAMPLE2.DEF.



Sample specification files are provided with your Mentor software to run the example tables discussed in this section. The files are located in either \CFMC\Mentor\ROADRUNR (DOS and UNIX ) or ROADRUNR.CFMC (MPE).

Following is the entire TABS.SPX file:

```
>ALLOW_INDENT

>PURGE_SAME

>RUN_LABEL="CfMC Sample Tables"

>DEFINE @STUDY rrunr

~INPUT @STUDY~,ALLOW_UPDATE,MAYBE_BACKUP,DOTS=100

>PRINT_FILE @STUDY~

>CREATE_DB @STUDY~1,DUPLICATE=WARN

~COMMENT
This is part of the Road Runner Study.

~SPEC_FILES @STUDY~

~DEFINE
TABLE_SET={global:
SET AUTOMATIC_TABLES
```

## BASIC TABLES

### 4.11 SAMPLE SPECIFICATION FILES

```
HEADER={: =Road Runner Fast Food Test Tables  
Prepared on #date# }
```

```
FOOTER{: =Tables prepared by Computers for Marketing  
Corp.  
Page #page_number#  
}
```

```
TABLE_SET={banner1:  
SET AUTOMATIC_TABLES
```

```
HEADER=: =Road Runner Fast Food Sample Tables  
Prepared on #date# }
```

```
FOOTER=: =Tables prepared by Computers for Marketing  
Corp.  
Page #page_number# }
```

```
EDIT=: -COLUMN_TNA, -ROW_NA, PERCENT_DECIMALS=0,  
STATISTICS_DECIMALS=2,  
COLUMN_WIDTH=7, STUB_WIDTH=40,  
RUNNING_LINES=1,  
TCON=(-TABLE_NAMES, PRINT_PAGE_NUMBERS, &  
-TCON_PAGE_NUMBERS, -FOOTER)
```



TABS.SPX file (con'd)

```

BANNER=:
|           <-----AGE----->  <-----INCOME----->  <---RATING--->
|
|           Under           Over  Under  $15-  Over           Neu-
| Total      35  35-54      54  $15k  $35k  $35k  Good  Poor
| -----  -----  -----  -----  -----  -----  -----  -----  ----- }

COLUMN=: TOTAL WITH &
          [1/51^1,2/3,4/5,6] WITH &
          [1/52^1/2,3/4,5,6] WITH &
          [1/47^4,5/1,2,3]
}

&@STUDY^DEF

~INPUT @STUDY~

~EXECUTE
  MAKE_TABLES

~END

```

Here are each of the lines of the spec file explained:

**>PURGE\_SAME**

Purges same named files. The default is to alpha-kick the first letter of the file name of the older file.

**>ALLOW\_INDENT**

Allows indentation of meta commands; by default they must start in column 1 of the specification file.

**>RUN\_LABEL= "CfMC Sample Tables"**

Identifies this run with this label. This label will print at the end of the run or if you type **CTRL-Y** during the program run.

**>DEFINE @STUDY rrunr**

Allows you to create user-defined variables to substitute a file name or to execute a command. In this case >DEFINE assigns the variable, @STUDY, to the file name, RRUNR. Wherever the file name RRUNR would normally appear, @STUDY is substituted. By replacing references to a particular file name with a variable, we can substitute a different file name for the same variable throughout a standardized specification file by specifying a new name on this line. We will make use of this command in the TABS.SPX file.

**~INPUT @STUDY~,ALLOW\_UPDATE,MAYBE\_BACKUP,DOT=100**

Loads the data file but allows you to modify it with the keyword ALLOW\_UPDATE. MAYBE\_BACKUP creates a copy (TRX) if one does not exist, preserving the original data file. DOT=100 will print a dot to the screen for every 100 cases read. We will be generating and storing new data for the banner. The tilde mark (~) acts as a delimiter between user-defined keywords.

**>CREATE\_DB @STUDY~1,DUPLICATE=WARN**

This line creates the db file. We are calling the DB file "study1" to make sure it doesn't write over an existing DB file that could have been generated from a Survent compile. Everything defined in the ~DEFINE block and all the tables built in the ~EXECUTE block will be stored in this DB file. The tilde character (~) after @STUDY~ is a delimiter. The tilde delimiter allows you to include options after the defined keyword name. DUPLICATE=WARN prints a warning every time an existing item is replaced in the file.

The DB file is a machine readable file, which means you cannot view or edit it in a word processing program. (Use the utility DBUTIL to look at the items in a DB file, see the *Utilities* manual.) It is an efficient way to store items generated by Mentor that you might want to use again. For instance, tables can be stored in a DB file. If you needed to reprint some or all of them to change the format, correct an error in the text, or add statistics rows you could instruct Mentor to look up the tables in the DB file and reprint them, without forcing the program to process the data file again for each table.

Table components such as stub labels, other text, and print options are already stored as a single entry in the DB file that Mentor can retrieve easily.

DB files are NOT REQUIRED for you to tabulate your data, but they do provide a way to store items that you might want to use again.

**~COMMENT**

Any text following this tilde command is considered a spec file comment and is not executed by Mentor. Everything is ignored until the next tilde command.

**~SPEC\_FILES @STUDY~**

This has Mentor generate TAB and LPR files.

**~DEFINE TABLE\_SET={global:**

Defines the initial TABLE\_SET. As you saw in “4.3 DEFINING TABLE ELEMENTS”, the TABLE\_SET structure can be used to define table elements that will control all the tables in a particular run such as SET commands, headers, footers, edit statement, banner text, and the column variable. You could define these individually, but this would require re-specifying each varname on its corresponding ~EXECUTE keyword. By defining them inside a TABLE\_SET you eliminate this extra step since TABLE\_SET= automatically causes the correct ~EXECUTE specification to be written to the program-generated TAB file (see ~SPEC\_FILES).

## BASIC TABLES

### 4.11 SAMPLE SPECIFICATION FILES

The global and banner TABLE\_SETs are defined before the file containing the stub TABLE\_SETs (RRUNR^DEF) is read in. That is because we want them in effect for all the tables made.

#### **SET AUTOMATIC\_TABLES**

Tells program to make tables from the elements specified whenever a ROW variable is seen.

#### **HEADER={ :=Road Runner Fast Food Test Tables Prepared on #date# }**

Defines the text that will print as the header on each table. The equal sign (=) centers the text, #date# prints the date in the format MM DD 19YY. (see *Appendix B: TILDE COMMANDS*, ~DEFINE LINES= to print system #time# or #table\_name# anywhere in the table titling). The lines will print centered for the page width. RUNNING\_LINES=1 will not affect these titles.

#### **FOOTER{ :=Tables prepared by Computers for Marketing Corp. Page #page\_number# }**

Defines the text that will print as the footer on each table. The equal sign (=) centers the text, #page\_number# prints the page number.

}

Ends this TABLE\_SET definition.

```
EDIT={ : -COLUMN_TNA  -ROW_NA  PERCENT_DECIMALS=0
          STATISTICS_DECIMALS=2
          COLUMN_WIDTH=7  STUB_WIDTH=40
          RUNNING_LINES=1
          TCON= (-TABLE_NAMES, PRINT_PAGE_NUMBERS, &
                -TCON_PAGE_NUMBERS, -FOOTER)
```

This is the main EDIT statement that will control the format of the printed tables. There are several new keywords not previously mentioned:

**-COLUMN\_TNA**

Suppresses printing of the default system-generated Total and No Answer summary columns.

**-ROW\_NA**

Suppresses printing of the default system-generated No Answer summary row.

**PERCENT\_DECIMALS=0**

Prints percents with zero decimal places. The default is one decimal place.

**STATISTICS\_DECIMALS=2**

Prints two decimal places on statistics rows. The default is one decimal place. This sample table run utilizes the DEF file made from PREPARE question specifications compiled with ~PREPARE COMPILE Mentor\_SPECS. The default is to generate statistics specifications for NUM questions. See “5.1 Expressions and Joiners” for details on defining expressions.

**COLUMN\_WIDTH=7**

Controls the width of table columns; the default is eight.

## BASIC TABLES

### 4.11 SAMPLE SPECIFICATION FILES

#### **STUB\_WIDTH=40**

Controls the width of table stubs; the default is 20.

#### **RUNNING\_LINES=1**

Controls how the table titles, header, and footer will print. The default is `RUNNING_LINES=0`, which prints the lines exactly as defined.

`RUNNING_LINES=1` will wrap the lines according to the `PAGE_WIDTH` setting (or default) and any `INDENT` (indents the whole table either by number specified (greater than 0) or the keyword `CENTER`). `RUNNING_LINES=2` means print the first line like `RUNNING_LINES=1`, then indent 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` specified. This keyword is useful when you want to set the question number off from the rest of the title text.

`RUNNING_LINES=1` or `2` is overridden if you specify a position character before the text: `=` to center, `>` to right-justify, or `<` to left-justify (default) all lines of text. See the definitions for `HEADER=` and `FOOTER=` for examples.

#### **TCON**

Prints a table of contents at the end of the print file. You can control what prints in the table of contents. Headers, footers, and all titles except `TITLE_5` print by default; you can suppress any of these. You can indent text with the keyword `INDENT=#`; the default is zero (0), do not indent. You must use an ampersand (`&`) to continue the `TCON=` specification to another line.

Justification commands (`<` to left-justify; `>` to right-justify; `=` to center) specified in titling do not affect how text is printed in the table of contents.

<code>-TABLE_NAMES</code>	Suppresses printing of the table name, e.g., T0002.
<code>PRINT_PAGE_NUMBERS</code>	Prints the page number.
<code>-TCON_PAGE_NUMBERS</code>	Suppresses the table of contents page number that prints at the top of each table of contents page.

-FOOTER

Suppresses printing of the footer in the table of contents.

BANNER= :

	<-----AGE----->	<-----INCOME----->	<---RATING--->
			Neu-
	Under	Over	tral/
Total	35 35-54	54	Good Poor
-----	-----	-----	----- } ----- }

- BANNER=** Defines the banner labels. Banner text formatting controls were explained in “4.8 FORMATTING BANNER TEXT”.
- &@STUDY~^DEF** Reads in the table definitions file either generated when PREPARE specifications were compiled with ~PREPARE COMPILE Mentor\_SPECS or written by you.
- EDIT\_DUMP** A debugging tool, prints the edit options in effect for each table.
- MULTIPLE\_WEIGHT\_STATISTICS** Allows T-tests using the STATISTICS= keyword on tables that have different weights on some columns.
- STATISTICS\_BASE\_AR** Sets the standard base for the T-tests executed by the table using the STATISTICS keyword to be the Any Response System row, rather than the Total row (the default).
- ~INPUT @STUDY~** Opens the data file RRUNR^TR.
- ~EXECUTE** This is the block where you execute tables or procedures against the data file.
- MAKE\_TABLES** This option builds and prints the tables. As explained earlier in 4.4 TABLE BUILDING, when you use SET AUTOMATIC\_TABLES, Mentor table building is triggered by the ROW= command. All of the table elements

## BASIC TABLES

### 4.11 SAMPLE SPECIFICATION FILES

either specified before the ROW= variable or defined in the same TABLE\_SET are in effect for that table. The EXECUTE command MAKE\_TABLES builds and prints each table. MAKE\_TABLES actually executes three separate commands: BUILD\_TABLES, RESET, and PRINT\_RUN, These commands utilize program-generated files created with the ~SPEC\_FILES command to read and store each TABLE\_SET definition, reset all table elements, then print the tables either to the open print file or list file.

## SAMPLE TABLE

Road Runner Fast Food Sample Tables

Prepared on 13 AUG 1992

TABLE 001

Q1. How much do you agree with the following statement: The fast food at Road Runners is worth what I pay for it.

	<-----AGE----->			<-----INCOME----->			<--RATING-->		
	Under		Over	Under	\$15-	Over		Neu- tral/	
Total	35	35-54	54	\$15k	\$35k	\$35k	Good	Poor	
	-----	-----	-----	-----	-----	-----	-----	-----	
Total	500	141	140	143	74	148	215	166	247
	100%	100%	100%	100%	100%	100%	100%	100%	100%
(5) Completely agree	88	21	29	23	10	30	36	26	51
	18%	15%	21%	16%	14%	20%	17%	16%	21%
(4) Somewhat agree	92	26	27	27	14	30	35	29	42
	18%	18%	19%	19%	19%	20%	16%	17%	17%
(3) Neither agree nor disagree	86	23	26	24	13	22	45	27	42
	17%	16%	19%	17%	18%	15%	21%	16%	17%



BASIC TABLES  
4.11 SAMPLE SPECIFICATION FILES



(2) Somewhat disagree	73	13	23	21	13	21	33	27	34
	15%	9%	16%	15%	18%	14%	15%	16%	14%
(1) Completely disagree	86	29	17	26	8	27	37	22	49
	17%	21%	12%	18%	11%	18%	17%	13%	20%
Don't Know/Refused to answer	75	29	18	22	16	18	29	35	29
	15%	21%	13%	15%	22%	12%	13%	21%	12%
Mean	3.05	2.97	3.23	3.00	3.09	3.12	3.00	3.08	3.06
Standard deviation	1.42	1.47	1.37	1.43	1.32	1.47	1.40	1.38	1.48
Standard error	0.07	0.14	0.12	0.13	0.17	0.13	0.10	0.12	0.10

Tables prepared by Computers for Marketing Corp.

Page 1

Road Runner Fast Food Test Tables

Prepared on 18 MAR 1994

T A B L E     O F     C O N T E N T S

P. 1 Q1. How much do you agree with the following statement: The fast food at Road Runners is worth what I pay for it.

**DEFINING A PROCEDURE FOR COMPLEX BANNERS**

The most basic use of the DEFINE command is to define variables, for example:

```
~DEFINE
    Banvar[1/80^1//9]
```

Then you can reference the variable later in the same spec file:

```
COLUMN=Banvar
```

As you saw in the cleaning chapter, you can also use the DEFINE statement to define procedures. You can use this capability to define elements when you have complex tables. For example, you can use DEFINE to generate and store banner data under a simple punch data variable. This is effective if you have a very complex banner or many rows. If you define the column variable or expression with TABLE\_SET, Mentor will have to generate the horizontal axis for each table during the build and print phase. Using DEFINE will speed processing of the data file. Here are lines we would add to the previous Roadrunner spec file:

```
~DEFINE
    Age[1/51^1,2/3,4/5,6]
    Income[1/52^1/2,3/4,5,6]
    Rating[1/47^4,5/1,2,3]

    Banexpr: TOTAL WITH Age WITH Income WITH Rating

    Banvar[1/80^1//9]

PROCEDURE= {make_ban:
    MODIFY banvar = banexpr
```

```
DO_TABLES }
```

And later, in the EXECUTE block:

```
~EXECUTE  
    READ_PROCEDURE=MAKE_BAN
```

Here we defined the three data variables that will form an expression for the tables' column variable.

**NOTE:** No colon is needed after the varname since these are simple variables containing no joiners or functions. It is not necessary for you to pre-define the variable components of an expression, but it is easier to reference them later by their variable names rather than retyping the definition. Here is each line described in detail:

**Banexpr: TOTAL WITH Age WITH Income WITH Rating**

Defines an expression connecting all the categories from the data variables referenced by AGE, INCOME, and RATING. In BANEXPR a user-defined Total column is created with the Mentor constant, TOTAL. Program-generated Total and No Answer columns will be suppressed at print time and replaced by this total. Since we want Total to be the first column printed in the table banner, it is specified at the beginning of the expression.

**Banvar [1/80^1//9]**

Defines a new data variable for the nine categories created by BANEXPR: 1 for Total, 3 for Age, 3 for Income and 2 for Rating. The data will be stored in column 80 of record 1 of the RRUNR data file (TR) which is currently blank. Note that BANVAR is defined as a punch data variable. Two slashes (//) define nine separate but consecutive categories.

**PROCEDURE=**

The ~DEFINE block keyword that allows you to perform some operation on every case in the data file or specified subset.

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

**{make\_ban:**

The name of the procedure.

**MODIFY banvar = banexpr**

A CLEANER command that modifies data. In this case the result of the expression BANEXPR will replace the data in BANVAR, blanking the categories first. See *Mentor Appendix B: TILDE COMMANDS, ~CLEANER* for more information.

**DO\_TABLES**

Another ~CLEANER command. DO\_TABLES allows you to make data modifications for the printed tables without changing the input data file. This way, Mentor only has to check in one place to read a complicated expression, and it speeds processing. A DO\_TABLES procedure is executed with the ~EXECUTE keyword READ\_PROCEDURE.

**}**

Ends the procedure.

**READ\_PROCEDURE=MAKE\_BAN**

This keyword will execute the DO\_TABLES procedure defined above.

## 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

You can use Survent questionnaire specifications to produce two table building specification files, ready to be used by Mentor, called the DEF file and the TAB file. You can use the ~PREPARE COMPILE Mentor\_SPECS command or CfMC's menu-assisted EZWriter application to create these files. The DEF file contains the syntax to define a table for each question and the TAB file contains the commands to build the tables defined in the DEF file. The DEF file can also be edited to add

basing, weights, statistics, print format controls, etc. The TAB file can be remade by Mentor to reflect any changes to the DEF file. Refer to “4.11 SAMPLE SPECIFICATION FILES” for details on using these files to make tables. All examples refer to the RRUNR sample specification files included with your Mentor software.

Refer to 2.2.1 FILES CREATED BY Survent in your Survent manual for more information on the DEF and TAB files.

Here is a basic specification file followed by examples of the DEF, TAB and LPR files, and then the table they created. For the purposes of this example we used a converted Survent variable stored in the DB file, RRUNR.DB. In “4.6 DEFINING DATA”, you can see the rules to define data variables in general and how to join data variables to form a complex banner. “4.8 FORMATTING BANNER TEXT” covers how to format the banner text.

```
>PURGE_SAME
>RUN_LABEL="First Sample Table"
>DEFINE @STUDY rrunr
>PRINT_FILE @STUDY~
>USE_DB @STUDY~      opens the DB file for access to a converted
Survent variable.

~SET AUTOMATIC_TABLES
~SPEC_FILES @STUDY~

~DEFINE
    EDIT={defedt: -COLUMN_NA, -ROW_NA, COLUMN_WIDTH=7,
    RUNNING_LINES=1, TCON
    }

&@STUDY~^DEF      this calls in the RRUNR.DEF file
```

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

```
~INPUT @STUDY~
```

```
~EXECUTE
```

```
    COLUMN=qn21
```

**is the converted Survent variable for respondent sex assigned as the table column. Mentor will generate banner labels from this variable's text.**

```
    EDIT=defedt
```

```
    MAKE_TABLES
```

```
~END
```

&@STUDY~^DEF reads in RRUNR^DEF which contains the definitions for each table. Here is the section of the definition file for the example table printed below. See “4.3 DEFINING TABLE ELEMENTS” and “4.6 DEFINING DATA” for details on defining tables and data variables.

```
TABLE_SET= { qn1_z:
```

```
TITLE=:
```

```
    Q1. How much do you agree with the following  
    statement:
```

```
    The fast food at Road Runners is worth what I pay  
    for it.}
```

```
STUB=:
```

```
    (5) Completely agree
```

```
    (4) Somewhat agree
```

```
    (3) Neither agree nor disagree
```

```
    (2) Somewhat disagree
```

```
    (1) Completely disagree
```

```
    Don't Know/Refused to answer }
```



```
ROW=: [1/6^5//1/10]
}
```

MAKE\_TABLES reads in the RRUNR^TAB and the RRUNR^LPR files. RRUNR.TAB has a list of tabsets and triggers table building, for this example it would have one line:

```
TABSET qn1_Z
```

RRUNR^LPR has load and print statements for each table. For this example, it would have one line:

```
LOAD T001 PRINT
```

```
TABLE 001
```

```
P. 1 Q1. How much do you agree with the following statement:
      The fast food at Road Runners is worth what I pay for it.
```

	Total	Male	Female
Total	500	263	237
	100.0%	100.0%	100.0%
(5) Completely agree	88	49	39
	17.6%	18.6%	16.5%
(4) Somewhat agree	92	43	49
	18.4%	16.4%	20.7%
(3) Neither agree nor disagree	86	48	38
	17.2%	18.3%	16.0%

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

(2) Somewhat disagree	73 14.6%	36 13.7%	37 15.6%
(1) Completely disagree	86 17.2%	43 16.4%	43 18.1%
Don't Know/Refused to answer	75 15.0%	44 16.7%	31 13.1%

#### T A B L E O F C O N T E N T S

PAGE 1

P.1 Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for it.

Here is a list of what Mentor table building components are generated from Survent:

<b>question label</b>	Generates the variable name for that table's definition
<b>question text</b>	Generates the table title
<b>question type</b>	Determines the row data type
<b>numeric range</b>	Generates the stub label set, the row data categories and default statistics for NUM questions
<b>response codes</b>	Generates the row data categories for CAT and FLD questions
<b>response text</b>	Generates the stub label set for CAT and FLD questions

Here are examples of Survent specs and the Mentor tabsets Survent generates for the basic question types (CAT, NUM, FLD).



**CAT QUESTION**

<b>Survent specs</b>	<b>Mentor tabset</b>
{ QN21: 1/57	tabset={ qn21_z:
Q21. Respondent Sex	title=:
	Q21. Respondent Sex}
! CAT	stub=:
1 Male	Male
2 Female	Female }
}	'qn21 (1/2)
	row=: [1/57.1^1/2]
	}

For CAT questions the comment line ' 'qn21 (1/2) indicates the Survent response codes which might differ from the corresponding punch codes found in the data for that question.

**FLD QUESTION**

<b>Survent specs</b>	<b>Mentor tabset</b>
{status: 1/5	tabset= { status_z:
Marital status	title=:
	Marital status}
! FLD	stub=:
M Married	Married
S Single	Single
W Widowed	Widowed}
}	row=: [1/5.1#M/D/S/W]

## BASIC TABLES

### 4.12 USING *Survent* TO GENERATE *Mentor* SPECIFICATION FILES

}

For FLD questions, the row variable for multiple-response FLD questions defaults to the \*F modifier, meaning net all responses across columns.

## SURVENT SPECS

```
{QN11: 1/8.20
Which brands are you aware of?
!FLD, ,10
01 COKE
02 COCA-COLA CLASSIC
03 NEW COKE
04 DIET COKE
05 CHERRY COKE
06 CAFFEINE FREE COKE
07 DR. PEPPER
08 REGULAR DR. PEPPER
09 DIET DR. PEPPER
10 GATORADE
11 MELLO YELLOW
12 REGULAR MELLOW YELLOW
13 OTHERS
99 DON'T KNOW }
```

## MENTOR TABSET

```
tabset= { qn11_z:
title=:
```

```

Which brands are you aware of?}
stub=
    COKE
    COCA-COLA CLASSIC
    NEW COKE
    DIET COKE
    CHERRY COKE
    CAFFEINE FREE COKE
    DR. PEPPER
    REGULAR DR. PEPPER
    DIET DR. PEPPER
    GATORADE
    MELLO YELLOW
    REGULAR MELLOW YELLOW
    OTHERS
    DON'T KNOW}
' 'qn11 (01/02/03/04/05/06/07/08/09/10/11/12/13/99)
row=: [1/8.2, 1/10, 1/12, 1/14, 1/16, 1/18, 1/20, 1/22,&
      1/24,1/26 *f#
      01/02/03/04/05/06/07/08/09/10/11/12/13/99]
    }

```

If you specify a recode table to be [SAMEAS label] or do the same in Script Composer by going to the F5 screen and filling in "Use the recode table from question\_\_\_\_", the DEF file will reflect that by automatically assigning that table element's variable name to mean 'the same as'.

## MENTOR TABSET

```
tabset= { qn2b_z:
```

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

```
        title=:
Q2B. Please rate the following characteristics:
The quality of service.}
stub= qn2a_s    <-- means use the stub variable qn2a_s.
'qn2b(5/4/3/2/1/0)
row=: [1/8.1^5/4/3/2/1/10]
}
```

## NUM QUESTION

### Survent specs

```
{ QN20: 1/55.2
    Q20. How many children are in your household?

! NUM, , , 0, 10, , RF
}
```

### Mentor tabset

```
tabset= { qn20_z:
title=:
Q20. How many children are in your household?}
stub=:
    0-3
    4-5
    6-10
    RF
    [stat] Mean
```

```

[stat] Standard deviation
[stat] Standard error}
row=: [1/55.2#0-3/4-5/6-10/"RF" ] &
      $[mean,std,se] [1/55.2 *ranges=0,10,,RF,,]
    }

```

By default, minimum and maximum allowable values for NUM questions are broken out into one to five categories of numeric values in the DEF file depending on the difference between the high and low values:

**Difference between high and low values:**                      **Number of categories:**

1	1
2-8	2
9-20	3
21-50	4
>50	5

If the difference between high and low is over 1000 then the first four categories are evenly divided with the remainder into the fifth category. The ranges are not meant to reflect any proportional divisions in the data. They are there to help you set up your own categories.

## SURVENT SPECS

```

{house: 1/9.6
  What is the current value of your home?
  !NUM,,,100000-500000,,RF}

```

**MENTOR TABSET**

```

tabset= { house_z:
title=:
What is the current value of your home?}
stub=
    100000-100125
    100126-100250
    100251-100375
    100376-100500
    100501-500000
    RF
    [stat] Mean
    [stat] Standard deviation
    [stat] Standard error}
row=: [1/9.6#100000-100125/100126-100250/100251-
100375/100376-100500/ &
    100501-500000/"RF" ]&
    $[mean,std,se] [1/9.6 *ranges=100000,500000,,RF,,]
}

```

You can specify a value for exception codes to be used in calculating statistics. Using the above example, we will assign a value of zero (0) to the exception code RF in the Survent specification. This will be reflected in the \*Ranges specification on the row variable.

**SURVENT SPECS**

```

{QN20: 1/55.2
Q20. How many children are in your household?

```

```
!NUM, , , 0-10, , RF=0}
```

## MENTOR TABSET

```
tabset= { qn20_z:
title=:
    Q20. How many children are in your household?}
stub=:
    0-3
    4-5
    6-10
    RF
    [stat] Mean
    [stat] Standard deviation
    [stat] Standard error}
row=: [1/55.2 # 0-3/4-5/6-10/"RF" ] &
    $[mean,std,se] [1/55.2 *ranges=0,10,,RF=0]
}
```

## CHANGING WHAT APPEARS IN THE DEF FILE

There are several options you can specify in your Survent questionnaire specification file that changes what goes into the DEF file. You can use the `~SPEC_RULES` command, the `!MISC` command, backslash options and compiler commands. The backslash commands control where text appears (Survent only, Mentor only, or both) while `~SPEC_RULES`, `!MISC`, the compiler commands have options that make general changes to the DEF file. These commands are described in detail below.

**!MISC OPTION*****CATEGORY\_STATISTICS (CATSTATS)***

Allows you to suppress statistical testing on rows that have the `$_[STATS=]` option. Using `-CATEGORY_STATISTICS` will prevent statistical testing on rows other than means.

The only way to use this option is in its negative form, `-CATEGORY_STATISTICS`. Including `CATEGORY_STATISTICS` (without the minus sign) in your `~PREPARE` specs does nothing.

`CATEGORY_STATISTICS` adds `[-STATS]` in front of the category part of a row variable in the DEF file. If your `~PREPARE` specifications look like:

```
{HOURS: 72.2
```

```
!MISC -CATEGORY_STATISTICS
```

```
In a typical day, how many hours do you spend listening  
to the radio?
```

```
!NUM, , , 1-24, , DK, VA=4 }
```

then the row variable for the tabset `HOURS_Z` will look like this in the .DEFfile:

```
ROW=: $_[-STATS] [ 72.2 # 1-5/10-13/14-24/"DK"/"VA" ] &  
      $_[STATS,MEAN,STD] [72.2 *ranges=1,24, ,DK,VA=4, ]
```

If you use `-CATEGORY_STATISTICS` on a table that does not have a mean, no statistical tests will be run on the table, and you will get a warning similar to the following:

```
(WARN #5385) Table T0003 with STATS= ban1_st but no stattests  
( are you using NUMITEMS() ? )
```



To customize the statistics written to the .DEF file, use the COLUMN\_MEAN option to ~SPEC\_RULES command in your ~PREPARE specs. (See *Mentor*; Appendix B: ~SPEC\_RULES.)

### NUM\_EXCEPTIONS=STUBTITLE,STUBTITLE

Adds additional stubs (e.g. exception codes or numbers outside the range) to NUM type questions. All questions with [SAMEAS label] will have the same additional stubs in the tabset unless a new !MISC NUM\_EXCEPTIONS is in the question definition.

If your ~PREPARE specifications look like:

```
{LISTHRS: 72.2
!MISC NUM_EXCEPTIONS="Don't Listen","Don't Know"
In a typical day, how many hours do you spend listening
to the radio?
!NUM, , , 1-24, 99, DK, VA=4 }
```

then the tabset in the DEF file will look like:

```
tabset= {listhrs_z:
title=:
    In a typical day, how many hours do you spend
    listening to the radio? }
stub=:
    1-5
    6-9
    10-13
    14-24
```

## BASIC TABLES

### 4.12 USING *Survent* TO GENERATE *Mentor* SPECIFICATION FILES

```
Don't Listen
Don't Know
VA
[stat] Mean
[stat] Standard Deviation
[stat] Standard Error }
row=: [72.4 # 1-5/6-9/10-13/14-24/99/"DK"/"VA" ] &
$ [mean,std,se] [74.2 *ranges=1,24,99,DK,VA=4,]
}
```

**RANGES="..."**

Specifies which numeric ranges to use for rows. For NUM questions only. Any legal range will be passed to the DEF file; no error checking is made on the ranges.

If your ~PREPARE specifications look like:

```
{ QN1b: 7
!MISC RANGES="1//10/11-20/21-50/51-98/99"
Q1b. How many catalogs are delivered to your office
weekly?
!NUM, , ,1,99
}
```

then the tabset in the DEF file will look like:

```
tabset= { qn1b_z:
title=:
  1b. How many catalogs are delivered to your office
weekly?}
stub=:
  1
  2
  3
  4
  5
  6
  7
  8
```

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

```
9
10
11-20
21-50
51-98
99
[stat] Mean
[stat] Standard deviation
[stat] Standard error
}
row=: [1/7.2 # 1//10/11-20/21-50/51-98/99 ] &
      $[mean,std,se] [1/7.2 *ranges=1,99,,,,]
}
```

## RATING=n

Generates default statistics for rating scale CAT questions. N is the number of categories in the rating scale. You may exclude Don't Know or Refused categories by specifying a lower number. You may use -n to reverse the scale. This can only be specified in a Survent spec file, not from EZWriter.

```
{QN1: 1/6
!MISC rating=5
ON A SCALE OF 1 TO 10, WHERE 10 MEANS YOU LIKE IT A LOT
AND 1 MEANS
YOU LIKE IT A LITTLE, HOW WOULD YOU RATE KRAL FOR PER-
SONAL APPEAL?
!CATEGORY
    5 (5) LIKE IT A LITTLE
    4 (4)
    3 (3)
    2 (2)
    1 (1)
    0 DON'T KNOW
}
```

Below is the tabset generated to the DEF file. The statistics rows added to the stub definition and the statistics calculations on the row variable. (Remember to have the AUTOPUNCHES option set in the spec file to have the definition match the response codes.)

```

tabset= { qn1_z:
title=:
    ON A SCALE OF 1 TO 10, WHERE 10 MEANS YOU LIKE IT A
    LOT AND 1 MEANS YOU LIKE IT A LITTLE, HOW WOULD YOU RATE
    KRAL FOR PERSONAL APPEAL?}
stub=:
    (5) LIKE IT A LITTLE
    (4)
    (3)
    (2)
    (1)
    DON'T KNOW
    [stat] Mean
    [stat] Standard deviation
    [stat] Standard error
    }
' 'qn1 (5/4/3/2/1/0)
row=: [1/6.1^5/4/3/2/1/10] &
    $[mean,std,se] [1/6.1 *ranges=1-5]
    }

```

For a 10-point rating scale using a single column (one through ten, and X for Don't Know), use !MISC RATING=10. If the answers were in column seven, the row definition in the DEF file would look like:

```

row=: [7^1/2/3/4/5/6/7/8/9/10/11] &
    $[mean,std,se] subscript([7^1//0])

```

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

#### **SAMEAS\_XXXX**

If you use the Survent option of [SAMEAS label], the !MISC options are also retained from the specified question.

#### **USER\_TEXT="string of text"**

This option allows you to put additional text within the body of the PREPARE spec file. While this text will NOT appear in the DEF file (or your table), it is text associated with the question and can be displayed with the ~CLEAN PRINT\_LINES \v5 option. See *Mentor, Appendix B ~CLEAN PRINT\_LINES* for details. You can continue along line of text by putting an ampersand at the end of the USER\_TEXT line.

If your ~PREPARE specifications look like:

```
{LISTHRS: 72.2
!MISC USER_TEXT="This question was Q4 of last year's
survey." &
"It will be used again next year."
In a typical day, how many hours do you spend listening
to the radio?
!NUM,,,1-24,99,DK,VA=4 }

~CLEAN
PRINT_LINES "\v2s" LISTHRS
PRINT_LINES "\v5s" LISTHRS

~END
```

then your list file will include:

```
PRINT_LINES "\v2s" LISTHRS
```

In a typical day, how many hours do you spend listening to the radio?

```
PRINT_LINES "\v5s" LISTHRYS
```

This question was Q4 of last year's survey. It will be used again next year.

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

**Backslash options** can be used in either question text or response text to control where text is used, either by Survent only, Mentor only, or both:

- \+ Displays text only on Survent interviewing screens.
- \- Prints this text only on tables or when the variable is referenced (e.g., by utilities such as HOLE).
- \\* Does both of the above. (default)

```
{ qn17: 1/52
\+Q17. Into which of the following categories does your
total family income
\+fall?
-Q17. Respondent Income Category

! CAT
1 Under $15,000
2 $15,000 - $24,9994
3 $25,000 - $34,999
4 $35,000 - $44,999
5 $45,000 - $49,999
6 $50,000 or more
9 Don't know/Refused
}

tabset= { qn17_z:
title=:
Q17. Respondent Income Category}
stub=:
    Under $15,000
```



```

$15,000 - $24,999
$25,000 - $34,999
$35,000 - $44,999
$45,000 - $49,999
$50,000 or more
Don't know/Refused}
''qn17(1/2/3/4/5/6/9)
row=: [1/52.1^1/2/3/4/5/6/9]
}
```

In this example the question presented on the interviewer's screen was recast as a statement for the table title. Only the text after the \- is passed to the DEF file and only the text after the \+ will display on a Survent screen.

Backslash commands, by default, are not passed to Mentor variables or spec files. You must specify the ~SPEC\_RULES option USE\_PRINT\_ENHANCEMENTS; then when a Mentor variable is made or specifications are generated from Survent the following rules are followed for backslash commands found in the text.

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

For standard text or Survent only (\\*):

<b>\A \H \OR# \O# \_ \T \(#,##,##)</b>	are removed.
<b>\B \U \I \F \E \N \^&lt;character&gt; \^&lt;hexdigit&gt;&lt;hexdigit&gt;</b>	are passed to the Mentor variable.
<b>\</b>	is changed to \N.
<b>\@</b>	is replaced with the either the question label or number.
<b>\DC+fb+fb \DDC+fb+fb \C+fb+fb \C+f_</b>	are passed to the Mentor variable as is.
<b>\:label:^-#</b>	is replaced with ANSWERFROM(label)
<b>\:col.width: ,  :col: .width, or  :col: ^width</b>	are replaced with DATAFROM(loc.wid).
<b>\:label +offset#:  ^-width</b>	is replaced with DATAFROM(label).
<b>\#(label)</b>	is replaced with ANSWERFROM(label,ALL).
<b>\#(label,1,4)</b>	is replaced with ANSWERFROM(label,1,4).
<b>\[col.wid]</b>	is replaced with PHONETEXT(col.wid).



For *Mentor* only text (\-):

**\G \X \T** \<color specs> \~<letter> \^<char> are all passed to the  
\' \' " \ \W \U \I \F \B \E \N \-W \-U \-I \-F \-B *Mentor* variable.

Refer to your *Survent* manual 2.5 *QUESTION TEXT* for information on *Survent* backslash commands, and to your *Mentor* manual *Appendix B: TILDE COMMANDS*, *~CLEANER PRINT\_LINES* for information on *Mentor* print controls.

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

~SPEC\_RULES specified before ~PREPARE COMPILE Mentor\_SPECS causes Survent to generate optional specifications to the DEF file.

**Syntax:** ~SPEC\_RULES option option...

The options are BASE, BASE\_COMMENT, CLN\_CHECK, COLUMN\_MEAN, DO\_LOOPS, NO\_BASE, STORE\_TABLES, STUB\_DEFAULT, and USE\_PRINT\_ENHANCMENTS, and they are described in the following pages.

**Options:    BASE**

converts Survent !IF statements to BASE= and TITLE\_4= definitions in the Mentor DEF file. Note: !IF statements are passed to the DEF file exactly as written in the Survent specification file, e.g., !IF QN6(1) becomes BASE=: QN6(1) in the DEF file. This would require that variable QN1 be available in an open DB file in order to correctly tabulate the data. TITLE\_4 definitions substitute the response text for the response code referenced in the !IF statement, e.g.,(QN6 is Yes). To get around this limitation write your !IF statements by referencing data locations instead of question labels, e.g., !IF [1/31^1].

If your ~PREPARE specifications look like:

```
{ QN6A:  1/31
!IF [1/30^1]
```

Q6a. Which entertainment was participated in during the past three months?

```
! CAT, ,6
1 Video games
2 Billiards
3 Fun House
```

```

4 Musical Revue
5 Dunk the Moose
6 Other
(-) 7 Don't know/refused
}

```

then the tabset in the DEF file will look like:

```

tabset= { qn6a_z:
base=: ([1/30^1])
T4=:
([1/30^1])}
title=:
Q6a. Which entertainment was participated in during the
past three months?}
stub=:
    Video games
    Billiards
    Fun House
    Musical Revue
    Dunk the Moose
    Other
    Don't know/refused}
' 'qn6a(1/2/3/4/5/6/7)
row=: [1/31.1^1/2/3/4/5/6/7]
}

```

### **BASE\_COMMENT**

Converts Survent !IF statements to BASE= and TITLE\_4= definitions in the DEF file, but comments them out.

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

#### CLN\_CHECK

Causes the compile option ~PREPARE COMPILE CLEANING\_SPECS to write out a CHECK [datavar] instead of the default EDIT varname statement to the CLN file.

#### COLUMN\_MEAN=

Customizes the default statistics written to the DEF file. You must provide your own stub labels for the statistics rows generated with this option. Default statistics are written for NUM questions and questions written with the !MISC option explained above.

**Syntax:** COLUMN\_MEAN="a" with b

#### Options:

"a" is the specification line needed to calculate statistics on the row variable in the DEF file, e.g., \$[MEAN,STD,SE,MEDIAN]

b what text, if any, you want added to the STUB= line in the Mentor DEF file. You can use this for a previously defined DB item, as in the example below.

```
>CREATE_DB rrunr
  >FILE_TO_DB statstub #
  statstub:
    [STATISTICS_ROW] Mean
    [STATISTICS_ROW] Standard Deviation
    [STATISTICS_ROW] Median }
  >END_OF_FILE
```

```
~SPEC_RULES COLUMN_MEAN="$ [MEAN, STD, MEDIAN] " WITH
statstub
```

```

~PREPARE COMPILE Mentor_SPECS
Survent specifications ...
~END

tabset= { qn20_z:
title=:
Q20. Children}
stub=
0-3
4-5
6-10
RF
[STATISTICS_ROW] Mean
    [STATISTICS_ROW] Standard Deviation
    [STATISTICS_ROW] Median }
row=: [1/55.2#0-3/4-5/6-10/"RF" ] &
    $[MEAN,STD,PTILE=.5] [1/55.2 *ranges=0,10,,RF,,]
}

```

Here are some of the lines from the previous spec file explained:

```

>CREATE_DB rrunr          Creates the DB file rrunr.
>FILE_TO_DB              Converts the spec lines into db items and stores them in
                          the open DB file.
statstub                 The name of the db item.
#                         Has >FILE_TO_DB read the following lines until an
                          >END_OF_FILE command is reached.
statstub:                 Name of the STUB= variable defined here.
[STATISTICS_ROW] Mean    the stub label.

```

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

>END\_OF\_FILE The end of the specifications.

Compiling Specifications From The Mentor Program:

```
~DEFINE
    STUB=statstub:
    [STATISTICS_ROW] Mean
    [STATISTICS_ROW] Standard Deviation
    [STATISTICS_ROW] Median }
    same specifications as example above. . .
```

If you are compiling Survent specifications with the Mentor program then follow the example above to define a stub label set for the statistics rows generated by ~SPEC\_RULES COLUMN\_MEAN=.

## DO\_LOOPS

Converts Survent LOOP type questions into a Mentor loop variable in the DEF file. The tabulated data will reflect all iterations of the loop in the frequency.

-DO\_LOOPS turns this option off, but then you will only get the first iteration of the loop. The loops are explained more after these specs:

```
tabset= { phones_z:
    title=:
        Which of the following types of telephones do you
have?}
    stub=:
        Wall phone
        Desk phone
        Cordless phone
        Cellular phone
        Novelty phone (Garfield, Snoopy, Mickey Mouse, etc.)
```



```

        None of the above}
''phones(01/02/03/04/05/06)
row=: [1/5.1^1/2/3/4/5/6]
    }

tabset= { purchase_z:
title=:
    Where did you purchase your PHONES?}
stub=:
    From an ATT phone store
    From a department store
    From an audio/video/electronics specialty store
    From a mail order house
    None of the above}
''purchase(01/02/03/04/05)
row=: [ (5,3) 1/8.1 ^ 1/2/3/4/5]
    }

```

(5,3) in the previous example indicates the maximum number of times to loop (5) and the number of columns or increment between iterations of the loop (3). In this example, each iteration of the loop uses three data columns (two for the loop controller and one for the response) for a total of 15 columns if the loop is executed the maximum number of times. When the data is tabulated Mentor will check for a response to the purchase question in the third column of all five iterations of the loop, columns: 8; 11; 14; 17; and 20.

### **NO\_BASE**

Does not convert !IF statements (because the tab writer will define all of the bases).  
(DEFAULT)

**STORE\_TABLES**

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.

**Example:**

```

tabset= { qn1_z:
title=:
Q1. How much do you agree with the following statement:
The fast food at Road Runners is worth what I pay for it.}
stub=:
    (5) Completely agree
    (4) Somewhat agree
    (3) Neither agree nor disagree
    (2) Somewhat disagree
    (1) Completely disagree
    Don't Know/Refused to answer
    [STATISTICS_ROW] Mean
    [STATISTICS_ROW] Standard deviation
    [STATISTICS_ROW] Standard error}
'qn1 (5/4/3/2/1/0)
row=: [1/6.1^5/4/3/2/1/10] &
      $[mean, std, se] [1/6.1*ranges=1-6]
store=T001
    }

```

You cannot use the SET AUTOMATIC\_TABLES command with this option since ROW= causes Mentor to make a table with the next available table name (in this example T001). STORE= also causes Mentor to make tables with either the name

assigned to it or the next available table name (see SET TABLE\_NAME=). In the example above, STORE= is already assigned the next available default table name T001. Mentor would print an error message for a duplicate table name since ROW= would make T001 then STORE=T001 would remake the same table.

Other SPEC\_RULES keywords are:

### **STUB\_DEFAULT=[options]**

Allows you to specify default stub [options] for stub labels written to the STUB= definition in the DEF file.

### **USE\_PRINT\_ENHANCEMENTS**

Passes Survent text enhancements (e.g., \B) or Mentor print enhancements to the titles and stub labels in the DEF file. Refer to *Backslash options* earlier in this section for a list of which commands are passed.

### ***Response Code Options***

There are several options available which allow you to control the conversion of Survent response codes in the Mentor row variable in the DEF file. You can create subtotal (or netted) categories, suppress categories from the row variable, mark a category as exclusive, and control which categories to include in statistics calculations.

**Syntax:** (<options>) == (<punches or codes>) Response text

### **Options:**

#### **(<options>)**

The standard Survent response code options: offset, <->punch, and SKIPTO; and these Mentor options:

#### **(-DISPLAY)**

## BASIC TABLES

### 4.12 USING *Survent* TO GENERATE *Mentor SPECIFICATION FILES*

Suppresses this response during *Survent* interviewing. The default is DISPLAY.

#### **(EXCLUSIVE)**

Specifies this as an exclusive response during *Survent* interviewing. The default is -EXCLUSIVE. This is the same as saying minus (-) or -punch in standard *Survent* spec writing. The keyword allows you say this anywhere inside the options parentheses. In the DEF file exclusive categories are marked in the row variable with the keyword, e.g. [6^5/4/3/2/1/(exclusive)0].

#### **(-Mentor\_SPEC)**

Says do not include this response as a category in the row variable in the DEF file. The default is Mentor\_SPEC. This also suppresses the comment, e.g., "qn1(5/4/3/2/1/0), in the DEF file.

#### **(STATISTICS)**

Controls which categories will be included in statistics calculations (only tests defined on the ~DEFINE STATISTICS= statement). The default is STATISTICS for all categories. However, if you specify (STATISTICS) on any response item, then the default for all others is (-STATISTICS). The statistics categories are marked in the row variable with the keyword stats: [2/10.2^1/2/(stats)3/4/5].

Statistics defined in the row variable, e.g., \$[MEAN,STD,SE], are not affected by this option.

=

A special response code indicating that this is a non-response item during interviewing. There must be as many equal = signs as response code length. Please refer to your *C-Survent* manual for more information.

#### **(<punches or codes>)**

The codes for the row variable category generated in the DEF file; punches for CAT questions and codes for FLD questions.

If there are no parentheses ( )s or an empty set of parentheses, then a [COMMENT] line is included in the stub definition in the DEF file. The variable is stored in the open DB file with the additional categories.

### Response Text

Specifies the text that will appear during Survent interviewing and on the stub label for this item.

In the example below, we have created two netted categories: punches four and five (5,4); and punches one and two (2,1). These items will not display as responses during Survent interviewing (-DISPLAY) and will be the only categories included in tests defined on the ~DEFINE STATISTICS= statement (STATISTICS). The last response will not be included in the row variable (-Mentor\_SPEC), but it will be an exclusive response during Survent interviewing (EXCLUSIVE).

```
{ QN1: 1/6.1
!MISC RATING=5
Q1. How much do you agree with the following statement:
The fast food at Road Runners is worth what I pay for it.
! CAT
(-DISPLAY, STATISTICS) = (5,4) TOP BOX
5 (5) Completely agree
4 (4) Somewhat agree
3 (3) Neither agree nor disagree
(-DISPLAY, STATISTICS) = (2,1) BOTTOM BOX
2 (2) Somewhat disagree
1 (1) Completely disagree
(-Mentor_SPEC, EXCLUSIVE) 0 Don't Know/Refused to answer
}
```

Here is the TABLE\_SET definition created in the DEF file. Note the stub and row definitions. The equal sign = response items and text specified on question QN1

## BASIC TABLES

### 4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES

were converted to stub labels and corresponding netted categories on the row variable.

```
tabset= { qn1_z:
title=:
    Q1. How much do you agree with the follow-
    ing statement:
    The fast food at Road Runners is worth what I pay for
it.}
stub=:
    TOP BOX
    (5) Completely agree
    (4) Somewhat agree
    (3) Neither agree nor disagree
    BOTTOM BOX
    (2) Somewhat disagree
    (1) Completely disagree
[stat] Mean
[stat] Standard deviation
[stat] Standard error}
row=: [1/6.1 ^ ( stats)5,4/5/4/3/( stats)2,1/2/1] &
    $ [mean,std,se] [1/6.1 *ranges=1-5]
}
```

#### ***Compiler Commands***

These are commands that you put in your Survent questionnaire specifications. With them, you can control which questions get passed to the DEF file. In addition, you can include Mentor specifications in your Survent questionnaire and designate the program-generated file they should be passed to: CLN, DEF, or TAB.

**{!-DO\_Mentor}**

Allows you to exclude a question or block of questions from the DB file and DEF file. The default is to convert each CAT, FLD, and NUM question to a variable in the DB file and a TABLE\_SET definition in the DEF file. {!DO\_Mentor} turns the default back on.

**{!Mentor\_CLN}, {!Mentor\_DEF}, and {!Mentor\_TAB}**

Used to pass Mentor specifications to the CLN, DEF, or TAB file, respectively. The Survent compiler treats these blocks as comments and no syntax checking is done. When you compile your questionnaire with either the option Mentor\_SPECS or CLEANING\_SPECS, these blocks will be passed to the appropriate file.

```
{!Mentor_DEF
{TABLE_SET=tab1:
  EDIT= -COLUMN_TNA }_
  TITLE= table title }_
  ROW= [2/5^1//8]
} }
```

**NOTE:** The first closing brace on the last line ends the TABLE\_SET and the second closing brace ends the !Mentor\_DEF block.

Brace underscore (}\_) allows you to imbed a right brace (}) inside the compiler command block when it is the last character on a line that you want passed as part of a specification. In the example above EDIT= and TITLE= each require a closing right brace. The underscore is stripped when the specification is passed to the file and the right brace is not interpreted as the end of the command block. The specification passed to the DEF file will look similar to this:

```
{TABLE_SET=tab1:
  EDIT= -COLUMN_TNA }
  TITLE= table title }
  ROW= [2/5^1//8]
```

```
}
```

## USING E-TABS

Mentor makes files that are usable by E-Tabs. E-Tabs is a software package that has a Writer module which can take a Mentor (or any number of other software packages) print file and prepare it for viewing by a Reader module. The end user can browse the tables, print the tables, do specific keyword searches, and cut/paste portions of the tables to other software packages such as Excel, Word, or PowerPoint. It can also create charts or graphs. E-Tabs will create a few files, one with an extension of .ZTE (ebook.zte) which is the final output file which the user will use.

The user can then modify that file as long as they keep the extension of .ZTE. E-Tabs has a Dos/Windows version and also SCO/Sun Solaris Unix version.

The E-Tabs program needs these 3 files which Mentor now creates:

- tab1.prt (our normal tables excluding the table of contents)
- cfmctoc.ite (table of contents)
- ebook.ini (description of table regions)

These Mentor features are needed to create the E-Tabs files:

- Use `~set etabs_printfiles`

This says create the three E-Tabs files.

- You must use `>printfile` statements to provide the names of the three E-Tabs files.





for tables file use:

```
>print_file tab1 #1 page_width=132
```

for toc file use:

```
>print_file cfmctoc.ite,user -formfeed #2 page_width=2000
```

for ebook file use

```
>print_file ebook.ini,user -formfeed #3 page_width=2000
```

- You must add this edit option: tcon=(separate\_file)

Spec writers should observe the following rules:

- 1) Lines in the table of contents cannot wrap, so make the page width long enough to accommodate this.
- 2) Works best with Mentor 8.1 banners created with make\_banner.
- 3) Banner headers (age, income, etc.) should be on the same level.

**This format is preferable:**

AGE

INCOME

RATING

## BASIC TABLES

### USING E-TABS

	Un-der			Un-der			Neu-tral	
	35	35-54	54	\$15k	\$15-35k	Over \$35k	Good	/Poor
Total	35	35-54	54	\$15k	\$35k	\$35k	Good	/Poor

**This format is *not* preferable:**

	AGE			INCOME			RATING	
	Under	Over		Under	\$15-	Over		Neutral
	35	35-54	54	\$15k	\$35k	\$35k	Good	/Poor
Total	35	35-54	54	\$15k	\$35k	\$35k	Good	/Poor

4) Edit tcon=() options should include:

- tcon\_page\_numbers
- header
- footer
- table\_names

5) Do not use these edit tcon=() options:

- first
- print\_page\_numbers

6) To create the E-Tabs ebook.zte file from the CfMC print files, make sure the table of contents file is named cfmtoc.ite and the description of table regions file is named ebook.ini.



If you are working with more than one set of tables in the same directory you might want to use the same root name when creating the CfMC files (tab1.prt, tab1.ite, tab1.ini). Then using a batch file, rename the CfMC files before creating the ebook.zte file. After the E-Tabs file is created, the same batch file can rename the ebook.zte to tab1.zte.

## More E-Tab Options

You can now change .prt, .ini and .ite files when using ~set etabs\_printfiles.

**Example:** ~set etabs\_printfiles

```
>printfile mt3581a #1
>printfile mt3581b #2
>printfile mt3581c #3
~exc
tabset=one
set printtcon
>printfile
```

```
>printfile mt3581d #1
>printfile mt3581e #2
>printfile mt3581f #3
~exc
tabset=one
set printtcon
>printfile
```



# INTERMEDIATE TABLES

## INTRODUCTION

In Chapter 4 you learned how to define the parts of a table and how to write a basic specification file to print tables. This chapter goes into more detail about the options in that chapter, including defining variable expressions and controlling the format of what prints on a table. In addition, it covers how to control printing on a row by row or column by column basis, how to run tables against multiple banners, how to assign your own table names, and how to reprint tables stored in a database file.

The specification file ALTTABS.SPX prints an alternate set of tables using the Road Runner sample questionnaire to illustrate several of the options covered in this chapter. See \CFMC\Mentor\ROADRUNR\ALTTABS.SPX (DOS/UNIX) or ALTABSPX.ROADRUNR.CFMC (MPE).

## 5.1 EXPRESSIONS AND JOINERS

Chapter 4 describes how to define a simple data variable, meaning a variable that defines one data field. This section describes how to write expressions. Expressions allow you to write definitions made up of one or more data variables and might include joiners, mathematical operators, functions, or commands to calculate statistics or build special tables. Refer to Chapter 6 “Advanced Tables” for special tables and mathematical operators and “9.3.2 Functions” for information on functions.

The end of “4.6 DEFINING DATA” defines an expression that includes a user-defined total column and joins the categories of two data variables to form the table's column variable. You will use expressions to build statistical calculations into either of the table's axes, to base or weight your tables, and to create special tables such as break or overlay tables.

## INTERMEDIATE TABLES

### 5.1 Expressions and Joiners

The basic syntax for an expression is:

```
varname: varname/[variable] joiner varname/[variable]
```

**varname** the name of this expression (optional if defined inside a TABLE\_SET structure).

**:** colon follows immediately to indicate this is an expression.

**varname/[variable]** can be either the name of a previously defined variable or expression or the variable definition itself inside brackets ([ ]).

**joiner** any one of the logical, vector, or mathematical joiners listed below used to join or connect the variables on either side of it.

```
A: (B WITH C) BY ([5^N1] AND D)
```

Parentheses are helpful to indicate hierarchy in an expression with changing joiners. Here is a brief summary of each joiner type followed by a more detailed explanation and examples.

#### LOGICAL JOINERS

**AND** Both sides of joiner must be true, i.e., responses exist in the data.

**OR** Either side is true.

**NOT** Reverses the truth of the statement following it.

#### VECTOR JOINERS

WITH	Extends the list of categories.
BY	Crosses each category combination (all categories on the left by all categories on the right).
WHEN	Like BY, but only one category is returned from the right side of the joiner.
INTERSECT	If categories on both sides are true, then the combined category is true (acts as an AND NET operation).
NET	Nets the corresponding categories (acts as an OR NET operation).
OTHERWISE	Uses the first category set if present, otherwise the second category set.

### MATH JOINERS

Relational:	EQ or =	Equal
	NE or <>	Not equal
	LT or <	Less than
	GT or >	Greater than
	LE OR <=	Less than or equal
	GE OR >=	Greater than or equal
Operators:	*	Multiply
	/	Divide
	+	Add
	-	Subtract
	%	Percentage
	++	Add (even if some element is missing)
	*=,/=, +=, -=, %=	Performs the operation on the item to the left of the equal = sign.

### 5.1.1 Logical Joiners

Logical joiners evaluate an expression by first converting each variable referenced from multi-category to single category, then combining the categories to determine whether the entire expression is true. The categories are not considered separately, rather they are "netted" into a single logical category, the result of a which is always either TRUE or FALSE.

**Example:** example1: (a OR b) AND NOT(c OR d)

Regardless of how many categories are in A and B, they are treated as one category by the OR joiner.

**NOTE:** Parentheses are recommended to indicate hierarchy in an expression with changing joiners.

You are most likely to use logical joiners to define a table base, e.g., respondent is female and has children, though it would also be appropriate to use them in a complex banner.

**AND** Combines two or more expressions into a single logical category and is true if some category of its components are true, but false if any of its components are false.

```
example1: [1/51^1] AND [1/52^5,6]
           Age           Income
           |             |
           1 category   1 category
           <35           >$35k
```

Creates one category (respondents under age 35) AND (income >\$35,000). The respondent either falls into this age category AND the income category (TRUE) or does not (FALSE).



example2: [1/51^1,2/3,4/5,6] AND [1/52^1/2,3/4,5,6]

Age	Income
3 categories	3 categories
<35/35-54/>54	<\$15k/\$15-\$35k/>\$35k

Also creates one category (age is <35 or 35-54 or >54) AND (income is <\$15k or \$15-\$35k or >\$35k). The respondent either falls into one of the three age categories AND one of the three income categories (TRUE) or does not (FALSE).

**OR** Combines expressions into a single logical category and is true if any of its components are true, but false if all of its components are false.

example1: [1/51^1] OR [1/52^5,6]

Age	Income
1 category	1 category
<35	>\$35k

Creates one category (respondents under age 35) OR (income >\$35K). The respondent can either fall into this age category OR the income category to be TRUE, but will only be FALSE if age is greater than 35 (or not specified) AND income is less than \$35K (or not specified).

**NOT** Creates a category with the opposite truth value of the expression stated. Strictly speaking, NOT is a function rather than a joiner because it is always specified as NOT([variable]) and can be combined with either the OR or AND joiner in an expression. How you place parentheses can affect the meaning of the entire expression. Think of all locations within a NOT statement as having an AND NOT relationship. If you want to have an OR NOT relationship, separate the variables with NOT preceding each one:

## INTERMEDIATE TABLES

### 5.1 Expressions and Joiners

example1:  $\text{NOT}([1/51^{1,2}] \text{ OR } [1/52^{1-3}])$

This expression defines one category including only those respondents who have neither a 1 nor a 2 punch in record 1 column 51 AND have neither a 1, 2, nor 3 punch in record 1 column 52.

example2:  $\text{NOT}([1/51^{1,2}]) \text{ OR } \text{NOT}([1/52^{1-3}])$

This expression also defines one category: respondents who have neither a 1 nor a 2 punch in record 1 column 51 OR who have neither a 1, 2, nor 3 punch in record 1 column 52

To define an OR condition within a single data column, each punch position must be defined with its own NOT statement.

example3:  $\text{NOT}([1/51^1]) \text{ OR } \text{NOT}([1/51^2])$

You could also write this as  $\text{example3}:[1/51^N1]) \text{ OR } [1/51^N2]$  to mean the same thing, record 1 column 51 is not a 1 punch OR a 2 punch.

Although you can use multi-category expressions with a logical joiner, remember that the joiner lumps the categories together into one. The two examples below return the same result.

$[1/45^{1,2/3,4}] \text{ OR } [1/47^{1/2}]$

$[1/45^{1-4}] \text{ OR } [1/47^{1,2}]$

### 5.1.2 Vector Joiners

Vector joiners combine expressions to alter their meaning or form other expressions.

**WITH**

Appends the categories from two or more variables to form one expression with all categories, read and printed left to right. This joiner is used most often to generate a complex banner (see “4.11 SAMPLE SPECIFICATION FILES” for an example table).

```
example1: [1/51^1,2/3,4/5,6] WITH [1/52^1/2,3/4,5,6]
```

Age	Income
3 categories	3 categories
<35/35-54/>54	<\$15k/\$15-\$35k/>\$35k

1st category is punches 1 or 2 from 1/51  
 2nd category is punches 3 or 4 from 1/51  
 3rd category is punches 5 or 6 from 1/51  
 4th category is punch 1 from 1/52  
 5th category is punches 2 or 3 from 1/52  
 6th category is punches 4 or 5 or 6 from 1/52

**BY**

Creates categories for every combination of its component expressions. It crosses all categories to the left of 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. This joiner is most often used to generate either a complex banner or stub.

```
example1: [1/51^1,2/3,4/5,6] BY [1/52^1/2,3/4,5,6]
```

Age	Income
3 categories	3 categories
<35/35-54/>54	<\$15k/\$15-\$35k/>\$35k

This expression creates nine categories as follows:

1st category (<35 and <\$15k)

## INTERMEDIATE TABLES

### 5.1 Expressions and Joiners

2nd category	(35-54 and <\$15k)
3rd category	(>54 and <\$15k)
4th category	(<35 and \$15-35k)
5th category	(35-54 and \$15-35k)
6th category	(>54 and \$15-35k)
7th category	(<35 and >\$35k)
8th category	(35-54 and >\$35k)
9th category	(>54 and >\$35k)

### WHEN

WHEN acts to combine an expression and a logical expression, i.e., "A WHEN B" means that the categories in A are counted only when B is true.

example1: [1/51^1, 2/3, 4/5, 6] WHEN [1/52^6]

Age	Income
3 categories	1 category
<35/35-54/>54	>\$35k

This expression produces three categories:

1st category counts respondents under age 35 only WHEN income is >\$35k

2nd category counts respondents aged 35-54 only WHEN income is >\$35k

3rd category counts respondents over age 54 only WHEN income is >\$35k

**NOTE:** If the item on the right side of the WHEN is a multi-category item it will be converted into one category returning either TRUE or FALSE.

### INTERSECT

Combines the categories of the multi-category variable on the left and the multi-category variable on the right in an AND operation. If BOTH categories are true then the resulting category is true. The expression must have the same number of

categories on each side of the INTERSECT joiner. A practical example might be as follows:

$$\begin{array}{ccc} [1/10^1//5] & \text{INTERSECT} & [1/11^1//5] \\ \text{saw advertisement} & & \text{purchased product} \end{array}$$

where the left side of the expression represents a multi-category question determining whether the respondent saw an advertisement for products and the right side represents another multi-category question determining which products were purchased. The intersection of these two variables combines the categories on both sides of the joiner with an AND operation: to be counted the respondent must have a response in record 1 column 10 AND a response in record 1 column 11.

The example above will return five categories:

```
1st category  1/10^1 AND 1/11^1
2nd category  1/10^2 AND 1/11^2
3rd category  1/10^3 AND 1/11^3
4th category  1/10^4 AND 1/11^4
5th category  1/10^5 AND 1/11^5
```

## NET

Combines the categories of the multi-category variable on the left and the multi-category variable on the right in an OR operation. If EITHER category is true, then the resulting category is true. Like INTERSECT, the expression must have the same number of categories on each side of the joiner.

Referring to the previous example, NET would combine the categories on both sides of the joiner in an OR operation: count respondent if he has a response in record 1 column 10 OR has a response in record 1 column 11.

That example would return five categories:

```
1st category  1/10^1 OR 1/11^1
```

## INTERMEDIATE TABLES

### 5.1 Expressions and Joiners

```
2nd category  1/10^2 OR 1/11^2
3rd category  1/10^3 OR 1/11^3
4th category  1/10^4 OR 1/11^4
5th category  1/10^5 OR 1/11^5
```

A more practical example might be netting the categories for aided and unaided awareness questions to create a total awareness category.

### OTHERWISE

Checks for some response in the variable specified to the left of the joiner. If responses exist in any of the categories defined then those are used. If no categories in the variable on the left are true, then the categories defined in the variable to the right of the joiner are used.

OTHERWISE is most often used with single response variables, e.g., if male respondent otherwise female respondent. Use this joiner with caution for multi-response data, i.e., only if one side or the other of the joiner could have data and not both, otherwise only the responses found for the left side of the expression will be used. If both sides could have data you should use NET.

```
example1: [1/5^1] OTHERWISE [1/5^2]
```

Means count respondents who have a 1 punch in record 1 column 5, otherwise check for a 2 punch in the same column.

### JOIN

Combines multiple text variables or a text string within a text variable. For example:

```
m [30.2$] = STRIP([11.3$]) JOIN STRIP([21.3$])
```

The STRIP function strips leading and trailing blanks from the field.

***Excluding Respondents From A Table***

This example illustrates using an expression to base your tables. Respondents can be excluded from a table or group of tables by assigning either a filter or base during tabulation. There are three important distinctions to note, however.

First, a filter returns a single category (even if the variable referenced has multiple categories) whereas a base returns the same number of categories as there are in the variable referenced. For example, if your filter references a variable for respondent sex it will return one category regardless of the response in the data. A base referencing the same variable would return two categories, male and female. Using respondent sex as a base would produce two sets of tables, one for the male respondents and one set for the females.

Second, the ~SET option DROP\_BASE affects only the variable defined with BASE=, not FILTER=.

Finally, only BASE= generates an automatic TITLE\_4 definition from the base variable specified (see related command EDIT= TITLE\_4\_FOR\_BASE).

You can define a filter for a table (e.g., women respondents only) and then use a base (e.g., married with children) to further subset the filtered group.

```
TABLE_SET= { qn1_z:
FILTER=: [1/57^2]
TITLE_2=:Filter is women respondents only \N }
BASE=: [1/54^1] AND [1/55.2#1-10]
TITLE_4=:\Nbase is married with children }
TITLE=:
```

Q1. How much do you agree with the following statement:

## INTERMEDIATE TABLES

### 5.1 Expressions and Joiners

```
The fast food at Road Runners is worth what I pay  
for it.}
```

```
STUB=:
```

```
(5) Completely agree  
(4) Somewhat agree  
(3) Neither agree nor disagree  
(2) Somewhat disagree  
(1) Completely disagree  
Don't Know/Refused to answer }
```

```
ROW=: [1/6.1^5//1/10]  
}
```

`FILTER=` is the keyword that defines a table filter. In this example, record 1 column 57 must have a 2 punch for the case to be included in this table.

`TITLE_2=` is the keyword that defines a table title that will print directly above the title defined with `TITLE=`. `\N` (new line) will cause a blank line to print after the text.

`BASE=` is the keyword that defines a table base. In this example, we have defined an expression with the `AND` joiner to further subset the filter already defined: to be included in this table the case must have a 1 punch in record 1 column 54 `AND` a number in the range 1-10 in record 1 columns 55 and 56.

`TITLE_4=` is the keyword that defines a table title that will print directly below the title defined with `TITLE=`. As explained for `TITLE_2`, `\N` will print a blank line before printing the `TITLE_4` text.



```

>PURGE SAME
~INPUT rrunr,MAYBE_BACKUP,DOT=100
>DEFINE @STUDY base
>PRINT FILE @STUDY~
~SPEC_FILE @STUDY~

~DEFINE
  TABLE_SET={tabtop:
  SET AUTOMATIC TABLES

  HEADER=: =Road Runner Fast Food Sample Tables
  Prepared on #date# }

  FOOTER=: =Tables prepared by Computers for Marketing Corp.
  Page #page_number# }

  EDIT=: -COLUMN_TNA, -ROW_NA, PERCENT_DECIMALS=0,
         COLUMN_WIDTH=7, PAGE_WIDTH=95, RUNNING_LINES=2 }

  BANNER=:
  |
  |      <-----AGE----->  <-----INCOME----->  <---RATING--->
  |
  |      Under          Over  Under  $15-   Over          Neu-
  |  TOTAL      35  35-54    54  $15k   $35k   $35k   Good  tral/
  |  -----  -----  -----  -----  -----  -----  -----  -----  ----- }
  |

COLUMN=: TOTAL WITH &
         [1/51^^1,2/3,4/5,6] WITH &    ''RESPONDENT AGE
         [1/52^^1/2,3/4,5,6] WITH &    ''INCOME
         [1/47^^4,5/1,2,3]             ''RATING
}

TABLE_SET= { qn1_z:
FILTER=: [1/57^2]
TITLE_2=:Filter is women respondents only \N }
BASE=: [1/54^1] AND [1/55.2#1-10]
TITLE_4=: \Nbase is married with children }
TITLE=:
  Q1. How much do you agree with the following statement:

```

```
    The fast food at Road Runners is worth what I pay for it. }
STUB=:
    (5) Completely agree
    (4) Somewhat agree
    (3) Neither agree nor disagree
    (2) Somewhat disagree
    (1) Completely disagree
    Don't Know/Refused to answer }
ROW=: [1/6^5//1/10]
    }

~EXECUTE    MAKE_TABLES
~END
```

Road Runner Fast Food Sample Tables  
Prepared on 24 AUG 1994

TABLE 001  
Filter is women respondents only

Q1. How much do you agree with the following statement: The fast food at Road Runners is worth what I pay for it.

Base is married with children

	<-----AGE----->				<-----INCOME----->			<---RATING--->	
	TOTAL	Under 35	35-54	Over 54	Under \$15k	\$15- \$35k	Over \$35k	Good	Neu- tral/ Poor
Total	34 100%	11 100%	11 100%	4 100%	3 100%	11 100%	14 100%	8 100%	18 100%
(5) Completely agree	8 24%	-	5 45%	-	1 33%	4 36%	1 7%	1 13%	5 28%
(4) Somewhat agree	4 12%	3 27%	1 9%	-	-	2 18%	2 14%	2 25%	1 6%
(3) Neither agree nor disagree	4 12%	2 18%	-	-	1 33%	-	3 21%	1 13%	3 17%
(2) Somewhat disagree	7 21%	2 18%	3 27%	1 25%	-	2 18%	4 29%	3 38%	3 17%
(1) Completely disagree	9 26%	4 36%	2 18%	2 50%	-	3 27%	4 29%	1 13%	5 28%
Don't Know/Refused to answer	2 6%	-	-	1 25%	1 33%	-	-	-	1 6%

### 5.1.3 Mathematical Joiners And Operators

This class of joiners and operators is most likely used in data cleaning and data generation procedures. This section provides an overview of each mathematical joiner and operator.

Using relational joiners, expressions can be formed by defining a comparison of variables and/or numbers using: less than (LT or <), less than or equal to (LE or <=), greater than (GT or >), greater than or equal to (GE or >=), equal (EQ or =), and not equal (NE or <>).

Here are some examples for logical comparisons that check for subsets among categorical variables:

```
var1 > var2
```

Means that every category in var2 must be in var1 (i.e., var2 is a subset of var1), but var1 may have some category that var2 does not; use LT (<) to reverse the comparison.

```
var1 = var2
```

Means that var1 and var2 have exactly the same categories, including none at all.

```
var1 <> var2
```

Means each variable has some category that the other does not, but they may have some categories in common.

The operators +, -, /, \*, and % are used to define the addition, subtraction, division, multiplication, and percentage of variables and/or constants. The ++ operator also

performs addition but treats missing elements as zero, unless all elements are missing:

A ++ B

Means 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 (multi-category variables), ++ evaluates on a category by category basis (like NET).

An equal sign (=) after the operator means perform the operation on the element to the left of the equation:

A += B returns the result of A + B and puts it in A  
A /= B returns the result of A / B and puts it in A  
A -= B returns the result of A - B and puts it in A  
A \*= B returns the result of A \* B and puts it in A  
A %= B returns the percent A is of B and puts it in A

## 5.2 Axis Commands/Cross-Case Operations

Axis commands are only used in expressions that form either your horizontal or vertical table axis, meaning they operate across all cases in the data set when a table is made. Axis commands are used to calculate statistical computations or make special types of tables. Statistical calculations are included in the sample table run files (see \CFMC\Mentor\ROADRUNR (DOS/UNIX) or ROADRUNR.CFMC (MPE)) provided with your Mentor software. Refer to Chapter 6: “Advanced Tables” for examples of special types of tables, break and overlay.

Syntax:      **AXIS=** expression \$[keywords] expression \$[ ]expression

AXIS is optional. Axis command keywords must be specified inside brackets ([ ]) which are preceded by a dollar sign (\$). Separate more than one keyword inside the brackets with either a comma or a space. Output from these keywords will print on the table in the order specified.

## INTERMEDIATE TABLES

### 5.2 Axis Commands/Cross-Case Operations

This section describes how to include three statistical calculations (mean, standard deviation, and standard error) to the row variable definition and how to add the correct labelling to the stub set for the statistics rows. See *Appendix B: TILDE COMMANDS* under `~DEFINE AXIS=` for a complete list of `$_[keywords]`.

**\$\_[MEAN]** Calculates the mean of the variables that follow. Means can be computed on more than one numeric variable by connecting them with the `WITH` joiner or referencing them within the brackets of a data variable. You can also reverse values or assign categories different values. See “6.2.1 Means on Rating Scales Using the Variable Definition” for more details.

**\$\_[SE]** Calculates the standard error of the sample from the mean of the variable(s).

**\$\_[STD]** Calculates the standard deviation of the sample from the mean of the variable(s).

Here is a `TABLE_SET` definition that includes these three statistical tests. See “4.4 TABLE BUILDING (THE INPUT AND EXECUTE STATEMENTS)” for information on `TABLE_SET`.

```
TABLE_SET= { qn4_z:
TITLE=:
    Q4. About how much do you pay per visit for Road
    Runner fast food - that is, not including entertain-
    ment? }
STUB=:
    $ 5-$10
    $11-$15
    $16-$20
    $21-$25
    $26-$30
    $31-$35
    $36-$40
```

```

$41-$45
$46-$50
Refused
[STATISTICS_ROW] Mean
[STATISTICS_ROW] Standard deviation
[STATISTICS_ROW] Standard error}
ROW=: [1/15.2# 5-10/11-15/16-20/21-25/26-30/31-35/36-
40/41-45/46-50/"RF" ] &
    $ [MEAN, STD, SE] [1/15.2 *RANGES=5-50]
}
ROW=: [1/15.2# 5-10/11-15/16-20/21-25/26-30/31-35/36-40/41-45/46-50/"RF" ]
defines the data type and its categories for the row variable.

```

&                      ampersand continues the definition to the next line.

\$(MEAN,STD,SE)        says calculate these statistics across all cases for this table.

[1/15.2                      is the location of data to be used in the statistical calculations.

\*RANGES=5-50]        specifies which data categories to include (minimum value of 5 through maximum value of 50) in the statistical calculations for this variable. (see “4.6 DEFINING DATA”)

Note that the stub label set has three lines at the end for the statistics calculated in the row variable.

[STATISTICS\_ROW] is referred to as a stub option. The keyword STATISTICS\_ROW is required to identify this as a statistics row. By default no percent sign will print and the number of decimal places of significance is one unless otherwise specified (see EDIT= STATISTICS\_DECIMALS= in “5.3 Changing Table Specifications”, *Print Options*).

Mean                      is the label that will print for this table row.

INTERMEDIATE TABLES

5.2 Axis Commands/Cross-Case Operations

Here is a sample table generated from this TABLE\_SET definition using the same banner and edit controls from the model table run described in "4.11 SAMPLE SPECIFICATION FILES".

Road Runner Fast Food Sample Tables  
Prepared on 13 AUG 1994

TABLE 001

Q4. About how much do you pay per visit for Road Runner fast food - that is, not including entertainment?

	<-----AGE----->			<-----INCOME----->			<--RATING-->		
	TOTAL	Under 35	35-54	Over 54	Under \$15k	\$15- \$35k	Over \$35k	Good	Neu- tral/ Poor
	-----	-----	-----	-----	-----	-----	-----	-----	-----
Total	500 100%	141 100%	140 100%	143 100%	74 100%	148 100%	215 100%	166 100%	247 100%
\$ 5-\$10	58 12%	23 16%	21 15%	10 7%	10 14%	14 9%	29 13%	23 14%	26 11%
\$11-\$15	52 10%	15 11%	16 11%	12 8%	6 8%	14 9%	22 10%	15 9%	27 11%
\$16-\$20	55 11%	11 8%	18 13%	17 12%	6 8%	14 9%	26 12%	21 13%	29 12%
\$21-\$25	42 8%	14 10%	9 6%	12 8%	9 12%	14 9%	13 6%	14 8%	19 8%
\$26-\$30	53 11%	16 11%	16 11%	12 8%	5 7%	12 8%	26 12%	11 7%	29 12%
\$31-\$35	47 9%	12 9%	9 6%	17 12%	6 8%	16 11%	18 8%	15 9%	27 11%
\$36-\$40	52 10%	12 9%	15 11%	18 13%	11 15%	14 9%	24 11%	16 10%	23 9%
\$41-\$45	51 10%	15 11%	17 12%	13 9%	10 14%	16 11%	21 10%	19 11%	23 9%



\$46-\$50	45 9%	12 9%	10 7%	18 13%	7 9%	21 14%	11 5%	16 10%	23 9%
Refused	45 9%	11 8%	9 6%	14 10%	4 5%	13 9%	25 12%	16 10%	21 9%
Mean	27.28	26.10	25.98	29.71	28.39	29.33	25.93	27.11	27.22
Standard deviation	13.27	13.77	13.55	12.91	13.52	13.62	13.00	13.81	13.11
Standard error	0.62	1.21	1.18	1.14	1.62	1.17	0.94	1.13	0.87

Tables prepared by Computers for Marketing Corp.

## 5.3 CHANGING TABLE SPECIFICATIONS

This section covers how you can control the format of what is printed on your tables either for all tables, or on a row by row or column by column basis. Printing controls allow you to override defaults for what is printed in the table's cells (e.g., frequency or percent only) and the format of what is printed (e.g., number of decimal places or percent sign), to alter the printed order of the tables rows (ranking), to enhance readability of tables (comments, underlining, and blank lines), to control what summary information is printed, and to control overall format such as page size, column and row widths.

Several of the print options covered in this section are illustrated in the sample table run `ALTTABS.SPX` in `\CFMC\Mentor\ROADRUNR` (DOS/UNIX) or `ALTABSPX.ROADRUNR.CFMC` (MPE).

### GLOBAL PRINT OPTIONS

The edit statement defines a format for printing all tables. In a given table run you could have many edit statements to control printing for specific tables or groups of

## INTERMEDIATE TABLES

### 5.3 Changing Table Specifications

tables (see *Appendix B: TILDE COMMANDS*, `~EXECUTE LOCAL_EDIT=`). Edit statements are defined, like all other table elements, in the `~DEFINE` program block with the keyword `EDIT=` followed by any number of the allowable edit options. You can include edit statements in a `TABLE_SET` structure as part of an entire table definition (see “4.11 *SAMPLE SPECIFICATION FILES*”).

#### **Syntax:** `~DEFINE`

`EDIT={name:options }`

`EDIT=` is the `~DEFINE` keyword used to specify table and page formatting controls.

{ left brace marks the beginning of the definition. (OPTIONAL)

name is the name of this definition. (OPTIONAL if defined within a `TABLE_SET` structure).

: colon immediately follows name.

options are separated by commas or one or more spaces. Options can be continued on as many lines as needed, with no continuation character needed.

} ends the definition.

## COLUMN PRINT OPTIONS

The `EDIT=` option `COLUMN_INFO=` allows you to specify what is printed on a table on a column by column basis. For example, you could specify a width for all columns or a different width for each column of the table with the `COLUMN_INFO` sub-option `WIDTH=`. This section includes a sample table to

illustrate this option. See *Appendix B: TILDE COMMANDS*, `~DEFINE EDIT=`  
`COLUMN_INFO=` for all column options.

**Syntax:** `EDIT={ edit1: COLUMN_INFO=(a/b/c/d/e/f) }`

- a is options for first printed column (i.e., System Total)
- b is options for second printed column, etc. (i.e., System No Answer).
- c is options for third printed column (i.e., first user-defined printed column), etc.

If you suppress the system Total and No Answer columns then the first user-defined column (e.g., TOTAL WITH) becomes the first printed column. See later in this section for more information on specific column options.

## ROW PRINT OPTIONS

Control what is printed on a table on a row by row basis with options specified on the `~DEFINE STUB=` statement.

**Syntax:** `~DEFINE`  
`STUB={ <name>:`  
`[options] row text or blank`  
`}`

`STUB=` is the `~DEFINE` keyword used to define the row labels or stub label set.

`{` left brace marks the beginning of the definition. (OPTIONAL)

`name` is the name of this definition. (OPTIONAL if defined within a `TABLE_SET` structure).

## INTERMEDIATE TABLES

### 5.3 Changing Table Specifications

: colon immediately follows name.

[options] are separated by commas, or one or more spaces.

row text or blank is the text that will print on the table for this row.

} ends the definition.

Where applicable, options or keywords are the same whether applied to all tables (EDIT=), a single column (EDIT=COLUMN\_INFO=), or a row (STUB=[option]). This section covers only selected options you will use often. Please see *Appendix B: TILDE COMMANDS*, ~DEFINE EDIT= for a complete list of edit options and ~DEFINE STUB= for stub options.

### SAMPLE TABLE PRINTED WITH DEFAULT OPTIONS

In the following table, notice the defaults:

System Total row and column

System No Answer row and column

Column width: 8 spaces

Row label width: 20 spaces

Frequencies with no decimal places

Frequencies with a value of zero print as a dash (-) in the cell

Vertical percentaging off the Total row to 1 decimal point.

Percent sign (%) prints. No horizontal percentaging.

Page length = 60 lines

Page width = 132 columns

TABLE 001

Q1. How much do you agree with the following statement:  
 The fast food at Road Runners is worth what I pay for it.

	Total	N/A	Male	Female
Total	500 100.0%	-	263 100.0%	237 100.0%
N/A	-	-	-	-
(5) Completely agree	88 17.6%	-	49 18.6%	39 16.5%
(4) Somewhat agree	92 18.4%	-	43 16.4%	49 20.7%
(3) Neither agree nor disagree	86 17.2%	-	48 18.3%	38 16.0%
(2) Somewhat disagree	73 14.6%	-	36 13.7%	37 15.6%
(1) Completely disagree	86 17.2%	-	43 16.4%	43 18.1%
Don't Know/Refused to answer	75 15.0%	-	44 16.7%	31 13.1%

## PRINT OPTIONS

Cell Manipulation Options:	Description	Default
COLUMN_INFO=(options)	Specifies column-specific options such as width, percent decimals, statistics, etc.	EDIT= options
-FREQUENCY <sup>1</sup>	Suppress printing frequencies in the cells; prints percentages only.	Print frequency
FREQUENCY_ONLY <sup>2</sup>	Prints frequency only, with no percentages.	Print frequency and vertical percent
FREQUENCY_DECIMALS=# <sup>1</sup>	Specifies the number of decimal places to print for frequencies. # can be 0-7.	0 (print whole number frequencies)
HORIZONTAL_PERCENT <sup>1</sup>	Prints horizontal percents off the System Total column.	Vertical percent
HORIZONTAL_PERCENT=x <sup>1</sup>	Specifies the horizontal percent base; x can be T for Total, AR for Any Response column, or # to specify the number of a particular column.	Total column
NUMBER_FORMAT=# <sup>1</sup> printed  mas	Prints frequencies in cells with commas and/or a dollar sign.  # can be 0 for default, 1 for commas, or 2 for commas and a dollar sign.	Numbers  without commas  or dollar sign.



**NOTE:** This option does not affect [STATISTICS\_ROW] unless specifically specified as a stub option.

PERCENT_DECIMALS=# <sup>1</sup>	Specifies number of decimal places to print for vertical and horizontal percents. Can be 0-7.	1
-PERCENT_SIGN <sup>1</sup>	Suppresses printing of % sign	Print %

---

An option is only available on the EDIT= statement unless marked as follows:

1. Means this option is available on EDIT=, EDIT=COLUMN\_INFO, and STUB=[option].
2. Means this option is available on both EDIT= and STUB= [option]

[ option] means this option is only available as a STUB [option]

**Cell Manipulation**  
**Options:** *(continued)*

**Description**

**Default**

STAR_PERCENT=#	Prints an asterisk (*) in the percent line of the cell if the value is less than the value specified here.	-1
	# can be -1, 0, or .001-100: 0 do not print asterisk -1 if PERCENT_DECIMALS=0 then print * when the percent is less than .5; if PERCENT_DECIMALS=1 then print * when the percent is less than .05, and if PERCENT_DECIMALS=2 then print * when the percent is	

## INTERMEDIATE TABLES

### 5.3 Changing Table Specifications

less than .005. (**Note:** You cannot specify -1.)

# print \* if the percent in the cell is less than this number.

**Note1:** A footnote showing the STAR\_PERCENT value prints on the last line of the last page (in the print position as a footer) of any table that has \* in a percent cell.

**Note2:** This option is usually used to note numbers that print as if they were 0%, but are actually slightly above 0%, especially on a percentage-only table or where very few respondents answered in some category.

STATISTICS_DECIMALS=# <sup>1</sup>	Number of decimal places to print for statistics. # can be 0-7.	1
TFRP	Total, No Answer, and base rows print as frequency only; all other rows print as percents only. (TFRP=Total frequency, Rows percents)	Frequency/ vertical percent
VERTICAL_PERCENT= <sup>1</sup>	Specifies the vertical percentage base for the table. Can be T for system Total row, AR for the Any Response row, >=1 for that data row, or (col,row) to specify a particular cell. Does not suppress frequency.	Total row

---

An option is only available on the EDIT= statement unless marked as follows:

1. Means this option is available on EDIT=, EDIT=COLUMN\_INFO, and STUB=[option].

2. Means this option is available on both EDIT= and STUB=[option]





[ option] means this option is only available as a STUB [option]

<b>Summary Information Manipulation Options:</b>	<b>Description</b>	<b>Default</b>
-COLUMN_NA	Suppress printing of No Answer column.	Prints
-COLUMN_TNA	Suppress printing of both the Total and No Answer columns.	Prints
NUMBER_OF_CASES	Prints the number of cases above the table title.	Does not print
-ROW_NA	Suppress printing of system No Answer row.	Prints
-ROW_TNA	Suppress printing of both the system Total and No Answer rows.	Prints
<b>Division Between Columns/Rows:</b>	<b>Description</b>	<b>Default</b>
COLUMN_WIDTH=#	Number of spaces to allot for each column of data	8
STUB_WIDTH=#	Number of spaces to allot for stub labels.	20
SKIP_LINES=# <sup>2</sup>	Number of lines to skip between rows. SKIP_LINES=0 condenses printing to fit the maximum number of rows per page.	1

---

An option is only available on the EDIT= statement unless marked as follows:



1. Means this option is available on EDIT=, EDIT=COLUMN\_INFO, and STUB=[option].
2. Means this option is available on both EDIT= and STUB= [option]

[ option] means this option is only available as a STUB [option]

Page Configuration Options:	Description	Default
BOTTOM_MARGIN=#	Specifies a margin of # lines at the bottom of each page.	6
CONTINUED=	Specifies what to print for continued pages of a table. Can be NONE (no indication of continuation), TOP prints (continued) next to the table number on the second and subsequent pages of the table, BOTTOM prints (continued) on the bottom of the page to indicate continuation.	TOP
CONTINUED_LOCATION	Specifies where to print “(continued)” on tables that span more than one page. Can be NONE, TOP, TOP_CENTER, TOP_RIGHT, BOTTOM, BOTTOM_CENTER, or BOTTOM_RIGHT.	TOP
CONTINUED_NUMBER	Adds a number suffix to the table name of tables that span more than one page. Can be NONE, AFTER_TABLE_NAME, or AFTER_CONTINUE. See <i>Mentor, Volume Two, Appendix B, ~DEF EDIT</i> CONTINUED_NUMBER for examples.	NONE
DATA_INDENT=# <sup>2</sup>	Specifies the number of spaces to indent the data columns.  <b>NOTE:</b> Data is right-justified according to the specified COLUMN_WIDTH. This option allows you to indent the data beyond the normal print position. It is useful to set off data in the table. Column headings are not affected by the indenting of the data columns.	0
INDENT=	Indents the entire table, text labelling included. Options are to	None



CENTER or the number of spaces to indent. Useful if tables are bound in a binder.

[STUB\_INDENT=#]

Specifies number of spaces to indent this stub. Useful to set off a row such as Sigma or Super-Sigma. You can also use a vertical bar (|) as a placeholder to print blank spaces before the start of the text. None

---

An option is only available on the EDIT= statement unless marked as follows:

1. Means this option is available on EDIT=, EDIT=COLUMN\_INFO, and STUB= [option].
2. Means this option is available on both EDIT= and STUB= [option]

[ option] means this option is only available as a STUB [option]

<b>Page Configuration</b> <b>Options:</b> <i>(continued)</i>	<b>Description</b>	<b>Default</b>
STUB_WRAP_INDENT=#	Number of spaces to indent subsequent stub lines when the label continues to more than one line. Default is 0 (do not indent), but may be any number 1+.	0
TOP_MARGIN=#	Number of rows to leave blank at the top of each page.	0
<b>Row Manipulation</b> <b>Options:</b>	<b>Description</b>	<b>Default</b>
[COMMENT]	Prints a comment label, with no corresponding row of data. This text will wrap at STUB_WIDTH=.	none
[N]	Kicks a single blank line before printing this row.	none
[P]	Kicks to a new page before printing this row.	none
[LONG_COMMENT]	Like [COMMENT] but will print as far across the page as there are characters, wrapping at the page width.	none
RANK_IF_INDICATED	Checks the stub label set for any [RANK_LEVEL=] commands and ranks the table rows if any are specified.	No ranking



RANK\_LEVEL=<sup>2</sup>

Level to rank all rows or specific row. Can be a number 0-9, where 0 means do not rank. Used as a stub [option] it can also be L for low, H for high, or some combination of #L or #H. See also [KEEP\_RANK=].

Level 1

---

An option is only available on the EDIT= statement unless marked as follows:

1. Means this option is available on EDIT=, EDIT=COLUMN\_INFO, and STUB=[option].
2. Means this option is available on both EDIT= and STUB=[option]

[ option] means this option is only available as a STUB [option]

<b>Row Manipulation</b>	<b>Description</b>	<b>Default</b>
-------------------------	--------------------	----------------

**Options:** *(continued)*

[STATISTICS_ROW]	Specifies this as a statistics row. See section “5.2 Axis Commands/Cross-Case Operations” for example specifications and table.	none
------------------	--	------

STUB_PREFACE=	Allows you to define your own summary row label set to be pre-pended to all tables; can be subset name, TNA, NONE, or ; to turn off a previous setting.	Total N/A
---------------	---	--------------

**NOTE:** This is very useful if you will be changing print options for these summary rows often in a run. By defining a stub preface you can use any allowable stub [option] to control these rows. See also STUB\_SUFFIX.

UNDERLINE <sup>2</sup>	Underlines this row. Default character is a dash (-), but allows a user-specified character with UNDERLINE=<character>	none
------------------------	--	------

<b>Miscellaneous Options:</b>	<b>Description</b>	<b>Default</b>
-------------------------------	--------------------	----------------

RUNNING_LINES=#	Controls how table text will be printed, i.e., titles, headers, footers. # can be 0 (default) print as written, 1 to wrap lines according to any PAGE_WIDTH= or INDENT= setting, and 2 which means print the first line like RUNNING_LINES=1, then indent the second and subsequent lines by the length of the first word in line one and any blanks immediately	0
-----------------	--	---





following.

**NOTE:** Text positional characters (= center, < left-justify (default), > right-justify) automatically set RUNNING\_LINES to 0 for that item. Table of Contents is not affected by RUNNING\_LINES, titles are printed according to the specified page width, with all lines indenting according to the default TCON format for titles.

TCON	Prints a table of contents of all tables, table number and title.	No TCON printed
------	---	-----------------

---

An option is only available on the EDIT= statement unless marked as follows:

1. Means this option is available on EDIT=, EDIT=COLUMN\_INFO, and STUB= [option].
2. Means this option is available on both EDIT= and STUB= [option]

[ option] means this option is only available as a STUB [option]

**Sample Tables**

*Title Print Positions*

Text specified on HEADER= prints on line one of the table.

TABLE 001

Text specified on TITLE\_2= prints on line three.

Text specified on TITLE= prints on line four.

Text specified on TITLE\_4= prints on line five.

	Total	N/A	Male	Female
Total	500 100.0%	-	263 100.0%	237 100.0%
N/A	-	-	-	-
(5) Completely agree	88 17.6%	-	49 18.6%	39 16.5%
(4) Somewhat agree	92 18.4%	-	43 16.4%	49 20.7%
(3) Neither agree nor disagree	86 17.2%	-	48 18.3%	38 16.0%
(2) Somewhat disagree	73 14.6%	-	36 13.7%	37 15.6%
(1) Completely disagree	86 17.2%	-	43 16.4%	43 18.1%
Don't Know/Refused to answer	75 15.0%	-	44 16.7%	31 13.1%

Text specified on TITLE\_5= prints as a footnote on each page.

Text specified on FOOTER= prints on the last line of each page.

**RUNNING\_LINES=2 PAGE\_WIDTH=60**

TABLE 001

Q1. How much do you agree with the following statement:  
 The fast food at Road Runners is worth what I pay for it.

	Total	N/A	Male	Female
Total	500 100.0%	-	263 100.0%	237 100.0%
N/A	-	-	-	-
(5) Completely agree	88 17.6%	-	49 18.6%	39 16.5%
(4) Somewhat agree	92 18.4%	-	43 16.4%	49 20.7%
(3) Neither agree nor disagree	86 17.2%	-	48 18.3%	38 16.0%
(2) Somewhat disagree	73 14.6%	-	36 13.7%	37 15.6%
(1) Completely disagree	86 17.2%	-	43 16.4%	43 18.1%
Don't Know/Refused to answer	75 15.0%	-	44 16.7%	31 13.1%

**-PERCENT\_SIGN**

TABLE 001

Q1. How much do you agree with the following statement:  
 The fast food at Road Runners is worth what I pay for it.

	Total	N/A	Male	Female
Total	500 100.0	-	263 100.0	237 100.0
N/A	-	-	-	-

**INTERMEDIATE TABLES***5.3 Changing Table Specifications*

(5) Completely agree	88	-	49	39
	17.6		18.6	16.5
(4) Somewhat agree	92	-	43	49
	18.4		16.4	20.7
(3) Neither agree nor disagree	86	-	48	38
	17.2		18.3	16.0
(2) Somewhat disagree	73	-	36	37
	14.6		13.7	15.6
(1) Completely disagree	86	-	43	43
	17.2		16.4	18.1
Don't Know/Refused to answer	75	-	44	31
	15.0		16.7	13.1

**FREQUENCY\_ONLY, FREQUENCY\_DECIMALS=2**

TABLE 001

Q1. How much do you agree with the following statement:  
 The fast food at Road Runners is worth what I pay for it.

	Total	N/A	Male	Female
Total	500.00	0.00	263.00	237.00
N/A	0.00	0.00	0.00	0.00
(5) Completely agree	88.00	0.00	49.00	39.00
(4) Somewhat agree	92.00	0.00	43.00	49.00
(3) Neither agree nor disagree	86.00	0.00	48.00	38.00
(2) Somewhat disagree	73.00	0.00	36.00	37.00
(1) Completely disagree	86.00	0.00	43.00	43.00
Don't Know/Refused to answer	75.00	0.00	44.00	31.00

INTERMEDIATE TABLES  
5.3 Changing Table Specifications

**-FREQUENCY\_ONLY (prints percents only), HORIZONTAL\_PERCENT,  
PERCENT\_DECIMALS=2, STAR\_PERCENT=-1 (default)**

TABLE 001

Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for it.

	Total	N/A	Male	Female
Total	100.00%	*	52.60%	47.40%
	100.00%		100.00%	100.00%
N/A	*	*	*	*
(5) Completely agree	100.00%	*	55.68%	44.32%
	17.60%		18.63%	16.46%
(4) Somewhat agree	100.00%	*	46.74%	53.26%
	18.40%		16.35%	20.68%
(3) Neither agree nor disagree	100.00%	*	55.81%	44.19%
	17.20%		18.25%	16.03%
(2) Somewhat disagree	100.00%	*	49.32%	50.68%
	14.60%		13.69%	15.61%
(1) Completely disagree	100.00%	*	50.00%	50.00%
	17.20%		16.35%	18.14%
Don't Know/Refused to answer	100.00%	*	58.67%	41.33%
	15.00%		16.73%	13.08%

**NOTE:** Percentage less than 0.005 printed as \*.

**SKIP\_LINES=0**

TABLE 001

Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for it.

	Total	N/A	Male	Female
Total	500	-	263	237
	100.0%		100.0%	100.0%
N/A	-	-	-	-
(5) Completely agree	88	-	49	39
	17.6%		18.6%	16.5%
(4) Somewhat agree	92	-	43	49
	18.4%		16.4%	20.7%
(3) Neither agree nor disagree	86	-	48	38
	17.2%		18.3%	16.0%
(2) Somewhat disagree	73	-	36	37
	14.6%		13.7%	15.6%
(1) Completely disagree	86	-	43	43
	17.2%		16.4%	18.1%
Don't Know/Refused to answer	75	-	44	31
	15.0%		16.7%	13.1%

**COLUMN\_INFO= Controlling Column Widths**

This example overrides the default column width of eight spaces for columns two, four, five, six, and seven to create a more even-appearing spacing between some of the banner points, and an obvious spacing between the different groupings (TOTAL, SEX, and AGE). Only the columns named on the COLUMN\_INFO option are affected. For a discussion on the banner used in this example refer to “4.8 FORMATTING BANNER TEXT”.

```
EDIT={edit1: COLUMN_INFO=(column=2 width=10/column=4 width=10/
column=5 width=7/column=6 width=7/column=7 width=9) }
```

TABLE 001

Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for it.

INTERMEDIATE TABLES

5.3 Changing Table Specifications

	<---SEX--->		<-----AGE----->				
	Total	Male	Female	Under 35	35-54	Over 54	Don't know/Refused
	-----	----	-----	-----	-----	-----	-----
Total	500	263	237	141	140	143	76
	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
(5) Completely agree	88	49	39	21	29	23	15
	17.6%	18.6%	16.5%	14.9%	20.7%	16.1%	19.7%
(4) Somewhat agree	92	43	49	26	27	27	12
	18.4%	16.4%	20.7%	18.4%	19.3%	18.9%	15.8%
(3) Neither agree nor disagree	86	48	38	23	26	24	13
	17.2%	18.3%	16.0%	16.3%	18.6%	16.8%	17.1%
(2) Somewhat disagree	73	36	37	13	23	21	16
	14.6%	13.7%	15.6%	9.2%	16.4%	14.7%	21.1%
(1) Completely disagree	86	43	43	29	17	26	14
	17.2%	16.4%	18.1%	20.6%	12.1%	18.2%	18.4%
Don't Know/Refused to answer	75	44	31	29	18	22	6
	15.0%	16.7%	13.1%	20.6%	12.9%	15.4%	7.9%



**Ranking Data Rows: EDIT= RANK\_IF\_INDICATED and STUB= [RANK\_LEVEL=]**

Here is an example stub definition indicating that the last four rows should be ranked low, i.e., below the other rows which are ranked at the default level of 1.

This table continues to a second page. By default, (continued) prints at the top of the continued page.

```
TABLE_SET= { qn8_z:
TITLE=:
    Q8. From your own experience and knowledge, what do you
especially like
    about Road Runner?}
STUB=:
    Good service/prompt service
    Dependable/continuous service
    The courteous employees they have/helpful
    I like the food/good food
    Food selection
    Good prices
    Computerized/accurate billing
    Helpful in explaining billing questions
    Good entertainment
    Variety of entertainment
    Nice family atmosphere
    Established place/has been around for awhile
    Accessible/Available/They're everywhere
    Like everything/good place
    [RANK_LEVEL=L] Other
    [RANK_LEVEL=L] No problem/No complaints
```

## INTERMEDIATE TABLES

### 5.3 Changing Table Specifications

```
[RANK_LEVEL=L] Don't know/No answer
[RANK_LEVEL=L] Nothing}
ROW=: [1/40.2^1//15/22//24]
}
```

Here is the EDIT statement for this example table. Note other EDIT= options used.

```
EDIT=: -COLUMN_NA, -ROW_NA, PERCENT_DECIMALS=0, COLUMN_WIDTH=7,
      STUB_WIDTH=40, RUNNING_LINES=1, RANK_IF_INDICATED }
```

TABLE 001

Q8. From your own experience and knowledge, what do you especially like about Road Runner?

	Total	Male	Female
Total	500	263	237
	100%	100%	100%
Like everything/good place	53	26	27
	11%	10%	11%
I like the food/good food	48	25	23
	10%	10%	10%
Computerized/accurate billing	47	21	26
	9%	8%	11%
Helpful in explaining billing questions	46	23	23
	9%	9%	10%
Established place/has been around for awhile	46	28	18
	9%	11%	8%
Variety of entertainment	43	25	18
	9%	10%	8%
The courteous employees they have/helpful	42	21	21
	8%	8%	9%

Food selection	40 8%	20 8%	20 8%
Good prices	40 8%	25 10%	15 6%
Nice family atmosphere	35 7%	21 8%	14 6%
Good entertainment	34 7%	17 6%	17 7%
Accessible/Available/They're everywhere	32 6%	11 4%	21 9%
Dependable/continuous service	31 6%	12 5%	19 8%
Good service/prompt service	28 6%	11 4%	17 7%
Other	44 9%	29 11%	15 6%
No problem/No complaints	21 4%	12 5%	9 4%
Don't know/No answer	19 4%	7 3%	12 5%

Q8. From your own experience and knowledge, what do you especially like about Road Runner?

	Total	Male	Female
Nothing	23 5%	15 6%	8 3%

**HORIZONTAL\_PERCENT= Changing the Percent Base**

User-Defined Total In The Banner

In this example the percent base has been changed from the default (system Total row/VERTICAL\_PERCENT=T) to the user-defined Total column. The default horizontal percent base is the system Total column. Since that is suppressed on the EDIT statement with -COLUMN\_TNA we have specified the user-defined Total column (column 1) as the percent base (HORIZONTAL\_PERCENT=1).

Edit statement for this table:

```
EDIT={ edit1: -COLUMN_TNA, -ROW_NA,
PERCENT_DECIMALS=0,HORIZONTAL_PERCENT=1,
        COLUMN_WIDTH=7, STUB_WIDTH=40, RUNNING_LINES=2,
PAGE_WIDTH=106}
```

TABLE 001

Q1. How much do you agree with the following statement:

The fast food at Road Runners is worth what I pay for it.

	<-----AGE----->			<-----INCOME----->		
	Under	Over	Under	\$15-	Over	
TOTAL	35	35-54	54	\$15k	\$35k	\$35k
----	-----	-----	-----	-----	-----	-----
Total	500	141	140	143	74	215
	100%	28%	28%	29%	15%	43%

(5) Completely agree	88	21	29	23	10	30	36
	100%	24%	33%	26%	11%	34%	41%
(4) Somewhat agree	92	26	27	27	14	30	35
	100%	28%	29%	29%	15%	33%	38%
(3) Neither agree nor disagree	86	23	26	24	13	22	45
	100%	27%	30%	28%	15%	26%	52%
(2) Somewhat disagree	73	13	23	21	13	21	33
	100%	18%	32%	29%	18%	29%	45%
(1) Completely disagree	86	29	17	26	8	27	37
	100%	34%	20%	30%	9%	31%	43%
Don't Know/Refused to answer	75	29	18	22	16	18	29
	100%	39%	24%	29%	21%	24%	39%

**NOTE:** The last two banner points were omitted from this table due to page size limitations.

### CHANGING PERCENT BASE WITHIN A STUB

In this example VERTICAL\_PERCENT is used as a STUB [option] to change the percent base for a row expression, respondent age by sex, where we want to show what percentage each age category is of each sex category rather than of the total sample. This table also illustrates several other options described above: EDIT=STUB\_PREFACE, -COLUMN\_TNA, PERCENT\_DECIMALS=, and STUB options: [UNDER\_LINE], [LONG\_COMMENT], [SUPPRESS], [SKIP\_LINES=], and [STUB\_INDENT=].

```
>PURGE SAME
~INPUT RRUNR
>DEFINE @study vper
```

## INTERMEDIATE TABLES

### 5.3 Changing Table Specifications

```
>PRINT_FILE @study~
~SPEC_FILE @study~

~DEFINE
STUB={ stubtop:           ''Defines the stub for the total and no answer rows.
[SUPPRESS] TOTAL         ''The stub option [SUPPRESS] prevents these rows from
[SUPPRESS] no answer}    ''printing (same as EDIT= -ROW_TNA).

TABLE_SET={ tab1:
SET AUTOMATIC_TABLES
EDIT=: -COLUMN_TNA, STUB_PREFACE=stubtop, PERCENT_DECIMALS=0 }
TITLE=: Respondent sex BY age with changing percent base}
STUB=:
[LONG_COMMENT] Percent base is the total row (suppressed)
[UNDER_LINE, VERTICAL_PERCENT=T] Male
[LONG_COMMENT] Percent base changed to first printed row: Male
[VERTICAL_PERCENT=1] | Under 25
| 25 to 34 ''| vertical bar acts as a placeholder on stub text
[STUB_INDENT=2] 35 to 44 ''[STUB_INDENT=2] does the same thing as |,
| 45 to 54           ''but also indents subsequent lines.
| 55 to 64
| 65 or over
[SKIP_LINES=3, LONG_COMMENT] Percent base changed back to total row
[UNDER_LINE, VERTICAL_PERCENT=T] Female
[LONG_COMMENT] Percent base changed to eighth printed row: Female
[VERTICAL_PERCENT=8] | Under 25
| 25 to 34
| 35 to 44
| 45 to 54
| 55 to 64
| 65 or over}
ROW=: [1/51^1-6/1//6] BY [1/57^1/2] ''1-6 nets age categories for all males
} ''and all females, then breaks out
''each category by male and female 1//6

~EXECUTE
COLUMN=TOTAL ''defines a total column and banner text1
MAKE_TABLES
~END
```

---

1. TOTAL is a System Constant. It says use the System TOTAL as the column variable and print "TOTAL" as the banner label. See "9.3.1 System Constants".

TABLE 001

Respondent sex BY age with changing percent base

TOTAL

Percent base is the total row (suppressed)

Male	229
----	46%

Percent base changed to first printed row: Male

Under 25	41
	18%
25 to 34	42
	18%
35 to 44	38
	17%
45 to 54	30
	13%
55 to 64	48
	21%
65 or over	30
	13%

Percent base changed back to total row

Female	195
-----	39%

Percent base changed to eighth printed row: Female

Under 25	24 12%
25 to 34	34 17%
35 to 44	33 17%
45 to 54	39 20%
55 to 64	32 16%
65 or over	33 17%

## 5.4 Printing Multiple Banners For Each Table Row

There may be occasions when you want to print the same row variable and stub label set against more than one column variable and banner label set, while maintaining the order of the table names. `SET COLUMN_REPEAT1` is the command that causes Mentor to print multiple banners consecutively for each row in a table run.

Syntax: `SET COLUMN_REPEAT=#`

`#` indicates the number of column variables each row variable will be cross-tabulated by. There is no practical maximum value for `#`.

When Mentor processes tables in `COLUMN_REPEAT` mode it assigns table names leaving room for the number of column variables specified. The program builds all tables defined for the first `COLUMN=` variable specified, then

---

1. Refer to `SET COLUMN_REPEAT_OVERRIDE` to override program defaults.



reprocesses the table definitions for the next column variable, and so on until the number of column variables specified on the COLUMN\_REPEAT has been satisfied. Tables are then printed in ascending table name order showing the same stub label set against however many banners were specified.

```
>PURGE _SAME
```

```
~INPUT rrunr, MAYBE_BACKUP, DOT=100
```

```
>DEFINE @STUDY colrep
```

```
>PRINT_FILE @STUDY~
```

```
~SPEC_FILE @STUDY~
```

```
~DEFINE
```

```
TABLE_SET={tabtop:
```

```
SET AUTOMATIC_TABLES, COLUMN_REPEAT=2, DROP_LOCAL_EDIT
```

```
HEADER=: =Road Runner Fast Food Sample Tables
```

```
Prepared on #date# }
```

```
FOOTER=: =Tables prepared by Computers for Marketing  
Corp.
```

```
Page #page_number# }
```

```
EDIT=: -COLUMN_TNA, -ROW_NA, PERCENT_DECIMALS=0,  
STATISTICS_DECIMALS=2,
```

```
VERTICAL_PERCENT=T, COLUMN_WIDTH=7,
```

```
STUB_WIDTH=40,
```

```
RUNNING_LINES=1, TCON }
```

## INTERMEDIATE TABLES

### *5.4 Printing Multiple Banners For Each Table Row*

}

```

TABLE_SET={banner1:
BANNER=:
|           <-----AGE----->  <-----INCOME----->  <---RATING--->
|
|           Under           Over  Under  $15-  Over           Neu-
| Total      35  35-54      54  $15k  $35k  $35k  Good  Poor
| -----  -----  -----  -----  -----  -----  -----  -----  ----- }

COLUMN=: TOTAL WITH &
          [1/51^1,2/3,4/5,6] WITH &  'AGE
          [1/52^1/2,3/4,5,6] WITH &  'INCOME
          [1/47^4,5/1,2,3]           'RATING
}

&@STUDY-^DEF  'reads in a file containing TABLE_SET definitions (title,
              'stub, row) for each table.

TABLE_SET={banner2:
LOCAL_EDIT=: -COLUMN_WIDTH}  'resets column width to default. Other options
                          'remain in effect.

BANNER=:
|           <-----STATUS----->  <---RATING--->
|
|           Living           Neu-
|           Di-   Wi-       to-   tral/
| Total Married vorced  dowed  Single  gether  Good  Poor
| -----  -----  -----  -----  -----  -----  -----  ----- }

COLUMN=: TOTAL WITH &
          [1/54^1//5] WITH &  'MARITAL STATUS, Refused category excluded
          [1/47^4,5/1,2,3]  'RATING
}

~EXECUTE MAKE_TABLES
~END

```

## 5.5 TABLE NAMES

Mentor automatically assigns table names starting with T001, though the T by default does not appear as part of the table name on the printed table. T001 represents the variable name for a particular table and is the name this table is stored under in a DB file for future retrieval in table manipulation (“9.2 TABLE MANIPULATION”) or to load and print a table stored in a DB file (see “5.6 Reprinting Tables”). Table names follow the same rules for other variable names.

## INTERMEDIATE TABLES

### 5.5 TABLE NAMES

They can be from 1 to 14 alphanumeric characters in length, must begin with an alpha character, and can include . (period), and \_ (underscore).

There are several options available to you regarding table names:

- Print the leading alpha character as part of the table name.
- Specify the starting table name for automatic table naming.
- Append up to a 16 character prefix or suffix to each table name.
- Change the default text 'TABLE ' preceding each table name.
- Specify a unique name for each table.
- Print a table name different from the name stored in the DB file (see “5.6 Reprinting Tables”).

### PRINTING LEADING ALPHA CHARACTER

EDIT= PRINT\_ALPHA\_TABLE\_NAMES prints the leading alpha character on every table name.

```
EDIT={edit1: PRINT_ALPHA_TABLE_NAMES }
```

TABLE T001

Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for it.

	Total	Male	Female
Total	500 100.0%	263 100.0%	237 100.0%
(5) Completely agree	88 17.6%	49 18.6%	39 16.5%
(4) Somewhat agree	92	43	49

	18.4%	16.4%	20.7%
(3) Neither agree nor disagree	86 17.2%	48 18.3%	38 16.0%
(2) Somewhat disagree	73 14.6%	36 13.7%	37 15.6%
(1) Completely disagree	86 17.2%	43 16.4%	43 18.1%
Don't Know/Refused to answer	75 15.0%	44 16.7%	31 13.1%

### SPECIFY STARTING NAME

The `~SET TABLE_NAME=name` option allows you to specify what the initial table name will be for automatic table naming by Mentor (i.e, `SET AUTOMATIC_TABLES, ~EXECUTE STORE=* or TABLE=*`). The program will generate table names from this initial name either by incrementing the last number (T001, T002, T003,...), or by alpha-kicking the last letter (T10A, T10B, T10C,...). The leading alpha character of the table name will not print on the tables unless otherwise specified on the EDIT statement as in the example above. This SET command can be used inside the `~DEFINE TABLE_SET=` structure.

```
~SET TABLE_NAME=T0001
```

**NOTE:** For this and subsequent tables, we are printing the titling only, not the entire table.

```
TABLE 0001
```

```
Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for  
it.
```

**PRINTING NAME WITH PREFIX OR SUFFIX**

The SUFFIX= and PREFIX= EDIT options allow you to prepend (prefix) or append (suffix) up to 16 characters of text, including blanks and special characters, to the table name (either default or user-specified). Double quotes are required when the text includes blanks or special characters, or when other EDIT options are specified on the same line. PREFIX=; or SUFFIX=; will reset the option to null.

```
EDIT={edit1: SUFFIX="_Road_Runner" }
```

```
TABLE 0001_Road_Runner
```

Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for  
it.

**REPLACING "TABLE"**

The default text 'TABLE ' precedes every table name printed before the title for each table. Use the CALL\_TABLE option either to suppress this text entirely or to specify your own. Maximum length is 16 characters including blanks and special characters. CALL\_TABLE=; will reset this option to the default 'TABLE' . EDIT= CALL\_TABLE=" " will suppress the printing of 'TABLE ' before the table name.

```
EDIT={edit1: CALL_TABLE="REPORT " }
```

This will replace TABLE with REPORT before each table name. Include at least one blank space after the text; if not, the table name will print immediately after this text.

```
REPORT 0001_Road_Runner
```

Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for  
it.

## SPECIFYING UNIQUE TABLE NAMES

In “4.11 *SAMPLE SPECIFICATION FILES*”, table production was triggered automatically by specifying `~SET AUTOMATIC_TABLES`. In this mode it is the `ROW=` keyword that causes Mentor to make a table from the elements defined and to assign it the next available table name (either from the default table name starting at T001 or from the user-specified beginning table name on the `~SET TABLE_NAME=` command).

The `~EXECUTE` keyword `STORE_TABLES=` also causes Mentor to process table specifications. Table names are generated from the name specified on the `STORE_TABLES=` command (e.g., `STORE_TABLES=T001`) or `STORE_TABLES=*` to assign the next available name (either default or user-specified on the `~SET TABLE_NAME` command).

In “4.3 *DEFINING TABLE ELEMENTS*”, we discussed the general form of the `TABLE_SET` structure to define tables. Any table building `~EXECUTE` command can be specified in this structure. `STORE_TABLES=` specified in the `TABLE_SET` structure allows you to do two things:

- specify a unique name for each table, overriding the default pattern of automatic table name generation by the program (see `STORE_TABLES=*`).
- store the table name as part of the `TABLE_SET` definition in a DB file, associating that set of table elements with this table name.

```
~DEFINE
TABLE_SET={ qn1_z:
TITLE=:
Q1. How much do you agree with the following statement:
The food at Road Runners is worth what I pay for it.}
STUB=:
    (5) Completely agree
    (4) Somewhat agree
    (3) Neither agree nor disagree
    (2) Somewhat disagree
    (1) Completely disagree
```

## INTERMEDIATE TABLES

### 5.5 TABLE NAMES

```
        Don't Know/Refused to answer }
ROW=: [1/6^5//1/10]
STORE_TABLES=RRUNR_QN1
}
```

Since STORE\_TABLES= causes Mentor to process table specifications defined to that point, specify it last in the TABLE\_SET definition. Table elements defined after STORE\_TABLES= would not be included in that table.

If you make tables using SET AUTOMATIC\_TABLES remember that the ROW= keyword causes Mentor to make tables in this mode. Referring to the example above, this TABLE\_SET would generate two tables: T001 when ROW= is processed (assuming that this is the first table made) and RRUNR\_QN1 with the same elements when the STORE\_TABLES= command is processed. SET - AUTOMATIC\_TABLES would produce one table.

```
TABLE_SET= { qn1_z:
TITLE=:
Q1. How much do you agree with the following statement:
The fast food at Road Runners is worth what I pay for
it.}
STUB=:
        (5) Completely agree
        (4) Somewhat agree
        (3) Neither agree nor disagree
        (2) Somewhat disagree
        (1) Completely disagree
        Don't Know/Refused to answer }
ROW=: [1/6.1^5/4/3/2/1/10]
BASE=: [1/57^2]
TITLE_4: Base is women respondents only
STORE_TABLES= T001_a
}
```

In this example SET AUTOMATIC\_TABLES mode would cause Mentor to make a table with the title, stub, and row defined above when the ROW= variable is processed (with the next available table name). STORE\_TABLES=T001\_a would



then cause Mentor to make a based version of the same table with the name T001\_a.

## PRINTING DIFFERENT TABLE NAMES

You can assign table names to be printed that are different from the actual table names stored in the DB file with the option PRINT\_TABLE\_NAME. The same rules apply as with regular table names: the name must begin with an alpha character; it must be between 1 and 14 characters long; and, if the first alpha character is followed by a number, the alpha character will be stripped.

Like regular table names, Mentor will automatically increment numbers or alpha characters on sequential tables. You can use either ~SET or ~DEFINE EDIT PRINT\_TABLE\_NAME.

**Example:** ~SET PRINT\_TABLE\_NAME=t 1

This will print table names Table1, Table 2, etc. See *Mentor, Appendix B*, ~DEFINE EDIT PRINT\_TABLE\_NAME for more details and examples.

## 5.6 Reprinting Tables

If you have stored your finished tables in a DB file then they are available for reprinting (see “4.11 SAMPLE SPECIFICATION FILES”). This is especially useful if you have a very large sample and need to make changes to the finished tables that do not require reprocessing the data file. Such changes might be corrections to any table text element, i.e., header, footer, banner text, stub labels, etc., different print options such as percentaging or number of decimal places, or adding printer statistics rows (meaning statistics computed on the existing numbers stored for a table's cells).

You will need the following files to reprint existing tables: the DB file that the tables are stored in and the LPR file generated automatically by Mentor when ~SPEC\_FILES is specified in a table run.

## INTERMEDIATE TABLES

### 5.6 Reprinting Tables

Each table is stored as an entry in the DB file under the table name, e.g., T001 would be the DB entry name for table 001. T001 stores the matrix (the numbers that make up this table) and the variable names for each of its table elements (banner and stub labels, titling, and EDIT statement). Each table element is stored as a separate entry in the DB file. Mentor reprints a table by searching through the open DB files for the actual entry that stores each of the table element variables. Up to five DB files can be open at any one time so it is not necessary to have all the parts of the table stored in the same DB file, i.e., the table definitions could be in one DB file and the actual tables in another. Mentor prints an error message if a table element cannot be located in the open DB files.

The LPR file contains a table printing statement for each table made in the original run. Edit this file to reprint only the tables wanted by deleting lines.

**Example:**   LOAD=T001 PRINT

This statement loads (into system memory) all of the variables needed to print the table named, and then prints the table.

LOAD\_TABLES= (abbreviated to LOAD=) is the ~EXECUTE keyword that loads the table specified and its elements from the open DB file into system memory. T001 is the name of the table to be loaded.

PRINT\_TABLES (abbreviated to PRINT) is the ~EXECUTE keyword that prints the table currently held in memory.

Here are sample specifications that reprint tables from the Road Runner example set changing the header and adding horizontal percentaging. Note that the data file is not reopened (~INPUT name); it is not needed to reprint existing tables.

```
>PURGE_SAME
>USE_DB rrunr1
>PRINT_FILE retabs
```

```

~DEFINE
TABLE_SET={ tabtop2=tabtop:
LOCAL_EDIT=: HORIZONTAL_PERCENT}
HEADER=:=THIS IS THE NEW TABLE HEADING }
}

~EXECUTE
TABLE_SET=tabtop2
&rrunr^LPR
~END

```

Here is an explanation of the commands from the previous spec file:

## ACCESSING THE DB FILE

>USE\_DB rrunr1                      Opens the DB file where the tables were stored from the original run (see “4.11 SAMPLE SPECIFICATION FILES”). Up to five DB files can be opened at one time (using USE\_DB and CREATE\_DB commands). This DB file is opened with Read\_only access (the default), so changes in this run will not be saved anywhere.

## DEFINING A NEW EDIT STATEMENT AND TABLE HEADER

```

~DEFINE
TABLE_SET={ tabtop2=tabtop:
LOCAL_EDIT=: HORIZONTAL_PERCENT}

HEADER=:=THIS IS THE NEW TABLE HEADING }
}

```

tabtop2=tabtop tells Mentor that TABTOP2 is the same as TABTOP (see “4.11 SAMPLE SPECIFICATION FILES”) except for the elements replaced. (see “4.4 TABLE

## INTERMEDIATE TABLES

### 5.6 Reprinting Tables

*BUILDING (The INPUT and EXECUTE statements)” for more on TABLE\_SET).*

LOCAL\_EDIT= is the ~EXECUTE keyword that appends to a previous edit statement, i.e., the options specified on the edit statement defined for TABTOP remain in effect.

HORIZONTAL\_PERCENT will add horizontal percentaging (based by default on the Total column) to the reprinted tables.

HEADER= replaces the HEADER= variable defined in TABTOP with the text specified here.

## REPRINTING THE TABLES

~EXECUTE

TABLE\_SET=tabtop\_e is the keyword to execute the new TABLE\_SET definition.

&rrunr^LPR reads in the LPR file (generated when TABS.SPX was run, see “4.11 SAMPLE SPECIFICATION FILES”), edited for the tables we want to reprint.

```
THIS IS THE NEW TABLE HEADING
TABLE 001
```

```
Q1. How much do you agree with the following statement:
The fast food at Road Runners is worth what I pay for it.
```

```

                <-----AGE----->  <-----INCOME----->
                Under           Over  Under  $15-  Over
TOTAL          35  35-54      54  $15k  $35k  $35k
-----  -----  -----  -----  -----  -----
```

Total	500	141	140	143	74	148	215
	100%	28%	28%	29%	15%	30%	43%
	100%	100%	100%	100%	100%	100%	100%
(5) Completely agree	88	21	29	23	10	30	36
	100%	24%	33%	26%	11%	34%	41%
	18%	15%	21%	16%	14%	20%	17%
(4) Somewhat agree	92	26	27	27	14	30	35
	100%	28%	29%	29%	15%	33%	38%
	18%	18%	19%	19%	19%	20%	16%
(3) Neither agree nor disagree	86	23	26	24	13	22	45
	100%	27%	30%	28%	15%	26%	52%
	17%	16%	19%	17%	18%	15%	21%
(2) Somewhat disagree	73	13	23	21	13	21	33
	100%	18%	32%	29%	18%	29%	45%
	15%	9%	16%	15%	18%	14%	15%
(1) Completely disagree	86	29	17	26	8	27	37
	100%	34%	20%	30%	9%	31%	43%
	17%	21%	12%	18%	11%	18%	17%
Don't Know/Refused to answer	75	29	18	22	16	18	29
	100%	39%	24%	29%	21%	24%	39%
	15%	21%	13%	15%	22%	12%	13%

Tables prepared by Computers for Marketing Corp.

Page 1

**NOTE:** The last two banner points were omitted from this table due to page size limitations.

### ADDING STATISTICS ROWS TO FINISHED TABLES

In “5.2 Axis Commands/Cross-Case Operations”, you saw how to compute statistical calculations on the data by defining axis commands, e.g., \$[MEAN], as part of the row variable. The statistics are calculated from the data as the table is built. Printer statistics rows can be computed when the table is printed, but they will be calculated off of the existing frequencies in the cells for that table and not the data itself.

## INTERMEDIATE TABLES

### 5.6 Reprinting Tables

```
>PURGE_SAME
>USE_DB rrunr1
>PRINT_FILE retabs2
~DEFINE
    TABLE_SET={ tabtop_s=tabtop:
    LOCAL_EDIT=: COLUMN_MEAN,
    COLUMN_STATISTICS_VALUES=VALUES(5,4,3,2,1,,),
                COLUMN_STD, COLUMN_SE }
}

~EXECUTE
    TABLE_SET= tabtop_s
&rrunr^LPR
~END
```

#### **Explanation:**

```
~DEFINE
TABLE_SET={ tabtop_s=tabtop:
LOCAL_EDIT=: COLUMN_MEAN, COLUMN_STATISTICS_VALUES=VAL-
                UES(5,4,3,2,1,,),
                COLUMN_STD, COLUMN_SE }
}
```

As explained in the previous example, only the elements defined in the new TABLE\_SET, TABTOP\_S, will replace or append to the elements already defined for the old TABLE\_SET, TABTOP.

This LOCAL\_EDIT defines additional EDIT options that will calculate statistics using the existing numbers in the tables. Corresponding row labels are generated automatically.

COLUMN\_MEAN calculates a printer mean and must be specified before COLUMN\_STD.

COLUMN\_STATISTICS\_VALUES=VALUES(5,4,3,2,1,,) assigns numeric values corresponding to the data categories for the row variable. These values are used to compute the mean. Comma delimiters indicate categories that will be excluded from any calculation. This has the same affect as the variable modifier \*RANGES=1-5, used to exclude categories from statistical calculations in a data variable definition (see the example in “5.2 Axis Commands/Cross-Case Operations”). In this example the last two commas exclude the Don't Know/Refused to answer category in the row variable (ROW=: [1/6^5//1/10]).

COLUMN\_STD COLUMN\_SE calculates a printer standard deviation and standard error using the values specified for COLUMN\_STATISTICS\_VALUES.

INTERMEDIATE TABLES

5.6 Reprinting Tables

Road Runner Fast Food Sample Tables

Prepared on 12 AUG 1994

TABLE 001

Q1. How much do you agree with the following statement:  
The fast food at Road Runners is worth what I pay for it.

	<-----AGE----->			<-----INCOME----->			
	Total	Under 35	35-54	Over 54	Under \$15k	\$15- \$35k	Over \$35k
Total	500	141	140	143	74	148	215
	100%	28%	28%	29%	15%	30%	43%
(5) Completely agree	88	21	29	23	10	30	36
	100%	24%	33%	26%	11%	34%	41%
(4) Somewhat agree	92	26	27	27	14	30	35
	100%	28%	29%	29%	15%	33%	38%
(3) Neither agree nor disagree	86	23	26	24	13	22	45
	100%	27%	30%	28%	15%	26%	52%
(2) Somewhat disagree	73	13	23	21	13	21	33
	100%	18%	32%	29%	18%	29%	45%
(1) Completely disagree	86	29	17	26	8	27	37
	100%	34%	20%	30%	9%	31%	43%
Don't Know/Refused to answer	75	29	18	22	16	18	29
	100%	39%	24%	29%	21%	24%	39%
Mean	3.05	2.97	3.23	3.00	3.09	3.12	3.00
Standard deviation	1.42	1.47	1.37	1.43	1.32	1.47	1.40
Standard error	0.07	0.14	0.12	0.13	0.17	0.13	0.10

Tables prepared by Computers for Marketing Corp.

Page 1

**NOTE:** The last two banner points were omitted from this table due to page size limitations.











# ADVANCED TABLES

## INTRODUCTION

**T**his chapter explains how to produce tables other than straightforward simple cross-tabulations. When producing more complex tables, the biggest problem is usually understanding how the table is constructed rather than using the proper syntax to construct the table. This chapter is designed to help you understand both.

This chapter is broken into major sections by the type of table you are trying to produce. Each section has a set of sequentially numbered tables starting from n01, where n is the sub-section number. The beginning of each section contains a description of when it is usually appropriate to use such a table construction. Each section is then broken down further into a particular type of table design. In each sub-section there is a description of the different ways to produce that table, and an example of how to produce it.

This chapter assumes a basic understanding of simple table building and of CfMC terminology. There are many references to things such as the EDIT statement and joiners like WITH; if you do not know what they are you may have difficulty following along. A review of Chapters 1 through 5 may be necessary.

### 6.1 TOP BOX/BOTTOM BOX SUMMARY TABLES

A Top Box table is one in which the highest rating for one of a series of brands or attributes is compared against the highest ratings of the others. The Top Box usually contains those respondents who have given the highest rating to that particular brand, but sometimes also contains those who have given the second and/or third highest rating as well. These tables may have a percentage base of either the total sample, or of the number of respondents that were asked about each brand if this might be different for each. To percentage each brand off the number responding to that brand you will need to define the percentage base along with the Top Box for each brand. In addition you will need to make sure that you use the

stub options [VERTICAL\_PERCENT=\*,SUPPRESS] on each base row so that it does not print (SUPPRESS) and the next row is percentaged off of it (VERTICAL\_PERCENT=\*).

You may also need to produce a Bottom Box table which is similar in design to the Top Box except it contains those respondents who have given the lowest rating to each brand.

### 6.1.1 Top Box Tables with Constant Percentage Base

We'll start with a Top Box table with a constant percentage base such as total or some sub-base which is applied to the entire table. For purposes of the example below, the respondent's gender is stored in column 5 while columns 7 through 11 hold the rating for five different brands (A-E), where the rating scale is from 1 to 4, 4 being the highest possible rating.

There are usually several ways to write any variable definition. In the examples below the shortest and most computer efficient usage is included in the tabset, while other perfectly reasonable definitions are also noted after the tabset. Note that any of these definitions (and many others also) will produce the same table and although we recommend the one inside the tabset, you can use any that is easy for you to understand.

**NOTE:** The following set of commands define a standard front end for the all the examples in this section, except where noted.

```
>PRINT_FILE TOPBX
~INPUT TOPBX
>CREATEDB TOPBX,DUPLICATE=WARN
~SET AUTOMATIC_TABLES,DROP_LOCAL_EDIT,
DROP_BASE,BEGIN_TABLE_NAME=T101

~DEFINE
```



```

TABLE_SET= {BAN1:
EDIT=: COLUMN_WIDTH=8, STUB_WIDTH=20, -COLUMN_TNA }
STUB_PREFACE=:
TOTAL
[SUPPRESS] NO ANSWER }
BANNER=:
|
|                GENDER
|                <----->
|  TOTAL      MALE  FEMALE
|  ----      ---  -}
COLUMN=: TOTAL WITH [5#1/2]
}

~EXECUTE
TABLE_SET= BAN1

```

These commands are specific to this example:

```

~DEFINE

TABLE_SET= {TAB101:
HEADER=: TOP BOX TABLE EXAMPLE}
TITLE=: TOP BOX SUMMARY TABLE FOR OVERALL RATING}
STUB=:
    BRAND A-TOP BOX
    BRAND B-TOP BOX
    BRAND C-TOP BOX
    BRAND D-TOP BOX
    BRAND E-TOP BOX }
ROW=: [07, ..., 11#4]
}

```

## ADVANCED TABLES

### 6.1 TOP BOX/BOTTOM BOX SUMMARY TABLES

Here are some alternate ways to write the row variable:

```
ROW101A:      [7#4] WITH [8#4] WITH [9#4] WITH [10^4]  
WITH [11#4]
```

```
ROW101B: &  
>REPEAT $A=07, . . . , 11; STRIP="WITH &"  
[$A#4] WITH &  
>END_REPEAT
```

```
~EXECUTE  
TABLE_SET= TAB101
```





Here is the table Mentor prints:

TOP BOX TABLE EXAMPLE

TABLE 101

TOP BOX SUMMARY TABLE FOR OVERALL RATING

	GENDER		
	TOTAL	MALE	FEMALE
	-----	-----	-----
TOTAL	100	52	48
	100.0%	100.0%	100.0%
BRAND A-TOP BOX	21	9	12
	21.0%	17.3%	25.0%
BRAND B-TOP BOX	27	14	13
	27.0%	26.9%	27.1%
BRAND C-TOP BOX	30	15	15
	30.0%	28.8%	31.3%
BRAND D-TOP BOX	19	11	8
	19.0%	21.2%	16.7%
BRAND E-TOP BOX	25	13	12
	25.0%	25.0%	25.0%

## 6.1.2 Top Box Tables with a Changing Percentage Base

When producing a Top Box table you may want to percentage each row off a different base. Suppose each rating scale was only asked of those who have used that brand; this base value will probably be different for each brand. Further suppose that the original rating scale tables were percentaged back to this base and for purposes of this summary table you want to maintain those same percentages. To do this, you will need to not only define each Top Box, but also its percentage base.

The syntax to produce this table is very similar to producing the Top Box table with constant percentage (See “6.1.1 Top Box Tables with Constant Percentage Base”), so make sure you understand that before proceeding. One very important difference is that you will need to create a new label set which controls the printing and percentage base in the table. Each base row label will want to have the [SUPPRESS] option to suppress its printing and the [VERTICAL\_PERCENT=\*] option to cause the next row to be percentaged off of it.

The following example assumes the same scenario as the previous example. The only valid response to one of the rating scale questions is 1,2,3,4 or 9. If you compare the frequencies in Table 102 with Table 101 you will see that they are the same, but the percentages are generally much higher in Table 102.

```
~DEFINE
```

```
TABLE_SET= {TAB102:
```

```
HEADER=: TOP BOX TABLE WITH DIFFERENT PERCENTAGE BASE  
ON EACH ROW}
```

```
TITLE=: TOP BOX SUMMARY TABLE FOR OVERALL RATING}
```

```
TITLE_4=: BASE= RESPONDENTS WHO USED THE BRAND}
```

```
STUB=:
```

```
[VERTICAL_PERCENT=*,SUPPRESS] BRAND A BASE-WON'T PRINT  
BRAND A TOP BOX
```



```
[VERTICAL_PERCENT=*,SUPPRESS] BRAND B BASE-WON'T PRINT  
BRAND B TOP BOX  
[VERTICAL_PERCENT=*,SUPPRESS] BRAND C BASE-WON'T PRINT  
BRAND C TOP BOX  
[VERTICAL_PERCENT=*,SUPPRESS] BRAND D BASE-WON'T PRINT  
BRAND D TOP BOX  
[VERTICAL_PERCENT=*,SUPPRESS] BRAND E BASE-WON'T PRINT  
BRAND E TOP BOX }  
ROW=: [07, . . . , 11#1-4, X/4]  
}
```

Here is an alternate way to write the row variable:

```
ROW102A: [7#1-4, X/4] WITH [8#1-4, X/4] WITH [9#1-4, X/4]  
WITH &  
[10#1-4, X/4] WITH [11#1-4, X/4]  
  
~EXECUTE  
TABLE_SET= TAB102
```

**ADVANCED TABLES**

*6.1 TOP BOX/BOTTOM BOX SUMMARY TABLES*

Here is the table that Mentor prints:

TOP BOX TABLE WITH DIFFERENT PERCENTAGE BASE ON EACH ROW

TABLE 102

TOP BOX SUMMARY TABLE FOR OVERALL RATING

BASE= RESPONDENTS WHO USED THE BRAND

	GENDER		
	<----->		
	TOTAL	MALE	FEMALE
	-----	----	-----
TOTAL	100 100.0%	52 100.0%	48 100.0%
BRAND A TOP BOX	21 22.3%	9 18.0%	12 27.3%
BRAND B TOP BOX	27 40.3%	14 35.9%	13 46.4%
BRAND C TOP BOX	30 34.5%	15 34.9%	15 34.1%
BRAND D TOP BOX	19 43.2%	11 42.3%	8 44.4%
BRAND E TOP BOX	25 28.4%	13 29.5%	12 27.3%

**NOTE:** When the percentage base is changing, you may want to suppress the Total row, since it is not the percentage base for the rows under it and this

might be confusing for anyone reading the table. To suppress the Total row you can use a different STUB\_PREFACE that looks like the one below.

```
STUB_PREFACE=  
[SUPPRESS] TOTAL  
[SUPPRESS] NO ANSWER }
```

If you wish to do significance testing on a top box table with a changing percentage base, you need to create a \$[base] row for every top box in the table. *See section 8.2.4* for an example of how to do this.

### 6.1.3 Ranking of Top Box Tables

Top box tables which are percentaged off a constant base, can be ranked by just using the RANK option on a LOCAL\_EDIT statement. It is much more difficult to rank a Top Box with a changing percentage base. Since you most likely want to rank this table off of the percentages, that percentage row must be created in the table.

As with the Top Box table with a changing percentage base (See “6.1.2 Top Box Tables with a Changing Percentage Base”), extra rows and stubs must be created in order to produce the desired result. In this instance, three rows will be created for each row printed in the table. The percentage needs to be created first, followed by the percentage base, followed by the frequency.

To create the percentage you must realize that a percentage is the same as a mean with a value of 100 assigned to those who are in the topbox and a value of 0 assigned to those who are in the base, but not in the topbox. This first row is set to rank level 1 while the 2 subsequent rows are set to rank level 2 so that however the percentage row is ranked the 2 other rows will remain below it. See “6.7.2 Ranking With Nets And Sub-Nets” for more information about setting rank levels.

## ADVANCED TABLES

### 6.1 TOP BOX/BOTTOM BOX SUMMARY TABLES

```
~DEFINE

TABLE_SET= {TAB103:
LOCAL_EDIT=: RANK_LEVEL=1, STUB_RANK_INDENT=0, RANK_COLUMN_BASE=1
}
HEAD=: RANKED TOP BOX TABLE WITH DIFFERENT PERCENTAGE BASE ON EACH
ROW}
TITLE=: TOP BOX SUMMARY TABLE FOR OVERALL RATING}
TITLE_4=: BASE= RESPONDENTS WHO USED THE BRAND}
STUB=:
[SUPPRESS,RANK=1] BRAND A TOP BOX PERCENTAGE (WON'T PRINT)
[VERTICAL_PERCENT=*,SUPPRESS,RANK=2] BRAND A BASE (WON'T PRINT)
[RANK=2] BRAND A TOP BOX
[SUPPRESS,RANK=1] BRAND B TOP BOX PERCENTAGE (WON'T PRINT)
[VERTICAL_PERCENT=*,SUPPRESS,RANK=2] BRAND B BASE (WON'T PRINT)
[RANK=2] BRAND B TOP BOX
[SUPPRESS,RANK=1] BRAND C TOP BOX PERCENTAGE (WON'T PRINT)
[VERTICAL_PERCENT=*,SUPPRESS,RANK=2] BRAND C BASE (WON'T PRINT)
[RANK=2] BRAND C TOP BOX
[SUPPRESS,RANK=1] BRAND D TOP BOX PERCENTAGE (WON'T PRINT)
[VERTICAL_PERCENT=*,SUPPRESS,RANK=2] BRAND D BASE (WON'T PRINT)
[RANK=2] BRAND D TOP BOX
[SUPPRESS,RANK=1] BRAND E TOP BOX PERCENTAGE (WON'T PRINT)
[VERTICAL_PERCENT=*,SUPPRESS,RANK=2] BRAND E BASE (WON'T PRINT)
[RANK=2] BRAND E TOP BOX
}
ROW=: &
>REPEAT $A=07,...,11; STRIP="&"
$[MEAN] SELECT_VALUE([$A#4/1-3,9],VALUES(100,0)) $[] [$A#1-4,X/4] &
>END_REPEAT
}

~EXECUTE
TABLE_SET= TAB103
```



Here is the table that Mentor prints:

RANKED TOP BOX TABLE WITH DIFFERENT PERCENTAGE BASE ON EACH ROW

TABLE 103

TOP BOX SUMMARY TABLE FOR OVERALL RATING

BASE= RESPONDENTS WHO USED THE BRAND

	GENDER		
	TOTAL	MALE	FEMALE
	-----	----	-----
TOTAL	100 100.0%	52 100.0%	48 100.0%
BRAND D TOP BOX	19 43.2%	11 42.3%	8 44.4%
BRAND B TOP BOX	27 40.3%	14 35.9%	13 46.4%
BRAND C TOP BOX	30 34.5%	15 34.9%	15 34.1%
BRAND E TOP BOX	25 28.4%	13 29.5%	12 27.3%
BRAND A TOP BOX	21 22.3%	9 18.0%	12 27.3%

Notice in the above table, that BRAND D prints highest on the table because it has the highest percentage, even though it has the lowest frequency. The only difference between Table 103 and Table 102 is the order in which the stubs are printed.

## 6.2 SUMMARY STATISTICS (MEANS)

This section shows how to produce summary statistics such as the mean, the standard deviation, the standard error, and the median on a table. It shows how you can efficiently and effectively produce these statistics no matter how the data was originally coded. In this section whenever the creation of a mean is discussed, you can replace the mean with any similar summary statistic (except where noted). For a complete list of all the summary statistics that can be produced see *Appendix B: TILDE COMMANDS, ~DEFINE AXIS=*.

If you wish to do significance testing on a top box table with a changing percentage base, you need to create a \$[base] row for every top box in the table. See *section 8.2.4* for an example of how to do this.

Means are usually produced on one of the following types of questions:

- Rating Scales
- A scale such as age where a number has been coded into a range and you want to use the midpoint of the range in order to calculate the mean.
- An actual value is stored in the data.

There are two different ways to produce means. You can either produce the mean directly on the variable definition or you can use the EDIT statement options. The variable definition method will allow you to produce an appropriate mean in all circumstances while the EDIT statement options are only appropriate on means of type 1 or 2 above if you are doing neither weighting nor dependent significance testing. See “6.2.5 Means And Medians Using The EDIT Options” for an explanation on how to produce summary statistics using the EDIT options. That section also discusses the differences in the two methods and the pros and cons of each. See “5.3 Changing Table Specifications” for a description of the syntax for creating a summary statistic as part of the variable definition.

It is very important to make sure that you create your own summary columns (total, no answer, etc.) when you are producing summary statistics. The values that



will be in the system-generated columns will reflect the number of respondents who were used in that statistic and not the true value of the statistic.

The number of decimal places that the statistic will print may be changed by using the STATISTICS\_DECIMALS=# option on the EDIT, STUB, or COLUMN\_INFO statements. The default decimal significance is 1 and valid settings are 0-7.

**NOTE:** The program prints a question mark (?) when the statistic is missing; i.e., it cannot be calculated because there is no respondent in that cell who has a valid number in the calculation. See Table 263 in “6.2.8 Summary Statistics with Arithmetic” for an example.

### 6.2.1 Means on Rating Scales Using the Variable Definition

Typically, creating a mean on a rating scale will follow one of these scenarios:

- No recode needed (values for mean = punch and DK coded as X or Y punch)
- Exclude values (values for mean = punch and DK coded as a number)
- Reverse the scale (values are in inverted order and DK coded as X or Y punch)
- Reverse the scale and exclude values (values are inverted and DK coded as a number)
- Scale is 1 to 10 with 10 coded as a 0, or scale is 0 to 10 with X or Y coded as 10.

#### WITH NO RECODING NEEDED

If the value of the rating has been coded with the corresponding punch and the Don't Know has been coded as a non-numeric code (X or Y punch), then there is no need to do any recoding on the mean. When no recoding is required you only need to define the row variable and specify the data location after the keyword \$[MEAN].

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

**NOTE:** In the stub you will need to include a label for the mean and you will want to make sure you mark it as a statistics row, so that it will be printed with the appropriate number of decimals and without any percentage.

In the example below, the gender question is stored in column 5 while column 7 is a 4 point rating scale with 4 equal to Excellent and Don't Know coded as an X punch. `STATISTICS_DECIMALS=2` is specified on the EDIT statement so that the statistics will print with two decimal places of significance.

**NOTE:** The following set of commands define a standard front end for the next set of examples.

```
>PRINT_FILE MEAN
~INPUT MEAN
~SET AUTOMATIC_TABLES,DROP_LOCAL_EDIT,DROP_BASE,
BEGIN_TABLE_NAME=T201

~DEFINE
TABLE_SET= {BAN1:
EDIT=:
COLUMN_WIDTH=8,STUB_WIDTH=20,-COLUMN_TNA,STATISTICS_DEC
IMALS=2 }
STUB_PREFACE=:
TOTAL
[SUPPRESS] NO ANSWER }
BANNER=:
|           GENDER
|           <----->
|  TOTAL  MALE  FEMALE
|  ----   ---   -----}
COLUMN=: TOTAL WITH [5^1/2]
}

~EXECUTE
TABLE_SET= BAN1
```

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

These commands are exclusive to this example.

```
~DEFINE
TABLE_SET= {TAB201:
HEADER=: MEAN AND OTHER SUMMARY STATISTICS ON A RATING
SCALE WITH NO RECODING}
TITLE=: OVERALL RATING OF PRODUCT A}
STUB=:
    EXCELLENT    (4)
    GOOD          (3)
    FAIR          (2)
    POOR          (1)
    DON'T KNOW
    [STATISTICS_ROW] MEAN
    [STATISTICS_ROW] STD DEV
    [STATISTICS_ROW] STD ERROR
    [STATISTICS_ROW, STATISTICS_DECIMALS=0] MEDIAN }
ROW=: [7^4//1/X] $[MEAN STD SE MEDIAN] [7]
}

~EXECUTE
TABLE_SET= TAB201
```



Here is the table Mentor prints:

MEAN AND OTHER SUMMARY STATISTICS ON A RATING SCALE WITH  
NO RECODING

TABLE 201

OVERALL RATING OF PRODUCT A

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
TOTAL		100	58	42
		100.0%	100.0%	100.0%
EXCELLENT	(4)	15	11	4
		15.0%	19.0%	9.5%
GOOD	(3)	15	10	5
		15.0%	17.2%	11.9%
FAIR	(2)	23	12	11
		23.0%	20.7%	26.2%
POOR	(1)	24	15	9
		24.0%	25.9%	21.4%
DON'T KNOW		23	10	13
		23.0%	17.2%	31.0%

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

MEAN	2.27	2.35	2.14
STD DEV	1.11	1.16	1.03
STD ERROR	0.13	0.17	0.19
MEDIAN	2	2	2

### WITH A NUMERIC DON'T KNOW EXCLUDED

If the value of the rating has been coded with the corresponding punch but the Don't Know has been coded as a numeric code (like 5), then if you do not put a qualifier on the mean, the program will use that value (5) in calculating the mean rather than excluding respondents who said Don't Know. In order to ensure proper calculation of the mean you need to use either the \*RANGES= modifier or the WHEN joiner to only include appropriate items in the calculation. The \*RANGES= modifier allows you to specify which numbers will be included in the statistic calculation. You can use either a dash to signify a range or a comma to separate individual items. See “5.1.2 Vector Joiners” for an explanation of the WHEN joiner.

The assumptions in the example below are the same as the previous example, except that the rating scale is now stored in column 8 and the Don't Know was coded as a 5 punch.

```
~DEFINE
```

```
TABLE_SET= {TAB202:  
HEADER=: MEAN EXCLUDING A DON'T KNOW CODED AS A NUMBER}  
TITLE=: OVERALL RATING OF PRODUCT B}
```



```
STUB= :  
    EXCELLENT   (4)  
    GOOD        (3)  
    FAIR        (2)  
    POOR        (1)  
    DON'T KNOW  
    [STATISTICS_ROW] MEAN }  
ROW= : [8^4//1/5] $ [MEAN] [8*RANGES=1-4]  
}
```

Here is an alternate way to write the row variable:

```
ROW202A: [8^4//1/5] $ [MEAN] [8] WHEN [8^1-4]  
  
~EXECUTE  
TABLE_SET= TAB202
```

**ADVANCED TABLES**  
 6.2 SUMMARY STATISTICS (MEANS)

Here is the table Mentor prints:

MEAN EXCLUDING A DON'T KNOW CODED AS A NUMBER  
 TABLE 202  
 OVERALL RATING OF PRODUCT B

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
TOTAL		100	58	42
		100.0%	100.0%	100.0%
EXCELLENT	(4)	25	13	12
		25.0%	22.4%	28.6%
GOOD	(3)	21	10	11
		21.0%	17.2%	26.2%
FAIR	(2)	24	17	7
		24.0%	29.3%	16.7%
POOR	(1)	18	12	6
		18.0%	20.7%	14.3%
DON'T KNOW		12	6	6
		12.0%	10.3%	14.3%
MEAN		2.60	2.46	2.81





## WITH THE SCALE REVERSED

Sometimes rating scales are coded such that a 1 signifies the highest possible rating and the highest number in the scale signifies the lowest rating. If you produce a mean on this variable without doing any recoding, then the lower the value of the mean, the higher the overall rating. In order to make a higher mean reflect a higher rating, you need to reverse the scale so that a 1 punch now has the highest value and the highest punch now has a value of 1. There are basically two different approaches, the subtraction method and the function method.

Using the subtraction method, you reverse the scale by taking one more than the highest value in the scale and subtracting the variable or column location from it. Suppose you have a 7 point scale coded in column 20, but you want the 1 punch to have a value of 7 on down to the 7 punch having a value of 1. So you would define the mean as  $[\text{MEAN}] 8 - [20]$ . This is how Mentor reverses a scale:

<i>If there is a</i>	1	<i>in column 20, it becomes</i>	$(8 - 1) = 7$
	2		$(8 - 2) = 6$
	3		$(8 - 3) = 5$
	4		$(8 - 4) = 4$
	5		$(8 - 5) = 3$
	6		$(8 - 6) = 2$
	7		$(8 - 7) = 1$

See below for an example of the subtraction method inside of a table set.

A second way to reverse the scale is by using a function to reassign the values you want to use for purposes of calculating the mean. You can use either the SUBSCRIPT or the SELECT\_VALUE function. The SUBSCRIPT function assigns the values as the subscript position of each category. This means that the first category in the variable is assigned a value of 1, the second a value of 2, and so on. If the variable is multiple then no value is assigned. The SELECT\_VALUE function assigns the values based on the position of the category in the variable and

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

by what values are written in the VALUES portion of the function. See 8.3.2 *FUNCTIONS* for more information on these functions.

In the example below, the 4 point rating scale is stored in column 10 with Excellent coded as a 1, Poor coded as a 4, and Don't Know coded as an X punch.

```
~DEFINE
```

```
TABLE_SET= {TAB203:
```

```
HEADER=: MEAN REVERSING THE SCALE AND DON'T KNOW CODED  
AS AN X OR Y}
```

```
TITLE=: OVERALL RATING OF PRODUCT C}
```

```
STUB=:
```

```
    EXCELLENT    (4)
```

```
    GOOD         (3)
```

```
    FAIR         (2)
```

```
    POOR         (1)
```

```
    DON'T KNOW
```

```
    [STATISTICS_ROW] MEAN }
```

```
ROW=: [10^1//4/X] $ [MEAN] 5 - [10]
```

```
}
```



Here are some alternate ways to write the row variable:

```
ROW203A: [10^1//4/X] $ [MEAN] SUBSCRIPT ([10^4//1])
ROW203B: [10^1//4/X] $ [MEAN]
SELECT_VALUE ([10^4//1], VALUES (1, 2, 3, 4))
```

```
~EXECUTE
TABLE_SET= TAB203
```

The printed table will look fundamentally the same as Table 202 above.

### WITH THE SCALE REVERSED AND DK/NA CODED AS NUMERIC

If the rating scale is coded in reverse as in table 203 and the Don't Know is a numeric code as in Table 202, then you will need to use one of the following methods to properly calculate the mean. Either you need to combine the subtraction method of Table 203 with the exclusion method of Table 202 or you can just use either the SUBSCRIPT or SELECT\_VALUE functions as in Table 203.

In the example below the 4 point rating scale is coded in column 9 with a 1 punch signifying Excellent, a 4 punch signifying Poor and a 5 punch standing for Don't Know.

```
~DEFINE

TABLE_SET= {TAB204:
HEADER=: MEAN REVERSING THE SCALE AND EXCLUDING DON'T
KNOW}
TITLE=: OVERALL RATING OF PRODUCT D}
```

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
STUB= :
      EXCELLENT   (4)
      GOOD        (3)
      FAIR        (2)
      POOR        (1)
      DON'T KNOW
      [STATISTICS_ROW] MEAN }
ROW= : [11^1//5] $ [MEAN] SUBSCRIPT ( [11^4//1] )
      }
```

Here are some alternate ways to write the row variable:

```
ROW204A: [11^1//5] $ [MEAN] 5 - [11*RANGES=1-4]
ROW204B: [11^1//5] $ [MEAN] (5 - [11]) WHEN [11^1-4]
ROW204C: [11^1//5] $ [MEAN]
SELECT_VALUE ( [11^1//4] , VALUES (4, 3, 2, 1) )

~EXECUTE
TABLE_SET= TAB204
```

The printed table will look fundamentally the same as Table 202 above.

### WITH 10 CODED AS A ZERO, AN X, OR Y

If you have a 10 point rating scale that has been coded in one column to save space, the 0 is probably used to stand for a rating of 10. If you do no recoding on this variable, the mean will be low as all those who rated the item a 10 will be assigned the value of 0 for purposes of calculating the mean. In order to properly calculate this mean you will want to use the SUBSCRIPT function.

In the example below a 10 point rating scale is coded in column 12, with Don't Know coded as a Y punch. If the scale is reversed, all you need to do is reverse the order of the categories within the SUBSCRIPT function.

```

~DEFINE

TABLE_SET= {TAB205:
HEADER=: MEAN USING THE SUBSCRIPT FUNCTION TO RECODE 0
AS 10}
TITLE=: 10 POINT OVERALL RATING OF PRODUCT E}
STUB=:
>REPEAT
$A=ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, "DK
/NA"
      $A
>END_REPEAT
      [STATISTICS_ROW] MEAN }
ROW=: [12^1//0/Y] $[MEAN] SUBSCRIPT([12^1//0])
}

```

Here is an alternate way to write the row variable:

```

ROW205A: [12^1//0/Y] $[MEAN]
SELECT_VALUE([12^1//0], VALUES(1, 2, . . . , 10))

```

```

~EXECUTE
TABLE_SET= TAB205

```

**ADVANCED TABLES**  
**6.2 SUMMARY STATISTICS (MEANS)**

Here is the table Mentor prints:

MEAN USING THE SUBSCRIPT FUNCTION TO RECODE 0 AS 10  
 TABLE 205  
 10 POINT OVERALL RATING OF PRODUCT E

	GENDER		
	TOTAL	MALE	FEMALE
	-----	-----	-----
TOTAL	100 100.0%	58 100.0%	42 100.0%
ONE	9 9.0%	4 6.9%	5 11.9%
TWO	13 13.0%	8 13.8%	5 11.9%
THREE	5 5.0%	4 6.9%	1 2.4%
FOUR	10 10.0%	6 10.3%	4 9.5%
FIVE	9 9.0%	4 6.9%	5 11.9%
SIX	9 9.0%	4 6.9%	5 11.9%
SEVEN	7 7.0%	2 3.4%	5 11.9%

(Table continued on next page)

EIGHT	8 8.0%	6 10.3%	2 4.8%
NINE	10 10.0%	5 8.6%	5 11.9%
TEN	11 11.0%	9 15.5%	2 4.8%
DK/NA	9 9.0%	6 10.3%	3 7.1%
MEAN	5.52	5.71	5.26

One other possible scenario very similar to this, is if the scale is from 0 to 10 with the 0 (or 10) punch standing for 0, and either the X or Y punch standing for 10. This row definition would look the same as above, except that you would probably always want to use the SELECT\_VALUE function. In the example below, assume that the 0 stands for 0, the X for 10, and the Y punch for Don't Know. Only the row variable definition is shown.

```
ROW205X: [12^0/1//9/X/Y] $[MEAN] &
SELECT_VALUE([12^0/1//9/X],VALUES(0,1,...,10))
```

## 6.2.2 Means For Range Type Variables

Sometimes you need to calculate a mean on data that has been coded as a range type variable with a code standing for a range of numbers. For example, you might have the age question coded so that a 1 punch means 18 to 30 and a 2 punch means 31 to 45 and so on. If you were to just do a straight mean on this table your result would be some number like 3.5, so to ensure you get a mean reflecting the real value you need to use the SELECT\_VALUE function to assign the midpoint of each range to the category. To determine the midpoint of the range, add the starting

point and the ending point of the range and divide by 2 (i.e.,  $(18+30)/2 = 24$ ). Open-ended categories (Under 18 or Over 60) will need someone to decide what the midpoint value should be. It is usually a good idea to include the value that was used on each stub to reduce possible confusion about how the mean was derived.

**NOTE:** If you try to calculate a regular median with this method, it will just be the midpoint value of where the middle respondent exists. Instead, you would want to use an interpolated median, which produces a median value that is the mid-point between the true median and next value beyond it. TABLE 206 below includes both the median and interpolated median for comparison.

## INTERPOLATED MEDIANS

If you try to calculate a median with this method, it will just be the midpoint value of where the middle respondent exists. You might want to use INTERPOLATED\_MEDIAN instead.

Mentor provides three different kinds of medians:

\$(MEDIAN)  
\$(INTERPOLATED\_MEDIAN) and  
~EDIT COLUMN\_MEDIAN.

\$(MEDIAN) is best for numeric data. \$(INTERPOLATED\_MEDIAN) and COLUMN\_MEDIAN are two different types of interpolated medians, and they are better for rating scales and range questions. It must be noted here that interpolated medians are considered a "junk statistic" by many marketing people.

A true median is always the value of an existing data element. To find this median, values must be sorted from low to high. If there are an even number of elements, the median will be the value in the  $N/2$  position, where  $N$  is the number of data elements. If there are an odd number of elements, then the median will be the item in the  $(N+1)/2$  position. The unfortunate result of this is that most rating scales and



range questions will always have the midpoint of the middle category as their median, because if the median falls anywhere in that category, the value is that is assigned is the midpoint. Interpolated medians breaks the range of data into pieces and can make the median higher or lower, depending on the number of people in each category. If there is an odd number of values, then MEDIAN and INTERPOLATED\_MEDIAN will produce the same result. If there is an even number of values, INTERPOLATED\_MEDIAN will find and average the two middle values. For example, if the values are 1, 3, 5, and 7, the INTERPOLATED\_MEDIAN is 4.

\$INTERPOLATED\_MEDIAN will work on unweighted data and data weighted with integer weights. Due to rounding issues, INTERPOLATED\_MEDIAN is not recommended with fractional weights.

In the following example, both MEDIAN and INTERPOLATED\_MEDIAN are included to show the difference between them. For details about ~EDIT COLUMN\_MEDIAN, see 6.2.5 MEANS AND COLUMN\_MEDIANS USING THE EDIT OPTIONS.

In the example below, the age question was coded as range variable in column 13 with a 1 punch signifying age 18 to 30, a 2 punch the 31 to 45 age group, a 3 punch the 46 to 60 group, a 4 punch the over 60 group, and a 5 punch for Don't Know.

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
~DEFINE

    TABLE_SET= {TAB206:
    HEADER=: MEAN AND MEDIAN USING THE SELECT FUNCTION
TO ASSIGN
    MIDPOINT VALUES TO CATEGORIES}
    TITLE: AGE OF RESPONDENT}
    STUB=:
    18 - 30    (24)
    31 - 45    (38)
    46 - 60    (53)
    OVER 60    (61)
    DON'T KNOW
    [STATISTICS_ROW] MEAN
    [STATISTICS_ROW, STATISTICS_DECIMALS=0] MEDIAN
    [STATISTICS_ROW, STATISTICS_DECIMALS=0] INTERPO-
LATED MEDIAN }
    ROW=: [13^1//5] $ [MEAN, MEDIAN, INTERPOLATED_MEDIAN]
&
    SELECT_VALUE( [13^1//4], VALUES(24,38,53,61) ) }

~EXECUTE
TABLE_SET= TAB206
```

Here is the table Mentor prints:

MEAN AND MEDIAN USING THE SELECT FUNCTION TO ASSIGN MIDPOINT  
VALUES TO CATEGORIES

TABLE 206

AGE OF RESPONDENT

GENDER

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
TOTAL		200	57	66
		100.0%	100.0%	100.0%
18 - 30	(24)	37	12	13
		18.5%	21.1%	19.7%
31 - 45	(38)	35	8	13
		17.5%	14.0%	19.7%
46 - 60	(53)	39	13	10
		19.5%	22.8%	15.2%
OVER 60	(61)	36	7	13
		18.0%	12.3%	19.7%
DON'T KNOW		30	10	8
		15.0%	17.5%	12.1%
MEAN		44.09	42.70	43.45
MEDIAN		53	38	38
INTERPOLATED_MEDIAN		53	46	38

### 6.2.3 Means For Numeric Data

Doing a mean on a numeric type variable usually falls into one of these three basic scenarios:

- The valid range of numbers is from  $j$  to  $k$  (where  $j$  and  $k$  are any real numbers) and DK is coded as some non-numeric value (i.e., DK or XX or --).
- The valid range of numbers is from  $j$  to  $k$  ( $j$  and  $k$  as above) and DK is coded as a number (i.e., 99 or 98 or -1).
- The valid range of numbers is from  $j$  to  $k$  ( $j$  and  $k$  as above) and some non-numeric code is used for some outer value (i.e., -- for 100 or && for .5).

#### WITH NO RECODING NECESSARY

If no recoding is necessary for the mean, then you can produce the mean by just referencing the data location or variable name in brackets. In the example below, suppose that the number of times the respondent has used product A in the last year is stored in columns 14 and 15, where 99 or more is coded as 99, and Don't Know is coded as DK. Since all items you wish to include in the mean are numeric and all you wish to exclude are non-numeric, no recoding is needed and you can just specify the location of the mean in brackets after the `$(MEAN)` keyword.



~DEFINE

```
TABLE_SET= {TAB207:
HEADER=: MEAN AND MEDIAN ON NUMERIC TYPE VARIABLE,
DON'T KNOW CODED AS DK}
TITLE=: NUMBER OF TIMES USED PRODUCT A IN LAST YEAR}
STUB=:
    0 - 10
    11 - 20
    21 - 50
    OVER 50
    DON'T KNOW
    [STATISTICS_ROW] MEAN
    [STATISTICS_ROW, STATISTICS_DECIMALS=0] MEDIAN }
ROW=: [14.2#0-10/11-20/21-50/51-99/"DK"] $ [MEAN MEDIAN]
[14.2]
}
```

~EXECUTE

```
TABLE_SET= TAB207
```

ADVANCED TABLES

6.2 SUMMARY STATISTICS (MEANS)

Here is the table Mentor prints:

MEAN AND MEDIAN ON NUMERIC TYPE VARIABLE, DON'T KNOW CODED AS DK

TABLE 207

NUMBER OF TIMES USED PRODUCT A IN LAST YEAR

	GENDER		
	TOTAL	MALE	FEMALE
	-----	-----	-----
TOTAL	100	58	42
	100.0%	100.0%	100.0%
0 - 10	11	6	5
	11.0%	10.3%	11.9%
11 - 20	7	3	4
	7.0%	5.2%	9.5%
21 - 50	28	16	12
	28.0%	27.6%	28.6%
OVER 50	45	28	17
	45.0%	48.3%	40.5%
DON'T KNOW	9	5	4
	9.0%	8.6%	9.5%
MEAN	50.63	52.43	48.11
MEDIAN	50	56	45

## WITH DON'T KNOW CODED AS A NUMBER

If the Don't Know response has been coded as a numeric value like 99, then if you do not do any recoding the mean will be much too high as all the Don't Know responses will be assigned a value of 99. You need to exclude these responses by either using the \*RANGES modifier or the WHEN joiner. See “6.2.1 Means on Rating Scales Using the Variable Definition” for a more detailed explanation of these.

In the example below the number of times the product was used in the past year has been coded as a two digit number in columns 16 and 17 while the Don't Know response was coded as 99.

```
~DEFINE
```

```
TABLE_SET= {TAB208:
HEADER=: MEAN AND MEDIAN ON NUMERIC TYPE VARIABLE WITH
DON'T KNOW CODED AS 99}
TITLE=: NUMBER OF TIMES USED PRODUCT B IN LAST YEAR}
STUB=:
    0 - 10
    11 - 20
    21 - 50
    OVER 50
    DON'T KNOW
    [STATISTICS_ROW] MEAN
    [STATISTICS_ROW, STATISTICS_DECIMALS=0] MEDIAN }
ROW=: [16.2#0-10/11-20/21-50/51-98/99] $ [MEAN,MEDIAN]
[16.2*RANGES=0-98]
}
```

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

Here is an alternate way to write the row variable:

```
ROW208A:  [16.2#0-10/11-20/21-50/51-98/99]  
$ [MEAN, MEDIAN] [16.2] WHEN &  
          [16.2#0-98]
```

```
~EXECUTE
```

```
TABLE_SET= TAB208
```

The printed table will look fundamentally the same as Table 207 above.

### WITH A NUMERIC VALUE CODED AS A NON-NUMERIC

If the data has been coded so that a non-numeric code like XX has been used to code a numeric value like 100, then you will need to use either the OTHERWISE joiner or the SUM function in combination with the SELECT\_VALUE function in order to properly recode the data. The SELECT\_VALUE function is used to assign the values to the non-numeric codes and the OTHERWISE joiner or the SUM function is used to combine these values with the good numeric values that exist in the data. See 5.1.2 VECTOR JOINERS for more information on the OTHERWISE joiner and 8.3.2 FUNCTIONS for details on the SUM function.

In the example below the data was coded in columns 18 and 19. 100 or more was coded as "--" and Don't Know was coded as DK. The [18.2] after the [\$MEAN] variable will assign the numeric value to all valid numeric codes, the SELECT\_VALUE function will assign a value of 100 to the "--" code, and the OTHERWISE joiner will cause the program to combine the two values using the SELECT\_VALUE function only if there is no valid numeric.

**NOTE:** The X punch is referenced as a "-" when you are inside a # type variable.



~DEFINE

```
TABLE_SET= {TAB209:
HEADER=: MEAN ON A NUMERIC TYPE VARIABLE WITH 100 CODED
AS -- (XX in punch mode)}
TITLE=: NUMBER OF TIMES USED PRODUCT C IN LAST YEAR}
STUB=:
    0   - 10
    11  - 20
    21  - 50
    51  - 99
    100
    DON'T KNOW
    [STATISTICS_ROW] MEAN }
ROW=: [18.2#0-10/11-20/21-50/51-99/--"/DK] $ [MEAN] &
      [18.2] OTHERWISE
SELECT_VALUE ([18.2#"--"],VALUES (100))
}
```

Here is an alternate way to write the row variable:

```
ROW209A: [18.2#0-10/11-20/21-50/51-99/--"/DK]
$ [MEAN] &

SUM ([18.2],SELECT_VALUE ([18.2#"--"],VALUES (100)))

~EXECUTE
TABLE_SET= TAB209
```

**ADVANCED TABLES**  
**6.2 SUMMARY STATISTICS (MEANS)**

Here is the table Mentor prints:

MEAN ON A NUMERIC TYPE VARIABLE WITH 100 CODED AS -- (XX in punch mode)

TABLE 209

NUMBER OF TIMES USED PRODUCT C IN LAST YEAR

	GENDER		
	<----->		
TOTAL	MALE	FEMALE	
-----	----	-----	
TOTAL	100	58	42
	100.0%	100.0%	100.0%
0 - 10	7	4	3
	7.0%	6.9%	7.1%
11 - 20	7	4	3
	7.0%	6.9%	7.1%
21 - 50	24	14	10
	24.0%	24.1%	23.8%
51 - 99	44	25	19
	44.0%	43.1%	45.2%
100 OR MORE	10	8	2
	10.0%	13.8%	4.8%
DON'T KNOW	8	3	5
	8.0%	5.2%	11.9%
MEAN	56.95	57.96	55.43

## 6.2.4 Summary Statistics in the Column Variable

All the summary statistics that can be produced for a row variable can also be produced for a column variable. The syntax for the definition of the column variable is exactly the same as for the row definition and any recoding would be specified the same way.

The one major difference of producing a mean in the column is that you will need to use the `COLUMN_INFO` option on the `EDIT` statement to make sure the data is formatted correctly. The `COLUMN_INFO` option allows you to set different print options for each banner point of the table just as the `STUB` options allow you to control how each row prints. See “5.3 Changing Table Specifications” for a description of the syntax for the `COLUMN_INFO` statement.

You will also need to make sure that you create your own summary rows (total, no answer, etc.) in the variable and suppress the system-generated ones, because the system-generated summary rows will print the number of valid respondents in the statistic and not the value of the statistic. In the example below suppose you want to produce a table similar to Table 207 in section 6.2.3, but wanted the number of times used the product in the column and the gender of the respondent as the row variable. Notice that there is a new `STUB_PREFACE` used to suppress the system-generated total and that both the `STUB` and the `ROW` have a user-generated total. Also note how the `COLUMN_INFO` option causes the 7th banner point to print as a statistic with 2 decimals and the 8th one as a statistic with 0 decimals.

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

~DEFINE

TABLE\_SET= {TAB210:

EDIT=:

COLUMN\_WIDTH=6,STUB\_WIDTH=10,-COLUMN\_TNA,STATISTICS\_DECIMALS=2,

PERCENT\_DECIMALS=0,VERTICAL\_PERCENT=1,

COLUMN\_INFO=(COLUMN=7,WIDTH=8,STATISTICS\_COLUMN

WIDTH=7,STATISTICS\_COLUMN,STATISTICS\_DECIMALS=0)}

STUB\_PREFACE=:

[SUPPRESS] TOTAL

[SUPPRESS] NO ANSWER }

BANNER=:

| NUMBER OF TIMES USED PRODUCT A IN PAST YEAR

| <=====

| TOTAL 0-10 11-20 21-50 51-99 DK MEAN MEDIAN

| -----

COLUMN=: TOTAL WITH [14.2#0-10/11-20/21-50/51-99/"DK"] \$[MEAN  
MEDIAN] [14.2]

HEADER=: SUMMARY STATISTICS IN THE BANNER}

TITLE=: GENDER OF RESPONDENT}

STUB=:

TOTAL

MALE

FEMALE }

ROW=: TOTAL WITH [5^1/2]

}

~EXECUTE

TABLE\_SET= TAB210

Here is the table Mentor prints:

SUMMARY STATISTICS IN THE BANNER  
TABLE 210  
GENDER OF RESPONDENT

	NUMBER OF TIMES USED PRODUCT A IN PAST YEAR						MEAN	MEDIAN
	TOTAL	0-10	11-20	21-50	51-99	DK		
	-----	-----	-----	-----	-----	--	-----	-----
TOTAL	100	11	7	28	45	9	50.63	50
	100%	100%	100%	100%	100%	100%		
MALE	58	6	3	16	28	5	52.43	56
	58%	55%	43%	57%	62%	56%		
FEMALE	42	5	4	12	17	4	48.11	45
	42%	45%	57%	43%	38%	44%		

**SUMMARY STATISTICS IN BOTH THE COLUMN AND THE ROW**

You can have both the column and row variables have summary statistics in them, but the cross of the two statistics will always be missing and will print as a "?".

You can change the "?" to some other character by using the EDIT=PUT\_CHARACTER option.

The example below crosses the number of times product A was used last year with the number of times product B was used last year. This will use the column and banner variables from the previous Table 210 and a similar definition to the row

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

and stub that were used for Table 208. The difference between this row and stub and those for Table 208 is that these must have a user-generated Total row.

```
~DEFINE
```

```
TABLE_SET= {TAB211:  
HEADER=: MEAN AND MEDIAN IN BOTH THE COLUMN AND THE ROW  
VARIABLE}  
TITLE=: NUMBER OF TIMES USED PRODUCT B IN LAST YEAR}  
STUB=:  
    TOTAL  
    0 - 10  
    11 - 20  
    21 - 50  
    OVER 50  
    DON'T KNOW  
    [STATISTICS_ROW] MEAN  
    [STATISTICS_ROW, STATISTICS_DECIMALS=0] MEDIAN }  
ROW=: TOTAL WITH [16.2#0-10/11-20/21-50/51-98/99] &  
    $ [MEAN,MEDIAN] [16.2*RANGES=0-98]  
}
```

```
~EXECUTE
```

```
TABLE_SET= TAB211
```

Here is the table Mentor prints:

MEAN AND MEDIAN IN BOTH THE COLUMN AND THE ROW VARIABLE  
TABLE 211  
NUMBER OF TIMES USED PRODUCT B IN LAST YEAR

	NUMBER OF TIMES USED PRODUCT A IN PAST YEAR						MEAN	MEDIAN
	TOTAL	0-10	11-20	21-50	51-99	DK		
	-----	-----	-----	-----	-----	--	-----	-----
TOTAL	100	11	7	28	45	9	50.63	50
	100%	100%	100%	100%	100%	100%		
0 - 10	13	1	2	5	5	-	44.54	37
	13%	9%	29%	18%	11%			
11 - 20	12	1	1	2	5	3	47.33	54
	12%	9%	14%	7%	11%	33%		
21 - 50	22	2	1	7	10	2	51.05	44
	22%	18%	14%	25%	22%	22%		
OVER 50	40	4	3	12	19	2	52.87	48
	40%	36%	43%	43%	42%	22%		
DON'T KNOW	13	3	-	2	6	2	52.00	68
	13%	27%		7%	13%	22%		
MEAN	45.61	50.88	38.29	44.27	47.77	39.86	?	?
MEDIAN	43	34	27	40	48	28	?	?

### 6.2.5 Means And Medians Using The EDIT Options

The summary statistics mean, standard deviation, standard error, variance, and median can be produced by using either the variable definition as explained in the previous sections or by using the EDIT options, COLUMN\_STATISTICS\_VALUES, COLUMN\_MEAN, COLUMN\_STD, COLUMN\_SE, COLUMN\_VARIANCE, and COLUMN\_MEDIAN to produce column statistics (extra rows will be generated at the bottom of the table) or ROW\_STATISTICS\_VALUES, ROW\_MEAN, ROW\_STD, ROW\_SE, ROW\_VARIANCE, and ROW\_MEDIAN to produce row statistics (extra columns will be generated on the right hand side of the table).

The advantages to creating the summary statistics by using the EDIT options are:

- The processing time of the computer is significantly less
- There is no change in the syntax regardless of how the Don't Know was coded
- The labels are automatically generated
- The system-generated summary columns/rows will have the correct values.

Limitations of creating summary statistics this way are:

- Values are only correct if every distinct number in the range is a category that is printed on the table (this will most likely not work for standard numeric type variables).
- Standard deviations, standard errors, and variances cannot be properly calculated on weighted data.
- Dependent statistical testing cannot be performed on the mean.
- Only one mean can be generated per table (No mean summaries).

The amount of processing time saved by doing summary statistics using the EDIT options is dependent on the number of respondents in the data file and the number of rows in the table. The greater the number of respondents causes the savings to be greater. For example, if you had 1000 respondents who answered a 5 point rating scale, then the EDIT options method would process the mean up to 100



times faster. This is not to imply that the entire table will process 100 times faster, but it might be as much as twice as fast. This could be a very significant difference if you increase the number of respondents to 100,000.

When you use the EDIT options to produce summary statistics you must tell the program what value you want to assign to each row in the table. Use the COLUMN\_STATISTICS\_VALUES=VALUES() option to set the values for a mean, standard deviation, or standard error. The COLUMN\_MEDIAN=VALUES() option is used to set the values for a median. In either case separate each row value with a comma; use a double comma to exclude a row from the calculation, and use the ellipses to generate a string of values.

**NOTE:** If you have more rows than values, all the extra rows will be excluded from any calculation.

## MEANS ON A RATING SCALE

Since it makes no difference how the Don't Know was coded, this example can replace all the rating scale examples in *6.2.1 MEANS ON RATING SCALES USING THE VARIABLE DEFINITION*. In this example the rating for brand A is stored in column 7. The COLUMN\_STATISTICS\_VALUES option is set to "VALUES(4,3,2,1)" which will cause the first row in the table to have a value of 4, the second row 3, the third row 2, and the fourth row 1. The fifth and any additional rows will not be included in the calculation.

**NOTE:** The statistical values for this table are exactly the same as Table 201.

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

~DEFINE

```
TABLE_SET= {TAB212:
HEADER=:
MEAN AND OTHER SUMMARY STATISTICS ON A RATING SCALE
USING THE EDIT OPTIONS}
TITLE=: OVERALL RATING OF PRODUCT A}
LOCAL_EDIT=: COLUMN_STATISTICS_VALUES=VALUES(4,3,2,1),
COLUMN_MEAN,
        COLUMN_STD, COLUMN_SE }
STUB=:
        EXCELLENT (4)
        GOOD (3)
        FAIR (2)
        POOR (1)
        DON'T KNOW }
ROW=: [7^4//1/X]
}
```

~EXECUTE

```
TABLE_SET= BAN1
TABLE_SET= TAB212
```

The printed table will look fundamentally the same as Table 201.

## MEANS ON A RATING SCALE WITH ROWS IN THE MIDDLE THAT NEED TO BE EXCLUDED

If you have additional rows in the middle of the table that need to be excluded from the calculation such as a top 2 box or a bottom 2 box, then you need to make sure that the COLUMN\_STATISTICS\_VALUES command has the appropriate number of commas.

This example is the same as Table 212 except it has both a top 2 box and bottom 2 box. Notice the COLUMN\_STATISTICS\_VALUES command now looks like "VALUES(4,3,,2,1)" which will cause the first and fourth rows in the table to be excluded from the statistics.

```

~DEFINE

TABLE_SET= {TAB213:
HEADER=:
MEAN USING THE EDIT OPTION AND EXCLUDING INTERNAL ROWS FROM
THE CALCULATION }
TITLE=: OVERALL RATING OF PRODUCT A}
LOCAL_EDIT=: COLUMN_STATISTICS_VALUES=VALUES(,4,3,,2,1),
COLUMN_MEAN }
STUB=:
    |TOP BOX
    |  EXCELLENT  (4)
    |  GOOD      (3)
    |BOTTOM BOX
    |  FAIR      (2)
    |  POOR     (1)
    |DON'T KNOW }
ROW=: [7^4,3/4/3/2,1/2/1/X]
}

~EXECUTE
TABLE_SET= TAB213
  
```

**ADVANCED TABLES**  
**6.2 SUMMARY STATISTICS (MEANS)**

Here is the table Mentor prints:

MEAN USING THE EDIT OPTION AND EXCLUDING INTERNAL ROWS FROM  
 THE CALCULATION

TABLE 213

OVERALL RATING OF PRODUCT A

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
TOTAL		100	58	42
		100.0%	100.0%	100.0%
TOP BOX		30	21	9
		30.0%	36.2%	21.4%
EXCELLENT	(4)	15	11	4
		15.0%	19.0%	9.5%
GOOD	(3)	15	10	5
		15.0%	17.2%	11.9%
BOTTOM BOX		47	27	20
		47.0%	46.6%	47.6%
FAIR	(2)	23	12	11
		23.0%	20.7%	26.2%
POOR	(1)	24	15	9
		24.0%	25.9%	21.4%
DON'T KNOW		23	10	13
		23.0%	17.2%	31.0%
Mean		2.27	2.35	2.14



## MEANS ON A RANGE VARIABLE

If you have a range variable you can produce the summary statistics by assigning the value of the midpoint of each range to each row. See “6.2.2 Means For Range Type Variables” for more information on how and why to use the midpoint.

**NOTE:** This will produce the same mean values as Table 206 in section 6.2.2.

```

~DEFINE

TABLE_SET= {TAB214 :
HEADER=: MEAN ON A RANGE TYPE VARIABLE USING THE EDIT
OPTION }
TITLE=: AGE OF RESPONDENT}
LOCAL_EDIT=:
COLUMN_STATISTICS_VALUES=VALUES (24 , 38 , 53 , 61) ,
COLUMN_MEAN }
STUB=:
    18 - 30    (24)
    31 - 45    (38)
    46 - 60    (53)
    OVER 60    (61)
    DON'T KNOW }
ROW=: [13^1//5]
}

~EXECUTE
TABLE_SET= TAB214

```

The printed table will look fundamentally the same as Table 206.

**CHANGING THE DEFAULT PRINT OPTIONS**

If you do not like the default labelling or default printing format for the system-generated summary statistics, you can use the PRINT\_ROW option on the stub label to change them. The PRINT\_ROW option allows you to print any system-generated row anywhere in the table, and when used in conjunction with all the other STUB options it allows you to format these lines any way in which you choose. The following example shows how to change the default labelling for the mean, standard deviation, and standard error.

**NOTE:** The numbers for this table are exactly the same as Table 212.

```

~DEFINE

TABLE_SET= {TAB215:
HEADER=:
SUMMARY STATISTICS USING THE EDIT OPTION AND CHANGING THE WAY
IT PRINTS }
TITLE=: OVERALL RATING OF PRODUCT A}
LOCAL_EDIT=: COLUMN_STATISTICS_VALUES=VALUES(4,3,2,1),
              COLUMN_MEAN,COLUMN_STD,COLUMN_SE }
STUB=:
      EXCELLENT (4)
      GOOD      (3)
      FAIR      (2)
      POOR      (1)
      DON'T KNOW
      [PRINT_ROW=MEAN] AVERAGE
      [PRINT_ROW=STD] STD DEV
      [PRINT_ROW=SE] STD ERR }
ROW=: [7^4//1/X]
}

~EXECUTE
TABLE_SET= TAB215

```

Here is the table Mentor prints:

SUMMARY STATISTICS USING THE EDIT OPTION AND CHANGING THE WAY  
IT PRINTS  
TABLE 215  
OVERALL RATING OF PRODUCT A

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
TOTAL		100	58	42
		100.0%	100.0%	100.0%
EXCELLENT	(4)	15	11	4
		15.0%	19.0%	9.5%
GOOD	(3)	15	10	5
		15.0%	17.2%	11.9%
FAIR	(2)	23	12	11
		23.0%	20.7%	26.2%
POOR	(1)	24	15	9
		24.0%	25.9%	21.4%
DON'T KNOW		23	10	13
		23.0%	17.2%	31.0%
AVERAGE		2.27	2.35	2.14
STD DEV		1.11	1.16	1.03
STD ERR		0.13	0.17	0.19

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

To rename the system-generated statistics for an entire run you can build a stub that formats the statistics the way you want and then use the STUB\_SUFFIX table element on any table that you are producing this type of statistic on. The example below produces the exact same table as Table 215.

```
~DEFINE
```

```
STUB= STUBBOT1:
```

```
    [PRINT_ROW=MEAN] AVERAGE
```

```
    [PRINT_ROW=STD] STD DEV
```

```
    [PRINT_ROW=SE] STD ERR }
```

```
TABLE_SET= {TAB216:
```

```
HEADER=:
```

```
SUMMARY STATISTICS USING THE EDIT OPTION AND CHANGING  
THE WAY IT PRINTS }
```

```
TITLE=: OVERALL RATING OF PRODUCT A}
```

```
STUB_SUFFIX= STUBBOT1
```

```
LOCAL_EDIT=: COLUMN_STATISTICS_VALUES=VALUES(4,3,2,1),  
COLUMN_MEAN,
```

```
    COLUMN_STD, COLUMN_SE }
```

```
STUB=:
```

```
    EXCELLENT (4)
```

```
    GOOD (3)
```

```
    FAIR (2)
```

```
    POOR (1)
```

```
    DON'T KNOW }
```

```
ROW=: [7^4//1/X]
```

```
}
```

```
~EXECUTE
```

```
TABLE_SET= TAB216
```



The printed table will look fundamentally the same as Table 215.

## COLUMN MEDIANS

COLUMN\_MEDIAN is type of interpolated median, which you can use instead of MEDIAN or INTERPOLATED\_MEDIAN. Because of the way it is calculated, a true median on rating scales and range questions is usually the midpoint of the middle category. The COLUMN\_MEDIAN does not use the midpoint of the median category. It breaks that category into ranges for each data element and, therefore, can make the median higher or lower, depending on the number of people in each category. (See “6.2.2 Means For Range Type Variables” for more information on interpolated medians).

COLUMN\_MEDIAN is best used with grouped data. The values for COLUMN\_MEDIANS are not the midpoints, but rather the starting point of each range. Each category then goes from its starting point to the starting point of the next category. You will need one extra value at the end to act as the ending point for the last category. If you have a Don't Know category, exclude it with an extra comma. Examples for setting up a COLUMN\_MEDIAN on rating scales and ranges follows. If you get a question mark instead of a median, see the section entitled Lost Medians.

### *Column Medians on Rating Scales*

When doing an column median on a rating scale with an ascending scale you will want to assign the starting value for each range as .5 less than its actual value. This will cause that value to have a range from .5 less than its value to .5 greater than its value. In the example below, the first row in the table has a range from .5 to 1.5, which is exactly what you want for the value of 1. You will also need to make sure the Don't Know is excluded and that there is an upper boundary for the last category. Notice that there is one more value in the COLUMN\_MEDIAN option than there are rows in the table. If you have a descending scale then you will want to assign the starting value as .5 higher than its actual value. The example below produces a column median on data column 7. Compare this with the median that

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

was printed in Table 003 in “6.2.1 Means on Rating Scales Using the Variable Definition”.

```
~DEFINE
```

```
TABLE_SET= {TAB216:  
HEADER=: COLUMN MEDIAN ON A RATING SCALE}  
TITLE=: OVERALL RATING OF PRODUCT A}  
LOCAL_EDIT=: COLUMN_MEDIAN=VALUES(4.5,3.5,2.5,1.5,,.5)  
}  
STUB=:  
    EXCELLENT (4)  
    GOOD      (3)  
    FAIR      (2)  
    POOR      (1)  
    DON'T KNOW }  
ROW=: [7^4//1/X]  
}
```

```
~EXECUTE
```

```
TABLE_SET= TAB216
```



Here is the table Mentor prints:

COLUMN MEDIAN ON A RATING SCALE  
TABLE 216

OVERALL RATING OF PRODUCT A

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
TOTAL		100	58	42
		100.0%	100.0%	100.0%
EXCELLENT	(4)	15	11	4
		15.0%	19.0%	9.5%
GOOD	(3)	15	10	5
		15.0%	17.2%	11.9%
FAIR	(2)	23	12	11
		23.0%	20.7%	26.2%
POOR	(1)	24	15	9
		24.0%	25.9%	21.4%
DON'T KNOW		23	10	13
		23.0%	17.2%	31.0%
Median		2.13	2.25	2.00

*Column Medians On Range Type Variables*

The approach for defining the values for a median on a range type variable is similar to that for a rating scale, except the value you assign to each row is the starting point for that row. Again, you will need to exclude any Don't Know category and make sure you assign an upper boundary to the last category.

If you do not define an upper boundary for the last category and the median should fall into that category, then a question mark (?) will print on the table denoting a missing median. When assigning the upper boundary, someone will have to decide which value to use. But, unlike the value assigned for a mean, the value of the upper boundary will only matter if the median falls into the last category. Otherwise, it will have no effect on the median at all.

```

~DEFINE

TABLE_SET= {TAB217:
HEADER=: COLUMN MEDIAN ON A RANGE TYPE VARIABLE }
TITLE=: AGE OF RESPONDENT}
LOCAL_EDIT=: COLUMN_MEDIAN=VALUES(18,31,46,61,,75) }
STUB=:
    18 - 30
    31 - 45
    46 - 60
    OVER 60
    DON'T KNOW }
ROW=: [13^1/5]
}

~EXECUTE

TABLE_SET= TAB217

```



Here is the table Mentor prints:

COLUMN MEDIAN ON A RANGE TYPE VARIABLE  
TABLE 217  
AGE OF RESPONDENT

	GENDER		
	<----->		
	TOTAL	MALE	FEMALE
	-----	----	-----
TOTAL	100	58	42
	100.0%	100.0%	100.0%
18 - 30	30	16	14
	30.0%	27.6%	33.3%
31 - 45	17	7	10
	17.0%	12.1%	23.8%
46 - 60	13	11	2
	13.0%	19.0%	4.8%
OVER 60	20	10	10
	20.0%	17.2%	23.8%
DON'T KNOW	20	14	6
	20.0%	24.1%	14.3%
Median	39.82	43.86	37.00

***Formula For Column Medians***

This is the procedure the program uses to determine what the column median is. You can use it if you want to verify that the median that is printing on the table is the one you expect (i.e., you have properly assigned the values for the median calculation).

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

The formula for the interpolated median is:

$$S + ((D * (E - S)) / F)$$

where:

- S is the starting point of the range of the row where the median exists
- E is the ending point of the range of the row where the median exists
- D is the difference that is left when you subtract out all the categories before the row where the median exists.
- F is the frequency of the row where the median exists

S, E, and F can all be determined once you know in which row the median exists. D can be determined in the process of finding out in which row the median exists.

To determine the row in which the median exists you must first sum up all the frequencies on the valid responses and divide that by 2. You then want to subtract the first valid frequency from the above total, and then subtract the second valid number from that, and so on until that subtraction would result in a number less than or equal to 0. The row that would cause this is the row where the median resides. The value you had before subtracting out this row is the value D above. So now you can calculate the median.

Here is an example to better explain this. Let us try to reproduce the median that the program produced in the total column for Table 217.

The pertinent numbers from that table are:

AGE	FREQUENCY	VALUES FROM COLUMN_MEDIAN
-----	-----	-----
18 - 30	30	18
31 - 45	17	31
46 - 60	13	46
OVER 60	20	60
		75

The sum of the valid responses is 80 (30+17+13+20) and the midpoint is (80/2)=40. Now you take 40 and subtract 30 from it and get 10. Now if you try to subtract 17 from 10 you will get -7 which is <=0 so the second row is the one where the median exists and 10 was the difference prior to the subtraction. So S=31, E=46, F=17, and D=10. Now substituting into the formula above:

$$\begin{aligned}
 S &+ \left( \frac{D * (E - S)}{F} \right) \\
 31 &+ \left( \frac{10 * (46 - 31)}{17} \right) \\
 31 &+ \left( \frac{10 * 15}{17} \right) \\
 31 &+ \left( \frac{150}{17} \right) \\
 31 &+ 8.82 = 39.82
 \end{aligned}$$

**NOTE:** If you have an odd number of valid responses then N/2 will be a number with .5 in it and so will D.

### ***Lost Medians***

Calculating a median is a more complicated statistic for the computer to do because it must maintain the entire collection of values in an array. (Arrays are sometimes described as "the number of buckets to hold values.") The default array size is 50, which is sometimes too low for Mentor to calculate a median. If the array size is too small and the median cannot be calculated, it is called a "lost median" and Mentor will print a question mark (?) where the median should be.

The three main causes of lost medians are as follows:

- There are too many values for Mentor to calculate the median. (For example, you are trying to get the median from a seven column field with hundreds of different values.)
- Over half of the values exist in either the highest or lowest value. (For example, over half of the values are zero.)

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

- The data is sorted by the variable you are trying to produce the median on. (This is NOT recommended! Try sorting the file on another variable.)

In each of these situations, you can increase your chances of recovering the lost medians by increase the array size when defining the median. (For example, `$(MEAN, MEDIAN(100))` ). Setting the array size too high slows processing and may cause memory problems. (Conversely, if you know you have less than 50 values, you may set the array below 50 to speed processing.)

An example of a table with a lost median follows. Notice the ? in the median row in the columns where the median was lost. Also included is an example of the warning that will appear in your list file when the median is lost. The same table is run again with larger array size to allow Mentor to calculate the median.

```
TABLE_SET= {TAB219:
HEADER=:  LOST MEDIANS }
TITLE=:  NUMBER OF PRODUCTS USED }
STUB=:
    1 - 1000
    1001 - 10000
    10001 - 20000
    OVER 20000
    DON'T KNOW
    [STATISTICS_ROW] MEAN
    [STATISTICS_ROW, STATISTICS_DECIMALS=0] MEDIAN }
ROW=:
[22.5#1-1000/1001-10000/10001-20000/20001-99999/DK] &
    $ [MEAN, MEDIAN] [22.5]
}

~EXECUTE
TABLE_SET= TAB219
```



Here is the table Mentor prints:

LOST MEDIANS  
TABLE 219  
NUMBER OF PRODUCTS USED

	GENDER		
	TOTAL	MALE	FEMALE
	-----	-----	-----
TOTAL	100	58	42
	100.0%	100.0%	100.0%
1 - 1000	28	21	7
	28.0%	36.2%	16.7%
1001 - 10000	11	6	5
	11.0%	10.3%	11.9%
10001 - 20000	5	2	3
	5.0%	3.4%	7.1%
OVER 20000	51	26	25
	51.0%	44.8%	59.5%
DON'T KNOW	5	3	2
	5.0%	5.2%	4.8%
MEAN	32913.46	28634.76	38796.68
MEDIAN	?	?	32232

Notice the question marks (?) that print in the median row for both the total column and the MALE column. This means that the program was unable to determine those medians.

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

Also the following message will print once in the list file for every median that is lost:

```
(ERROR #5070) for T219 number buckets is 50, try ${median(501) or [ptile.xx(501]}  
**error** Randomizing the input file so that values are not consecutive may help.  
**error** ~set median_cells= can be used to change the number of  
**error** buckets for the entire run. If this is a common problem then  
**error** this set option may be added to your mentinit file. Be aware  
**error** however that increasing the number of buckets will cause runs  
**error** with medians or pfiles to take a bit longer.
```

The number 50 above is the current number of buckets that were used. The number 501 is just a guess by the program (1 plus 10 times the current setting). You can guarantee the median will be calculated if you set the number of buckets to the number of different categories that have values in the range. You can run a frequency count to determine this number.

And here is the example to recover the lost medians. Notice the medians are now correct for the total column and the MALE column.

```
~DEFINE  
  
TABLE_SET= {TAB220:  
HEADER=:CHANGING THE DEFAULT NUMBER OF BUCKETS TO RECOVER  
LOST MEDIANS }  
ROW=: [22.5#1-1000/1001-10000/10001-20000/20001-99999/DK] &  
$ [MEAN, MEDIAN(200)] [22.5]  
}  
  
~EXECUTE  
TABLE_SET= TAB220
```



Here is the printed table. Only the median row is printed for this table as the rest of the table would appear the same as Table 219.

CHANGING THE DEFAULT NUMBER OF BUCKETS TO RECOVER LOST  
MEDIAN

TABLE 220

NUMBER OF PRODUCTS USED

	GENDER		
	<----->		
	TOTAL	MALE	FEMALE
	-----	----	-----
TOTAL	100	58	42
	100.0%	100.0%	100.0%
MEDIAN	23849	12035	32232

## PERCENTILES

Percentiles are values that exist in the data such that a certain percentage of data is below that value. A median is equivalent to the 50th percentile, since 50 percent of the values in the range are below it. Other percentiles that are often used are the 25th, 75th, and 90th percentiles. You define a percentile as follows:

\$(PERCENTILE=.NN), where NN is the percentile (percentage) you are looking for. Following is a sample table showing how to produce multiple percentiles on a single table.

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
~DEFINE
```

```
TABLE_SET= {TAB221:
```

```
HEADER=: PRODUCING PERCENTILES }
```

```
TITLE=: NUMBER OF PRODUCTS USED IN THE LAST MONTH }
```

```
STUB=:
```

```
    1 - 25
```

```
    26 - 50
```

```
    51 - 75
```

```
    OVER 75
```

```
    DON'T KNOW
```

```
[[STATISTICS_ROW,STATISTICS_DECIMALS=0] MINIMUM
```

```
[[STATISTICS_ROW,STATISTICS_DECIMALS=0] 25TH PERCENTILE
```

```
[STATISTICS_ROW,STATISTICS_DECIMALS=0] 50TH PERCENTILE/MEDIAN
```

```
[STATISTICS_ROW,STATISTICS_DECIMALS=0] 75TH PERCENTILE
```

```
[STATISTICS_ROW,STATISTICS_DECIMALS=0] MAXIMUM }
```

```
ROW=: [25.2#1-25/26-50/51-75/76-99/" " ] &
```

```
$ [MINIMUM, PERCENTILE=.25, MEDIAN, PERCENTILE=.75, gMAXIMUM]
```

```
[25.2]
```

```
}
```

```
~EXECUTE
```

```
TABLE_SET= TAB221
```

Here is the printed table.

PRODUCING PERCENTILES

TABLE 221

NUMBER OF PRODUCTS USED IN THE LAST MONTH

	GENDER		
	TOTAL	MALE	FEMALE
TOTAL	100	58	42
	100.0%	100.0%	100.0%
1 - 25	18	11	7
	18.0%	19.0%	16.7%
26 - 50	31	16	15
	31.0%	27.6%	35.7%
51 - 75	30	20	10
	30.0%	34.5%	23.8%
OVER 75	16	8	8
	16.0%	13.8%	19.0%
DON'T KNOW	5	3	2
	5.0%	5.2%	4.8%
MINIMUM	1	1	3
25TH PERCENTILE	29	28	29
50TH PERCENTILE/ MEDIAN	49	51	45
75TH PERCENTILE	66	66	65
MAXIMUM	99	97	99

**NOTE:** Percentiles may be “lost” just as medians are and you can increase the array size in the same way as medians to recover those that are “lost”.

### 6.2.6 Mean Summary Tables

Mean summary tables are usually produced when you have a series of rating scales or a series of numeric type responses and you want to compare the means. Creating a mean summary table with no recoding is quite simple and can be done either by using the WITH joiner between each data location or by putting multiple data locations in the same set of brackets. If recoding of the mean is required then you will need to combine the recoding with either of the above approaches.

The first and most important thing to understand when doing a mean summary table is that once you have specified \$[MEAN] all the categories after that will produce a mean until there is another \$[ ] command (See “5.2 Axis Commands/Cross-Case Operations” and *Appendix B: TILDE COMMANDS, ~DEFINE AXIS=* for more explanation of \$[ ]).

#### RATING SCALES WITH NO RECODING

A typical example of when a mean summary table would be used is when you have a series of rating scales. There are a number of different ways to construct the row variable definition for this type of table, but the easiest is to tell the program you are now creating means by using \$[MEAN] and then following that with each location you wish to do the mean on in a single set of brackets. You can also simplify this if the locations are consecutive by using the ellipses.

In the following example the gender of the respondent has been stored in column 5 and the rating of 5 different brands has been stored in columns 6 through 10.

**NOTE:** The following set of commands define a standard front end for the next set of examples.

```

>PRINT_FILE MNSUM
~INPUT MNSUM
~SET AUTOMATIC_TABLES,DROP_LOCAL_EDIT,
DROP_BASE,BEGIN_TABLE_NAME=T251

~DEFINE

STUB= STUBTOP1:
[SUPPRESS] TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= {BAN1:
EDIT=:
COLUMN_WIDTH=8,STUB_WIDTH=20,-COLUMN_TNA,STATISTICS_DEC
IMALS=2 }
STUB_PREFACE= STUBTOP1
BANNER=:
|
|                                GENDER
|                                <----->
|    TOTAL    MALE    FEMALE
|    -----  - - -  - - - - - }
COLUMN=:    TOTAL WITH [5^1/2]
}

~EXECUTE
TABLE_SET= BAN1

```

These commands are specific to this example:

```
~DEFINE
```

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
TABLE_SET= {TAB251:
HEADER=: STRAIGHT MEAN SUMMARY TABLE WITH NO RECODING}
TITLE=: MEAN OVERALL RATING FOR BRANDS}
LOCAL_EDIT=: STUB_EXTRA=[STATISTICS_ROW] }
STUB=:
    BRAND A
    BRAND B
    BRAND C
    BRAND D
    BRAND E}
ROW=: $ [MEAN] [06, ..., 10]
}
```

Here are some alternate ways to write the row variable:

```
ROW251A: $[MEAN] [6] $[MEAN] [7] $[MEAN] [8] $[MEAN] [9] $[MEAN] [10]
ROW251B: $[MEAN] [6] WITH [7] WITH [8] WITH [9] WITH [10]
ROW251C: $[MEAN] [6,7,8,9,10]
```

```
~EXECUTE
```

```
TABLE_SET= TAB251
```





Here is the table Mentor prints:

STRAIGHT MEAN SUMMARY TABLE WITH NO RECODING  
TABLE 251  
MEAN OVERALL RATING FOR BRANDS

	GENDER		
	TOTAL	MALE	FEMALE
	-----	----	-----
BRAND A	2.87	2.67	3.11
BRAND B	2.93	3.07	2.76
BRAND C	3.04	3.00	3.09
BRAND D	2.94	3.09	2.76
BRAND E	2.73	2.93	2.49

### RATING SCALES WITH RECODING NEEDED

If the rating scales were coded so that some recoding is needed for the mean to be calculated you can combine the recoding process discussed in “6.2.1 Means on Rating Scales Using the Variable Definition” with the processes discussed above. There are also some shortcuts you can use here to keep from having to explicitly define each mean. The four types of possible recoding as previously discussed are (A) Exclude a numeric Don't Know code, (B) Reverse the scale, (C) Both A and B, and (D) Recode the zero punch as 10.

### RATING SCALES WITH THE DON'T KNOW CODED AS A NUMERIC

If the Don't Know response was coded as a numeric value you will need to exclude it from the calculation. If the same code was used for all the rating scales then you

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

can just use the \*RANGES modifier to exclude it from each mean. If different codes were used for different scales then you will have to exclude them individually (See ROW252A below). The example below assumes the same things as the example for TAB251, except that now the Don't Know response was coded as a 5 for each rating scale. The TITLE, STUB, and LOCAL\_EDIT are omitted here since they would be exactly the same as TAB251.

```
~DEFINE
```

```
TABLE_SET= {TAB252:  
HEADER=: MEAN SUMMARY TABLE WITH DON'T KNOW CODED AS A  
NUMERIC}  
ROW=: $[MEAN] [6,...,10*RANGES=1-4]  
}
```

Here are some alternate ways to write the row variable:

```
ROW252A: $[MEAN] [6*RANGES=1-4] WITH [7*RANGES=1-4]  
WITH &  
[8*RANGES=1-4] WITH [9*RANGES=1-4] WITH  
[10*RANGES=1-4]  
ROW252B: $[MEAN] ([6] WHEN [6^1-4]) WITH ([7] WHEN  
[7^1-4]) WITH &  
([8] WHEN [8^1-4]) WITH ([9] WHEN [9^1-4]) WITH  
([10] WHEN [10^1-4])  
ROW252C: $[MEAN] [6,...,10] INTERSECT [6,...,10^1-4]
```

```
~EXECUTE
```

```
TABLE_SET= TAB252
```

The printed table will look fundamentally the same as Table 251 above.



## RATING SCALES WITH THE SCALE REVERSED

If the scale needs to be reversed you can use the subtraction method or the SUBSCRIPT function method (see “6.2.1 Means on Rating Scales Using the Variable Definition” for more information about these methods). If you use the subtraction method you can use scaler/vector arithmetic to significantly reduce the specification writing needed.

Scaler/vector arithmetic works in the following manner. A scaler is a single numeric value (number, data location, etc.). If you join a scaler to a vector with any of the arithmetic joiners (+, -, \*, /), then that arithmetic operation will be performed on every category in the vector. This allows you to multiply a series of fields by the same number or to reverse the scale on a series of fields. The example below assumes the same things as TAB251 except that the 5 point scale needs to be reversed. The TITLE, STUB, and LOCAL\_EDIT have been omitted since they are exactly the same as TAB251.

```
~DEFINE
```

```
TABLE_SET= {TAB253 :
HEADER=: MEAN SUMMARY TABLE WITH THE SCALE REVERSED}
ROW=: $[MEAN] 5 - [6, ..., 10]
}
```

Here are some alternate ways to write the row variable:

```
ROW253A: $[MEAN] (5 - [6]) WITH (5 - [7]) WITH (5 - [8])
WITH &
(5 - [9]) WITH (5 - [10])
ROW253B: $[MEAN] &
>REPEAT $A=06, ..., 10; STRIP="WITH &"
```

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
                SUBSCRIPT( [$A^4//1] ) WITH &  
>ENDREPEAT  
  
~EXECUTE  
TABLE_SET= TAB253
```

The printed table will look fundamentally the same as Table 251 above.

### **RATING SCALES WITH THE SCALE REVERSED AND DON'T KNOW CODED AS A NUMERIC**

If the scale needs to be reversed and there is a numeric Don't Know code you can either combine methods from above or use the SUBSCRIPT function. In all cases where you use the SUBSCRIPT function on a mean summary table you need to individually define each mean and join them using the WITH joiner. This example assumes the same things as TAB251, except that the Don't Know was coded as a 5, and the 4 point scale needs to be reversed. The TITLE, STUB, and LOCAL\_EDIT have been omitted since they are exactly the same as TAB251.

```
~DEFINE  
  
TABLE_SET= {TAB254:  
HEADER= :  
MEAN SUMMARY TABLE WITH THE SCALE REVERSED AND DON'T  
KNOW CODED AS NUMERIC}  
ROW=: $ [MEAN] 5 - [06, ..., 10*RANGES=1-4]  
}
```

Here are some alternate ways to write the row variable:

```
ROW254B: $ [MEAN] &
```

```
>REPEAT $A=06, . . . , 10; STRIP="WITH &"
      SUBSCRIPT ([$A^4//1]) WITH &
>ENDREPEAT
```

```
~EXECUTE
TABLE_SET= TAB254
```

The printed table will look fundamentally the same as table 251 above.

### RATING SCALES WITH 10 CODED AS A ZERO (0)

If you want to do a mean summary table on a series of ratings where the scale goes from 1 to 10 and 10 was coded as a 0, then you will want to use the SUBSCRIPT function to recode them and use the WITH joiner to combine all the different ratings. The TITLE, STUB, and LOCAL\_EDIT have been omitted since they are exactly the same as TAB251. The TITLE, STUB, and LOCAL\_EDIT have been omitted since they are exactly the same as TAB251.

```
~DEFINE

TABLE_SET= {TAB255:
HEADER=: MEAN SUMMARY TABLE WITH 10 CODED AS 0}
ROW=: $ [MEAN] &
>REPEAT $A=06, . . . , 10; STRIP="WITH &"
      SUBSCRIPT ([$A^1//0]) WITH &
>END_REPEAT
}

~EXECUTE
TABLE_SET= TAB255
```

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

The printed table will look fundamentally the same as Table 251 above.

#### RANGE VARIABLES

To produce a mean summary table on range type variables you need to define each mean using the SELECT\_VALUE function and connect them using the WITH joiner (See “6.2.2 Means For Range Type Variables” for an explanation of how to use the SELECT\_VALUE function to define each mean). The values chosen for the SELECT\_VALUE should be the midpoint of each range in the variable. The example below assumes the ranges are 1 to 10, 11 to 20, 21 to 30, and 31 to 50. The TITLE, STUB, and LOCAL\_EDIT have been omitted since they are exactly the same as TAB251.

```
~DEFINE
```

```
TABLE_SET= {TAB256:
```

```
HEADER=: MEAN SUMMARY TABLES OR RANGE VARIABLES}
```

```
ROW=: $ [MEAN] &
```

```
>REPEAT $A=06, ..., 10; STRIP="WITH &"
```

```
SELECT_VALUE ( [ $A^1 / 4 ] , VALUES ( 5.5 , 15.5 , 25.5 , 40.5 ) ) WITH  
&
```

```
>ENDREPEAT
```

```
}
```

```
~EXECUTE
```

```
TABLE_SET= TAB256
```

The printed table will look fundamentally the same as Table 251 above.



## NUMERIC DATA WITH THE DON'T KNOW CODED AS A NON-NUMERIC

Producing a mean summary table on numeric data consists of combining the production of a mean from numeric data (See “6.2.3 Means For Numeric Data”) with the methods that were discussed for mean summary tables on rating scales. A mean summary table for a mean where the Don't Know was coded as non-numeric and all the numeric values were coded as their existing values requires no recoding and looks very similar to the mean summary on a rating scale without any recoding.

In the example below, assume there are 6 brands and each respondent is asked how many times they have used the brand in the past year. The answers for Brand A through F are stored in consecutive 2 column fields starting with Brand A in columns 11 and 12. The Don't Know was coded as DK.

```
~DEFINE

TABLE_SET= {TAB257:
HEADER=:
MEAN SUMMARY TABLE OF NUMERIC FIELDS WITH DON'T KNOW
CODED AS NON-NUMERIC}
TITLE=: AVERAGE NUMBER OF TIMES USED PRODUCT IN LAST
YEAR}
LOCAL_EDIT=: STUB_EXTRA=[STATISTICS_ROW] }
STUB=:
    BRAND A
    BRAND B
    BRAND C
    BRAND D
    BRAND E
    BRAND F}
ROW=: $ [MEAN] [11.2, . . . , 21]
```

ADVANCED TABLES

6.2 SUMMARY STATISTICS (MEANS)

}

~EXECUTE

TABLE\_SET= TAB257

Here is the table Mentor prints:

MEAN SUMMARY TABLE OF NUMERIC FIELDS WITH DON'T KNOW CODED AS  
NON-NUMERIC

TABLE 257

AVERAGE NUMBER OF TIMES USED PRODUCT IN LAST YEAR

	GENDER		
	<----->		
	TOTAL	MALE	FEMALE
	-----	-----	-----
BRAND A	46.08	46.80	45.18
BRAND B	56.81	62.08	50.39
BRAND C	53.56	56.36	49.97
BRAND D	54.97	53.90	56.24
BRAND E	53.90	54.81	52.83
BRAND F	59.38	60.78	57.68





## NUMERIC DATA WITH THE DON'T KNOW CODED AS NUMERIC

If the Don't Know response is coded as a numeric value such as 99, then you must exclude this from the calculation. The easiest way to do this is to use the \*RANGES modifier. The example below assumes the same things as TAB257, except here the Don't Know was coded as 99 for all the questions. The TITLE, STUB, and LOCAL\_EDIT have been omitted since they are exactly the same as TAB257.

```
~DEFINE

TABLE_SET= {TAB258:
HEADER=:
MEAN SUMMARY TABLE OF NUMERIC FIELDS WITH DON'T KNOW
CODED AS NUMERIC}
ROW=: $ [MEAN] [11.2, . . . , 21*RANGES=0-98]
}

~EXECUTE
TABLE_SET= TAB258
```

The printed table will look fundamentally the same as Table 257 above.

## NUMERIC DATA WITH A NUMERIC VALUE CODED AS A NON-NUMERIC CODE

When producing a mean summary table on a set of numeric questions where a non-numeric code like "XX", was used to mean a numeric value like 100, you must use one of the methods discussed in "6.2.2 Means For Range Type Variables" to recode the mean and then use the WITH joiner to combine them all. The TITLE, STUB, and LOCAL\_EDIT have been omitted since they are exactly the same as TAB257.

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
~DEFINE
```

```
TABLE_SET= {TAB259:  
HEADER=: MEAN SUMMARY TABLE OF NUMERIC FIELDS WITH "XX"  
CODED AS 100}  
ROW=: $ [MEAN] &  
>REPEAT $A=11,13,...,21; STRIP="WITH &"  
      ([$.2] OTHERWISE  
SELECT_VALUE([$.2#"--"],VALUES(100))) WITH &  
>ENDREPEAT  
}
```

Here is an alternate way to write the row variable:

```
ROW258A: $ [MEAN] &  
>REPEAT $A=11,13,...,21; STRIP="WITH &"  
  
SUM([$.2],SELECT_VALUE([$.2#"--"],VALUES(100))) WITH  
&  
>ENDREPEAT  
  
~EXECUTE  
TABLE_SET= TAB259
```

The printed table will look fundamentally the same as Table 257 above.

## USING THE “BY” JOINER

A mean summary table using BY is one in which both the banner and the stub are normal demographic type variables, but the entire table is reporting the mean of some other variable. For instance, you are crossing AGE by SEX, but the entire table is reporting the mean on the overall rating of a product. A given cell tells you the mean for the age group of that sex. To produce this table you need to define the mean for the rating as you normally would and then use the BY joiner to break that mean out by all the age categories. See the example below for the exact syntax.

In the example below, the age question is stored in columns 11 and 12, and the overall rating of product A is in column 6 (no recoding needed). If recoding was needed you would just use whatever recoding method was needed before applying the BY joiner.

```
~DEFINE

TABLE_SET= {TAB260:
HEAD=: USING BY TO CREATE MEAN SUMMARY TABLE}
TITLE=: AGE BY MEAN FOR OVERALL RATING OF PRODUCT A}
LOCAL_EDIT=: STUB_EXTRA=[STATISTICS_ROW] }
STUB=:
    UNDER 18
    18 - 30
    31 - 45
    46 - 60
    OVER 60
    REFUSED
    NO ANSWER}
ROW=: $[MEAN] [6] BY
[11.2#0-17/18-30/31-45/46-60/61-99/"RF"/"--"]
}
```

**ADVANCED TABLES**  
 6.2 SUMMARY STATISTICS (MEANS)

~EXECUTE  
 TABLE\_SET= TAB260

Here is the printed table.

USING BY TO CREATE MEAN SUMMARY TABLE  
 TABLE 260  
 AGE BY MEAN FOR OVERALL RATING OF PRODUCT A

	SEX		
	TOTAL	MALE	FEMALE
UNDER 18	2.96	2.92	3.00
18 - 30	3.17	2.60	3.57
31 - 45	2.33	2.57	2.00
46 - 60	3.00	2.86	3.25
OVER 60	2.77	2.50	3.15
REFUSED	3.50	3.50	3.50
NO ANSWER	2.83	2.00	3.67

## 6.2.7 Means Scattered Throughout The Table

You may want to produce summary statistics (means) interspersed in the table with other frequency type numbers. In order to do this you must specify any time you want to switch from producing frequencies to means and vice-versa. As we have seen throughout this section, to switch from frequencies to means you need to use the keyword \$[MEAN], but to switch from means to frequencies you will need to use the keyword \$[ ]. The empty set of brackets tells the program that you want to go back to the default mode of producing whatever that category returns (usually a frequency, but sometimes a number). A good example of when you need to intersperse means and frequencies in a single table is a summary table where you are printing both the top box and the mean for each of several brands. In order to produce this table you will need to alternate between producing a frequency (top box) and a summary statistic (mean).

The example below is a combination of Table 102 and Table 251 in this chapter.

The following set of commands define a standard front end for the next set of examples.

```
>PRINT_FILE  MENIN
~INPUT  MENIN
~SET  AUTOMATIC_TABLES, DROP_LOCAL_EDIT,
DROP_BASE, BEGIN_TABLE_NAME=T261

~DEFINE

STUB=  STUBTOP1 :
[SUPPRESS]  TOTAL
[SUPPRESS]  NO ANSWER  }

TABLE_SET=  {BAN2 :
```

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
EDIT= :
COLUMN_WIDTH=8, STUB_WIDTH=20, -COLUMN_TNA, STATISTICS_DEC
IMALS=2 }
STUB_PREFACE= STUBTOP1
BANNER= :
|
|                SEX
|                <----->
|  TOTAL      MALE  FEMALE
|  -----   ----  -}
COLUMN=:  TOTAL WITH [5^1/2]
}

~EXECUTE
TABLE_SET= BAN2
```

These commands are exclusive to this example.

```
~DEFINE
```

```
TABLE_SET= {TAB261:
HEADER=: EXAMPLE OF A TABLE WITH MEANS AND FREQUENCIES
INTERMIXED}
TITLE=: TOPBOX AND MEAN SUMMARY TABLE}
TITLE_4=: BASE= RESPONDENTS WHO USED THE BRAND}
STUB=:
>REPEAT $A=A, B, C, D, E
      [COMMENT, UNDERLINE]   BRAND $A
      [VERTICAL_PERCENT=*, SUPPRESS] PERCENT BASE FOR
BRAND $A (DOESN'T PRINT)
      [STUB_INDENT=2, -FREQUENCY] TOPBOX PERCENTAGE
      [STUB_INDENT=2, STATISTICS_ROW] MEAN
```

```

>END_REPEAT
}
ROW=: &
>REPEAT $A=07, . . . , 11; STRIP="$ [] &"
      [$A^1-4/4] $ [MEAN] [$A] $ [] &
>END_REPEAT
}

ROW261: [7^1-4/4] $ [MEAN] [7] $ [] [8^1-4/4] $ [MEAN] [8]
&
      $ [] [9^1-4/4] $ [MEAN] [9] $ [] [10^1-4/4] $ [MEAN]
[10] &
      $ [] [11^1-4/4] $ [MEAN] [11]

~EXECUTE
TABLE_SET= TAB261

```

Here is the printed table.

EXAMPLE OF A TABLE WITH MEANS AND FREQUENCIES INTERMIXED

TABLE 261  
TOPBOX AND MEAN SUMMARY TABLE  
BASE= RESPONDENTS WHO USED THE BRAND

**ADVANCED TABLES**  
 6.2 SUMMARY STATISTICS (MEANS)

	SEX		
	<----->		
	TOTAL	MALE	FEMALE
	----	----	-----
BRAND A			
-----			
TOPBOX PERCENTAGE	28.4%	27.5%	29.3%
MEAN	2.70	2.70	2.71
BRAND B			
-----			
TOPBOX PERCENTAGE	30.8%	29.3%	32.4%
MEAN	2.62	2.68	2.54
BRAND C			
-----			
TOPBOX PERCENTAGE	26.3%	14.6%	38.5%
MEAN	2.66	2.49	2.85
BRAND D			
-----			
TOPBOX PERCENTAGE	22.2%	20.5%	24.3%
MEAN	2.35	2.30	2.41
BRAND E			
-----			
TOPBOX PERCENTAGE	15.8%	16.3%	15.2%
MEAN	2.29	2.28	2.30





## MEAN/FREQUENCY SUMMARY TABLE

Another common type of mean summary table prints the number of respondents who make up each mean value, along with the mean value. This is sometimes referred to as a mean/frequency summary table. If you use the syntax in the next example the number generated in the frequency row will be the number of respondents who have a valid number in that data location. You can combine this with any of the recoding discussed above.

```

~DEFINE
TABLE_SET= TAB262:
HEADER=: EXAMPLE OF A SUMMARY TABLE OF MEANS AND EACH
MEANS FREQUENCY }
TITLE=: MEAN OVERALL RATING FOR BRANDS }
LOCAL_EDIT=: -VERTICAL_PERCENT }
STUB=:

                                | BRAND A - FREQUENCY
[STATISTICS_ROW,SKIP_LINES=0 ] |           MEAN
                                | BRAND B - FREQUENCY
[STATISTICS_ROW,SKIP_LINES=0 ] |           MEAN
                                | BRAND C - FREQUENCY
[STATISTICS_ROW,SKIP_LINES=0 ] |           MEAN
                                | BRAND D - FREQUENCY
[STATISTICS_ROW,SKIP_LINES=0 ] |           MEAN
                                | BRAND E - FREQUENCY
[STATISTICS_ROW,SKIP_LINES=0 ] |           MEAN }
ROW=: $ [FREQUENCY,MEAN] [07, . . . ,11]
}

~EXECUTE
TABLE_SET= TAB262

```

ADVANCED TABLES

6.2 SUMMARY STATISTICS (MEANS)

Here is the printed table:

EXAMPLE OF A SUMMARY TABLE OF MEANS AND EACH MEANS  
FREQUENCY

TABLE 262

MEAN OVERALL RATING FOR BRANDS

	TOTAL	SEX	
		MALE	FEMALE
BRAND A - FREQUENCY	81	40	41
MEAN	2.70	2.70	2.71
BRAND B - FREQUENCY	78	41	37
MEAN	2.62	2.68	2.54
BRAND C - FREQUENCY	80	41	39
MEAN	2.66	2.49	2.85
BRAND D - FREQUENCY	81	44	37
MEAN	2.35	2.30	2.41
BRAND E - FREQUENCY	76	43	33
MEAN	2.29	2.28	2.30

**NOTE:** To print the Mean before the frequency, reverse the order of the keywords MEAN and FREQUENCY inside the \$[ ] and also flip the order of the stub definitions so that the text for the Mean row is first.

## 6.2.8 Summary Statistics with Arithmetic

You may sometimes need to perform some arithmetic operation on the mean before displaying it. This usually entails moving the decimal point to the right or to the left, or adding, subtracting, multiplying or dividing two or more fields. To move the decimal point to the left you can either divide the mean by the appropriate amount (10 or 100 or 1000) or you can use the \*F modifier. When using a data location to define a numeric field you can put the modifier \*F<number> right before the close bracket. The <number> says how many implied decimals to read that field with. In other words if 1028 is stored in columns 11 through 14 and you reference that field as [11.4\*F2], the Mentor program will read that number as 10.28 (2 implied decimal places). To move the decimal place to the right you need to multiply the variable by the appropriate amount. If you wish to perform any arithmetic operation, just use the appropriate symbol (+, -, \*, /, \*\*).

In the following example frequencies, sums, means, and standard deviations are produced for a number of fields. First compare the statistics for the stubs [11.4] and [11.4\*F2] noting that the frequency is the same, but the other numbers are all 100 times smaller. Second, compare the stubs [15.4] and 3\*[15.4] again noting that the frequency is the same, but the other values are all 3 times higher. Then note that the stub [19.4] has only one respondent with a valid answer, so that the standard deviation is missing and the mean is even missing under the males column, because there are no males who have a valid answer.

**NOTE:** In the example below the mean of the stub [11.4] + [15.4] is equal to the mean of the stub [11.4] + the mean of the stub [15.4]. This will only occur when everyone who has a valid answer in one of those fields has a valid answer in both.

~DEFINE

## ADVANCED TABLES

### 6.2 SUMMARY STATISTICS (MEANS)

```
TABLE_SET= {TAB263:
TITLE=:SUMMARY STATISTICS WITH ARITHMETIC }
LOCAL_EDIT=: -VERTICAL_PERCENT }
STUB=:
>REPEAT
$A=" [11.4] ", " [11.4*F2] ", " [15.4] ", "3* [15.4] ", " [19.4] ", " [
11.4] + [15.4] "
      [COMMENT,UNDERLINE] STATS FOR $A
      [STUB_INDENT=2,SKIP_LINES=0] FREQUENCY
      [STUB_INDENT=2,SKIP_LINES=0] SUM
      [STATISTICS_ROW,STUB_INDENT=2,SKIP_LINES=0] MEAN
      [STATISTICS_ROW,STUB_INDENT=2,SKIP_LINES=0] STD
DEV
>END_REPEAT
}
ROW=: $ [FREQUENCY,SUM,MEAN,STD] [11.4] WITH [11.4*F2]
WITH [15.4] WITH &
      (3* [15.4]) WITH [19.4] WITH ([11.4]+[15.4])
}

~EXECUTE
TABLE_SET= TAB263
```



Here is the printed table:

TABLE 263  
SUMMARY STATISTICS WITH ARITHMETIC

	GENDER		
	<----->		
	TOTAL	MALE	FEMALE
	-----	-----	-----
STATS FOR [11.4]			
-----			
FREQUENCY	110	65	45
SUM	307009	174141	132868
MEAN	2790.99	2679.09	2952.62
STD DEV	3210.92	3004.85	3516.00
STATS FOR [11.4*F2]			
-----			
FREQUENCY	110	65	45
SUM	3070	1741	1329
MEAN	27.91	26.79	29.53
STD DEV	32.11	30.05	35.16
STATS FOR [15.4]			
-----			
FREQUENCY	110	65	45
SUM	387726	263648	124078
MEAN	3524.78	4056.12	2757.29
STD DEV	3001.15	3236.99	2460.11

(Table continued on next page)

## ADVANCED TABLES

### 6.3 WEIGHTED TABLES

STATS FOR 3\*[15.4]

-----

FREQUENCY	110	65	45
SUM	1163178	790944	372234
MEAN	10574.35	12168.37	8271.87
STD DEV	9003.44	9710.97	7380.33

STATS FOR [19.4]

-----

FREQUENCY	1	-	1
SUM	945	-	945
MEAN	945.00	?	945.00
STD DEV	?	?	?

STATS FOR [11.4] + [15.4]

-----

FREQUENCY	110	65	45
SUM	694735	437789	256946
MEAN	6315.77	6735.22	5709.91
STD DEV	4384.49	4531.66	4137.25

## 6.3 WEIGHTED TABLES

A weighted table is one in which each case is multiplied by some appropriate factor so that it carries a higher or lower weight in the sample. This is usually done when you want the distribution in the tables across some demographic group to reflect the universal distribution rather than your sample distribution. A typical example would be as follows: You're interviewing in some city and need to contact 110 respondents, but in this city females are much easier to contact than males, so you end up contacting 70 females and only 40 males. You know that in this city half the people are male and half are female, so for purposes of the tables you want to down weight the 70 females so it looks like there are only 55. To do this you would assign a weight of .786 to each ( $.786 \times 70 = 55.02$ ). You also want to up weight the 40 males to become 55. You would assign them a weight of 1.375 ( $40 \times 1.375 = 55$ ).

If the weighted total will be equal to the sample total, then you can calculate the weight by doing the following: take the universe for the group (expressed as a percentage or number) and divide it by the sample (percentage or number).

$$\text{Weight} = \frac{\text{targeted \# (or \%)}}{\text{actual \# (or \%)}} =$$

In our example above, that would give us 55/40 (or 50/36 if using percentages) = 1.375 for the male group.

Sometimes when you weight responses you want to weight the numbers up so that they actually reflect the numbers in the universe. In the example above, suppose the city we were calling had 10,000 residents. Then you would want the 70 females in the sample to look like 5000 (10000 \* 50%) and you would want the 40 males to look like 5000 also. The weights in this case would be 71.43 for females and 125 for males. In this case, use the same formula, but use numbers rather than percents. The target value is now 5000, not 55, for males, and our weight is 5000/40=125.

In order to do weighting, you need to either store this weight value somewhere in the data record or create an expression that holds the value. This is usually done with the SELECT function, although it can be done several other ways also. You can use the file GENWT.SPX located in the Mentor subdirectory to not only calculate the weights for you based on the target percentages, but also store them in each data record for future use.

You can either weight the entire table to affect every cell in the table, or you can weight either the column or row variable to apply different weights (including none) to different categories in the variable. Once you have determined the weights for the table and how the table is to be weighted you can use any of the table elements WEIGHT, COLUMN\_WEIGHT, COLUMN\_SHORT\_WEIGHT, ROW\_WEIGHT, or ROW\_SHORT\_WEIGHT to apply the weight to the table.

The WEIGHT element weights the entire table, while the others allow you to create cells with different weights. The WEIGHT element cannot be used in conjunction with any of the others, but you can do both column weighting and row weighting simultaneously.

**NOTE:**

- Weighted statistics are calculated properly, except for the EDIT options COLUMN\_STD, COLUMN\_SE, COLUMN\_VARIANCE, ROW\_STD, ROW\_SE, and ROW\_VARIANCE.
- When doing weighting, frequency counts may not add back to the total, and percentages may not add up to 100% due to rounding
- If data cases are not assigned a weight because either the weight field has a missing value or the SELECT function does not account for it, you will get the following warning message in the compile:

**(WARN #8873) tables with MISSING table weights.**

If you get this warning it is likely there is an error in the weight definition and the data cases with a missing weight will be dropped from the table.

### 6.3.1 Weighting with Weight Value already Stored in the Data

It is quite simple to weight an entire table if the weight has already been stored in the data. All you need to do is use the table element WEIGHT= and equate it to the location where the weight is stored. If the weight is stored with implied decimals you will need to use the \*F modifier. See “6.2.8 Summary Statistics with Arithmetic” for more information on the \*F modifier.

In the example below it is assumed that a weight value has previously been stored in columns 7-10 of the data file. The pertinent part of this example is the WEIGHT= line inside TABLE\_SET TAB301. This weight value could have come from a generation done in a previous Mentor run. See “6.3.4 Storing the Weight in the Data” to see how to store the weight in the data.



**NOTE:** The following set of commands define a standard front end for the all the examples in this section, except where noted.

```

>PRINT_FILE WGHT
~INPUT WGHT,ALLOW_UPDATE
~SET DROP_LOCAL_EDIT,DROP_BASE,BEGIN_TABLE_NAME=T301

~DEFINE

STUB= STUBTOP1:
WEIGHTED TOTAL
[SUPPRESS] WEIGHTED NO ANSWER }

TABLE_SET= {BAN1:
EDIT=:
COLUMN_WIDTH=8,STUB_WIDTH=20,-COLUMN_TNA,STATISTICS_DECIMALS=
2 }
STUB_PREFACE= STUBTOP1
BANNER=:
|
|                GENDER
|                <----->
|   TOTAL      MALE   FEMALE
|   -----   ----   -}
COLUMN=:   TOTAL WITH [5^1/2]
}

TABLE_SET= {TAB301:
TITLE=: OVERALL RATING OF PRODUCT A}
STUB=:
        EXCELLENT   (4)
        GOOD        (3)
        FAIR        (2)
        POOR        (1)
        DON'T KNOW
        [STATISTICS_ROW] MEAN }
ROW=: [6^4//1/5] $[MEAN] [6*RANGES=1-4]

```

## ADVANCED TABLES

### 6.3 WEIGHTED TABLES

```
}
```

```
~EXECUTE
```

```
TABLE_SET= BAN1
```

```
TABLE_SET= ROW1
```

And here is our example:

```
~DEFINE
```

```
TABLE_SET= {TAB301:
```

```
HEADER=: WEIGHTED TABLE USING PREVIOUSLY STORED  
WEIGHT}
```

```
WEIGHT=: [7.4]
```

```
STORE_TABLES=*
```

```
}
```

```
~EXECUTE
```

```
TABLE_SET= TAB301
```

Here is the table that is printed.

WEIGHTED TABLE USING PREVIOUSLY STORED WEIGHT  
TABLE 301  
OVERALL RATING OF PRODUCT A

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
WEIGHTED TOTAL		100	55	45
		100.0%	100.0%	100.0%
EXCELLENT	(4)	17	6	11
		17.0%	10.4%	25.2%
GOOD	(3)	20	12	8
		20.1%	22.0%	17.8%
FAIR	(2)	31	20	11
		31.3%	36.4%	25.0%
POOR	(1)	14	9	5
		13.8%	16.1%	10.9%
DON'T KNOW		18	8	9
		17.8%	15.1%	21.1%
MEAN		2.49	2.31	2.73

Notice that ~SET AUTOMATIC\_TABLES was not specified. This option looks for a ROW= to trigger the making of a table. In this example, we wanted to specify the ROW= separately, so we left off the AUTOMATIC\_TABLES and put a STORE\_TABLES=\* in the TABLE\_SET.

Also, notice in the above table that the DONT KNOW row has a value of 18 for the Total column and values of 8 and 9 for the Male and Female columns. Since the data is weighted the actual values in those cells are 17.8, 8.4, and 9.4 respectively. Each of these values round up to produce numbers that do not seem to add up.

### 6.3.2 Weighting using the SELECT Function

If the weight has not been previously stored in the data you can directly create a variable that contains the weights by using the SELECT\_VALUE function. In the following example the respondent's city is stored in column 11 which is the variable being used to assign the weights. City 1 will have a weight factor of .86, City 2 a weight of 0.66, City 3 a weight of 1.39, and City 4 a weight of 1.67 (See “9.3.2 Functions”, *Number Returning Functions* for detailed information on the SELECT\_VALUE function). The only difference between this example and the previous one is the WEIGHT= table element.

```
~DEFINE

TABLE_SET= {TAB302:
HEADER=: WEIGHTED TABLE USING THE SELECT FUNCTION}
WEIGHT=:
SELECT_VALUE ( [11^1//4] , VALUES ( .86 , .66 , 1.39 , 1.67 ) )
STORE_TABLES=*
}
```

The printed table will look fundamentally the same as Table 301 above.

### 6.3.3 Printing Both a Weighted and an Unweighted Total Row

You may want to print both a weighted and unweighted total row, so that you can easily tell both what the percentage base was for a particular column and the actual

number of respondents in that column. The easiest way to do this is to use the SET option UNWEIGHTED\_TOP in conjunction with a special STUB\_PREFACE. The UNWEIGHTED\_TOP option causes the program to create two additional summary rows at the top of the table, the unweighted total and the unweighted no answer.

The example below is the same as that for Table 301, except for the SET option and the different STUB\_PREFACE. In the STUB\_PREFACE notice that the vertical percentage has been turned off on the unweighted total because it makes no sense on this row.

```
~DEFINE
```

```
STUB= STUBTOP2:
```

```
[-VERTICAL_PERCENT] UNWEIGHTED TOTAL
```

```
[SUPPRESS] UNWEIGHTED NO ANSWER
```

```
WEIGHTED TOTAL
```

```
[SUPPRESS] WEIGHTED NO ANSWER }
```

```
TABLE_SET= {TAB303:
```

```
HEADER=: WEIGHTED TABLE WITH UNWEIGHTED TOTAL ROW }
```

```
SET UNWEIGHTED_TOP
```

```
STUB_PREFACE= STUBTOP2
```

```
STORE_TABLES=*
```

```
}
```

```
~EXECUTE
```

```
TABLE_SET= TAB303
```

ADVANCED TABLES

6.3 WEIGHTED TABLES

Here is the table Mentor prints:

WEIGHTED TABLE WITH UNWEIGHTED TOTAL ROW

TABLE 303

OVERALL RATING OF PRODUCT A

	TOTAL	GENDER	
		MALE	FEMALE
	-----	-----	-----
UNWEIGHTED TOTAL	100	56	44
WEIGHTED TOTAL	100	55	45
	100.0%	100.0%	100.0%
EXCELLENT (4)	17	6	11
	17.0%	10.4%	25.2%
GOOD (3)	20	12	8
	20.1%	22.0%	17.8%
FAIR (2)	31	20	11
	31.3%	36.4%	25.0%
POOR (1)	14	9	5
	13.8%	16.1%	10.9%
DON'T KNOW	18	8	9
	17.8%	15.1%	21.1%
MEAN	2.49	2.31	2.73

If you wish to print the Unweighted Any Response row in addition to, or instead of the Unweighted Total row, you need to use the keyword PRINT\_ROW=UAR in your STUB\_PREFACE. If you only wanted to print the Unweighted and Weighted Any Response Rows, the STUB\_PREFACE would look like the following:

```
STUB_PREFACE= :
[SUPPRESS] UNWEIGHTED TOTAL
[SUPPRESS] UNWEIGHTED NO ANSWER
[PRINT_ROW=UAR, -VERTICAL_PERCENT] UNWEIGHTED ANY
RESPONSE
[SUPPRESS] WEIGHTED TOTAL
[SUPPRESS] WEIGHTED NO ANSWER
[PRINT_ROW=AR] WEIGHTED ANY RESPONSE }
```

### 6.3.4 Storing the Weight in the Data

If the weight variable is not already stored in the data, you may want to store it there for ease of future reference. Once the weights are stored in the data it is a very easy process to weight the tables. You may want to do this if you are going to be running multiple runs all with the same weighting scheme and you don't want the program to have to recalculate the weights each time.

To store the weights in the data you will need to define and execute a procedure. Inside the procedure you will want to use the SELECT\_VALUE function to store the various weights in the data. You can either explicitly put the decimal point in the data or you can store the weight as a whole number and then read the field as having implied decimals using the \*F modifier. In the example below the MODIFY command is storing the values without a decimal point, while the PRINT\_DATA command is storing it with a decimal point. See TAB304 for how to reference the data from the MODIFY command and see TAB305 for the data from the PRINT\_DATA command. On an actual table run you would use one or the other.

## ADVANCED TABLES

### 6.3 WEIGHTED TABLES

```
~DEFINE PROCEDURE= {GENWT1:  
MODIFY [51.3] =  
SELECT_VALUE ( [11^1//4] , VALUES (86,66,139,167) )  
PRINT_DATA [61.4] "\4.2F"  
SELECT_VALUE ( [11^1//4] , VALUES (.86, .66,1.39,1.67) )  
}  
~EXECUTE  
PROCEDURE= GENWT1
```

The weight statement will look like this if you used the MODIFY command above:

```
~DEFINE  
TABLE_SET= {TAB304:  
HEADER=: WEIGHTED TABLE USING A WEIGHT STORED WITH AN  
IMPLIED DECIMAL POINT} WEIGHT=: [51.3*F2]  
STORE_TABLES=* }  
~EXECUTE  
TABLE_SET= TAB304
```

The printed table will look basically the same as Table 301 above.

The weight statement will look like this if you used the PRINT\_DATA command above:

```
~DEFINE  
TABLE_SET= {TAB305:  
HEADER=: WEIGHTED TABLE USING A WEIGHT STORED WITH AN  
EXPLICIT DECIMAL POINT} WEIGHT=: [61.4]  
STORE_TABLES=* }  
~EXECUTE  
TABLE_SET= TAB305
```

The printed table will look fundamentally the same as Table 301 above.



### 6.3.5 Assigning Different Weights to Different Banner Points

Sometimes when producing weighted tables, you will need to apply different weighting factors to different banner points. For example, you might want some of the banner points weighted by the sex variable and others weighted by the city variable. An even more common occurrence is that you want some of the banner points weighted and others unweighted. An unweighted column is just a column where the weight for everyone in that column is one.

To produce a table with different weights across the banner, you will need to use the `COLUMN_WEIGHT` table element to set the weights. This element needs to have the same number of categories as the number of categories in your banner definition, if you do not, you will get an appropriate error message. Since most banners have a fair amount of banner points, it is usually a good idea to predefine any weight variables you need to use and then just reference them by name on the `COLUMN_WEIGHT` statement.

In the example below the banner is broken into two parts for each of the original banner points, one based on one weight variable, and the second on a different weight variable. Since the column variable has six categories in it, then the `COLUMN_WEIGHT` variable must also have six categories in it. Notice in Table 306 how the numbers and frequencies change between the two columns with the different weights.

**NOTE:** All system-generated columns will be unweighted when `COLUMN_WEIGHT` or `COLUMN_SHORT_WEIGHT` are specified.

## ADVANCED TABLES

### 6.3 WEIGHTED TABLES

~DEFINE

WGHT1: SELECT ([11<sup>1</sup>/4], VALUES (.86, .66, 1.39, 1.67))

WGHT2: SELECT ([12<sup>1</sup>/4], VALUES (.89, .59, 1.92, 1.47))

TABLE\_SET= {BAN2:

EDIT=:

COLUMN\_WIDTH=8, STUB\_WIDTH=20, -COLUMN\_TNA, STATISTICS\_DECIMALS=2 }

STUB\_PREFACE= STUBTOP1

BANNER=:

```
|
|                                     GENDER
|                                     <----->
|   WGHT1  WGHT2  WGHT1  WGHT2  WHGT1  WGHT2
|   TOTAL   TOTAL   MALE   MALE   FEMALE FEMALE
|   -----  - - - - -  - - - - -  - - - - -  - - - - -  - - - - - }
```

COLUMN\_WEIGHT=: WGHT1 WITH WGHT2 WITH WGHT1 WITH WGHT2 WITH WGHT1 WITH WGHT2

COLUMN=: TOTAL WITH TOTAL WITH [5<sup>1</sup>/1/2/2]

}

TABLE\_SET= {TAB306:

HEADER=: WEIGHTED TABLE USING DIFFERENT WEIGHTS ON DIFFERENT BANNER POINTS}

STORE\_TABLES=\*

}

~EXECUTE

TABLE\_SET= BAN2

TABLE\_SET= TAB306

Here is the table that is printed.

WEIGHTED TABLE USING DIFFERENT WEIGHTS ON DIFFERENT BANNER POINTS  
TABLE 306  
OVERALL RATING OF PRODUCT A

		GENDER					
		<=====>					
		WGHT1	WGHT2	WGHT1	WGHT2	WGHT1	WGHT2
		TOTAL	TOTAL	MALE	MALE	FEMALE	FEMALE
		-----	-----	-----	-----	-----	-----
WEIGHTED TOTAL		100	100	55	58	45	41
		100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
EXCELLENT	(4)	17	23	6	9	11	15
		17.0%	23.5%	10.4%	14.7%	25.2%	35.9%
GOOD	(3)	20	18	12	11	8	6
		20.1%	17.9%	22.0%	19.5%	17.8%	15.7%
FAIR	(2)	31	28	20	20	11	8
		31.3%	28.4%	36.4%	34.3%	25.0%	20.3%
POOR	(1)	14	14	9	9	5	4
		13.8%	13.8%	16.1%	16.0%	10.9%	10.7%
DON'T KNOW		18	16	8	9	9	7
		17.8%	16.3%	15.1%	15.5%	21.1%	17.5%
MEAN		2.49	2.61	2.31	2.39	2.73	2.93

### 6.3.6 Printing Both a Weighted and an Unweighted Total Column

If you need to print both a weighted and an unweighted total column you can easily do this by using the COLUMN\_SHORT\_WEIGHT table element. This option is very similar to the COLUMN\_WEIGHT option in that it allows you to assign different weights to different columns in the banner, but it also allows you to define more categories in your column variable than in your weight variable and it just uses the last weight variable for all the additional categories. This means if your first banner point is an unweighted total followed by the rest of the banner points weighted, you can just define a two category weight variable. In order to create an unweighted category in a weight variable you need to assign it the keyword TOTAL. So in general, your weight variable will look like TOTAL WITH WEIGHTNAME. Notice that the column definition starts with TOTAL WITH TOTAL to create both the unweighted and weighted total columns.

```
~DEFINE
```

```
TABLE_SET= {BAN3:
EDIT=:  COLUMN_WIDTH=8,STUB_WIDTH=20,-COLUMN_TNA,
STATISTICS_DECIMALS=2 }
BANNER=:
|
|                               GENDER
|  UNWGHT    WGHT  <----->
|   TOTAL    TOTAL    MALE  FEMALE
|  -----  -----  ----  -----}
COLUMN=:  TOTAL WITH TOTAL WITH [5^1/2]
COLUMN_SHORT_WEIGHT=: TOTAL WITH WGHT1
}
```

```
TABLE_SET= {TAB307:
HEADER=:  WEIGHTED TABLE WITH UNWEIGHTED AND WEIGHTED TOTAL
COLUMNS}
STORE_TABLES=*
}
```

(Continued on next page)



```
~EXECUTE  
TABLE_SET= BAN3  
TABLE_SET= TAB307
```

ADVANCED TABLES

6.3 WEIGHTED TABLES

Here is the table that is printed.

WEIGHTED TABLE WITH UNWEIGHTED AND WEIGHTED TOTAL COLUMNS

TABLE 307

OVERALL RATING OF PRODUCT A

	UNWGHT TOTAL -----	WGHT TOTAL -----	GENDER <----->	
			MALE ----	FEMALE -----
WEIGHTED TOTAL	100 100.0%	100 100.0%	55 100.0%	45 100.0%
EXCELLENT (4)	21 21.0%	17 17.0%	6 10.4%	11 25.2%
GOOD (3)	19 19.0%	20 20.1%	12 22.0%	8 17.8%
FAIR (2)	30 30.0%	31 31.3%	20 36.4%	11 25.0%
POOR (1)	14 14.0%	14 13.8%	9 16.1%	5 10.9%
DON'T KNOW	16 16.0%	18 17.8%	8 15.1%	9 21.1%
MEAN	2.56	2.49	2.31	2.73

### 6.3.7 Assigning Different Weights To Different Rows

The process for weighting a table by the row variable is exactly the same as for weighting by the column variable, except you will want to use the table elements *ROW\_WEIGHT* and *ROW\_SHORT\_WEIGHT* instead of *COLUMN\_WEIGHT* and *COLUMN\_SHORT\_WEIGHT*. See “6.3.5 Assigning Different Weights to Different Banner Points” and “6.3.6 Printing Both a Weighted and an Unweighted Total Column” for the process of weighting by the column variable.

If you want to create a row in the middle of the table that is unweighted, you can also do this by using the `$_[RAW_COUNT]` keyword. This will cause all categories defined after it to be unweighted. In the following example, suppose you wanted to create both a weighted and unweighted Don't Know row on a table. You would want to define the weighted Don't Know row as you normally do and then define it again, after you have specified the `$_[RAW_COUNT]` keyword.

```
~DEFINE

TABLE_SET= {TAB308:
TITLE=: OVERALL RATING OF PRODUCT A}
STUB=:
        EXCELLENT   (4)
        GOOD        (3)
        FAIR        (2)
        POOR        (1)
        DON'T KNOW
        [-VERTICAL_PERCENT] UNWEIGHTED DK
        [STATISTICS_ROW] MEAN }
ROW=: [6^4//1/5] $_[RAW_COUNT] [6^5] $_[MEAN] [6*RANGES=1-4]
}

TABLE_SET= {TAB308:
HEADER=: USING THE RAWCOUNT OPTION TO PRODUCE WEIGHTED
AND UNWEIGHTED ROWS }

(Continued on next page)
```

## ADVANCED TABLES

### 6.3 WEIGHTED TABLES

```
STORE_TABLES=* }
```

```
~EXECUTE
```

```
TABLE_SET= BAN1
```

```
TABLE_SET= TAB308
```





Here is the table that is printed.

USING THE RAWCOUNT OPTION TO PRODUCE WEIGHTED AND UNWEIGHTED ROWS

TABLE 308

OVERALL RATING OF PRODUCT A

		GENDER		
		<----->		
		TOTAL	MALE	FEMALE
		-----	----	-----
WEIGHTED TOTAL		100	55	45
		100.0%	100.0%	100.0%
EXCELLENT	(4)	21	6	11
		21.0%	10.4%	25.2%
GOOD	(3)	19	12	8
		19.0%	22.0%	17.8%
FAIR	(2)	30	20	11
		30.0%	36.4%	25.0%
POOR	(1)	14	9	5
		14.0%	16.1%	10.9%
DON'T KNOW		16	8	9
		16.0%	15.1%	21.1%
UNWEIGHTED DK		16	8	8
MEAN		2.56	2.31	2.73

### 6.3.8 WEIGHTING USING MULTIPLE FACTORS

Sometimes, when tables are weighted, multiple factors are used. For instance, you might want to weight the table by both GENDER and AGE variables. There are three different approaches to accomplishing this.

The first and easiest approach is to just multiply the two weights together. The problem with doing this is that it does not take into account how the different weights will affect each other. The second approach is actually to assign weights to each cross section of the two variables. For instance, in the above example, you would have to assign a weight for Males under the age of 25, Females under the age of 25, and so on. If you do not know the cross-sectional universe percentages, you can estimate them by multiplying the two target percentages that make up the cross-section. The last approach is to use a procedure called “sample balancing.” You need to use this if you are weighting by enough factors such that a respondent does not exist in each and every cross-section.

### 6.4 SUMMARY TABLES (MARKET SHARE)

A table of sums (sometimes called a market share table) is usually created when you have a series of numeric type questions about how many purchases have been made of particular brands in a specified time period. The table you create needs to show what percentage a particular brand's purchases constitute of all purchases.

In order to create this table you must realize that if you just specify a data location [col.wid] and nothing else, the program will calculate the sum of all the valid numbers in that field. To create the overall sum though, you need to add all the individual fields together making sure that you are doing any recoding if necessary (9=DK or -- = 100). You may want to use the SUM function, the ++ joiner, or an exclamation point (!) after each open bracket so that missing values are counted as zero for purposes of the total sum. After you have created the sum you need to access each of the appropriate fields and join them using the WITH joiner or use multiple locations inside the same set of brackets. You will also want to make sure that you suppress the printing of the system total row (usually with a STUB\_PREFACE) and that if you are printing the vertical percentage you change the percentage base to the total sum row.

```

>PRINT_FILE SUMS
~INPUT SUMS
~SET AUTOMATIC_TABLES,DROP_LOCAL_EDIT,
DROP_BASE BEGIN_TABLE_NAME=T401

~DEFINE

STUB= STUBTOP1:
[SUPPRESS] TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= {BAN1:
EDIT=: COLUMN_WIDTH=8,STUB_WIDTH=20,-COLUMN_TNA,
STATISTICS_DECIMALS=2 }
STUB_PREFACE= STUBTOP1
BANNER=:
|                GENDER
|                <----->
|  TOTAL      MALE  FEMALE
|  -----   ----  -}
COLUMN=:  TOTAL WITH [5^1/2]
}

~EXECUTE
TABLE_SET= BAN1

~DEFINE

TABLE_SET= {TAB401:
HEADER=: SUMS TABLE OR MARKET SHARE}
TITLE=: TOTAL GALLONS USED}
LOCAL_EDIT=: VERTICAL_PERCENT=1 }
STUB=:
      TOTAL GALLONS
  
```

## ADVANCED TABLES

### 6.4 SUMMARY TABLES (MARKET SHARE)

```
        BRAND A
        BRAND B
        BRAND C
        BRAND D
        BRAND E
        BRAND F
        BRAND G
        OTHER}
ROW=: SUM([11.4,15,...,39]) WITH [11.4,15,...,39]
}
```

Here are some alternate ways to write the row variable:

```
ROW401A: ([!11.4] + [!15.4] + [!19.4] + [!23.4] + [!27.4] + &
          [!31.4] + [!35.4] + [!39.4]) WITH [11.4,15,...,39]
ROW401B: ([11.4] ++ [15.4] ++ [19.4] ++ [23.4] ++ [27.4] ++ &
          [31.4] ++ [35.4] ++ [39.4]) WITH [11.4,15,...,39]
```

```
~EXECUTE
TABLE_SET= TAB401
```

Here is the table that is printed.

```
SUMS TABLE OR MARKET SHARE
TABLE 401
TOTAL GALLONS USED
```

	SEX	
	<----->	
TOTAL	MALE	FEMALE
-----	-----	-----

TOTAL GALLONS	166066	71569	94497
	100.0%	100.0%	100.0%
BRAND A	26299	12674	13625
	15.8%	17.7%	14.4%
BRAND B	2989	1450	1539
	1.8%	2.0%	1.6%
BRAND C	30939	12787	18152
	18.6%	17.9%	19.2%
BRAND D	25885	12904	12981
	15.6%	18.0%	13.7%
BRAND E	21075	9347	11728
	12.7%	13.1%	12.4%
BRAND F	19183	8191	10992
	11.6%	11.4%	11.6%
BRAND G	16170	5817	10353
	9.7%	8.1%	11.0%
OTHER	23526	8399	15127
	14.2%	11.7%	16.0%

## 6.5 HOLECOUNT AND BREAK TABLES

A "holecount" table is CfMC's term for a table that cannot be built in a simple cross-tab format. There are many table designs that you may want to see that cannot be created by crossing one variable by another. Being able to recognize when you need to create a holecount table is usually much more difficult than creating the table itself. The idea behind the table is that you will define all the categories as if the table was a long one column (or one row) table and use the keyword `$(BREAK)` to break the definition at a certain category and force the next set of categories to print as the table's next column (or row, depending on the table orientation).

The easiest example of when you need to use a holecount table is when you have a series of rating scales on different brands or attributes and you want to create a summary table in which the brands are the stub or banner and the rating scale itself is the other axis. Suppose for this example that the brands are being used for the banner and the rating scale for the stub, then notice that there are no two variables that you can create to describe each of these items since the data in each column of the table is coming from a different data location.

"Break" tables are a variation of holecount tables. You have the same basic situation, but under each banner heading you are showing data from a current and previous wave, or two different products that are being compared. See “6.5.4 Break Table with a Multi-level Banner” for a quick preview of a break table. The data for Brand A and Brand B is coming from two different data locations, and it is also being "broken" by city. Again, you cannot write a true cross-tab but you can create the table as though you were creating Brand B's mentions after Brand A's and then use the `$(BREAK)` keyword to wrap the data so that it formats properly.

There are a number of things to be aware of when you try to create a `$(BREAK)` table. One very important point is whether or not a total column and/or row will be needed for the table, and if so what it should look like. This is very important because you often need to create your own summary totals because the system summary rows and columns will be the total base for the table, unless you have used the `$(BREAK_CONTROL)` option (see “6.5.3 Holecount Table with a Varying Percentage Base” for more information on the `$(BREAK_CONTROL)`).

**NOTE:** The number of categories in each piece of the BREAK variable must be the same.

### 6.5.1 Holecount Table with Different Brands (Locations) in the Banner

Suppose you need to produce a table with a number of different brands in your banner and a rating scale for each brand down the side. You cannot create a normal cross-tab because each rating scale is located in a different location. The first thing to determine is whether you want to define each column as you go and then break to the next column or if you want to define each row as you go and then break to the next row. Either way can give you the appropriate table, and the method of choice is usually determined by which of those two variables is easier to write and understand.

In the first example below, it is assumed that the rating scales for five different brands of colas are stored in data columns 7 through 11. To create this table by defining each column one at a time, you would define the column variable as TOTAL, and then you define the row variable by defining the first column (possibly the total), then use the \$[BREAK] keyword, and then define the second column and so on. This table could also be written by defining the row variable as TOTAL and defining each row in the column variable.

**NOTE:** The following set of commands define a standard front end for the all the examples in this section, except where noted.

```
>PRINT_FILE HOLE
~INPUT HOLE
~SET AUTOMATIC_TABLES, DROP_LOCAL_EDIT, DROP_BASE,
BEGIN_TABLE_NAME=T501

~DEFINE

STUB= STUBTOP1:
[SUPPRESS] TOTAL
```

## ADVANCED TABLES

### 6.5 HOLECOUNT AND BREAK TABLES

```
[SUPPRESS] NO ANSWER }
```

These commands are exclusive to this example.

```
TABLE_SET= {TAB501:
HEADER=: HOLECOUNT TABLE WITH THE DIFFERENT BRANDS
(LOCATIONS) IN THE BANNER}
TITLE=: RATING OF COLAS}
TITLE_4=: BASE= TOTAL SAMPLE}
EDIT=: COLUMN_WIDTH=8,STUB_WIDTH=20-COLUMN_TNA,
VERTICAL_PERCENT=1}
STUB_PREFACE= STUBTOP1
BANNER=:
|          <----- COLAS ----->
|  TOTAL  BRND A  BRND B  BRND C  BRND D  BRND E
|  -----  -----  -----  -----  -----  -----}
COLUMN=:  TOTAL
STUB=:
        TOTAL
        VERY GOOD
        GOOD
        FAIR
        POOR
        VERY POOR
        DON'T KNOW }
ROW=: [07,...,11*L^1-12,B/1//6] $[BREAK] TOTAL WITH
[07^1//6] &
        $[BREAK] TOTAL WITH [08^1//6] $[BREAK] TOTAL WITH
[09^1//6] &
        $[BREAK] TOTAL WITH [10^1//6] $[BREAK] TOTAL WITH
[11^1//6]
```





}

Here is an alternate way to write the table:

```
COL501A: [7, ..., 11*L^1-12, B] WITH [7, ..., 11^1-12, B]
$ [BREAK] &
>REPEAT $A=1, ..., 6; STRIP="$ [BREAK] &"
           [7, ..., 11*L^$A] WITH [7, ..., 11^$A] $ [BREAK] &
>END_REPEAT
ROW501A: TOTAL

~EXECUTE
TABLE_SET= TAB501
```

Here is the table Mentor prints:

```
HOLECOUNT TABLE WITH THE DIFFERENT BRANDS (LOCATIONS) IN
THE BANNER
TABLE 501
RATING OF COLAS
BASE= TOTAL SAMPLE
```

	<----- COLAS ----->					
	TOTAL	BRND A	BRND B	BRND C	BRND D	BRND E
	-----	-----	-----	-----	-----	-----
TOTAL	500	100	100	100	100	100
	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
VERY GOOD	83	25	15	17	13	13

## ADVANCED TABLES

### 6.5 HOLECOUNT AND BREAK TABLES

	16.6%	25.0%	15.0%	17.0%	13.0%	13.0%
GOOD	85	15	22	12	16	20
	17.0%	15.0%	22.0%	12.0%	16.0%	20.0%
FAIR	81	14	19	18	10	20
	16.2%	14.0%	19.0%	18.0%	10.0%	20.0%
POOR	98	17	18	16	25	22
	19.6%	17.0%	18.0%	16.0%	25.0%	22.0%
VERY POOR	74	12	11	19	17	15
	14.8%	12.0%	11.0%	19.0%	17.0%	15.0%
DON'T KNOW	79	17	15	18	19	10
	15.8%	17.0%	15.0%	18.0%	19.0%	10.0%

#### 6.5.2 Holecount Table with Rating Scales (Different Values) in Banner

Suppose you want to produce the same table as in the previous section except that you want to rotate the table 90 degrees so that the rating scale is now in the banner and the different brands are in the stub. You will want to use the same logic as the preceding table, but you will want to flip the column and row variable definitions, along with redefining your banner and stub labels to match. One other important difference is that you will probably want to percentage horizontally instead of vertically.

```
~DEFINE
```

```
TABLE_SET= {TAB502:
```

```
HEADER=:
```

```

HOLECOUNT TABLE WITH THE RATING SCALES (DIFFERENT
VALUES) IN THE BANNER}
EDIT=:
COLUMN_WIDTH=8,STUB_WIDTH=20,-COLUMN_TNA,-VERTICAL_PERC
ENT,
      HORIZONTAL_PERCENT=1}
BANNER=:
|
|                                OVERALL RATING
|
|
|<=====>
|          VERY                VERY  DON'T
|  TOTAL   GOOD   GOOD   FAIR   POOR   POOR   KNOW
|  -----  ----  ----  ----  ----  ----  -----}
COLUMN=: [07,...,11*L^1-12,B/1//6] $[BREAK] TOTAL WITH
|[07^1//6] &
|          $[BREAK] TOTAL WITH [08^1//6] $[BREAK] TOTAL WITH
|[09^1//6] &
|          $[BREAK] TOTAL WITH [10^1//6] $[BREAK] TOTAL WITH
|[11^1//6]
STUB=:
      TOTAL
      BRAND A
      BRAND B
      BRAND C
      BRAND D
      BRAND E}
ROW=:  TOTAL
}

~EXECUTE
TABLE_SET= TAB502
  
```

ADVANCED TABLES  
6.5 HOLECOUNT AND BREAK TABLES

Here is the table Mentor prints:

HOLECOUNT TABLE WITH THE RATING SCALES (DIFFERENT  
VALUES) IN THE BANNER

TABLE 502  
RATING OF COLAS

						OVERALL
RATING						
<=====>						
VERY	DON'T		VERY			
POOR	POOR	KNOW	TOTAL	GOOD	GOOD	FAIR
----	----	-----	-----	----	----	----
TOTAL			500	83	85	81
98	74	79	100.0%	16.6%	17.0%	16.2%
19.6%	14.8%	15.8%				
BRAND A			100	25	15	14
17	12	17	100.0%	25.0%	15.0%	14.0%
17.0%	12.0%	17.0%				
BRAND B			100	15	22	19
18	11	15	100.0%	15.0%	22.0%	19.0%
18.0%	11.0%	15.0%				

BRAND C			100	17	12	18
16	19	18				
			100.0%	17.0%	12.0%	18.0%
16.0%	19.0%	18.0%				
BRAND D			100	13	16	10
25	17	19				
			100.0%	13.0%	16.0%	10.0%
25.0%	17.0%	19.0%				
BRAND E			100	13	20	20
22	15	10				
			100.0%	13.0%	20.0%	20.0%
22.0%	15.0%	10.0%				

### 6.5.3 Holecount Table with a Varying Percentage Base

If you are trying to create a table similar to the one above, but each column wants to be percentaged off a different number like those who purchased that brand, then you want to create a `$(BREAK_CONTROL)` variable which allows you to define an individual base for each piece of the `$(BREAK)` variables that will be created. The `$(BREAK_CONTROL)` variable will immediately precede the beginning of the `$(BREAK)` variable (see example following).

Suppose that you want to create a table similar to the one in “6.5.1 Holecount Table with Different Brands (Locations) in the Banner”, except that now you want each brand's column based on the fact that the respondent purchased that brand (I.E. You want the percentage base to be any response). If brand purchase was stored in column 6 with BRAND A stored as a 1, BRAND B as a 2, BRAND C as a 3, BRAND D as a 4, and BRAND E as a 5, then you can write the `$(BREAK_CONTROL)` variable as `$(BREAK_CONTROL=[6^1//5])` or if you have previously defined `[6^1//5]` with a name like `BRAND_PUR`, then you define it as `$(BREAK_CONTROL=BRAND_PUR)`. This will put a base of column 6 a 1 punch on the first `$(BREAK)` item, a base of column 6 a 2 punch on the second,

## ADVANCED TABLES

### 6.5 HOLECOUNT AND BREAK TABLES

and so forth. This will also put the base on the system summary rows so that you will not have to create your own.

**NOTE:** If you do not have the variable pre-defined you will need to terminate the `$(BREAK_CONTROL)` option with two right brackets, one to close the variable and the second to close the `$(BREAK_CONTROL)`. Use of the `BREAK_CONTROL` option can significantly reduce processing time, if most respondents do not fall into most of the `BREAK` categories.

```
~DEFINE
```

```
STUB= STUBTOP2:
```

```
TOTAL
```

```
[SUPPRESS] NO ANSWER }
```

```
TABLE_SET= {TAB503:
```

```
HEADER=: HOLECOUNT TABLE WITH REDUCED BASE}
```

```
TITLE=: RATING OF COLAS }
```

```
TITLE_4=: BASE= THOSE WHO HAVE PURCHASED THE COLA}
```

```
EDIT=: COLUMN_WIDTH=8,STUB_WIDTH=20,-COLUMN_TNA }
```

```
STUB_PREFACE= STUBTOP2
```

```
BANNER=:
```

```
| <----- COLAS ----->
| BRND A BRND B BRND C BRND D BRND E
| ----- }
```

```
COLUMN=: TOTAL
```

```
STUB=:
```

```
VERY GOOD
```

```
GOOD
```

```
FAIR
```

```

POOR
VERY POOR
DON'T KNOW }
ROW=: $[BREAK_CONTROL=[06^1//5]] [07^1//6] $[BREAK]
[08^1//6] &
      $[BREAK] [09^1//6] $[BREAK] [10^1//6] $[BREAK]
[11^1//6]
}

~EXECUTE
TABLE_SET= TAB503

```

Here is the table Mentor prints:

```

HOLECOUNT TABLE WITH REDUCED BASE
TABLE 503
RATING OF COLAS
BASE= THOSE WHO HAVE PURCHASED THE COLA

```

	<----- COLAS ----->				
	BRND A	BRND B	BRND C	BRND D	BRND E
TOTAL	40	41	37	37	37
	100.0%	100.0%	100.0%	100.0%	100.0%
VERY GOOD	9	2	7	3	4
	22.5%	4.9%	18.9%	8.1%	10.8%
GOOD	9	11	4	9	7

	22.5%	26.8%	10.8%	24.3%	18.9%
FAIR	4	8	10	2	8
	10.0%	19.5%	27.0%	5.4%	21.6%
POOR	5	7	4	11	8
	12.5%	17.1%	10.8%	29.7%	21.6%
VERY POOR	5	6	7	5	5
	12.5%	14.6%	18.9%	13.5%	13.5%
DON'T KNOW	8	7	5	7	5
	20.0%	17.1%	13.5%	18.9%	13.5%

### 6.5.4 Break Table with a Multi-level Banner

If your banner is split into multiple levels so that the higher level is a demographic item such as area and the lower level is multiple products with the data for each product in a different location, then you also need to use the `$(BREAK)` keyword to produce this table. Instead of setting the column equal to the total as we have in the above examples, for this situation you set the column equal to the upper level variable. The row definition then is each of the variables for the two or more products joined together with the `$(BREAK)` keyword. Notice that the number of banner points in the finished table will be the number of categories in the column definition times the number of different breaks in the row definition. The number of rows in the table will be the number of categories in any one piece of the break variable. In the example below the rating for two products (A and B) are stored in locations 7 and 8 and the city designation is stored in location 5. The `$(BREAK_CONTROL)` keyword is used so that the System Total row will show correct values.





**ADVANCED TABLES**  
**6.5 HOLECOUNT AND BREAK TABLES**

Here is the table Mentor prints:

\$(BREAK) TABLE WITH A MULTI-LEVEL BANNER

TABLE 504

RATING OF BRAND A AND BRAND B BY CITY

	CITY									
	<=====>									
	SAN									
	TOTAL		FRANCISCO		LOS ANGELES		NEW YORK		BOSTON	
	<----->		<----->		<----->		<----->		<----->	
	BRD A	BRD B	BRD A	BRD B	BRD A	BRD B	BRD A	BRD B	BRD A	BRD B
	-----									
TOTAL	100	100	28	28	30	30	20	20	22	22
	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
VERY GOOD	25	15	9	6	7	4	4	3	5	2
	25.0%	15.0%	32.1%	21.4%	23.3%	13.3%	20.0%	15.0%	22.7%	9.1%
GOOD	15	22	4	3	6	8	4	5	1	6
	15.0%	22.0%	14.3%	10.7%	20.0%	26.7%	20.0%	25.0%	4.5%	27.3%
FAIR	14	19	3	3	3	4	1	6	7	6
	14.0%	19.0%	10.7%	10.7%	10.0%	13.3%	5.0%	30.0%	31.8%	27.3%
POOR	17	18	3	4	7	9	3	4	4	1
	17.0%	18.0%	10.7%	14.3%	23.3%	30.0%	15.0%	20.0%	18.2%	4.5%

6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)



VERY POOR	12	11	5	5	3	2	2	-	2	4
	12.0%	11.0%	17.9%	17.9%	10.0%	6.7%	10.0%		9.1%	18.2%
DON'T KNOW	17	15	4	7	4	3	6	2	3	3
	17.0%	15.0%	14.3%	25.0%	13.3%	10.0%	30.0%	10.0%	13.6%	13.6%

**6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)**

Whenever you create a table where multiple responses were collected from a given respondent and each response was collected in a separate location, you can refer to that as a "multiple location" table. Very often when you have this design you want to report information across all the locations, rather than across respondents, especially if a respondent can give the same answer in more than one location. A typical example of this might be if you ask each respondent the make of each automobile they own and now you want to produce a table that talks about automobiles instead of respondents. For instance, you want to know what percentage of all the automobiles owned are manufactured by Chevrolet, Ford, Toyota etc.

**NOTE:** A given respondent can own more than one car of the same make.

This scenario can be further complicated if there are follow up questions asked about each item originally mentioned. Suppose as above the respondent is asked the make of each automobile they own, but then is asked other questions about each automobile like what model year it is, how many miles it has on it, or if they like the service they have received on it. This is sometimes refer to as a loop or a loop structure because you will loop through a series of questions about each automobile until you have talked about all the automobiles. The difficulty in these constructions is that the information for a particular make may be in any of the data positions allocated and each respondent may have gone through the loop a different number of times.

There are three different basic approaches that can be used to create the above tables and the one you will want to use is determined by the data layout and the

type of table you wish to produce. The three methods are: using a multiple location variable, using a looped variable, and creating an overlay structure.

To create a multiple location table using a multi-location variable you need to have the data collected in the same format in each of the different locations, only one axis of the table has an item that has data coming from different locations, and no statistical testing is being done. If so, then just specify each of the data locations in a single set of brackets and use either the \*F or \*L modifier to either net or sum the locations together. See 4.3 *DEFINING DATA VARIABLES* for more information on the \*F and \*L modifiers.

To create a multiple location table using a loop variable you again need to have the data collected in the same format in each of the different locations and the locations must be the same distance from each other. Unlike using multi-location variables you can have a loop variable in more than one axis of the table, but if two or more of the axes are looped, they must contain the same number of iterations.

The third method to create a multiple location table is to use an overlay structure. This method has no restrictions on the data layout and you can have more than one axis overlaid, except that each axis that is overlaid must have the same number of overlays.

Multi-location variable:

```
MULTI_LOC: [11,12,13,14*L^1//6/9]
```

Loop variable:

```
LOOPVAR: [(4,1)11^1//6/9]
```

Overlay Structure:

```
OVERLAY: [11^1//6/9] $[OVERLAY] [12^1//6/9]
$[OVERLAY] &
[13^1//6/9] & $[OVERLAY] [14^1//6/9]
```

The above three variables would all produce a similar table to each other depending upon what the other axes look like. If both the column and the base were single location variables then the tables would be exactly the same. If you are producing a simple multiple location table, then approach one is preferred, but if the data is stored in a loop structure and either the base or the column variable is also dependent upon the loop then you must use either the second or third approach.

For loop or overlay variables, the program treats each iteration of the loop or overlay piece as a distinct case. Suppose you have 100 respondents in your sample, but you create a variable with five overlay pieces in it, then the program will act as though there are 500 possible cases for that table. If one of the axes is not overlaid, then that variable will be used for each piece of the overlay. The same logic also applies to loop variables.

**NOTE:** You often want to create your own summary rows and columns because the system-generated numbers may not be appropriate when using overlaid or loop variables.

The maximum number of iterations in a loop variable is 99. See also ~SET LOOP\_KICKOUT.

### 6.6.1 Simple Multiple Location Tables

If you create a table with a typical demographic banner and the row is a question that is stored in multiple locations for each respondent with the same coding scheme and this table will count all mentions together regardless of which location the answer came from, then you create this table by creating a multi-location variable.

For example, you ask respondents about the type of banking they do. You ask each respondent about each of the different banks they use including what kinds of accounts they have at each bank and their overall satisfaction with that account. In the example below, the sex of the respondent has been stored in column 5 and we are collecting information for up to six different banks. For each account mentioned there is a 10 column field which holds information just about that bank. The first two columns of each field are a two digit number which indicates which bank the

## ADVANCED TABLES

### 6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)

account is held at. The third column is a punch to note the type of account. The fourth column is an overall rating of that bank's service for that account and the fifth through the tenth columns are the total dollar amount held in that account. The first mention is stored in columns 11-20, the second in 21-30, the third in 31-40, the fourth in 101-110, the fifth in 111-120, and the last in 121-130.

This first table that is produced is a simple multiple location table with the percentage base for the table being the total number of banks which is much greater than the total number of respondents, so you will need to generate your own total.

**NOTE:** The following set of commands define a standard front end for the next set of examples.

```
>PRINT_FILE OVERL
~INPUT OVERL
~SET AUTOMATIC_TABLES, DROP_LOCAL_EDIT, DROP_BASE,
BEGIN_TABLE_NAME=T601

~DEFINE

STUB= STUBTOP1:
[SUPPRESS] TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= {BAN2:
EDIT=: COLUMN_WIDTH=8, STUB_WIDTH=25, -COLUMN_TNA,
STATISTICS_DECIMALS=2, VERTICAL_PERCENT=1}
STUB_PREFACE= STUBTOP1

BANNER=:
|                                GENDER
```



```
|          <----->
|  TOTAL    MALE  FEMALE
|  -----  ----  -}
COLUMN=:  TOTAL WITH [5^1/2]
}
```

```
~EXECUTE
TABLE_SET= BAN2
```

Here are the commands exclusive to this example.

```
~DEFINE

TABLE_SET= {TAB601:
HEADER=: TABLE WITH MENTIONS FROM MULTIPLE LOCATIONS
ADDED TOGETHER}
TITLE=: ACCOUNTS HAVE AT ANY BANK}
TITLE_4=: BASE: TOTAL ACCOUNTS OF BANKS}
STUB=:
    TOTAL
    CHECKING ACCOUNT
    SAVINGS ACCOUNT
    VISA/MASTERCARD
    MORTGAGE/HOME LOAN
    CAR/PERSONAL LOAN
    ATM CARD
    DK/NA/REF}
ROW=: [13,23,33,103,113,123*L^1-6,9/1//6/9]
}
```

ADVANCED TABLES

6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)

Here is an alternate way to write the row variable:

```

ROW601A: [13^1-6,9/1//6/9] $ [OVERLAY] [23^1-6,9/1//6/9]
$ [OVERLAY] &
          [33^1-6,9/1//6/9] $ [OVERLAY] [103^1-6,9/1//6/9]
$ [OVERLAY] &
          [113^1-6,9/1//6/9] $ [OVERLAY]
[123^1-6,9/1//6/9]

~EXECUTE
TABLE_SET= TAB601

```

Here is the table Mentor prints:

```

TABLE WITH MENTIONS FROM MULTIPLE LOCATIONS ADDED
TOGETHER
TABLE 601
ACCOUNTS HAVE AT ANY BANK
BASE: TOTAL ACCOUNTS OF BANKS

```

	TOTAL	SEX	
		MALE	FEMALE
TOTAL	285	126	159
	100.0%	100.0%	100.0%
CHECKING ACCOUNT	37	16	21
	13.0%	12.7%	13.2%



SAVINGS ACCOUNT	45	16	29
	15.8%	12.7%	18.2%
VISA/MASTERCARD	36	18	18
	12.6%	14.3%	11.3%
MORTGAGE/HOME LOAN	37	17	20
	13.0%	13.5%	12.6%
CAR/PERSONAL LOAN	48	22	26
	16.8%	17.5%	16.4%
ATM CARD	38	17	21
	13.3%	13.5%	13.2%
DK/NA/REF	44	20	24
	15.4%	15.9%	15.1%

A second table could be produced using this construction, but instead of counting each account as many times as it appears you only want to count it once no matter how many times it appears. The syntax only changes slightly for the multi-location approach, as the \*L now becomes \*F. In the overlay approach the keyword OVERLAY is now replaced with the keyword NET\_OVERLAY. The loop variable approach cannot produce this table.

The previous table tells you how many of each account there are, while this next table tells you how many respondents have that type of account. Notice that the percentage base for this table is the true total number of respondents.

## ADVANCED TABLES

### 6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)

```
~DEFINE
```

```
STUB= STUBTOP2:
```

```
TOTAL
```

```
[SUPPRESS] NO ANSWER }
```

```
TABLE_SET= {TAB602:
```

```
HEADER=: TABLE WITH MENTIONS FROM MULTIPLE LOCATIONS  
NETTED TOGETHER}
```

```
LOCAL_EDIT=: VERTICAL_PERCENT=T}
```

```
STUB_PREFACE= STUBTOP2
```

```
STUB=:
```

```
    CHECKING ACCOUNT
```

```
    SAVINGS ACCOUNT
```

```
    VISA/MASTERCARD
```

```
    MORTGAGE/HOME LOAN
```

```
    CAR/PERSONAL LOAN
```

```
    ATM CARD
```

```
    DK/NA/REF}
```

```
ROW=: [13,23,33,103,113,123*F^1//6/9]
```

```
}
```

Here is an alternate way to write the row variable:

```
ROW602B: [13^1//6/9] $[NET_OVERLAY] [23^1//6/9]
```

```
$(NET_OVERLAY) [33^1//6/9] &
```

```
    $(NET_OVERLAY) [103^1//6/9] $(NET_OVERLAY]
```

```
[113^1//6/9] &
```

```
    $(NET_OVERLAY] [123^1//6/9]
```



```
~EXECUTE  
TABLE_SET= TAB602
```

ADVANCED TABLES

6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)

Here is the table Mentor prints:

TABLE WITH MENTIONS FROM MULTIPLE LOCATIONS NETTED TOGETHER

TABLE 602

ACCOUNTS HAVE AT ANY BANK

		SEX	
		<----->	
	TOTAL	MALE	FEMALE
	-----	-----	-----
TOTAL	100	42	58
	100.0%	100.0%	100.0%
CHECKING ACCOUNT	32	14	18
	32.0%	33.3%	31.0%
SAVINGS ACCOUNT	33	12	21
	33.0%	28.6%	36.2%
VISA/MASTERCARD	32	17	15
	32.0%	40.5%	25.9%
MORTGAGE/HOME LOAN	31	14	17
	31.0%	33.3%	29.3%
CAR/PERSONAL LOAN	35	17	18
	35.0%	40.5%	31.0%



ATM CARD	28	13	15
	28.0%	31.0%	25.9%
DK/NA/REF	30	16	14
	30.0%	38.1%	24.1%

### 6.6.2 Tables With Both the Row and the Base Overlaid

By far the most useful example of an overlay table or a loop variable is when you have a loop structure as described above and the table you want to produce has only information about a particular product in it. You want the program to look in each of the data locations, but only report information from those locations when it is referring to the particular product you are interested in. In this example suppose you want to produce a table similar to the ones above, but now you only want to report on accounts at a particular bank. In order to create this table you must define an overlay variable for the base. Similar syntax is used for defining the row definition and you must have the same number of overlay pieces in the base definition as the row definition. The resultant table will be a compilation of the six separate rows and bases with the banner. Again, you will want to produce your summary rows and columns to ensure you get the numbers you are expecting.

**TIP:** When creating overlay tables it is often helpful to think of each separate loop location as a separate table. Usually each of these individual tables is simple to create, and then all you need to do is combine these tables by making each of them a separate piece in the overlay. For instance, in the example below if the data was only stored in columns 11-13, then you would write the table with a BASE of [11.2#01] and the ROW of [13^1-6,9/1//6/9]. This then becomes the first overlay piece in each of the base and the row definitions. Now looking only at the second iteration, the data was stored in columns 21-23 and therefore it would have a BASE of [21.2#01] and a ROW of [23^1-6,9/1//6/9]. This continues for the rest of the iterations.

**NOTE:** The same row spec would be used regardless of which bank you were reporting on as only the base definition would change.

## ADVANCED TABLES

### 6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)

~DEFINE

TABLE\_SET= {TAB603:

HEADER=: MULTIPLE MENTION TABLE BASED ON A SINGLE ITEM  
FROM THE LOOP STRUCTURE}

TITLE=: ACCOUNTS HAVE AT BANK A}

TITLE\_4=: BASE: THOSE WHO USE BANK A}

BASE=: [11.2#01] \$[OVERLAY] [21.2#01] \$[OVERLAY]  
[31.2#01] \$[OVERLAY] &

[101.2#01] \$[OVERLAY] [111.2#01] \$[OVERLAY]  
[121.2#01]

STUB\_PREFACE= STUBTOP1

STUB=:

TOTAL

CHECKING ACCOUNT

SAVINGS ACCOUNT

VISA/MASTERCARD

MORTGAGE/HOME LOAN

CAR/PERSONAL LOAN

ATM CARD

DK/NA/REF}

ROW=: [13^1-6,9/1//6/9] \$[OVERLAY] [23^1-6,9/1//6/9]  
\$[OVERLAY] &

[33^1-6,9/1//6/9] \$[OVERLAY] [103^1-6,9/1//6/9]  
\$[OVERLAY] &

[113^1-6,9/1//6/9] \$[OVERLAY] [123^1-6,9/1//6/9]

}

~EXECUTE

TABLE\_SET= TAB603



Here is the table Mentor prints:

MULTIPLE MENTION TABLE BASED ON A SINGLE ITEM FROM THE LOOP STRUCTURE

TABLE 603

ACCOUNTS HAVE AT BANK A

BASE: THOSE WHO USE BANK A

	SEX		
	<----->		
	TOTAL	MALE	FEMALE
	-----	-----	-----
TOTAL	44	17	27
	100.0%	100.0%	100.0%
CHECKING ACCOUNT	7	3	4
	15.9%	17.6%	14.8%
SAVINGS ACCOUNT	6	2	4
	13.6%	11.8%	14.8%
VISA/MASTERCARD	5	2	3
	11.4%	11.8%	11.1%
MORTGAGE/HOME LOAN	5	1	4
	11.4%	5.9%	14.8%
CAR/PERSONAL LOAN	7	2	5
	15.9%	11.8%	18.5%

## ADVANCED TABLES

### 6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)

ATM CARD	5	4	1
	11.4%	23.5%	3.7%
DK/NA/REF	9	3	6
	20.5%	17.6%	22.2%

#### 6.6.3 Overlay Tables With Summary Statistics (Means)

If the loop structure contains variables that you would like to produce statistics on, then you can follow the procedures as above, but you also must now overlay the definition of the statistic (MEAN). Suppose a rating scale for each bank was located in columns 14, 24, 34, 104, 114, and 124 and you want to produce the mean of that rating. You will need to specify the mean for each piece of the overlay separately and the program will combine them to give you an overall mean.

**NOTE:** Any needed recoding of the mean would follow the normal procedures and would have to be done for each piece of the overlay. As in the table before you will need to overlay the base definition if you want to produce the table on a particular bank.

```
~DEFINE
```

```
TABLE_SET= {TAB604:
HEADER=: OVERLAY TABLE WITH SUMMARY STATISTICS LIKE A
MEAN}
TITLE=: OVERALL RATING OF BANK A}
TITLE_4=: BASE: THOSE WHO USE BANK A}
BASE=: [11.2#01] $ [OVERLAY] [21.2#01] $ [OVERLAY]
[31.2#01] $ [OVERLAY] &
[101.2#01] $ [OVERLAY] [111.2#01] $ [OVERLAY]
[121.2#01]
STUB=:
```





```

TOTAL
EXCELLENT (5)
VERY GOOD (4)
GOOD (3)
FAIR (2)
POOR (1)
DK/NA

[STATISTICS] MEAN
[STATISTICS] STANDARD DEVIATION }
ROW=: [14^1-5,9/5//1/9] $[MEAN,STD] [14*R=1-5]
$[OVERLAY] [24^1-5,9/5//1/9] &
    $[MEAN,STD] [24*R=1-5] $[OVERLAY]
[34^1-5,9/5//1/9] $[MEAN,STD] &
    [34*R=1-5] $[OVERLAY] [104^1-5,9/5//1/9]
$[MEAN,STD] [104*R=1-5] &
    $[OVERLAY] [114^1-5,9/5//1/9] $[MEAN,STD]
[114*R=1-5] $[OVERLAY] &
    [124^1-5,9/5//1/9] $[MEAN,STD] [124*R=1-5]
}

~EXECUTE
TABLE_SET= TAB604
    
```

Here is the table Mentor prints:

```

OVERLAY TABLE WITH SUMMARY STATISTICS LIKE A MEAN
TABLE 604
OVERALL RATING OF BANK A
BASE: THOSE WHO USE BANK A
    
```

SEX

**ADVANCED TABLES**

*6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)*

		<----->		
		TOTAL	MALE	FEMALE
		-----	-----	-----
TOTAL		42	16	26
		100.0%	100.0%	100.0%
EXCELLENT	(5)	7	3	4
		16.7%	18.8%	15.4%
VERY GOOD	(4)	11	4	7
		26.2%	25.0%	26.9%
GOOD	(3)	7	1	6
		16.7%	6.3%	23.1%
FAIR	(2)	7	4	3
		16.7%	25.0%	11.5%
POOR	(1)	5	1	4
		11.9%	6.3%	15.4%
DK/NA		5	3	2
		11.9%	18.8%	7.7%
MEAN		3.22	3.31	3.17
STANDARD DEVIATION		1.34	1.38	1.34



### 6.6.4 Tables with the Banner and the Row Overlaid

If you have a scenario where the banner also contains a category that was collected in the loop structure you then will have to create the column variable with an overlay also. A typical example of this is that the banner might consist of all the banks and the stub consists of the services used at each bank. In order to create this table you will need to overlay all three parts of the table, row, banner, and base, because a given cell in the table is dependent upon both the row and column variable coming from the same iteration of the loop. Again, the easiest way to write this table is to pretend that you are writing six different tables each only coming from one location and then use the OVERLAY keyword to combine them all.

This type of table would also be required if you were creating a table that crosses the amount of money in the bank by type of accounts have, since both variables are inside the loop structure.

~DEFINE

```
TABLE_SET= {TAB605:
HEADER=: TABLE WITH BANNER, ROW, AND BASE ALL WITH
OVERLAY STRUCTURES}
TITLE=: OVERALL RATING OF EACH BANK}
TITLE_4=: BASE: THOSE WHO USED THAT BANK}
BASE=: [11.2#1-10] $ [OVERLAY] [21.2#1-10] $ [OVERLAY]
[31.2#1-10] $ [OVERLAY] &
           [101.2#1-10] $ [OVERLAY] [111.2#1-10] $ [OVERLAY]
[121.2#1-10]
BANNER=:
|
|          <----- USED ----->
|          BANK    BANK    BANK    BANK    BANK
|  TOTAL      A      B      C      D      E
|  -----  - - - -  - - - -  - - - -  - - - -  - - - - }
```

**ADVANCED TABLES**

*6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)*

```

COLUMN=: [11.2#1-10/1//5] $ [OVERLAY] [21.2#1-10/1//5]
$ [OVERLAY] &
        [31.2#1-10/1//5] $ [OVERLAY] [101.2#1-10/1//5]
$ [OVERLAY] &
        [111.2#1-10/1//5] $ [OVERLAY] [121.2#1-10/1//5]
STORE_TABLES=*
}

~EXECUTE
TABLE_SET= TAB605
    
```

Here is the table Mentor prints:

TABLE WITH BANNER, ROW, AND BASE ALL WITH OVERLAY  
STRUCTURES

TABLE 605

OVERALL RATING OF EACH BANK

BASE: THOSE WHO USED THAT BANK

USED ----->		<-----			
BANK	BANK	TOTAL	BANK A	BANK B	BANK C
D	E				
----	----	-----	----	----	----
TOTAL		285	42	37	43
46	34				
100.0%	100.0%	100.0%	100.0%	100.0%	100.0%

ADVANCED TABLES  
6.6 MULTIPLE LOCATION TABLES (OVERLAY AND LOOP STRUCTURES)

EXCELLENT	(5)	33	7	7	3
3	3				
		11.6%	16.7%	18.9%	7.0%
6.5%	8.8%				
VERY GOOD	(4)	57	11	10	5
12	4				
		20.0%	26.2%	27.0%	11.6%
26.1%	11.8%				
GOOD	(3)	49	7	6	8
5	8				
		17.2%	16.7%	16.2%	18.6%
10.9%	23.5%				
FAIR	(2)	62	7	10	8
11	9				
		21.8%	16.7%	27.0%	18.6%
23.9%	26.5%				
POOR	(1)	43	5	2	10
10	3				
		15.1%	11.9%	5.4%	23.3%
21.7%	8.8%				
DK/NA		41	5	2	9
5	7				
		14.4%	11.9%	5.4%	20.9%
10.9%	20.6%				
MEAN		2.90	3.22	3.29	2.50
2.68	2.81				

STANDARD DEVIATION	1.32	1.34	1.25	1.31
1.33	1.18			

## 6.7 LONG BRAND LISTS

Very often when you produce a table from a long list of brands, attributes, or other similar type mentions, you can make the table easier to read by doing any of the following:

- 1 Producing nets of categories that are similar.
- 2 Ranking the table with those categories mentioned most printing first.
- 3 Suppressing rows that have no mentions.
- 4 Collapsing rows with few mentions into an All Other category.

### 6.7.1 Producing Net Categories

You may want to produce an additional category which is a net of other mentions. There are a number of different ways to do this, most of which are also discussed in other parts of the manual. They are gathered here to show the difference between them. The way you choose to do the net depends on how the data is coded and personal preference.

The easiest way to create a net is inside an existing punch or numeric variable. If you use the dash it will net from the code before the dash to the code after the dash. [11^1-4] will produce one category which is the net of punches 1 through 4 and [12.2#1-10] will produce one category which is the net of numbers 1 through 10. See “4.6.1 Summary of Rules for Defining Data” for more information on this. Examples in this sub-section just show the stub and the row definition for a table.

```
TABLE_SET= {NET_DASH:
```

```

STUB= :
11^1-4
12.2#1-10 }
ROW= : [11^1-4] WITH [12.2#1-10]
}

```

All the other forms of netting depend on how the data is coded. If you need to form a net of a punch category with a numeric category, two different punches from two different columns, or two different numbers from two different fields, then you will need to use the OR joiner. See “5.1.1 Logical Joiners” for more information on the OR joiner.

```

TABLE_SET= {NET_OR :
STUB= :
11^1 OR 53^3
11^2 OR 12.2#45
12.2#45 OR 14.2#28 }
ROW= : ([11^1] OR [53^3]) WITH &
        ([11^2] OR [12.2#45]) WITH &
        ([12.2#45] OR [14.2#28])
}

```

If you wish to form nets as above, but want the net category to appear in front of all the inside categories, then you will want to use the NET function. The NET function allows you to define an expression and add one additional category to the front of it which is the net of anyone who is in any of the categories in the expression.

```

TABLE_SET= {NET_FUNC :
STUB= :
11^1-4 OR 12.2#1-20
11^1

```

## ADVANCED TABLES

### 6.7 LONG BRAND LISTS

```
11^2
11^3
11^4
12.2#1-10
12.2#11-20 }
ROW=: NET([11^1//4] WITH [12.2#1-10/11-20])
}
```

This is the way the variable would look if you used the OR joiner instead of the NET function.

```
NO_NET_FUNC: ([11^1-4] OR [12.2#1-20]) WITH &
              [11^1//4] WITH [12.2#1-10/11-20]
```

If you are netting the same code from a number of different locations, then you can use the \*F modifier to net the locations. This is most useful if you have a set of answers in two different fields where the codes are the same for the two fields and you want to net all the answers together.

```
TABLE_SET= {NET_STAR_F:
STUB=:
11^1 OR 12^1
11^2 OR 12^2
11^3 OR 12^3
11^4 OR 12^4 }
ROW=: [11,12*F^1//4]
}
```

This is the way the variable would look if you used the OR joiner instead of the NET function.



```
NO_NET_STAR_F: ([11^1] OR [12^1]) WITH ([11^2] OR
[12^2]) WITH &
                ([11^3] OR [12^3]) WITH ([11^4] OR
[12^4])
```

If you need to net two variables together, such that the first category of each is netted, and then the second, and so on, then you will want to use the NET joiner. This is most useful for grids where the unaided awareness was coded as a long list and the aided awareness was coded as yes/no (1 / 2).

```
TABLE_SET= {NET_JOINER:
STUB=:
11^1 OR 15^1
11^2 OR 16^1
11^3 OR 17^1
11^4 OR 18^1 }
ROW=: [11^1//4] NET [15,16,17,18^1]
}
```

This is the way the variable would look if you used the OR joiner instead of the NET joiner.

```
NO_NET_JOINER: ([11^1] OR [15^1]) WITH ([11^2] OR
[16^1]) WITH &
                ([11^3] OR [17^1]) WITH ([11^4] OR [18^1])
```

### 6.7.2 Ranking With Nets And Sub-Nets

When you rank a table with nets and sub-nets you need to assign rank levels to the different stubs, so that like items stay together under their appropriate nets and sub-nets. This is done by setting RANK\_LEVEL=# on each stub item, where the # is the rank level. Highest order nets, stubs that are not under any other net, and stubs like ALL OTHER, DON'T KNOW, REFUSED, and NONE should all be set

to level 1. All items directly under a net are then assigned level 2 including any sub-nets. Any items directly under a sub-net are then assigned the next level (3), including any sub-sub-nets. Continue with this process till every item has been assigned a rank level. Also be sure to force any items low or high in their level as needed by putting an L or H after the number (i.e., any Other type response should be kept low in its rank level). The default rank level is 1 unless it is specified on the EDIT statement or you have used the KEEP\_RANK option which says keep this rank level in effect until you see another rank level command.

After you have assigned the rank level to each stub item, you then can invoke ranking either by using RANK\_LEVEL=1 or RANK\_IF\_INDICATED on the EDIT statement.

When the program then ranks such an annotated stub, it first ranks all the level 1 items from high to low, forcing any high or low as indicated. It then takes all the level 2 items that were under the level 1 item that has come to the top and ranks them. It then ranks any level 3 items under the first level 2 items. It continues with this until it has no higher level to rank, and then goes to the next lower level item in the rank.

**NOTE:** Any given item always will stay under the first item above it with a lower number.

In the following example is the list of stubs that will be printed:

```
DOMESTIC (NET)
CHRYSLER
FORD
GENERAL MOTORS
OTHER DOMESTIC
EUROPEAN (NET)
BRITISH (SUB-NET)
ALFA ROMEO
```

JAGUAR  
RANGE ROVER  
ROLLS-ROYCE  
STERLING  
OTHER BRITISH  
GERMAN (SUB-NET)  
BMW  
MERCEDES  
VOLKSWAGEN/PORSCHE/AUDI  
OTHER GERMAN  
OTHER EUROPEAN (SUB-NET)  
FIAT  
PEUGEOT  
SAAB  
VOLVO  
YUGO  
OTHER EUROPEAN  
ASIAN (NET)  
JAPANESE (SUB-NET)  
HONDA  
MAZDA  
NISSAN  
SUBARU  
TOYOTA  
OTHER JAPANESE  
OTHER ASIAN (SUB-NET)  
HYUNDAI  
OTHER ASIAN  
NONE  
DON'T KNOW

To assign the rank levels for this stub, you must first find all the major net categories and any items not in a net and assign them level 1. In this case they are the nets DOMESTIC, EUROPEAN, and ASIAN, along with NONE and DON'T KNOW. In addition, NONE and DON'T KNOW should be forced low. Next you need to look at all the items between DOMESTIC and EUROPEAN. CHRYSLER, FORD, GENERAL MOTORS, and OTHER DOMESTIC are all the same level and should be assigned level 2 with OTHER DOMESTIC being forced low. Next, look at all the items between EUROPEAN and ASIAN. Notice these items are not all at the same level so you must find all the sub-nets and items which do not belong to any sub-net and assign them level 2. OTHER EUROPEAN should be forced low. Continue on with this for the rest of the stub. See the stub below to see the rank level that was assigned for each stub item.

Often in conjunction with nets and sub-nets you want to underline the label of the nets and sub-nets. You can do this by using the UNDERLINE keyword on those stubs. You also will probably want to use the LINES\_LEFT option which says skip to a new page if you do not have this many lines left on the page. This will keep a net line from printing at the bottom of the page and having all the items under it print at the top of the next page. However, setting this too high will cause a lot of blank space at the bottom of some pages.

Indenting is done automatically for levels 2 and higher. Use the EDIT option STUB\_RANK\_INDENT= to change the default of two characters of indentation per rank level greater than 1.

**NOTE:** The following set of commands define a standard front end for the next set of examples.

```
>PRINT_FILE LISTS
~INPUT LISTS
~SET AUTOMATIC_TABLES, DROP_LOCAL_EDIT, DROP_BASE,
BEGIN_TABLE_NAME=T701

~DEFINE
```

```

STUB= STUBTOP1:
TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= {BAN2:
EDIT=: COLUMN_WIDTH=8,STUB_WIDTH=30,-COLUMN_TNA }
STUB_PREFACE= STUBTOP1
BANNER=:
|                SEX
|                <----->
|  TOTAL      MALE  FEMALE
|  -----   -    -}
COLUMN=:  TOTAL WITH [5^1/2]
}

~EXECUTE
TABLE_SET= BAN2

```

These commands are exclusive to this example.

```

~DEFINE

TABLE_SET= {TAB701:
HEADER=: RANKING LARGE BRAND LIST WITH NETS AND
SUB-NETS}
TITLE=: AUTOMOBILE MANUFACTURERS}
LOCAL_EDIT=: RANK_LEVEL=1 }
STUB=:

```

ADVANCED TABLES

6.7 LONG BRAND LISTS

[RANK\_LEVEL=1, UNDERLINE, LINES\_LEFT=17] DOMESTIC  
(NET)  
[KEEP\_RANK=2] CHRYSLER  
FORD  
GENERAL MOTORS  
[RANK\_LEVEL=2L] OTHER DOMESTIC  
[RANK\_LEVEL=1, UNDERLINE, LINES\_LEFT=26] EUROPEAN  
(NET)  
[RANK\_LEVEL=2, UNDERLINE, LINES\_LEFT=23] BRITISH  
(SUB-NET)  
[KEEP\_RANK=3] ALFA ROMEO  
JAGUAR  
RANGE ROVER  
ROLLS-ROYCE  
STERLING  
[RANK\_LEVEL=3L] OTHER BRITISH  
[RANK\_LEVEL=2, UNDERLINE, LINES\_LEFT=17] GERMAN  
(SUB-NET)  
[KEEP\_RANK=3] BMW  
MERCEDES  
VOLKSWAGEN/PORSCHE/AUDI  
[RANK\_LEVEL=3L] OTHER GERMAN  
[RANK\_LEVEL=2L, UNDERLINE, LINES\_LEFT=23] OTHER  
EUROPEAN (SUB-NET)  
[KEEP\_RANK=3] FIAT  
PEUGEOT  
SAAB  
VOLVO  
YUGO  
[RANK\_LEVEL=3L] OTHER EUROPEAN

```
[RANK_LEVEL=1, UNDERLINE, LINES_LEFT=26] ASIAN (NET)
[RANK_LEVEL=2, UNDERLINE, LINES_LEFT=23] JAPANESE
(SUB-NET)
[KEEP_RANK=3] HONDA
MAZDA
NISSAN
SUBARU
TOYOTA
[RANK_LEVEL=3L] OTHER JAPANESE
[RANK_LEVEL=2L, UNDERLINE, LINES_LEFT=11] OTHER
ASIAN (SUB-NET)
[RANK_LEVEL=3] HYUNDAI
[RANK_LEVEL=3L] OTHER ASIAN
[RANK_LEVEL=1L, LINES_LEFT=8] NONE
[RANK_LEVEL=1L] DON'T KNOW }
ROW=: [11.3^1-4/1//4/5.20/5-10/5//10/11.14/11//14/15-
20/15//20/21-28/21-26/&
21//26/27,28/27/28/29/30]
}
~EXECUTE
TABLE_SET= TAB701
```

Only the first and last part of the table are printed here.

```
RANKING LARGE BRAND LIST WITH NETS AND SUB-NETS
TABLE 701
AUTOMOBILE MANUFACTURERS
```

ADVANCED TABLES

6.7 LONG BRAND LISTS

	SEX		
	TOTAL	MALE	FEMALE
TOTAL	200	109	91
	100.0%	100.0%	100.0%
ASIAN (NET)	88	42	46
-----	44.0%	38.5%	50.5%
JAPANESE (SUB-NET)	86	42	44
-----	43.0%	38.5%	48.4%
NISSAN	23	11	12
	11.5%	10.1%	13.2%
HONDA	21	11	10
	10.5%	10.1%	11.0%
TOYOTA	13	7	6
	6.5%	6.4%	6.6%
SUBARU	9	4	5
	4.5%	3.7%	5.5%
MAZDA	8	5	3
	4.0%	4.6%	3.3%
OTHER JAPANESE	23	9	14





	11.5%	8.3%	15.4%
OTHER ASIAN (SUB-NET)	5	2	3
-----	2.5%	1.8%	3.3%
HYUNDAI	5	2	3
	2.5%	1.8%	3.3%
OTHER ASIAN	-	-	-
-----			

**ADVANCED TABLES**

*6.7 LONG BRAND LISTS*

DOMESTIC (NET)	46	25	21
-----	23.0%	22.9%	23.1%
CHRYSLER	20	13	7
	10.0%	11.9%	7.7%
GENERAL MOTORS	20	10	10
	10.0%	9.2%	11.0%
FORD	12	5	7
	6.0%	4.6%	7.7%
OTHER DOMESTIC	-	-	-
NONE	37	20	17
	18.5%	18.3%	18.7%
DON'T KNOW	22	15	7
	11.0%	13.8%	7.7%

### 6.7.3 Suppressing Blank Rows in a Large List

To suppress a row with no mentions in it, you need only to do is use the option `MINIMUM_FREQUENCY=1` on the `EDIT` or `LOCAL_EDIT` statement to then suppress all rows which do not have at least one respondent in them. If you have rows that you want to print even though the frequency is zero, like an `OTHER` category or `DON'T KNOW` category (that you always want to show was calculated on the table), then you will want to use the option `-MINIMUM_FREQUENCY` on all those stubs. Also be aware that if you have weighted data, then you may not want to set `MINIMUM_FREQUENCY=1`, but rather to some other value like either 0.5 or 0.01 depending upon what you want to



do with a row that has less than 0.5 as the frequency. This frequency would round to zero, but there is at least one valid respondent in that category. A value of 0.5 would suppress this row, while 0.01 would print it with a dash.

This example uses comment labels and stub indentation to mark the different groupings although no net categories have been defined.

## ADVANCED TABLES

### 6.7 LONG BRAND LISTS

~DEFINE

TABLE\_SET= TAB702:

HEADER=: SUPPRESSING ZERO ROWS IN A LARGE BRAND LIST  
WITH COMMENTS }

LOCAL\_EDIT=: MINIMUM\_FREQUENCY=1 }

STUB=:

[COMMENT,UNDERLINE] DOMESTIC

[STUB\_INDENT=2] CHRYSLER

[STUB\_INDENT=2] FORD

[STUB\_INDENT=2] GENERAL MOTORS

[STUB\_INDENT=2, -MINIMUM\_FREQUENCY] OTHER DOMESTIC

[COMMENT,UNDERLINE] EUROPEAN

[COMMENT,STUB\_INDENT=2,UNDERLINE] BRITISH

[STUB\_INDENT=4] ALFA ROMEO

[STUB\_INDENT=4] JAGUAR

[STUB\_INDENT=4] RANGE ROVER

[STUB\_INDENT=4] ROLLS-ROYCE

[STUB\_INDENT=4] STERLING

[STUB\_INDENT=4, -MINIMUM\_FREQUENCY] OTHER BRITISH

[COMMENT,STUB\_INDENT=2,UNDERLINE,LINES\_LEFT=16]

GERMAN

[STUB\_INDENT=4] BMW

[STUB\_INDENT=4] MERCEDES

[STUB\_INDENT=4] VOLKSWAGEN/PORSCHE/AUDI

[STUB\_INDENT=4, -MINIMUM\_FREQUENCY] OTHER GERMAN

[COMMENT,STUB\_INDENT=2,UNDERLINE,LINES\_LEFT=22]

OTHER EUROPEAN

[STUB\_INDENT=4] FIAT

```

[STUB_INDENT=4] PEUGEOT
[STUB_INDENT=4] SAAB
[STUB_INDENT=4] VOLVO
[STUB_INDENT=4] YUGO
[STUB_INDENT=4, -MINIMUM_FREQUENCY] OTHER EUROPEAN
[COMMENT, UNDERLINE, LINES_LEFT=25] ASIAN
[COMMENT, STUB_INDENT=2, UNDERLINE, LINES_LEFT=22]
JAPANESE
[STUB_INDENT=4] HONDA
[STUB_INDENT=4] MAZDA
[STUB_INDENT=4] NISSAN
[STUB_INDENT=4] SUBARU
[STUB_INDENT=4] TOYOTA
[STUB_INDENT=4, -MINIMUM_FREQUENCY] OTHER JAPANESE
[COMMENT, STUB_INDENT=2, UNDERLINE, LINES_LEFT=10]
OTHER ASIAN
[STUB_INDENT=4] HYUNDAI
[STUB_INDENT=4, -MINIMUM_FREQUENCY] OTHER ASIAN
[-MINIMUM_FREQUENCY, LINES_LEFT=8] NONE
[-MINIMUM_FREQUENCY] DON'T KNOW }
ROW=: [11.3^1//30]
}

```

~EXECUTE

TABLE\_SET= TAB702

Only the first and last part of the table are printed here.

SUPPRESSING ZERO ROWS IN A LARGE BRAND LIST WITH  
COMMENTS

ADVANCED TABLES

6.7 LONG BRAND LISTS

TABLE 702  
AUTOMOBILE MANUFACTURERS

	SEX		
	TOTAL	MALE	FEMALE
TOTAL	200	109	91
	100.0%	100.0%	100.0%
DOMESTIC			
CHRYSLER	20	13	7
	10.0%	11.9%	7.7%
FORD	12	5	7
	6.0%	4.6%	7.7%
GENERAL MOTORS	20	10	10
	10.0%	9.2%	11.0%
OTHER DOMESTIC	-	-	-
EUROPEAN			
BRITISH			
JAGUAR	1	1	-

	0.5%	0.9%	
RANGE ROVER	4	2	2
	2.0%	1.8%	2.2%
STERLING	5	3	2
	2.5%	2.8%	2.2%
OTHER BRITISH	4	2	2
	2.0%	1.8%	2.2%
GERMAN			
-----			
BMW	9	5	4
	4.5%	4.6%	4.4%
MERCEDES	11	6	5
	5.5%	5.5%	5.5%
VOLKSWAGEN/PORSCHE/AUDI	13	9	4
	6.5%	8.3%	4.4%
OTHER GERMAN	2	1	1
	1.0%	0.9%	1.1%
-----			

SUPPRESSING ZERO ROWS IN A LARGE BRAND LIST WITH COMMENTS

TABLE 702 (continued)

AUTOMOBILE MANUFACTURERS

	SEX		
	TOTAL	MALE	FEMALE
NONE	37 18.5%	20 18.3%	17 18.7%
DON'T KNOW	22 11.0%	15 13.8%	7 7.7%

### 6.7.4 Collapsing Low Mentions into another Category

Sometimes when you have a long list you not only want to suppress blank rows, but you may want to suppress rows that contain less than a certain frequency or percentage value. If you suppress these rows however, you are actually removing numbers from the table and will probably want to print them in what is called a collapsed category at the bottom of the table. This allows you to take long lists and shorten them so that they are easier to read because they only have the top mentions and are not cluttered up with lots of rows with only a couple of mentions in each.

Whenever a row is suppressed due to the MINIMUM\_FREQUENCY or MINIMUM\_PERCENT option, the system sums all the numbers from that row in a system row called SUPPRESSED. You can then use the PRINT\_ROW option to print that row at the bottom of the table.





In the example below all rows with less than 5 percent of the total are being suppressed along with all the Other categories to produce one global Other category. MINIMUM\_PERCENT=5 is set on the EDIT and then it is set to 100 on the Other categories to make sure they are suppressed. It is also set to -MINIMUM\_PERCENT on the DON'T KNOW and REFUSED to make sure they do print even if they do not have many mentions.

**NOTE:** This table is also ranked to highlight the fact that no row with under 5 percent mentions is printed.

## ADVANCED TABLES

### 6.7 LONG BRAND LISTS

~DEFINE

TABLE\_SET= TAB703:

HEADER=:

COLLAPSING LOW MENTIONS INTO AN ALL OTHER CATEGORY ON A  
LARGE BRAND LIST}

LOCAL\_EDIT=: MINIMUM\_PERCENT=5,RANK\_LEVEL=1}

STUB=:

CHRYSLER

FORD

GENERAL MOTORS

[MINIMUM\_PERCENT=100] OTHER DOMESTIC

ALFA ROMEO

JAGUAR

RANGE ROVER

ROLLS-ROYCE

STERLING

[MINIMUM\_PERCENT=100] OTHER BRITISH

BMW

MERCEDES

VOLKSWAGEN/PORSCHE/AUDI

[MINIMUM\_PERCENT=100] OTHER GERMAN

FIAT

PEUGEOT

SAAB

VOLVO

YUGO

[MINIMUM\_PERCENT=100] OTHER EUROPEAN

HONDA

MAZDA

```

NISSAN
SUBARU
TOYOTA
[MINIMUM_PERCENT=100] OTHER JAPANESE
HYUNDAI
[MINIMUM_PERCENT=100] OTHER ASIAN
[PRINT_ROW=SUPPRESS,RANK_LEVEL=1L
-MINIMUM_PERCENT] ALL OTHER COMPANIES
[RANK_LEVEL=1L, -MINIMUM_PERCENT ] NONE
[RANK_LEVEL=1L, -MINIMUM_PERCENT ] DON'T KNOW }
ROW=: [11.3^1//30]
}

~EXECUTE
TABLE_SET= TAB703

```

Here is the table that is printed.

COLLAPSING LOW MENTIONS INTO AN ALL OTHER CATEGORY ON A  
LARGE BRAND LIST

TABLE 703  
AUTOMOBILE MANUFACTURERS

	SEX		
	<----->		
TOTAL	MALE	FEMALE	
----	----	-----	
TOTAL	200	109	91
	100.0%	100.0%	100.0%

**ADVANCED TABLES**

*6.7 LONG BRAND LISTS*

NISSAN	23	11	12
	11.5%	10.1%	13.2%
HONDA	21	11	10
	10.5%	10.1%	11.0%
CHRYSLER	20	13	7
	10.0%	11.9%	7.7%
GENERAL MOTORS	20	10	10
	10.0%	9.2%	11.0%
VOLVO	16	8	8
	8.0%	7.3%	8.8%
VOLKSWAGEN/PORSCHE/AUDI	13	9	4
	6.5%	8.3%	4.4%
TOYOTA	13	7	6
	6.5%	6.4%	6.6%
FORD	12	5	7
	6.0%	4.6%	7.7%
MERCEDES	11	6	5
	5.5%	5.5%	5.5%
ALL OTHER COMPANIES	95	49	46
	47.5%	45.0%	50.5%

NONE	37	20	17
	18.5%	18.3%	18.7%
DON ' T KNOW	22	15	7
	11.0%	13.8%	7.7%

### 6.7.5 Printing Subtotal Rows

Five subtotal rows are maintained by the system, which can help you create summary rows, or other types of rows on the printed table. You can store the information of any row into a given subtotal by using either the stub option `SUBTOTAL#` or `KEEP_SUBTOTAL#`. The # can be a number 1 to 5 or blank (the default) which will use subtotal 1. `SUBTOTAL#` will only add this row into the `SUBTOTAL`, while `KEEP_SUBTOTAL` will continue to add rows into the subtotal until another subtotal stub option is used. A given row can be added into multiple subtotal rows. To print a subtotal you need to use the `PRINT_ROW` option and set it equal to either `SUBTOTAL#_CLEAR` or `SUBTOTAL#_NO_CLEAR`. If you set `PRINT_ROW` equal to `SUBTOTAL#_CLEAR` it prints and clears the subtotal value while `SUBTOTAL#_NO_CLEAR` will print it and not clear it. The default is to clear any subtotal when you print it.

The following is an example of using two different subtotal options to create summary rows at the bottom of the table.

```
~DEFINE
TABLE_SET= TAB704:
HEADER=: USING SUBTOTALING OPTIONS TO PRODUCE NET
CATEGORIES }
TITLE=: RATING OF BRAND X }
```

ADVANCED TABLES

6.7 LONG BRAND LISTS

```

STUB= :
[SUBTOTAL1] VERY GOOD
[SUBTOTAL1] GOOD
FAIR
[SUBTOTAL2] POOR
[SUBTOTAL2] VERY POOR
DON'T KNOW
[PRINT_ROW=SUBTOTAL1] SUBTOTAL GOOD
[PRINT_ROW=SUBTOTAL2] SUBTOTAL POOR }
ROW=: [21^5//1/X] }

~EXECUTE
TABLE_SET= TAB704

```

Here is the table that is printed.

USING SUBTOTALING OPTIONS TO PRODUCE NET CATEGORIES

TABLE 704

RATING OF BRAND X

	SEX		
	TOTAL	MALE	FEMALE
TOTAL	200	109	91
	100.0%	100.0%	100.0%
VERY GOOD	33	21	12

	16.5%	19.3%	13.2%
GOOD	57 28.5%	27 24.8%	30 33.0%
FAIR	53 26.5%	36 33.0%	17 18.7%
POOR	32 16.0%	14 12.8%	18 19.8%
VERY POOR	11 5.5%	5 4.6%	6 6.6%
DON'T KNOW	14 7.0%	6 5.5%	8 8.8%
SUBTOTAL GOOD	90 45.0%	48 44.0%	42 46.2%
SUBTOTAL POOR	43 21.5%	19 17.4%	24 26.4%

Another possible use of the subtotalling feature is to use it to print a row twice on a given table. This can be useful for printing multiple vertical percentages for each row in the table. The example below demonstrates printing a vertical percentage both of the total row and those who rated the brand.

## ADVANCED TABLES

### 6.7 LONG BRAND LISTS

~DEFINE

TABSET= TAB705:

HEADER=: USING SUBTOTALING OPTIONS TO PRINT MULTIPLE  
VERTICAL PERCENTAGES }

TITLE=: RATING OF BRAND X }

STUB=:

[SUBTOTAL1, VERTICAL\_PERCENT=T] THOSE WHO RATED BRAND X

[PRINT\_ROW=SUBTOTAL1, SKIP\_LINES=0, -FREQUENCY,  
VERTICAL\_PERCENT=1]

[SUBTOTAL1, VERTICAL\_PERCENT=T] VERY GOOD

[PRINT\_ROW=SUBTOTAL1, SKIP\_LINES=0, -FREQUENCY,  
VERTICAL\_PERCENT=1]

[SUBTOTAL1, VERTICAL\_PERCENT=T] GOOD

[PRINT\_ROW=SUBTOTAL1, SKIP\_LINES=0, -FREQUENCY,  
VERTICAL\_PERCENT=1]

[SUBTOTAL1, VERTICAL\_PERCENT=T] FAIR

[PRINT\_ROW=SUBTOTAL1, SKIP\_LINES=0, -FREQUENCY,  
VERTICAL\_PERCENT=1]

[SUBTOTAL1, VERTICAL\_PERCENT=T] POOR

[PRINT\_ROW=SUBTOTAL1, SKIP\_LINES=0, -FREQUENCY,  
VERTICAL\_PERCENT=1]

[SUBTOTAL1, VERTICAL\_PERCENT=T] VERY POOR

[PRINT\_ROW=SUBTOTAL1, SKIP\_LINES=0, -FREQUENCY,  
VERTICAL\_PERCENT=1]

[SUBTOTAL1, VERTICAL\_PERCENT=T] DON'T KNOW }

T5=: \N

FIRST PERCENTAGE IS OFF OF TOTAL RESPONDENTS

SECOND PERCENTAGE IS OFF THOSE WHO RATED THE BRAND }

ROW=: [21^1-5/5//1/X]

}





~EXECUTE  
TABSET= TAB705

Here is the table that is printed.

USING SUBTOTALING OPTIONS TO PRINT MULTIPLE VERTICAL  
PERCENTAGES

TABLE 705  
RATING OF BRAND X

	SEX		
	<----->		
	TOTAL	MALE	FEMALE
	-----	-----	-----
TOTAL	200	109	91
	100.0%	100.0%	100.0%
THOSE WHO RATED BRAND X	186	103	83
	93.0%	94.5%	91.2%
	100.0%	100.0%	100.0%
VERY GOOD	33	21	12
	16.5%	19.3%	13.2%
	17.7%	20.4%	14.5%
GOOD	57	27	30
	28.5%	24.8%	33.0%

## ADVANCED TABLES

### 6.8 MASTER-TRAILER PROCESSING

	30.6%	26.2%	36.1%
FAIR	53	36	17
	26.5%	33.0%	18.7%
	28.5%	35.0%	20.5%
POOR	32	14	18
	16.0%	12.8%	19.8%
	17.2%	13.6%	21.7%
VERY POOR	11	5	6
	5.5%	4.6%	6.6%
	5.9%	4.9%	7.2%
DON'T KNOW	14	6	8
	7.0%	5.5%	8.8%

FIRST PERCENTAGE IS OFF OF TOTAL RESPONDENTS

SECOND PERCENTAGE IS OFF THOSE WHO RATED THE BRAND

## 6.8 MASTER-TRAILER PROCESSING

When you collect general information in a master questionnaire, collect additional information in trailer questionnaires, and then generate tables from those related questionnaires, it is called Master-Trailer processing. In the following example, household information has been collected in the master questionnaire, and then information about each person and the trips they take out of the house has been collected in the trailer questionnaire. Using Master-Trailer processing, you can generate tables based on households, individuals, or a combination of information from both groups.

This example uses two data files: one for household information and one for individual trip destinations during a test week. Household information includes the home address, which is the origin of all trips. The destinations of up to four persons in the household, a maximum of three trip destinations per person has been recorded in the trailer questionnaire. The number weekday and weekend trips to each destination has also been recorded.

This example uses the spec file trips3.spx that creates three tables. Information about the household can be reported and combined with destination data using both data files as both files have a Household ID# as part of the Case ID#. The data files are simple and concise and contain information about only those destinations actually visited.

The data files for the trips example are organized as follows:

The hhdata.tr (household data file) is 80 columns long.

<u>Item</u>	<u>Location</u>
Household ID#	1.5
Origin of Trip	50.4

The indata.tr (individual destination data file) is 240 columns long.

<u>Item</u>	<u>Location</u>
Household ID#	1.5
Person #	6
Trip #	7
Destination	130.4
Number of Weekday Trips	231.2
Number of Weekend Trips	233.2

Important keywords in this example include:

- `~INPUT $ studyname=<name>`

The phantom `~INPUT` statement uses a "studyname" so that the phantom file can assume the `CASE_WIDTH`, `WORK_LENGTH`, `TOTAL_LENGTH`, `TEXT_WIDTH`, `TEXT_START`, and `CASE_START` from the input file with that "studyname".

- `STUDYNAME![data description]`

This refers to a location or variable in a specific input file associated with the studyname. There is also `STUDYNAME^[data description]` for identifying locations or variables in a specific dbfile.

- `READ_PROCEDURE=<name> ON studyname`

This executes the specified procedure on the input file associated with the studyname.

- `WHILE MATCHING "indata" indata![1.5$]`

This command lets you do hierarchical jobs (such as master-trailer processing) on sets of files where the secondary files are sorted in order by the relevant match fields. Here is how `WHILE MATCHING` is used in the following example:

- 1 Start with the primary file (hhdata).
- 2 Do a choosefile on the secondary file (indata), which is the file associated with the studyname "indata".
- 3 Starting with the first case look for the matching field `indata![1.5$]` to match `hhdata!case_id`. The matching field can be any expressions resulting in a string (for example, `studyname![1.5$] JOIN studyname![8$]`).
- 4 If the matching field is less, then read forward in the secondary file. If it is greater, then quit the loop. This only works if the file given by studyname is sorted by the matching field.



- 5 Now execute the interior of the WHILE .... ENDWHILE for every case that passes any SELECT= on the ~INPUT statement and also has a matching field with the same CASE\_ID in the primary file.
- 6 Do a choosefile back to the primary file and go on.

ADVANCED TABLES  
6.8 MASTER-TRAILER PROCESSING

```
~COMMENT trips3.spx

~DEFINE

PROC={proc1:

    WHILE MATCHING "indata" indata![1.5$]

        COPY inwork![130.4] = indata![130.4]           ''destinations
        COPY inwork![231.2] = indata![231.2]           ''total weekday trips
        COPY inwork![233.2] = indata![233.2]           ''total weekend trips

        DO_TABLES

    ENDWHILE

}

trips: CFUNC(-34,&
inwork![130.4#707/708/726/727/728/729/730/746/747/748/780/783/784])

TABSET=&
{global:
    FOOTER={:
        =Page #page_number#
    }
    HEADER={:
        =1995 County Transit Survey
        December 1995\n
    }
    GLOBAL_EDIT={:
        -COLTNA
        CALL_TABLE=""
        -TCON
        PDEC=0
        PUTCHAR=-z--
    }
}
```



```

                SUPPRESS_ROWS_BASE=1
            }
    }

TABSET=&
{ban1:
    EDIT={:
        CWIDTH=5
        SWIDTH=24
    }
    BANNER={:
        |
        |                               Origins
        |=====
        |Total  707  708  725  726  727  729  730  746  780  783
        |-----
        |
    }
    COL=: NET([50.4#707/708/725/726/727/729/730/746/780/783])
}

```

```

TABSET=&
{qwkday:
    LOCAL_EDIT={:
        MINFREQ=1
        -ROWTNA
        -VPER
        SKIP=0
    }

    TITLE={:
        Trips (one-way) from Glenview Estates Area\n
    }

    TITLE_4={:
        Total weekday trips
    }

    STUB={:
        Total trips
    }
}

```

ADVANCED TABLES  
6.8 MASTER-TRAILER PROCESSING

```
[comment, underline, skiplines=1] Destinations within Glenview
Estates
    [stub_indent=4] 707
    [stub_indent=4] 708
    [stub_indent=4] 726
    [stub_indent=4] 727
    [stub_indent=4] 728
    [stub_indent=4] 729
    [stub_indent=4] 730
    [stub_indent=4] 746
    [stub_indent=4] 747
    [stub_indent=4] 748
    [stub_indent=4] 780
    [stub_indent=4] 783
    [stub_indent=4] 784
                                Outside Glenview Estates
    }
ROW=: trips * inwork![231.2]
}

TABSET=&
{qwkend=qwkday:
    TITLE_4={:
        Total weekend trips
    }
    ROW=: trips * inwork![233.2]
}

TABSET=&
{qttotal=qwkday:
    TITLE_4={:
        Total trips
    }
    ROW=: trips * (inwork![231.2] ++ inwork![233.2])
}
```



```
~INPUT hhdata      STUDY_NAME=hhdata  NUMBER_INPUT_BUFFERS=3
~INPUT indata      STUDY_NAME=indata  NEW_BUFFER
~INPUT $           STUDY_NAME=inwork  NEW_BUFFER

>PRINTFILE trips3 PAGE_WIDTH=80

~EXC

TABSET=global

READ_PROC=proc1 on hhdata
TABSET=ban1
  TABSET=qwkday      TAB=*
  TABSET=qwkend      TAB=*
  TABSET=qttotal     TAB=*

~END
```

ADVANCED TABLES  
 6.8 MASTER-TRAILER PROCESSING

1995 County Transit Survey  
 December 1995

Trips (one-way) from Glenview Estates Area

Total weekday trips

	Origins										
	Total	707	708	725	726	727	729	730	746	780	783
Total trips	561	47	92	18	16	28	33	133	30	159	5
Destinations within Glenview Estates											
707	17	-	-	-	-	-	-	-	-	17	-
708	3	-	-	-	-	-	-	3	-	-	-
726	5	-	-	-	-	-	-	-	-	5	-
727	7	-	7	-	-	-	-	-	-	-	-
729	15	-	10	-	-	-	-	3	-	2	-
746	5	-	-	-	-	-	-	5	-	-	-
747	12	-	8	-	-	-	-	-	-	4	-
Outside Glenview Estates	497	47	67	18	16	28	33	122	30	131	5



1995 County Transit Survey  
December 1995

Trips (one-way) from Glenview Estates Area

Total weekend trips

	Origins										
	=====										
Total	707	708	725	726	727	729	730	746	780	783	
	-----										
Total trips	70	9	20	2	2	4	2	10	5	16	-
Destinations within											
Glenview Estates											
-----											
707	3	-	-	-	-	-	-	-	-	3	-
726	1	-	-	-	-	-	-	-	-	1	-
727	4	-	4	-	-	-	-	-	-	-	-
729	4	-	3	-	-	-	-	-	-	1	-
746	2	-	-	-	-	-	-	-	2	-	-
747	1	-	1	-	-	-	-	-	-	-	-
780	2	-	2	-	-	-	-	-	-	-	-
Outside Glenview Estates	53	9	10	2	2	4	2	10	3	11	-

ADVANCED TABLES  
 6.8 MASTER-TRAILER PROCESSING

1995 County Transit Survey  
 December 1995

Trips (one-way) from Glenview Estates Area

Total trips

	Origins										
	Total	707	708	725	726	727	729	730	746	780	783
Total trips	631	56	112	20	18	32	35	143	35	175	5

Destinations within  
 Glenview Estates

-----											
707	20	-	-	-	-	-	-	-	-	20	-
708	3	-	-	-	-	-	-	3	-	-	-
726	6	-	-	-	-	-	-	-	-	6	-
727	11	-	11	-	-	-	-	-	-	-	-
729	19	-	13	-	-	-	-	3	-	3	-
746	7	-	-	-	-	-	-	5	2	-	-
747	13	-	9	-	-	-	-	-	-	4	-
780	2	-	2	-	-	-	-	-	-	-	-
Outside Glenview Estates	550	56	77	20	18	32	35	132	33	142	5

# TABLES VIEWABLE THROUGH A BROWSER

## INTRODUCTION

This chapter explains how to customize Mentor specs to produce HTML output files that allow you to view tables through a Web browser. It includes information about WebTables, Cascading Style Sheets (CSS), and On-Demand tables. This chapter assumes that there is a basic understanding of tables and CfMC terminology. If you are not familiar with creating tables, review Chapters 4, 5, and 6.

At the end of this chapter there is a section on how to simultaneously create a print file, a delimited file, and an HTML file. For more information, go to “Preparing Mentor Output Files For Post Processing”

**NOTE:** Any references to “Class” statements in this chapter pertain to use of style sheets. This option is available in Mentor 7.7 or above. If using prior versions, the alternate commands must be used.

For information on Dynamic Charts, contact CFMC Customer Support at [support@cfmc.com](mailto:support@cfmc.com).

## WebTables Overview

The HTML (.htm) files created by Mentor must be placed in a Web-accessible directory. The .htm files created by Mentor must be created in or moved to this Web-accessible directory

Make sure that you have JavaScript enabled in your browser. The default in most browsers will enable JavaScript. However, if you would like to make sure that this default is turned on:

- To check this in your Netscape Browser, go to File Preferences Enable JavaScript (there are three boxes to check).

- In IE, go to **Tools** ==> **Internet Options** ==> **Advanced**, and check box **JIT Compiler -enabled**. These steps will vary with different browser versions.
- In Mozilla, Firefox, go to **Tools** ==> **Options** ==> **Web Features**, check **Enable Java, JavaScript** boxes.
- In Safari, click on **Preferences, Security**, click on **Enable Java** and **JavaScript** boxes.

## BROWSER COMPATIBILITY

You can use any version of IE 5 or above or Netscape 4 or above to view WebTables. Versions 6.0 or higher are the only browsers that will function fully.

The following sections describe basic and then complex WebTables.

## EXAMPLE MENTOR FILES AVAILABLE ONLINE

For an example of Mentor files, go to <http://distrib.cfmc.com/hints/webtab77/>. The files on this website are based on the Roadrunner survey that can be found in CfMC's Mentor and Survent example directories.

Certain aspects of the specs are explained within the files. The explanations are offset by two single quotation marks ("). These are commonly used in CfMC files. It is known as "commenting out" text or code.

## 7.1 Basic WebTables

Using Mentor enables you to transform plain ASCII tables into tables with HTML enhancements to produce user-friendly tables on the Internet, called WebTables.

## WRITING SPECS

Mentor produces the enhanced HTML code that allows the tables to be viewed through a browser. Included in this chapter is a Mentor .spx file that contains all of the necessary coding to create colorful web tables. Notes explaining the code or portions of code will be offset with ' ' (two single quotes).

## MANAGING YOUR SPECS

In general, most web table-related commands can go in the `~set` block or the `~edit` statement within your spec file. However, **colinfo** and **tcon options** MUST go in the `~edit` statement. In addition,

```
~set
web_tables= (
css_file_check=/www/htmldocs/cfmcweb/css/wmentor.css
css_file_path="http://product01.cfmc.com/cfmcweb/css/wmentor.css"
on Resize="resetColors()"
)
```

MUST be put in a `~set` block.

**NOTE:** Useful hint - using the full path for the CSS makes it easier to use the same specs from one job to another.

## USING SET COMMANDS IN TABS.SPX

```
~SET
WEB_TABLES=(
css_file_check=/www/htmldocs/cfmcweb/css/wmentor.css
css_file_path="http://product01.cfmc.com/cfmcweb/css/wmentor.css"
bgcolor=#F0F8FF "is best set in the body statement on the css
web_format_ban=(class=banner) "below is the alternative not using a CSS
" or web_format_ban=(bgcolor=lime font=(color=red size=1))
web_format_stub=(class=stub)
web_format_freq=(class=freq)
```

```

web_format_vper=(class=vper
web_format_stat=(class=stat)
web_format_comment=(class=comm)
web_format_hper=(class=hper)

***** rows alternate lime/red for bgcolor*****
" web_format_odd_row=( bgcolor=red)
" web_format_even_row=( bgcolor=lime)

WEB_FORMAT_TABLE=(
" The following commands are all handled in the body statement in the css
"   class=vd10 "font=(face="verdana" size=2)
"   border=10
"   width=100% " or width=200
"   cellspacing=1%
"   cellpadding=1%
"   bordercolor=#333366
"   )
" )
)

```

### GLOBAL EDIT STATEMENTS IN THE TABS.SPX

```

tabset=global:
global_edit=-:coltna
tcon=(first,
      web_format_tcon_anchor=(class=TocBody)
      web_format_tcon_anchor=(class=Toctabnum)
      )
RANK_IF_INDICATED
RANK_COL_BASE=1
PDEC=1
SDEC=2
PUTCHARS=-Z-
-t4base
stats_on_separate_line
minfreq=1
dostats=.90 ALL_POSSIBLE_PAIRS_TEST
running_lines=1
}

```



```

***** cols alternate size *****
" Colinfo DONE IN EDIT STATEMENT in banner_a.def
" web_format_odd_col=( bgcolor=lime font=(color=red size=5))
" web_format_even_col=( bgcolor=pink font=(color=lime size=4))

```

***For colinfo:***

```

colinfo=(
  c=1 webformat=(bgcolor=lime) font=(size=5)) /
  c=2 webformat=(bgcolor=yellow) /
  c=3=2 /
  c=4=2 /
  c=5 webformat=(bgcolor=lime) /
  c=6=5 /
  c=7=5 /
  c=8 webformat=(bgcolor=yellow) /
  c=9=8
)

```

When setting up your Web banner, the following will produce .prt, .dlm and .htm banners.

```

BANNER=: make_banner
[      level=2]
[underline=- level=1]      Total
[underline== level=2 colspan=3] AGE
[underline=- level=1]      Under 35
[underline=- level=1]      35 - 54
[underline=- level=1]      Over 54
[underline== level=2 colspan=3] INCOME
[underline=- level=1]      Under $15k
[underline=- level=1]      $15 - $35
[underline=- level=1]      Over $35k
[underline== level=2 colspan=2] RATING
[underline=- level=1]      Good
[underline=- level=1]      Neutral /Poor

```

```

}
COLUMN=: TOTAL WITH &
  [1/51^1,2/3,4/5,6] WITH &"respondent age
  [1/52^1/2,3/4,5,6] WITH &"income
  [1/47^4,5/1,2,3] "rating
}

```

***Stub Front Options:***

```

[web_format_row=(bgcolor="ccccff") ] Top 2 box
[web_format_row=(bgcolor="ccccff") ] Bottom 2 box
[stat Web_format_stub=(bgcolor="ccfff")]Mean
[stat web_format_stub=(bgcolor="ccfff")]& Standard deviation

```

**Files Needed to Create WebTables**

Similar to a Mentor run in version 7.6, you need a few files in order for your tables to work in 7.7. You will start off with the following files.

- Tabs.spx (Table specs including stub options, etc)
- Banner\_a.def (Specifies all the banner points and edits associated with that particular banner)
- Studyname\_x.def (Defines all question-by-question table definitions)

The banner\_a.def and <studyname>\_x.def can be ampersanded in to your tabs.spx as separate files, or they can be included in the tabs.spx (cut and pasted in).

To simplify the process of putting up WebTables, the file examples in this documentation are broken up into separate files that are ampersanded into the tabs.spx file.

**NOTE:** If there are multiple banners within one run, then all of the banners can be put in the same directory. However, if there are multiple banners with separate runs, placing each banner in separate folders helps keep the job

organized. For example, if there are three banners then you can have folders for each run: ban01, ban02, ban03. Since you will have different output (html,dlm,prt,...) for each banner, you will end up with a large number of files. If you have them in separate folders, it will be easier to maintain file organization.

## Files Produced by the Mentor Run

After you run Mentor, the following files are produced:

- An HTML file - **<studyname>.htm**. This file brings all of the parts of the table together. It calls in all the files that have been created by Mentor to produce the desired tables. Mentor produces files with a .htm extension because, by default, Mentor only allows three-character extensions.

**NOTE:** There is a meta command (>cfmc\_extension htm=html) that can be put in the tabs.spx spec that will produce .html extensions.

- An ASCII print file– **<studyname>.prt** (This is used for downloadable ASCII tables.)
- A comma-delimited file – **<studyname.dlm>** (This can be imported into applications, such as Excel.)

### NOTE FOR WINDOWS/DOS CLIENTS

If Mentor is run on a Windows/DOS PC, then the html files generated by Mentor are created with UPPERCASE names (TAB14.HTM). When these files are transferred to the server, they must be changed to lower-case names.

### WHEN MANUALLY TESTING AND PREPARING TO GO LIVE

After you run all of your specs through Mentor without errors and after your HTM files are generated, then the files are copied to the Linux server or the HP-UX server.

**NOTE:** CfMC spec writers use UltraEdit© for writing specs and putting up tables.

These HTM files must reside in a Web-accessible directory on your server that would have a path, such as this (if CfMC install directions were followed):

`/var/www/html/<studyname>`

### **Basic steps to Running a Live Mentor 7.7 Job**

- 1** Write the specs as if it's a straight Mentor job, adding the necessary web-related commands to the tabs.spx file. This will allow you to produce an htm file.
- 2** The live data must be transferred with fastcopy to a directory where it can be accessed. This step will prevent the corruption of the data set. (Fastcopy is a CfMC SUPER command. For more information, see Chapter 4 of your Survent Manual.)
- 3** Run the tables.
- 4** Once the tables are made, the htm (html) file must be transferred to the Web-accessible directory on the server, unless the tables have been run in that area of the server. They will then be viewable through a Web browser.

### **Setting up Directories and Running WebTables**

CfMC follows a set procedure in setting up directories and running web tables. This procedure is specific to how the machines are set up. Your system administrator will have to tailor your procedure according to the setup of your own machines. CfMC's support team can provide some guidance, if necessary.

### **Automating or Running Tables on a Set Schedule**

In order to run tables automatically, at certain times, you will need to have a cronjob. This would have to be set up by your system administrator.

- 1 A CfMC cron job script is included in this documentation. It follows the example spec files. It is included as a guideline tool. So, in order to automatically run tables, you must:
- 2 Set up the cron job.
- 3 Consider password protecting your tables. (See the Webpass script, which is also included in this document.)
- 4 If more than one banner is used or you would like to access both Web Tables and downloadable ASCII tables, an index page would be a good way to achieve this. (An example of an index.html file is also included.)

## 7.2 Complex WebTables

When creating WebTables, using colors and different fonts in specs can produce state of the art tables. Although sometimes dependent on the browser, creating tables can be done with little effort. Mentor 7.7 has the ability to create these tables with simple commands or by using Cascading Style Sheets (CSS).

The following section will show how these commands can be used.

### BASIC COLORS

<b>Color</b>	<b>Hexadecimal Code</b>
Aqua	"#00FFFF"
Black	"#000000"
Blue	"#0000FF"
Fuchsia	"#FF00FF"
Gray	"#808080"
Green	"#008000"
Lime	"#00FF00"
Maroon	"#800000"
Navy	"#000080"

Olive	"#808000"
Purple	"#800080"
Red	"#FF0000"
Silver	"#C0C0C0"
Teal	"#008080"
Yellow	"#FFFF00"
White	"#FFFFFF"

## HELPFUL INTERNET WEBSITES FOR CHOOSING COLORS

- [http://hotwired.lycos.com/webmonkey/reference/color\\_codes](http://hotwired.lycos.com/webmonkey/reference/color_codes)
- <http://www.zspc.com/color/index-e.html>
- <http://www.visibone.com/colorlab/>
- <http://www.december.com/html/spec/colorsafe.html>
- <http://www.simplythebest.net/info/216color.html>

## SOURCES FOR CSS HELP

There are many sources for help with CSS. Here are a few:

[http://www.devguru.com/Technologies/css/quickref/css\\_index.html](http://www.devguru.com/Technologies/css/quickref/css_index.html)

<http://www.w3schools.com/css/default.asp>

<http://glish.com/css/#tutorials>

<http://hotwired.lycos.com/webmonkey/>

## VALIDATION WEBSITES

There are useful validation websites which include the following:

For HTML: <http://validator.w3.org/>

For CSS: <http://jigsaw.w3.org/css-validator/validator-uri.html>

## Basic Commands with and without Cascading Style Sheet

Without CSS	With CSS	CSS Commands
~set web_tables=(	~set web_tables=(	
	css_file_check=/www/htmldocs/sued/rrunr/css/style.css	Path that mentor uses to validate styles
	css_file_path="/sued/rrunr/css/style.css"	Path that is inserted into html page
html_title="Road Runner Fast Food Sample tables"	html_title="Road Runner Fast Food Sample tables"	
web_format_ban=(bgcolor="ffffff" font=(color=black))	web_format_ban=(class=banner)	.banner { background-color: #ffff00; text-align:center;(default is right) color: #000000; }
web_format_stub=(bgcolor=yellow)	web_format_stub=(class=stub)	.stub { background-color: #ffff00; }
web_format_freq=(bgcolor=yellow font=(color=green))	web_format_freq=(class=freq)	.freq { background-color: #ffff00; color: #008000; }
web_format_vper=(class=vper)	web_format_vper=(class=vper)	.vper { color:#000066; text-align:center; }
web_format_stat=(font=(color=red))	web_format_stat=(class=stat)	.stat { color:red; }
web_format_hp=(bgcolor="ccffff" font=(color=purple))	web_format_hp=(class=hper)	.hper { background-color: #ccffff; color: purple; }
web_format_cumulative_percent=(bgcolor="cccfff")	web_format_stat=(class=cper)	.cper { background-color: #ccffff; }

web_format_comment=( bgcolor="ffffff")	web_format_comment=(class=com m)	.comm { }
web_format_odd_col=(b gcolor=yellow)	web_format_odd_col=(class=oddc ol)	.oddc ol { color:yellow; }
web_format_even_col=( bgcolor=yellow)	web_format_even_col=(class=even col)	.evencol { color:yellow; }
web_format_odd_row=( bgcolor=yellow)	web_format_odd_row=(class=oddr ow)	.oddr ow { color:yellow; }
web_format_even_row= (bgcolor=yellow)	web_format_even_row=(class=eve nrow)	.evenrow { color:yellow; }
bgcolor=aqua font=(face=arial) border= width=90% cellspacing=10 cell padding=20%	The look of the overall tables is set in the body statement. Anything added after that will override the statements in the body statement	body { padding: 2em 3em 2em 3em; border-top: 1.7em solid #006699 ; border-left: 1.3em solid #b0c4de; border-bottom: 1.7em solid 006699; border-right: 1.3em solid #b0c4de; margin: 0; font-family: verdana, sans, arial; font-size: .8em; }

## Example Files and How They Work

The specs in subsequent sections create CfMC’s “Road Runner” demo. An example of a Table of Contents and the table it produces follows the spec files needed to produce them.

Explanations, which can be seen throughout the files, explain certain steps in the specs and the functions that they perform.

### THE TABS.SPX FILE



The tabs.spx file shown below controls the following:

- The ability to turn off frequencies or percentages in the delimited tables.
- The ability to turn off the footer and the header.
- The ability to turn off the tables of contents for comma-delimited tables that are used for many other purposes.

```
>filldef
>PURGESAME
>DEFINE @STUDY rrnr      "* NAME STUDY (REMEMBER CONSISTENCY)

>CREATEDDB tables S=L

~set -varnames

>-printrep

~specfile @STUDY
~SET AUTOTAB drop_local_edit drop_base drop_t4 stat_base_ar
  DELIMITED_TABLES=(delimiter=comma banner -quoted_banner_text
                    stats_on_separate_line)
WEB_TABLES=(
css_file_check=/www/htmldocs/cfmcweb/css/wmentor.css
css_file_path="http://product01.cfmc.com/cfmcweb/css/wmentor.css"
'*****
'PLEASE NOTE: If the background color is set in the banner edit or in the body
'statement in the CSS it will override all other bgcolor statementsw below
'*****
'***** Contents alter colors within the elements of the tables*****
web_format_ban=(class=banner) 'below is the alternative not using a CSS
" web_format_ban=(bgcolor=lime font=(color=red size=1))
web_format_stub=(class=stub)
web_format_freq=(class=freq)
web_format_vper=(class=vper)
web_format_stat=(class=stat)
web_format_comment=(class=comm)
web_format_hper=(class=hper)
```

```

***** rows alternate lime/red for bgcolor*****
" web_format_odd_row=( bgcolor=red)
" web_format_even_row=( bgcolor=lime)

WEB_FORMAT_TABLE=(
" The following commands are all handled in the body statement in the css
"   class=vd10 "font=(face="verdanal" size=2)
"   border=10
"   width=100% " or width=200
"   cellspacing=1%
"   cellpadding=1%
"   bordercolor=#333366
  )
)
~DEFINE

STUB=stubpref1:
[-Vper SKIP_LINES=0 freq]Total Sample
[-VPER]No Answer
[Prt=Ar Vper=* SKIP_LINES=0 freq]Any Response
[comment]
}

STUB=stubpref2:
[supp -vper -freq ] "TOTAL
[SUPP] NO ANSWER
[comment]
}

tabset=global:
global_edit=-:coltna

***** THIS HANDLES TABLE OF CONTENTS APPEARANCE ***
tcon=(first,
" web_format_tcon=(class=TocBody)           "this is for the title portion of toc
" web_format_tcon_anchor=(class=Toctabnum) "this is for the left side (TABLE #) )

```

```
RANK_IF_INDICATED
RANK_COL_BASE=1
PDEC=1
SDEC=2
PUTCHARS=-Z-
-t4base
stats_on_separate_line
minfreq=1
dostats=.90 ALL_POSSIBLE_PAIRS_TEST
}
```

```
footer:=\k(h)<center>\k(h,p)Tables prepared by Computers for Marketing
Corp.\k(h)<br>
\k(h,p)PAGE #PgNum#\k(h)<br>
\k(h,p)LAST UPDATED #DATE# #TIME# \k(h)<br>
```

If you would like to download an ASCII version of these tables to your computer,<br>click "file", then "save as" AFTER

```
<span class="foot"><a href="./@STUDY~.prt">clicking here</a></span></center>
}
}
&banner_a.def
&rrunr_x^def
```

```
~in @STUDY~, DOT=100
```

```
>PRINTFILE @STUDY~
```

```
~EXECUTE
```

```
MAKE_TABLES
```

```
~end
```

The following scripts are used to transfer tables to a different directory

```
"BOB 6/23/03 Everything else will be ignored for now
```

```
~set printtcon "to flush out table of contents (if tcon is on)
```

```

~specfile "to close "specfiles" (e.g., .htm,.dlm)
~
>prtfile "to close the print file
~
"do perform whatever system commands you want with this output
>sys cp @STUDY~.prt /www/htmldocs/<directory_name>
>sys cp @STUDY~*.htm /www/htmldocs/<directory_name>
>sys cp @STUDY~.dlm /www/htmldocs/<directory_name>
~END

```

### THE BANNER\_A.DEF FILE

```

TABSET=BANNER_A:
STUBPREF=STUBPREF1
stat=:I=BCD,I=EFg,I=HI

EDIT=:
  SWID=30
  CWID=7
"colinfo=(
"  c=1 webformat=( bgcolor=#ffffff) /
"  c=2=1 /
"  c=3=1 /
"  c=4=1 /
"  c=5=1 /
"  c=6=1 /
"  c=7=1 /
"  c=8=1 /
"  c=9=1
"  )
}

BANNER=: make_banner
[level=2]    'space holder
[underline=- level=1]    Total
[underline== level=2 colspan=3] AGE
[underline=- level=1]    Under 35
[underline=- level=1]    35 - 54
[underline=- level=1]    Over 54

```

```

[underline== level=2 colspan=3] INCOME
[underline=- level=1] Under $15k
[underline=- level=1] $15 - $35
[underline=- level=1] Over $35k
[underline== level=2 colspan=2] RATING
[underline=- level=1] Good
[underline=- level=1] Neutral /Poor
}
COLUMN=: TOTAL WITH &
    [1/51^1,2/3,4/5,6] WITH & "respondent age
    [1/52^1/2,3/4,5,6] WITH & "income
    [1/47^4,5/1,2,3] "rating
}

```

### THE RRUNR\_X.DEF FILE

```

tabset= { qn1_z:
title=:
Q1. How much do you agree with the following statement: The fast food at Road
    Runners is worth what I pay for it.}
stub=:
(5) Completely agree
(4) Somewhat agree
(3) Neither agree nor disagree
(2) Somewhat disagree
(1) Completely disagree
Don't Know/Refused to answer
[comment]
[ stat] Mean
[stat] Standard deviation
[stat] Standard error}
"qn1(5/4/3/2/1/0)
row=: [1/6.1^5/4/3/2/1/10] &
    $[mean,std,se] [1/6.1 *ranges=1-5]
}

tabset= { qn2a_z:

```

```

title=
  Q2a. Please rate the following characteristics: The quality of the food.}
stub=
  (5) Very good
  (4) Good
  (3) Neither poor nor good
  (2) Poor
  (1) Very poor
  Don't know/refused to answer
  [comment]
  [ stat] Mean
  [stat] Standard deviation
  [stat] Standard error}
"qn2a(5/4/3/2/1/0)
row=: [1/7.1^5/4/3/2/1/10] &
  $[mean,std,se] [1/7.1 *ranges=1-5]
  }

tabset= { qn2b_z:
title=
  Q2B. Please rate the following characteristics: The quality of service.}
" SAMEAS QN2A
stub=qn2a_s
"qn2b(5/4/3/2/1/0)
row=: [1/8.1^5/4/3/2/1/10] &
  $[mean,std,se] [1/8.1 *ranges=1-5]
  }

tabset= { qn2c_z:
title=
  Q2C. Please rate the following characteristics: The cleanliness of the restaurant.}
" SAMEAS QN2A
stub=qn2a_s
"qn2c(5/4/3/2/1/0)
row=: [1/9.1^5/4/3/2/1/10] &
  $[mean,std,se] [1/9.1 *ranges=1-5]
  }

tabset= { qn2d_z:

```

```
title=:
  Q2D. Please rate the following characteristics: The prices on the menu.}
" SAMEAS QN2A
stub=qn2a_s
"qn2d(5/4/3/2/1/0)
row=: [1/10.1^5/4/3/2/1/10] &
  $[mean,std,se] [1/10.1 *ranges=1-5]
}

tabset= { qn2e_z:
title=:
  Q2E. Please rate the following characteristics: The accuracy of your bill.}
" SAMEAS QN2A
stub=qn2a_s
"qn2e(5/4/3/2/1/0)
row=: [1/11.1^5/4/3/2/1/10] &
  $[mean,std,se] [1/11.1 *ranges=1-5]
}

tabset= { qn2f_z:
title=:
  Q2F. Please rate the following characteristics: The cleanliness of the restrooms.}
" SAMEAS QN2A
stub=qn2a_s
"qn2f(5/4/3/2/1/0)
row=: [1/12.1^5/4/3/2/1/10] &
  $[mean,std,se] [1/12.1 *ranges=1-5]
}

tabset= { qn2g_z:
title=:
  Q2g. What is your opinion of the Road Runners billing format? Is it...}
stub=:
  Easy to understand
  Hard to understand
  Neither easy nor hard to understand
  Don't know}
"qn2g(1/2/3/0)
row=: [1/13.1^1/2/3/10]
```

}

Those are the three files that you have to set up properly and compile to test for errors. The rest of the files are generated by Mentor.

**THE INDEX.HTML FILE**

The index.html file is created by the programmer. It provides access to both the WebTables and downloadable ASCII tables within a browser if you choose to set it up this way. The file’s source code follows. This is for a single-bannered table.




---

## CFMC DEMO FOR TABULATION PRODUCTS

### Road Runner Study

ON-LINE TABLES	<a href="#">Web Tables</a>	<a href="#">Quick upload/Printable</a>
DYNAMIC CHARTS	<a href="#">Web Tables with Dynamic Charts</a>	<a href="#">Quick Overview of how to use Dynamic Charts</a>
ON-LINE TABLES	<a href="#">Excel Import No Header or Footer</a>	<a href="#">Excel Import</a>
ON-DEMAND TABLES	<a href="#">Web Tables On-Demand</a>	<a href="#">Quick Overview of how to use On-Demand tables</a>

---

This is what the index page looks like through a browser.





```
</div>  
</div>  
<br /><br />  
<div class="hdivide">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</div>  
</div>  
</body>  
</html>
```

**EXAMPLE OUTPUT OF A WEBTABLE**

In this example, Table 001 was chosen, and it produced a nicely organized table.

**TABLE 001**

Q1. How much do you agree with the following statement: The fast food at Road Runners is worth what I pay for it.

	Total	AGE			INCOME			RATING	
	501	Under 35	35 - 54	Over 54	Under \$15k	\$15 - \$35	Over \$35k	Good	Neutral/Poor
Total Sample	501	160	132	148	55	172	204	165	242
	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)
Any Response	501	160	132	148	55	172	204	165	242
	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
(5) Completely agree	77	34	20	24	4	24	34	22	34
	15.4%	16.0%	15.2%	16.2%	7.3%	14.0%	16.7%	13.3%	14.0%
(4) Somewhat agree	92	27	19	36	16	36	28	32	47
	18.4%	16.9%	14.4%	24.3%	29.1%	20.9%	13.7%	19.4%	19.4%
(3) Neither agree nor disagree	94	34	23	22	10	37	36	33	47
	18.8%	21.3%	17.4%	14.9%	18.2%	21.5%	17.6%	20.0%	19.4%
(2) Somewhat disagree	90	31	27	24	9	29	38	27	40
	18.0%	19.4%	20.5%	16.2%	16.4%	16.9%	18.6%	16.4%	20.2%
(1) Completely disagree	76	22	20	23	10	22	33	28	32
	15.2%	13.8%	15.2%	15.5%	18.2%	12.8%	16.2%	17.0%	13.2%
Don't Know/Refused to answer	72	22	23	19	6	24	35	23	33
	14.4%	13.8%	17.4%	12.8%	10.9%	14.0%	17.2%	13.9%	13.6%
Mean	3.01	3.00	2.93	3.11	2.90	3.07	2.95	2.96	3.01
Standard deviation	1.36	1.33	1.38	1.39	1.29	1.30	1.41	1.35	1.32
Standard error	0.07	0.11	0.13	0.12	0.18	0.11	0.11	0.11	0.09

(sig=.10) (all\_pairs) columns tested BCD, EFG, HI

Tables prepared by Computers for Marketing Corp.

PAGE 1

LAST UPDATED 29 JAN 2004 9:18 AM

If you would like to download a flat ASCII version of these tables to your computer, click "file", then "save as" AFTER clicking here

[BACK TO TABLE OF CONTENTS](#)

**Custom CfMC Scripts**

**THE WEBPASS SCRIPT**

This script was written to give clients the ability to password protect real-time results . This script is included with the install and instructions for customizing this script are provided in the following sections.

**Preparation**

Edit your httpd's access.conf (or httpd.conf if access.conf is no longer used) and alter "AllowOverride" in <Directory (\*\* html doc home \*\*)> </Directory> block to be "AllowOverride AuthConfig".

**NOTE:** Make sure that `httpd.conf` has all of the `ServerNames` listed:

**Example:**

```
ServerName pd1.cfmc.com
```

```
    ServerName www.survey1.cfmc.com
```

This will stop the password screen from coming up multiple times.

Place this file in any directory in your `$PATH` (e.g., `"/usr/local/bin"`)

**Usage:**

```
"webpass <document name> <subdirectory of main html document directory>"
```

or

```
"webpass" (and follow the prompts)
```

Once this is run for a particular directory, it does not have to be run again. Replace documents in or copy new documents into the password-protected web document directory.

You may need to change the following variables:

```
$docloc = "/var/www/html"; change this to your main html doc location
```

```
$authname = "Password Protected Location"; Appears on username/password screen
```

```
$passfile = ".htpasswd"; Name of file for passwords; this will be in document directory.
```

Once you have customized your script, run it and set up a username and password, you can then e-mail the client with the url where the html docs reside and their username and password. The script example follows.

## 7.3 ON-DEMAND TABLES

The On-Demand utility enables you to perform advanced data manipulation, such as performing quick cross tabs between two variables. For example, any question in a study can be crossed with any other question. On-Demand is very beneficial for analyzing a study.

There is a simple command that activates the utility on the server where the study is running. If the study is not running on the server, the .qpx and .tr can be placed in a Web-accessible directory. Both options are easy to use.

### 7.3.1 Installation Requirements

If the software was properly installed, a cfmcweb directory will be created in the Web-accessible area (html document directory) of the server. This directory will probably be /var/www/html/ (CfMC's default), but check with your system administrator if you do not know what it is. This cfmcweb directory will contain subdirectories containing CfMC-specific applications.

There are two subdirectories in cfmcweb that pertain to the creation of On-Demand Tables:

**1** /var/www/html/cfmcweb/php/ondemand: holds php scripts. The files are:

- makedb.php - php script for setting up On-Demand presentation screen
- maketab.php - php script for making tables
- cleanup.php - cleans up tables directory files that were created during the run that are no longer needed.

**NOTE:** You should NEVER edit these files

**2** /var/www/html/cfmcweb/css: for html cascading style sheets. CfMC has included:

- ondemand.css

**NOTE:** You may add your own .css files to this directory based on the structure of ondemand.css. It is not advisable to edit the ondemand.css, the default style sheet. If Mentor 7.7+ is reinstalled, it will overwrite ondemand.css. Instead, rename the file and make changes to the looks and options in this new file. It can then be brought in on the command line during the original setup using the following syntax: /var/www/html/rrunr/css/<cssfilename>.css. If a style sheet is added later, the “ondeminit” can be edited to reflect that change.

Likewise, there will also be a cfmc.cgi subdirectory under the main cgi-bin directory in the Web-accessible area. The Web server default location for this is /var/www/cgi-bin/cfmc.cgi. But, once again, your system administrator should know the actual location. There will be one file in this cfmc.cgi subdirectory, a perl script "ondem.pl". You should NEVER edit this file.

### 7.3.2 Installation requirements

The following conditions must be met in order to use On-Demand Tables:

- A UNIX platform with a Web server that supports php (v4.3.0+).
- CfMC Mentor 7.7+ must be installed on the server where the On-Demand Tables will run.

#### SETUP

**Remember that the example specs provided here show the way it is done at CfMC.** They are just suggestions to help you run the tables that will be viewed through the browser.



There are two possible scenarios for running On-Demand Tables.

- 1 1 - A CfMC cron job script is included in this documentation. It follows the example spec files. It is included as a guideline tool. So, in order to automatically run tables, you must:
  - The Web survey is running on the server
  - .qpx (possibly a .def) is on the server
  - data is on the server
  - A Web area exists
- 2 2 - The study is not running on the server

Create a directory in a Web-accessible area of the server (ex. /var/www/html/rrunr/ondemand/test)

Place the .qpx or .def and data file in that directory or subdirectory.

Place any auxiliary files needed (webtab.add) in that directory

Place data in that directory or somewhere on the server where it can be accessed.

In the directory where the .qpx/.def file type ==>ondemand.pl

The following screen will appear:

#### Welcome to On-Demand

#### COMMAND LINE USAGE:

```
/cfmc/dev.80/go/ondemand.pl <specfile> (MUST BE .qpx or .def) <option1>  
<option2>...
```

Any or all of the following options may be used on the command line:

```
data=<fully qualified data file name (def:/cfmc/dev.80/data/_STUDY_.tr)  
wwwloc=<web location of tables> (def:/var/www/html/_STUDY_/ondemand)  
cssfile=<fully qualified css file name  
(def:/var/www/html/cfmcweb/css/ondemand.css)  
add[=filename (def:webtab.add)]  
batch
```

**setpass=<yes/no>**

**debug**

**Please enter a spec file name (or press Enter to quit)**

**(NOTE: This MUST have a .qpx or .def file extension)==>rrunr.def**

**NOTE:** The.qpx or .def extension must be used

**NOTE:** If at the original prompt (/var/www/html/rrunr), you typed

**==>ondemand.pl rrunr.qpx**, the previous screen would be skipped.

Next, the default data location can be changed.

**Please enter the Data File, i.e., the fully qualified name of the data file that your tables will use**

**Press <Enter> for /cfmc/dev8.0/data/rrunr.tr**

**==>/var/www/html/rrunr/tables/rrunr.tr**

Next, you will get the prompt for the Web-accessible directory (public area):

**Please enter the Tables Location, i.e., the location in the web area for your tables**

**Press <Enter> for /var/www/html/rrunr/ondemand**

**==>/var/www/html/rrunr/ondemand/test**

If the directory does not have proper read/write access, the following warning will appear.:

**WARNING: Permissions for web location**

**/var/www/html/rrunr/ondemand/test are 775 (drwxrwxr-x).**



**They should be 777**

**Press enter to set permission to 777 or break to get out**

Next, the prompt for the style sheet that will be used appears:

**Please enter the CSS File, i.e., the fully qualified name of the style sheet (css) file**

**Press <Enter> for /var/www/html/cfmcweb/css/ondemand.css**

**==>/var/www/html/rrunr/ondemand/test/ondemand2.css**

**NOTE:** You may add your own .css files, based on the structure of ondemand.css, to this directory. It is not advisable to edit the ondemand.css (the default style sheet). If Mentor 7.7+ is reinstalled, it will overwrite ondemand.css. Instead, rename the file and make changes to the looks and options in this new file. . It can then be brought in on the command line during the original setup using the following syntax:  
/var/www/html/rrunr/css/<cssfilename>.css. If a style sheet is later added the "ondeminit" can be edited to reflect that change. It can then be brought in on the command line.

The following prompt will appear. If an add file with additional specs is to be used, type "y" at the prompt:

Do you have a file with additional spec that you would like to add to rrunr.def?

(y/(n))=>y

**Please enter the this file's name (enter for webtab.add)==> <enter>**

The following will appear:

**NOTE:** t the first running of ondemand.pl, a file called "ondeminit" will be placed in the \${CFMC}control/ directory. This will contain general defaults for definitions of data, wwwloc and cssfile based on your environment. For every run

of `ondemand.pl`, a file (also called `ondeminit`) will be placed in the directory local to where `ondemand.pl` is run. This will allow you to run `ondemand.pl` repeatedly without the need to redefine these variables if they don't need redefining. NOTE: By default, `ondemand.pl` will still prompt for all of these variables unless the "batch" option is used on the command line.

**NOTE: The initial file "ondeminit" is created. It contains:**

```
data=/var/www/html/rrunr/tables/rrunr.tr  
wwwloc=/var/www/html/rrunr/ondemand/test  
cssfile=/var/www/html/rrunr/test/ondemand2.css  
add=webtab.add
```

**This will allow you to use the "batch" option in future runs of**  
**/cfmc/dev.80/go/ondemand.pl**

**So here's recap**

**The data file used will be /var/www/html/rrunr/tables/rrunr.tr**

**The index.html and tables will be stored in**  
**/var/www/html/rrunr/ondemand/test**

**The .css file used will be /var/www/html/rrunr/test/ondemand2.css**

**Additional definitions will come from webtab.add**

**Press enter to continue or break to start over ==><enter>**

At this point, you would have changed any defaults at the command line. The file that holds these changes is "ondeminit" It is in the directory local to where `ondemand.pl` is run. This file can be edited at any time.

The next prompt will deal with password protecting the directory. Type "y" to set a password.

**Shall we set a password in /var/www/html/rrunr/ondemand/test?**

**(y/(n)) ==>y**

The following prompt asking for a user name appears. In this case, "test" is the user name.

**Please enter a user name or press enter for no more ==>test**

The prompt will ask you for a password. At the next prompt you will be asked to re-type the password. The next prompt allows for another user name. If this is not necessary, press enter to proceed.

**New password:**

**Re-type new password:**

**Adding password for user test**

**Please enter a user name or press enter for no more ==>**

Everything needed has been entered and you are ready to proceed. The following instructions appear, and you will be returned to the original prompt:

**all done!**

**Now we will create the necessary files to run tables.**

**This may take a few moments, please be patient.**

**Now go to <http://product01.cfmc.com/rrunr/ondemand/test> to continue**

The URL that is needed to access On-Demand Tables can be copied and pasted into your browser. In this example, *http://product01.cfmc.com/rrunr/ondemand/test* would be copied and pasted.

If the debug option is used, */var/www/html/rrunr/ondemand/test>ondemand.pl rrunr.def debug* the following is displayed:

**Default Data File set to /cfmc/dev.80/data/\_STUDY\_.tr per /cfmc/dev.80/control/ondeminit file**

**Default Tables Location set to /var/www/html/\_STUDY\_/ondemand per /cfmc/dev.80/control/ondeminit file**

**Default CSS File set to /var/www/html/cfmcweb/css/ondemand.css per /cfmc/dev.80/control/ondeminit file**

**Default Data File set to /var/www/html/sued/rrunr/tables/rrunr.tr per ondeminit file**

**Default Tables Location set to /var/www/html/sued/rrunr/ondemand/test per ondeminit file**

**Default CSS File set to /var/www/html/cfmcweb/css/ondemand.css per ondeminit file**

**Default Add File set to test.add per ondeminit file**

### 7.3.3 Editing

At any time, the specs used for On-Demand Tables can be edited. There are many reasons for making changes. It might be to change something in the .def file, adding banners or changing settings, to mention a few. After this is done, On-Demand must be reinitialized. To avoid going through the prompts the "batch" command can be used /cfmc/dev.80/go/==>ondemand.pl rrunr.def batch. The ondeminit file created previously will be read and the following will be displayed:

**Here's a recap:**

**data=/cfmc/dev8.0/data/rrunr.tr**

**index.html and tables will be stored in /var/www/html/rrunr/ondemand**

**css file used will be /var/www/html/cfmcweb/css/ondemand.css**

**additional definitions will come from webtab.add**

**Now we will create the necessary files to run tables.**

**This may take a few moments, please be patient.**

Now go to <http://product01.cfmc.com//rrunr/ondemand/test/test> to continue  
[sued@product01 /www/htmldocs/rrunr/ondemand/test/test dev80]>

The "ondeminit" file created at the initial setup can be edited at any time. Once an edited copy has been created, it will be placed in the directory local to where ondemand.pl is run. If edited, the study would need to be reinitialized by typing:

```
/cfmc/dev.80/go/ondemand.pl rrunr.def batch
```

```
data=/cfmc/dev8.0/data/rrunr.tr
```

```
wwwloc=/var/www/html/rrunr/ondemand
```

```
cssfile=/var/www/html/cfmcweb/css/ondemand.css
```

```
add=webtab.add
```

Once again, this reflects the choices made in the setup, and it reflects the settings created for this CfMC demo. Paths and directories will reflect the choices made at setup by your system administrators.

If, at any time, a password needs to be added after the initial setup, the following command can be used.

```
==>ondemand.pl rrunr.def batch setpass=yes
```

This will skip all prompts until it gets to the password prompts.

## **WEBTAB.ADD**

Using the webtab.add on the command line allows for bases, weights, banners, headers and logos to be added at setup time. Just as the .qpx/.def file has to be in the local directory (i.e., the directory where you type "ondemand.pl"), so does your ".add" file. The add file will be copied to the Web directory along with the .qpx/.def file. If you change anything in an existing On-Demand setup, the study must be reinitialized.

{!MENTOR\_DEF} and {!MENTOR\_TAB} are used to pass Mentor specifications to the .def or .tab file. The SURVENT compiler treats these blocks as comments and no syntax checking is done. When you compile your questionnaire with the option MENTOR\_SPECS, these blocks will be passed to the appropriate file.

The close brace (}) in a ~prepare block is interpreted as the end of a command block. The close brace followed by an underscore (}) allows you to embed a close brace (}) inside the compiler command block. The underscore is removed when compiled and passed to the .def file.

The following is an example of the what is contained in the webtab.add file:




If using a .qpx	If using a .def
<pre> webtab.add  {!MENTOR_DEF  tabset={base_ny: t4=:BASE: Location: New York }_ "base=:loc(4) base=: [58.2#NY] }_  tabset={base_ca: t4=:BASE: Location: California }_ "base=:loc(4) base=: [58.2#CA] }_  tabset={weight_01: t2=:WEIGHT: Gender: Males:10 Females:100 }_ weight=:select([57^1/2],values(10,100)) }_  tabset={weight_02: t2=:WEIGHT: Martial Status: Married:10 Divorced:20 Single:100 }_ weight=:select([54^1/2/4],&amp; values(10,20,100)) }_ } </pre>	<pre> webtab.add  tabset={base_ny: t4=:BASE: Location: New York } "base=:loc(4) base=: [58.2#NY] }  tabset={base_ca: t4=:BASE: Location: California } "base=:loc(4) base=: [58.2#CA] }  tabset={weight_01: t2=:WEIGHT: Gender: Males:10 Females:100 } weight=:select([57^1/2],values(10,100)) }  tabset={weight_02: t2=:WEIGHT: Martial Status: Married:10 Divorced:20 Single:100 } weight=:select([54^1/2/4],values(10,20,100) ) } </pre>

### 7.3.4 Adding a Base or Weight

A base or weight can be defined in three different ways.

- 1 In the .qpx
- 2 Adding it with a file. This file can be named whatever you want it be, but by default a file called webtab.add will be looked for if the add option is used.
- 3 Defining base or weight directly on the On-Demand selection screen define box.

 **Define Base and Weight**

Base	Definition
<input type="radio"/> Define:	<input type="text"/>
<input checked="" type="radio"/> None	

Weight	Definition
<input type="radio"/> Define:	<input type="text"/>
<input checked="" type="radio"/> None	

» **Open Table Options**

#### *Using the qpx*

This section describes how the .qpx would look.



**NOTE:** Use the t4 statement to define your base text, and the t2 statement to define text associated with weights. For the benefit of the final user, be sure to at least define what question label the base or weight is defined from.

**NOTE:** Fully qualify the file names of all ampersand/include files so the .qpx when copied to the web area and compiled will find them. If ampersanding in files to the .qpx make sure paths are fully qualified. (example /cfmc/websurv/studies/rrunr/webtab.add)

**NOTE:** Multi language studies need the >language speaking=<language> option set in the .qpx. Survent doesn't but using the db file (which is used in On-Demand Tables) requires it. you must choose a language to display results.

>purgesame

~PREP COMPILE

[RRUNR,640,"EXAMPLE JOB",SPECWID=128,&  
TEXT\_START=5/1,WORK\_START=4/1]

{!AUTO\_PUNCHES}  
{!BLANK\_LINES=1}

{!MENTOR\_DEF

tabset={base\_ny:  
t4=:BASE: Location: New York  
}\_  
"base=:loc(4)  
base=: [58.2#NY]  
}\_

tabset={base\_ca:  
t4=:BASE: Location: California  
}\_

```

"base=:loc(4)
base=: [58.2#CA]
}_

tabset={weight_01:
t2=:WEIGHT: Gender: Males:10 Females:100
}_
weight=:select([57^1/2],values(10,100))
}_

tabset={weight_02:
t2=:WEIGHT: Martial Status: Married:10 Divorced:20 Single:100
}_
weight=:select([54^1/2/4],values(10,20,100))
}_
}

{ QN1: 1/6.1
!MISC RATING=5
Q1. How much do you agree with the following statement:
The fast food at Road Runners is worth what I pay for it.
! CAT
5 (5) Completely agree
4 (4) Somewhat agree
3 (3) Neither agree nor disagree
2 (2) Somewhat disagree
1 (1) Completely disagree
0 Don't Know/Refused to answer
}

{ QN2A: 1/7.1
!MISC RATING=5
Q2a. Please rate the following
characteristics:
The quality of the food.
! CAT
5 (5) Very good
4 (4) Good
3 (3) Neither poor nor good

```

```

2 (2) Poor
1 (1) Very poor
0 Don't know/refused to answer
}

~end

```

### ADDING A FILE ON THE COMMAND LINE

A file name `weftab.add` can be used to define these bases and weights as well. This file will be placed in the directory where the On-Demand tables will be run. In this case, that would be `/var/www/html/rrunr/ondemand` (directory used in previous examples).

**NOTE:** Use the `t4` statement to define your base text, and the `t2` statement to define text associated with weights. For the benefit of the final user, be sure to at least define what question label the base or weight is defined from.

#### **weftab.add**

```

{!MENTOR_DEF

tabset={base_ny:
t4=:BASE: Location: New York
}_
"base=:loc(4)
base=: [58.2#NY]
}_

tabset={base_ca:
t4=:BASE: Location: California
}_
"base=:loc(4)
base=: [58.2#CA]
}_

```

```

tabset={weight_01:
t2=:WEIGHT: Gender: Males:10 Females:100
}_
weight=:select([57^1/2],values(10,100))
}_

tabset={weight_02:
t2=:WEIGHT: Martial Status: Married:10 Divorced:20 Single:100
}_
weight=:select([54^1/2/4],values(10,20,100))
}_
}

```

If you add the option "add" to your `ondemand.pl` command line, the file "webtab.add" will be looked for in the local directory and added to the questionnaire spec. If you embellish the "add" command with, for example, "add=somestuff.spx", a file called "somestuff.spx" will be added to the questionnaire spec. Only one file may be added for now. The spec should be a complete `!mentor_def` block and it will be inserted at the very end of the db file.

This is done at setup time. The syntax is `/var/www/html/<study>/ondemand.pl <study> add`. Just as the `qpx` file has to be in the local directory (i.e., the directory where your type "ondemand.pl"), so does your "add" file. The add file will be copied to the web directory just as the `qpx` file is (if you're not running `ondemand.pl` in this web directory, of course). This feature was developed to add bases and weights at initialization. If you want to add pre-defined bases and weights to an existing `ondemand` setup, reinitialize the study.

## USING THE DEFINE BOX

As in any CfMC utility (hole,scan,...) a base or a weight can be set for the tables. You can use a data location to set the base or weight. An example would be a base of `[51^1]`, which means only include those cases that have a one-punch in column fifty one. You can also use variables from previous Mentor runs. The DB file in which the variables are defined is accessible in this utility. For further information, see the *Utilities* manual.

Define Base and Weight

Base	Name	Description
<input type="radio"/>	BASE_NY	BASE: Location: New York
<input type="radio"/>	BASE_CA	BASE: Location: California
<input checked="" type="radio"/>	DEFINE: Male:qn21(1)	
<input type="radio"/>	NONE	

Weight	Name	Description
<input type="radio"/>	WEIGHT_01	WEIGHT: Gender: Males:10 Females:100
<input type="radio"/>	WEIGHT_02	WEIGHT: Martial Status: Married:10 Divorced:20 Single:100
<input type="radio"/>	DEFINE:	
<input checked="" type="radio"/>	NONE	

The following are examples of base definitions:

- RATING(1,2,Y) : named question code list
- STATE(<AZ-CA,NV) : code range/ (<=not)
- SEX(M) OR [AGE#18-24,99] : combinations (AND/OR)
- Example location specs: : ,=net -=range
- [2/10.2^1-20,24,B] : punches (N=not,B=blank)
- [TIMES#1-10,99,DK,RF," "] : numbers or letters
- [10.2,....,20\*F#1-20] : number nets across columns

WARNING: Do not use "/"s between categories!

The following are examples of weight definitions:

Use a pre-defined name, (i.e. "WGHTIT"), OR if weight is in the data, use the format:"[5/23.3\*F2]" for Record 5, column 23, length 3, with 2 implied decimals.

To specify a new weight variable, use the format:

“Select([1/6.1^1//5],Values(1.23,2.5,0.76,1.03,1))” which assigns weight values using punches 1-5 of column 6.

### EDIT OPTIONS

When the index page appears, there is a section on the bottom where you can edit table options.

» [Open Table Options](#)

**Make Table**

When Open edit options is clicked, the following will appear. If the radio button is “on,” the menu will expand. Below the Statistical Testing is “of,” therefore the menu is collapsed. These options are explained in previous chapters.



« Close Table Options

Frequencies: on  off

Decimal Places <input type="text" value="0"/>
<input type="checkbox"/> Suppress rows that are less than <input type="text" value="0"/>
Format: <input checked="" type="radio"/> "1000" <input type="radio"/> "1,000" <input type="radio"/> "\$1,000"

Percentages: on  off

Decimals Places <input type="text" value="0"/>		
<input checked="" type="checkbox"/> Vertical Percentaging	Based on: <input type="radio"/> Total <input checked="" type="radio"/> All Responding	<input type="checkbox"/> Suppress rows that are less than <input type="text" value="0"/> %
<input type="checkbox"/> Horizontal Percentaging	Based on: <input checked="" type="radio"/> Total <input type="radio"/> All Responding	
<input type="checkbox"/> Cumulative Percentaging		
<input type="checkbox"/> Suppress Percent (%) sign		
<input type="checkbox"/> Show Frequencies Only for Summary Rows (T/NA/AR), Percentages Only for Body		
<input type="checkbox"/> Replace Percentages Below <input type="text" value="0"/> with "*" (and add footnote)		

Statistical Testing: on  off

Miscellaneous Options

Statistics Decimal Places <input type="text" value="2"/>					
(i.e., Decimal places for Mean, Standard Deviation, Standard Error, etc. values)					
Summary Rows to Print:	<input type="checkbox"/> Total <input type="checkbox"/> No Answer <input checked="" type="checkbox"/> All Responding				
What to print for ZERO / MISSING / EMPTY CELLS:					
Integer Zero (0)	<input type="radio"/> 0	<input type="radio"/> Blank	<input type="radio"/> ?	<input checked="" type="radio"/> -	<input type="radio"/> other: <input type="text"/>
Floating-Point Zero (0.0)	<input checked="" type="radio"/> 0	<input type="radio"/> Blank	<input type="radio"/> ?	<input type="radio"/> -	<input type="radio"/> other: <input type="text"/>
Missing	<input type="radio"/> 0	<input type="radio"/> Blank	<input checked="" type="radio"/> ?	<input type="radio"/> -	<input type="radio"/> other: <input type="text"/>
Empty Cell	<input type="radio"/> 0	<input type="radio"/> Blank	<input type="radio"/> ?	<input checked="" type="radio"/> -	<input type="radio"/> other: <input type="text"/>

**Make Table**

## FILES CREATED

During the setup phase, the following files will be created in your On-Demand Web table directory when the .qpx is compiled by the utility:

- <study>.sum
- <study>.def
- <study>.db
- <study>.qpx
- mkdb.db
- index.html - this file allows access to the On-Demand tables

**NOTE:** An index.html file will be created in this directory. If you are using a pre-existing Web area for your On-Demand Tables, make sure you don't overwrite anything important, such as the index.html file.

Once the On-Demand Tables are accessed, other files will appear in the directory. These files will all have an .html extension. There will be a different file for each table.



### 7.3.5 Creating On-Demand Tables

#### ON-DEMAND SELECTION SCREEN

Use the On-Demand selection screen to choose the variables that are to be crossed. Any question in the .qpx can be crossed with any other question in the same .qpx.. A screen listing all the variables will be displayed. To choose the banner, click the radio button in the left column. For the stub, click the radio button in the second column. A Base or Weight can be added. Once the selection is made, click **Submit** on the bottom of the page.

### On-Demand Tables

Select Banner and Stub

Banner	Stub	Question Title
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Total
<input type="radio"/>	<input type="radio"/>	qn1: Q1. How much do you agree with the following statement: The fast food at Road Runners is worth what I pay for it
<input type="radio"/>	<input type="radio"/>	qn2a: Q2a. Please rate the following characteristics: The quality of the food.
<input type="radio"/>	<input type="radio"/>	qn2b: Q2B. Please rate the following characteristics: The quality of service.
<input type="radio"/>	<input type="radio"/>	qn2c: Q2C. Please rate the following characteristics: The cleanliness of the restaurant.
<input type="radio"/>	<input type="radio"/>	qn2d: Q2D. Please rate the following characteristics: The prices on the menu.
<input type="radio"/>	<input type="radio"/>	qn2e: Q2E. Please rate the following characteristics: The accuracy of your bill.
<input type="radio"/>	<input type="radio"/>	qn2f: Q2F. Please rate the following characteristics: The cleanliness of the restrooms.
<input type="radio"/>	<input type="radio"/>	qn12: Q12. Overall, how would you rate Road Runners?
<input type="radio"/>	<input type="radio"/>	qn13: Q13. How long have you lived in your current home?
<input type="radio"/>	<input type="radio"/>	qn14: Q14. What is the occupation of the principal wage earner in your home?
<input type="radio"/>	<input type="radio"/>	qn15: Q15. What is the highest level of formal education completed by the principal wage earner in your home?
<input type="radio"/>	<input type="radio"/>	qn16: Q16. Which of the following best describes your age?
<input type="radio"/>	<input type="radio"/>	qn17: Q17. Into which of the following categories does your total family income fall?
<input type="radio"/>	<input type="radio"/>	qn18: Q18. Would you classify your ethnic background as . . .
<input type="radio"/>	<input type="radio"/>	qn19: Q19. And how would you describe your marital status?
<input type="radio"/>	<input type="radio"/>	qn20: Q20. How many children ages 18 and under are in living at home?
<input type="radio"/>	<input type="radio"/>	qn21: Q21. Respondent Sex
<input type="radio"/>	<input type="radio"/>	last: STATE CODE

Define Base and Weight

	Base	Name	Description
<input type="radio"/>	BASE_NY		BASE: Location: New York
<input type="radio"/>	BASE_CA		BASE: Location: California
<input type="radio"/>	DEFINE:	<input type="text"/>	
<input checked="" type="radio"/>	NONE		

	Weight	Name	Description
<input type="radio"/>	WEIGHT_01		WEIGHT: Gender: Males:10 Females:100
<input type="radio"/>	WEIGHT_02		WEIGHT: Martial Status: Married:10 Divorced:20 Single:100
<input type="radio"/>	DEFINE:	<input type="text"/>	
<input checked="" type="radio"/>	NONE		



« Close Table Options

Frequencies: on  off

Decimal Places	<input type="text" value="0"/>
<input type="checkbox"/> Suppress rows that are less than	<input type="text" value="0"/>
Format:	<input checked="" type="radio"/> "1000" <input type="radio"/> "1,000" <input type="radio"/> "\$1,000"

Percentages: on  off

Decimals Places	<input type="text" value="0"/>	
<input checked="" type="checkbox"/> Vertical Percentaging	Based on: <input type="radio"/> Total <input checked="" type="radio"/> All Responding	<input type="checkbox"/> Suppress rows that are less than <input type="text" value="0"/> %
<input type="checkbox"/> Horizontal Percentaging	Based on: <input checked="" type="radio"/> Total <input type="radio"/> All Responding	
<input type="checkbox"/> Cumulative Percentaging		
<input type="checkbox"/> Suppress Percent (%) sign		
<input type="checkbox"/> Show Frequencies Only for Summary Rows (T/NA/AR), Percentages Only for Body		
<input type="checkbox"/> Replace Percentages Below <input type="text" value="0"/> with "*" (and add footnote)		

Statistical Testing: on  off

Miscellaneous Options

Statistics Decimal Places	<input type="text" value="2"/>		
(i.e., Decimal places for Mean, Standard Deviation, Standard Error, etc. values)			
Summary Rows to Print:	<input type="checkbox"/> Total	<input type="checkbox"/> No Answer	<input checked="" type="checkbox"/> All Responding
What to print for ZERO / MISSING / EMPTY CELLS:			
Integer Zero (0)	<input type="radio"/> 0	<input type="radio"/> Blank	<input type="radio"/> ? <input checked="" type="radio"/> - <input type="radio"/> other: <input type="text"/>
Floating-Point Zero (0.0)	<input checked="" type="radio"/> 0	<input type="radio"/> Blank	<input type="radio"/> ? <input type="radio"/> - <input type="radio"/> other: <input type="text"/>
Missing	<input type="radio"/> 0	<input type="radio"/> Blank	<input checked="" type="radio"/> ? <input type="radio"/> - <input type="radio"/> other: <input type="text"/>
Empty Cell	<input type="radio"/> 0	<input type="radio"/> Blank	<input type="radio"/> ? <input checked="" type="radio"/> - <input type="radio"/> other: <input type="text"/>

**Make Table**

## FINAL OUTPUT OF TABLES

Each table will appear in a pop-up window. If you minimize the table window, more than one table can be viewed at the same time.

26 JAN 2005 1:29 PM

Q20. How many children ages 10 and

under are in living at home?

Q14. What is the occupation of the principal

wage earner in your home?

	QN20				
	Total	N/A	0-3	4-5	6-10
Base	501	52	151	88	210
	100%	100%	100%	100%	100%
	%	%	%	%	%
		(B)	(C)	(D)	(E)
Professional/Technical	55	7	18	10	20
	11%	13%	12%	11%	10%
Clerical	43	7	9	9	18
	9%	13%	6%	10%	9%
		c			
Skilled laborer	56	4	17	12	23
	11%	8%	11%	14%	11%
Unskilled laborer	63	6	22	7	28
	13%	12%	15%	8%	13%
Housewife/Student	62	4	20	12	26
	12%	8%	13%	14%	12%
Unemployed	45	8	8	9	20
	9%	15%	5%	10%	10%
		C			
Retired	64	7	24	8	25
	13%	13%	16%	9%	12%
Other	52	4	16	7	25
	10%	8%	11%	8%	12%
Refused	61	5	17	14	25
	12%	10%	11%	16%	12%

(sig=.05 + sig=.10) (all\_pairs) columns tested B-E

Close Window

## Preparing Mentor Output Files For Post Processing

You can create a print file, a delimited file and an html file simultaneously using these minimum keywords:

- ~specfile <rootname>
- >printfile <filename>
- ~set webtables delimited\_tables

You can run tables from your own tabsets or you can use the .ban file and the .def file created by ~prepare. The most efficient way to make your own banner tabsets is by using the banner element option of make\_banner.

1 Creating three files simultaneously by making your own tabsets.

If you make your own banner and use the make\_banner option, a .ban is made for you and is named using the ~specfile <rootname>. It contains all of the necessary print, delimited and html banner specifications.

Notice in the .ban file that when you use the default delimiter of tabs, the delimited banner is created using "\t" in place of the tabs.

Mentor specs (post01.spx)

```
tabset= { qn12_y:
banner=:
make_banner
[level=2]
[level=1] Total
[level=2] Overall Road Runners Rating
[level=1] (5) Very good
[level=1] (4) Good
[level=1] (3) Neither poor nor good
[level=1] (2) Poor
[level=1] (1) Very poor
[level=1] Don't know/refused to answer
}
statistics=: b-g
col=: total with [47.1^5/4/3/2/1/10]
}
```

Mentor banner (post01.ban)

banner=qn12\_bn:

\k(p)

Overall Road Runners Rating

```
=====
=
```

			(3)			Don't
			Neither			know/
			poor		(1)	refused
	Very	(4)	nor	(2)	Very	to
Total	good	Good	good	Poor	poor	answer
-----	-----	-----	-----	-----	-----	-----

\k(d)

\tOverall Road Runners Rating\tOverall Road Runners Rating\tOverall Road Runners Rating\tOverall Road Runners ...

\tTotal\t(5) Very good\t(4) Good\t(3) Neither poor nor good\t(2) Poor\t(1) Very poor\tDon't know/refused to answer

\k(h)

<tr>

<td colspan="1">

<td colspan="1">

<td colspan="6" align="center"> Overall Road Runners Rating

</tr>

<tr>

<td colspan="1">

<td colspan="1" align="right"> Total

<td colspan="1" align="right"> (5) Very good

<td colspan="1" align="right"> (4) Good

```
<td colspan="1" align="right"> (3) Neither poor nor good  
<td colspan="1" align="right"> (2) Poor  
<td colspan="1" align="right"> (1) Very poor  
<td colspan="1" align="right"> Don't know/refused to answer  
</tr>  
}
```

## 2 Creating three files simultaneously by using the .ban file and .def file created by ~prepare

This setup ampersands in (brings in) the .ban file and the .def file created by ~prepare. The ~set option auto\_banner\_heading=title is used to provide a 30 character banner heading. The ~specrules option of column\_statistics is used to add a statistics element to the banner tabset.

Mentor banner from ~prepare (rrunr.ban)

```
tabset= { qn12_y:  
  banner_title=:  
    Q12. Overall, how would you rate Road  
    Runners?}  
  banner=:  
  make_banner  
  [level=2] Q12. Overall, how would you...  
  [level=1] (5) Very good  
  [level=1] (4) Good  
  [level=1] (3) Neither poor nor good  
  [level=1] (2) Poor  
  [level=1] (1) Very poor  
  [level=1] Don't know/refused to answer  
}
```

```

statistics=: a-f
"qn12(5/4/3/2/1/0)
col=: [47.1^5/4/3/2/1/10]
}

```

If you use ~specrules base option and have bases or filters that look like this:  
 base=: (QN6(1)), you'll want to add the ~prepare created dbfile to this run. If you  
 typically use these ~set options:

```

drop_base
drop_title_4

```

you will also want to add these ~set options:

```

drop_filter
drop_filter_title

```

A new tabset element has been added to Mentor 7.7 to join banner tabsets called  
 joined\_tabsets. This setups joins a user created banner tabset called usertotal and  
 the Q12 banner tabset from the ~prepare created .ban file.

Mentor specs (post02.spx)

```

tabset= { usertotal:
banner=:
make_banner
[level=2]
[level=1] Total
}
col=: total
}

```





```
joined_tabset={ banner1_y:
usertotal with &
qn12_y
}
```

Mentor banner (post02.ban)

banner=banner1\_bn:

\k(p)

Q12. Overall, how would you...

=====

			(3)			Don't
			Neither			know/
	(5)		poor		(1)	refused
	Very	(4)	nor	(2)	Very	to
Total	good	Good	good	Poor	poor	answer
-----	-----	-----	-----	-----	-----	-----

\k(d)

\t\tQ12. Overall, how would you...\tQ12. Overall, how would you...\tQ12.

Overall, how would you...\tQ12. Overall, ...

\tTotal\t(5) Very good\t(4) Good\t(3) Neither poor nor good\t(2) Poor\t(1) Very  
 poor\tDon't know/refused to answer

\k(h)

<tr>

<td colspan="1">

<td colspan="1">

<td colspan="6" align="center"> Q12. Overall, how would you...

</tr>

<tr>

```

<td colspan="1">
<td colspan="1" align="right"> Total
<td colspan="1" align="right"> (5) Very good
<td colspan="1" align="right"> (4) Good
<td colspan="1" align="right"> (3) Neither poor nor good
<td colspan="1" align="right"> (2) Poor
<td colspan="1" align="right"> (1) Very poor
<td colspan="1" align="right"> Don't know/refused to answer
</tr>
}

```

The `joined_tabset` option automatically adjusts the statistics element in the `~prepare .ban` file from `statistics=: a-f` to `statistics=: b-g` when the `usertotal` banner tabset was joined with the `Q12` banner tabset.

Mentor print file (`post02.prt`)

### 3 Customizing the delimited file

There are a number of `~set delimited_tables` options available that let you select what is passed to the delimited file and these options have no affect on the print file or the html file. Stats letters and footnotes always go to the delimited file but column stats labels like (A) do not. There is no table of contents associated with a delimited file. All of the following items can be turned on or off. All of these items are automatically passed to the delimited file except for the banner and the percent signs:

```

BANner
BANner_TITLE
COMMENT
do_TABLE_NAME
FILTER_TITLE

```



FOOTer  
HEADer  
PERcent\_SIGN  
STUB  
TITLE  
Title\_2  
Title\_4  
Title\_5

These additional ~set delimited\_tables options are also available:

ADD\_SPACE\_BEFORE\_STATS  
Column\_WIDTH=  
DELIMITER=  
FILL\_LINES  
FILL\_TABLE  
LABELS  
QUOTED\_BANNER\_TEXT  
STATS\_ON\_SEPARATE\_LINE  
Stub\_WIDTH=  
USE\_AS\_PRINTED

By default, cells that are in the print file as blank, zero or dash, are passes to the delimited file as zeros. The option:

```
~set delimited_tables=(use_as_printed)
```

allows you to create a delimited file with cells containing the same characters as the printed table.

You can now change the CfMC assigned extensions on files. If you want to make a comma delimited file and you want an extension that is recognizable as such in Excel, you can change the extension by using:

```
>cfmc_extension dlm=csv
```

Mentor specs (post03.spx)

```
~set delimited_tables=(
    banner
    -quoted_banner_text
    delimiter=comma
    -footer
    percent_sign
    stats_on_separate_line
    use_as_printed
)
```

Mentor default delimited file (post02.dlm)

Mentor enhanced delimited file (post03.csv)

#### 4 Customizing an html file

There are a number of ~set webtables options available that let you select what is passed to the html file and these options have no affect on the print file or the delimited file. Stats letters, column stats labels like (A), footnotes, comments and percent signs always go to the html file. A "back to table of contents" anchor is added to the end of each table. All of the following items can be turned on or off. All these items are automatically passed to the html file:



BANner  
BANner\_TITLE  
do\_TABLE\_NAME  
FILTER\_TITLE  
FOOTer  
HEADer  
STUB  
TITLE  
Title\_2  
Title\_4  
Title\_5

These additional ~set webtables options are also available:

BGCOLOR=  
CSS\_FILE\_CHECK=  
CSS\_FILE\_PATH=  
FRAMESet  
FRAMETABLE=  
HTML\_TITLE=  
TCON\_ANCHOR\_TEXT=  
WEB\_FORMAT\_BANner=()  
WEB\_FORMAT\_COMMENT=()  
WEB\_FORMAT\_Cumulative\_Percent\_items=()  
WEB\_FORMAT\_EVEN\_COL=()  
WEB\_FORMAT\_EVEN\_ROW=()  
WEB\_FORMAT\_FREQuency\_items=()  
WEB\_FORMAT\_Horizontal\_Percent\_items=()  
WEB\_FORMAT\_ODD\_COL=()  
WEB\_FORMAT\_ODD\_ROW=()

```
WEB_FORMAT_STATistical_test_items=()
WEB_FORMAT_STATistics_ROW=()
WEB_FORMAT_STUB=()
WEB_FORMAT_TABLE=()
WEB_FORMAT_Vertical_Percent_items=()
```

All of these items are used specifically with Dynamic Charts:

```
BANNER_LEVELS=
DOCTYPE=
DYNAMIC_SCRIPTS=
DYNAMIC_TABLES
GROUPS=()
HTML_SRC=
ONRESIZE=
SURVEY=
SURVEY_NAME=
```

Mentor specs (post04.spx)

You can use the \k(p,h,d) options to pass specific text to specific files. To change the footer specs to remove the page numbers from the html file and the delimited file you can use:

```
footer={:=
Tables prepared by Computers for Marketing Corp.
\k(p)Page #PAGE_NUMBER#\k(p,h,d)
}
```



The html files are designed to match the printed files in content. To print significance letters on a separate line, you need to use the appropriate edit statement and this also affects the printed tables. The table of contents is most useful if it is printed first. Edit options that control the page numbering in the table of contents are ignored when creating an html file.

```
global_edit={:  
  stats_on_separate_line  
  tcon=(first  
    print_page_numbers  
    -tcon_page_numbers  
  )  
}
```

This example uses the following webtables options to customize the html file.

```
~set webtables=(  
  html_title="Road Runner Survey Title"  
  -header  
  tcon_anchor_text="***BACK TO THE TABLE OF CONTENTS***"  
  web_format_statistical_test_items=(font=(color=red))  
)
```

Mentor default html file (post02.htm)

Mentor enhanced html file (post04.htm)

## 5 Putting the table of contents in a frame in the html file

Mentor specs (post05.spx)

For this example, two banner tabsets from the ~prepare created .ban file will be joined to create the banner.

```
joined_tabset={ banner1_y:  
usertotal with &  
qn12_y with &  
qn16_y  
}
```

The only other change in this example is to put the table of contents in a frame separate from the tables using:

```
~set webtables=(frameset)
```

By putting the table of contents in a separate frame, three html files are created instead of one. The starter file is named using the ~specfile <rootname>, the table of contents file has a "t" added to the end of the <rootname> and all of the tables are in a file with "aa" added to the end of the <rootname>. Notice that the "back to table of contents" anchor is removed because it is no longer necessary when the table of contents is in its own separate frame.

Mentor starter html file (post05.htm)

Mentor table of contents html file (post05t.htm)

Mentor tables html file (post05aa.htm)





## Augmenting Prepare Specs to Enhance Tables

You can use Survent questionnaire specifications to produce two table building specification files, ready to be used by Mentor, called the DEF file and the TAB file. You can use the `~PREPARE COMPILE CMentor_SPECS` command or CfMC's menu-assisted EZWriter application to create these files. The DEF file contains the syntax to define a table for each question and the TAB file contains the commands to build the tables defined in the DEF file. The DEF file can also be edited to add basing, weights, statistics, print format controls, etc. The TAB file can be remade by Mentor to reflect any changes to the DEF file. For more information see “4.12 USING Survent TO GENERATE Mentor SPECIFICATION FILES”.

Refer to “4.11 SAMPLE SPECIFICATION FILES” for details on using these files to make tables. All examples refer to the RRUNR sample specification files included with your Mentor software.

## **TABLES VIEWABLE THROUGH A BROWSER**

*Augmenting Prepare Specs to Enhance Tables*

# STATISTICS (SIGNIFICANCE TESTING)

## INTRODUCTION

This chapter explains how to do significance testing (T-tests), chi square tests, ANOVA tests, and various other tests using the Mentor software. It also describes some of the program's methodology and some common problems that occur with this type of testing. The actual statistical formulas are not presented in this chapter, but you can find them in *Appendix A: STATISTICAL FORMULAS*.

This chapter assumes a basic understanding of table building and of CfMC terminology. If you are not familiar with table building, review chapters 4, 5 and 6. Although a basic understanding of statistics is not needed to produce the tests, it is very useful for understanding the process. In most cases a brief statistical description is provided, but for a more detailed reference consult a statistics textbook.

Significance testing can be created in two different ways. Testing can be done (1) while the program is reading data cases (table building phase), or (2) from the numbers that are printed on the table (table printing phase). Tests created during the table building phase work in almost all situations, while tests created during the table printing phase only work in a select number of situations (independent and unweighted). Most of this chapter deals with tests created during the table-building phase. See “8.5 PRINT PHASE STATISTICAL TESTING” for information on the table-printing phase tests.

The Mentor program can report significance testing in two ways: (1) marking cells that are significantly greater than other cells with the letter associated with the lesser column or (2) printing the actual statistical value and/or its significance. Most of this chapter describes marking the significantly greater cells. See “8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES” for information on printing the actual T values on the table.

You can test both columns and rows, although column tests are more common. Most of this chapter describes column tests. See “8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)” for information on row testing.

You can only test means or percentages produced from simple frequencies. A simple frequency is one in which a given respondent has a value of 1 or 0 in a given cell (except for any weighting). Items such as sigma, sums, and medians cannot be tested. For a detailed list of what cannot be tested, see “8.10 NOTES ON SIGNIFICANCE TESTING”.

## 8.1 SIGNIFICANCE TESTING TO MARK CELLS

To use significance testing on a table to mark cells, two options are required: the STATISTICS statement (as an element of the table) and the EDIT statement option DO\_STATISTICS. The STATISTICS statement defines which columns will be tested, while the DO\_STATISTICS option sets the confidence level. The system defaults are set to the following:

- System total row is the statistical base
- Tables can only have a single weight
- Confidence level is set to 95%
- The All Possible Pairs Test is used for multiple column tests

### 8.1.1 The STATISTICS Statement

The STATISTICS statement defines which columns and/or rows to test against each other and whether the test is independent, dependent, inclusive, or if t values are to be printed on the table. An *independent* test is one in which no case is in more than one of the cells being tested. A *dependent* test is one in which at least one case is in more than one of the cells. An *inclusive* test is one in which the first column in a test completely contains all the cases in all the rest of the columns. A

*printable* test can only test pairs of columns and no column can be the last column in more than one pair.

To set up a column test, first assign a name to the statement, and then list the letters associated with each column (no spaces). The first column in the table is assigned the letter A, the second the letter B, and so on. To test the first three columns of the table against each other the statement would look like this:

```
STATISTICS= STAT1 : ABC
```

Assigning a name to the STATISTICS= variable is optional, but doing so allows you to reference the statement by its name; for instance in a TABLE\_SET definition or in the ~EXECUTE block. If there is more than one test that needs to be performed on a given table, end the first test with a comma, and then enter the next set of columns. To identify the test as independent, put I= in front of that particular set of columns. If a test is inclusive, put T= in front of the columns set. If t values will be printed on the table, use the PRINTABLE\_T option before the list of columns.

A typical STATISTICS statement looks like this:

```
STATISTICS= STAT2 : I=BC,DEF, I=GHI, T=AKLM, BDGJ
```

This statement would create the following five tests:

- 1 An independent test on columns B and C
- 2 A dependent test on columns D, E, and F
- 3 An independent test on columns G, H, and I
- 4 An inclusive test on columns A, K, L, and M
- 5 A dependent test on columns B, D, G, and J

**NOTE:** The STATISTICS statement, like all other table elements, can be assigned a default name when it is defined inside of a TABLE\_SET. All examples in this chapter use this convention.

### 8.1.2 Independent, Dependent, Inclusive, and Printable Tests

An important distinction exists between independent and dependent tests. The formula for the dependent test contains additional calculations that rely entirely on the distribution of the individual data cases. Therefore, you generally cannot verify the results of a dependent test by examining the values printed on the table. These calculations cause dependent tests to require a smaller percentage difference to be marked as significant. For example, suppose the mean rating of brand A is being tested against brand B. If one respondent rated brand A higher than brand B, he has a clear preference for brand A. However, if one person rated brand A higher than another rated brand B, the difference in the rating might be because the first person always rates high and the other person always rates low, so you must distribute the difference between the difference in the brands and the difference in the respondents.

If all the columns in a given test are independent, then the extra calculations for the dependent test drop out of the calculation. Computer processing time can be reduced and additional error checking performed on independent tests if these tests are specified as independent (I=) on the STATISTICS= statement. This allows the program to drop the additional unnecessary calculations and prints an error message if the tests are not really independent.

#### NEW PROTECTION VALUE IN SIGNIFICANCE TESTING

In the calculation of the variance used in Mentor's significance testing, the protection limit on the lowest possible value in the denominator was changed from 0.0001 to 0.000001.

There is a new ~set tinyv option which can be used to set this value back to the original default value of 0.0001 in the unlikely event the user needs this to reproduce past results.

An inclusive test is one in which all the subsequent columns are completely contained in the first column. This test, sometimes referred to as *completely dependent*, is modified in the following way. The resultant test is an independent test that is exactly the same test as testing the contained column against everyone who is in the other column, but not in the contained column.

**Example:**

- testing the total against males results in the same test as testing the females against males
- testing people from California against people from San Francisco results in the same test as testing people from California, but not from San Francisco against people from San Francisco

If all the columns in a test have this property the program can perform additional error checking if the test has been specified as inclusive. This will cause the program to print an error message if it finds a case that is not in the first column, but is in a subsequent column.

If the t values or their significance are to be printed on the table, you can error check at definition time to make sure that there will be room on the table to print the values. Since the t value will be printed under the second column in the test, only pairs of columns may be tested and no column can be the second column in more than one pair. The use of the PRINTABLE\_T option will error check the STATISTICS statement to make sure it follows these rules. If the PRINTABLE\_T option is not used and more than one value is created in any cell, the error will not be reported until the table is printed.

### 8.1.3 Setting the Confidence Level

Significance testing is done in a two-step process:

## STATISTICS (SIGNIFICANCE TESTING)

### 8.1 SIGNIFICANCE TESTING TO MARK CELLS

First, the program reads the data and does all the initial calculations determined by the elements on the STATISTICS statement.

Second, it prints the table using the confidence level specified on the DO\_STATISTICS option on the EDIT statement.

The DO\_STATISTICS= option can be set to any of the following: .80, .90, .95, .99, and APPROXIMATELY .nn (where nn is any value). You can combine options with a plus sign (+). DO\_STATISTICS without an option has a default significance level of 95%.

**NOTE:** Neither DO\_STATISTICS=.80 nor APPROXIMATELY .nn can be used in conjunction with the Newman-Keuls testing procedure discussed later in this chapter.

Here are some example settings of the DO\_STATISTICS option:

```
DO_STATISTICS=.90
```

Sets the confidence level to 90%; the significance level to 10%

```
DO_STATISTICS=.99
```

Sets the confidence level to 99%; the significance level to 1%

```
DO_STATISTICS=.90+.95
```

Performs bi-level testing at both the 95% and 90% confidence levels.

```
DO_STATISTICS=APPROXIMATELY .85
```



Sets the confidence level to 85%, but uses an approximation formula for the t-values rather than looking it up in a table.

```
DO_STATISTICS=.95+APPROXIMATELY .88
```

Performs bi-level testing at both the 95% confidence level (table look-up) and at the 88% level using an approximation formula.

The program will mark significantly higher cells by putting the letter associated with the lower cell to the right of the frequency of the higher cell. Multiple letters run into the next cell then are continued on the next line, until all the letters are printed. Normally the printed letters print in upper case, unless you are doing bi-level testing. If the difference is significant at the higher level the letter prints in upper case, and lower case if it is different at the lower level, but not at the higher level. For an example of bi-level testing, see “*8.1.5 Changing the Confidence Level*”.

When significance testing is done, the program will print a percentage sign, along with the appropriate letter for that column under the System Total row. This makes it easy to distinguish both the statistical base and the columns that were tested. Columns that are not tested will not have a letter designation under the total row. It will also print an automatic footnote at the bottom of each page which includes the significance level, the test used, and which columns were tested.

**NOTE:** A 95% confidence level is equivalent to a .05 significance level.

#### 8.1.4 Standard Significance Testing

The minimum specification required to do significance testing is a STATISTICS statement, along with the EDIT option DO\_STATISTICS. If either of these statements is specified without the other, then no testing will be done and no footnote will print on the table.

In most of the examples in this chapter, the STATISTICS statement and the EDIT options that affect the significance testing are specified in the TABLE\_SET definition for a given table, but in general these options would be specified in the TABLE\_SET that defines the banner and therefore apply to the entire run. Switching settings from table to table may be both confusing and difficult to check. Here is an example of standard significance testing at the 95% confidence level.

**NOTE:** The following set of commands defines a standard front end for the next set of examples.

```
>PURGESAME
>PRINT_FILE STAT1
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T101

~DEFINE

STUB= STUBTOP1:
[BASE_ROW] TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= { BAN1:
EDIT=:
COLUMN_WIDTH=7,STUB_WIDTH=30,-COLUMN_TNA,STATISTICS_DECIMALS=2,
-PERCENT_SIGN }
STUB_PREFACE= STUBTOP1
BANNER=:
|           SEX           AGE           ADVERTISING AWARENESS
|           <=====> <=====> <=====>
| TOTAL MALE FEMALE 18-30 31-50 51-70 BRND A BRND B BRND C BRND D
| ----- }

```



```
COLUMN= TOTAL WITH [5^1/2] WITH [6^1//3] WITH [7^1//4]
}
```

Here is the first example:

```
TABLE_SET= { TAB101:
STATISTICS=: I=BC,I=DEF,GHIJ
LOCAL_EDIT=: DO_STATISTICS }
HEADER=: TABLE WITH SIGNIFICANCE TESTING }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
NET GOOD
| VERY GOOD
| GOOD
FAIR
NET POOR
| POOR
| VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: [11^4,5/5/4/3/1,2/2/1/X] $[MEAN,STD,SE] [11]
STORE= T101 }

~EXECUTE

TABLE_SET= BAN1
TABLE_SET= TAB101
```

STATISTICS (SIGNIFICANCE TESTING)

8.1 SIGNIFICANCE TESTING TO MARK CELLS

Here is the table that is printed:

TABLE WITH SIGNIFICANCE TESTING

TABLE 101

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	196	204	125	145	113	91	108	107	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD	197	120C	77	57	63	74DE	42	55	54	105G
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70C	32	35	38	27	19	32	30	60G
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47DE	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28F	44F	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20	19	27J	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30B	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4



VERY POOR	44	9	35B	17	13	10	11	14J	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79GH
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
 (sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

In Table 101, notice the "C" next to the frequency of 120 in the MALE column and NET GOOD row. This signifies that the percentage of males who were a NET GOOD mention is significantly higher at the 95% level than the percentage of females who were a NET GOOD mention. The letter "C" is used because the FEMALES column is the third column in the banner and it is so designated under the TOTAL row at the top of the table. Also notice that there is no "A" under the TOTAL column because it was not included in any of the tests.

The footnote on the bottom of the page tells you which significance level was used (.05 or a 95% confidence level), which test was used (All Possible Pairs), and which columns were tested (BC, DEF, and GHIJ).

### 8.1.5 Changing the Confidence Level

The confidence level easily can be changed by using the EDIT option DO\_STATISTICS. For instance, to test at the 99% confidence level, you would specify DO\_STATISTICS=.99. Below is an example of a table that would be tested

**STATISTICS (SIGNIFICANCE TESTING)**

*8.1 SIGNIFICANCE TESTING TO MARK CELLS*

at the 99% level. Table settings that are the same as Table 101 are restated here for clarity.

```
TABLE_SET= { TAB102:
STATISTICS=: I=BC,I=DEF,GHIJ
LOCAL_EDIT=: DO_STATISTICS=.99 }
HEADER=: TABLE WITH SIGNIFICANCE TESTING AT THE 99%
CONFIDENCE LEVEL }
TITLE= TAB101
TITLE_4= TAB101
STUB= TAB101
ROW= TAB101
STORE_TABLES=* }
```

Here is the table that is printed:

TABLE WITH SIGNIFICANCE TESTING AT THE 99% CONFIDENCE LEVEL

TABLE 102

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	196	125	145	113	91	108	107	176	
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	
	%	%	%	%	%	%	%	%	%	
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD	197	120C	77	57	63	74DE	42	55	54	105

STATISTICS (SIGNIFICANCE TESTING)  
8.1 SIGNIFICANCE TESTING TO MARK CELLS

	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70C	32	35	38	27	19	32	30	60
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47DE	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28	44F	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20	19	27J	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30B	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4
VERY POOR	44	9	35B	17	13	10	11	14	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79G
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
(sig=.01) (all\_pairs) columns tested BC, DEF, GHIJ

If you compare this table to Table 101, you will notice that there is no "G" in the NET GOOD row under the banner point BRND D. This is because these two cells are significantly different at the 95% level, but not at the 99% level. Also, notice the change in the table footnote.

### 8.1.6 Bi-Level Testing (Testing at Two Different Confidence Levels)

To do bi-level testing, or testing at two different confidence levels, on the same table specify the EDIT option DO\_STATISTICS= to the two different levels and join them with a plus sign (+). For example, to test at both the 99% and 95% levels say DO\_STATISTICS=.99+.95. The example below is the same as Table 101 above, except for the DO\_STATISTICS option.

The program will use upper- and lower-case letters to distinguish between those cells that are significant at the upper and lower levels. The upper-case letter also will be associated with the higher significance level regardless of the order in which they are specified on the DO\_STATISTICS command.

```
TABLE_SET= { TAB103:
STATISTICS=: I=BC, I=DEF, GHIJ
LOCAL_EDIT=: DO_STATISTICS=.95+.99 }
HEADER=:TABLE WITH SIGNIFICANCE TESTING AT THE 95% AND
99% CONFIDENCE LEVELS }
TITLE= TAB101
TITLE_4= TAB101
STUB= TAB101
ROW= TAB101
STORE_TABLES=* }
```

Here is the table that is printed:

```
TABLE WITH SIGNIFICANCE TESTING AT THE 95% AND 99%
CONFIDENCE LEVELS
TABLE 103
RATING OF SERVICE
```



STATISTICS (SIGNIFICANCE TESTING)  
8.1 SIGNIFICANCE TESTING TO MARK CELLS

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	<----->		<----->			<----->				
	TOTAL	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	
TOTAL	400	196	204	125	145	113	91	108	107	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD	197	120C	77	57	63	74DE	42	55	54	105g
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70C	32	35	38	27	19	32	30	60g
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47DE	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28f	44F	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20	19	27J	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30B	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4
VERY POOR	44	9	35B	17	13	10	11	14j	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79Gh
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
 (sig=.01 + sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

If you compare the above table to both Table 101 and Table 102, you will notice that it is actually a combination of the two. Every letter marking from Table 102 is replicated on this table, and every letter marking from Table 101 that was not on Table 102 is printed here with a lower case letter. Notice the lower case "g" in the NET GOOD row under the BRND D column. This means that this cell is significantly greater than column G at the 95% level, but not at the 99% level. The footnote (sig=.01 + sig=.05) indicates testing at both levels.

### 8.1.7 Using Nonstandard Confidence Levels

The program can test at confidence levels other than 99%, 95%, 90%, or 80% by using the APPROXIMATELY option in conjunction with the DO\_STATISTICS option on the EDIT statement. Use this option to test at any other confidence level that is desired. Be aware though that it uses a formula to determine whether an item is significant or not rather than looking up the t value in a table. This formula has the same error rates as the table values (i.e. its marking will be wrong the same percent of the time), but for values on the border of being significant it may be different.

**NOTE:** This option cannot be used in conjunction with the Newman-Keuls procedure.

This example is the same as Table 101, except for the DO\_STATISTICS option on the EDIT statement

```
TABLE_SET= { TAB104 :
STATISTICS=: I=BC, I=DEF, GHIJ
```

```
LOCAL_EDIT=: DO_STATISTICS= APPROXIMATELY .93 }  
HEADER=: TABLE WITH SIGNIFICANCE TESTING AT AN OTHER  
CONFIDENCE LEVEL (93%) }  
TITLE= TAB101  
TITLE_4= TAB101  
STUB= TAB101  
ROW= TAB101  
STORE_TABLES=* }
```

The printed table would look similar to Table 101, except for some of the statistical markings and the footnote. The footnote would be as follows:

(sig= apprx 0.07) (all\_pairs) columns tested BC, DEF, GHIJ

### 8.1.8 Inclusive T Tests

If you have a situation where you are testing a set of columns against the total column or against a column that contains all the values in those columns, you may want to mark the test as inclusive by using the T= option in front of the list of columns. This option will verify that all subsequent columns are contained in the first column in the list and report an error if it finds a data case that is not contained.

This example is the same as Table 101 except for the STATISTICS statement.

```
TABLE_SET= { TAB105:  
STATISTICS=: T=ABC,T=ADEF,GHIJ  
LOCAL_EDIT=: DO_STATISTICS=.95 }  
HEADER=:TABLE WITH INCLUSIVE SIGNIFICANCE TESTING AT THE  
95% CONFIDENCE LEVEL}  
TITLE= TAB101  
TITLE_4= TAB101
```

```
STUB= TAB101
ROW= TAB101
STORE_TABLES=* }
```

The printed table would look similar to Table 101, except for some of the statistical markings and the footnote. The footnote would be as follows:

```
(sig=.05) (all_pairs) columns tested T= ABC, T= ADEF,
GHIJ
```

Notice that the test of GHIJ is a dependent test, even though the others are inclusive.

## 8.2 Changing the Statistical Base

If the table is to represent percentages from a row other than the System Total row, you must specify which row will be the base. The program checks that the percentage base and the statistical base are the same row. If you change one, you must change the other. This is done to keep you from printing significance on cells that have no relationship to the percentage that is printing.

**NOTE:** Significance testing for means is unaffected by the statistical base, instead the base for each mean is calculated internally as part of calculating the mean.

### 8.2.1 Changing to the Any Response Row

If the System Any Response row is the percentage base, then you can change it to the default statistical base by using the SET option STATISTICS\_BASE\_AR. If you do not use this option and percentage off the Any Response row, the program will print an error message that the percentage base is different than the statistical base. If the percentage base is the System Any Response row, you will probably want to create a new STUB\_PREFACE to mark it as the base row for the test.

This STUB\_PREFACE should use the STUB option BASE\_ROW along with the PRINT\_ROW=AR, so that the program will print the percentage signs and column letters under the Any Response row. If the System Total row is also printed, the STUB option -BASE\_ROW on its definition can be used to suppress the percentage signs and letters from printing under it.

You can set the default statistical base back to the System-generated Total row by using the SET option -STATISTICS\_BASE\_AR. However, be careful not to make mistakes if you turn this option on and off multiple times in one run. It is recommended that in a given run you use either the System Total or System Any Response row as the percentage base and not to flip back and forth.

**NOTE:** The following set of commands defines a standard front end for the next set of examples

```
>PURGESAME
>PRINT_FILE STAT2
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T201

~DEFINE

TABLE_SET= { BAN1 :
EDIT=:
COLUMN_WIDTH=7,STUB_WIDTH=30,-COLUMN_TNA,STATISTICS_DECIMALS=2,
        -PERCENT_SIGN }
BANNER=:
|
|           SEX                AGE                ADVERTISING AWARENESS
|
|           <=====>  <=====>  <=====>
| TOTAL   MALE FEMALE  18-30  31-50  51-70 BRND A BRND B BRND C BRND D
| -----  -----  -----  -----  -----  -----  -----  -----  ----- }
```

```
COLUMN=: TOTAL WITH [5^1/2] WITH [6^1//3] WITH [7^1//4]
}
```

**Example:**

```
STUB= STUB_TOP_AR:
[-VERTICAL_PERCENT -BASE_ROW] TOTAL
[-VERTICAL_PERCENT] NO ANSWER
[PRINT_ROW=AR,BASE_ROW] TOTAL RESPONDING }

TABLE_SET= { TAB201:
SET STATISTICS_BASE_AR
STATISTICS=: I=BC,I=DEF,GHIJ
LOCAL_EDIT=: VERTICAL_PERCENT=AR,DO_STATISTICS=.95 }
STUB_PREFACE= STUB_TOP_AR
HEADER=: TABLE WITH SIGNIFICANCE TESTING BASED ON THE
ANY RESPONSE ROW }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= THOSE RESPONDING }
STUB=:
NET GOOD
| VERY GOOD
| GOOD
FAIR
NET POOR
| POOR
| VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
```

```
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: [12^4, 5/5/4/3/1, 2/2/1/X] $ [MEAN, STD, SE] [12]
STORE_TABLES=* }
```

STATISTICS (SIGNIFICANCE TESTING)

8.2 Changing the Statistical Base

Here is the table that is printed:

TABLE WITH SIGNIFICANCE TESTING BASED ON THE ANY  
RESPONSE ROW

TABLE 201

RATING OF SERVICE

BASE= THOSE RESPONDING

	SEX		AGE			ADVERTISING AWARENESS				
	TOTAL	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D
TOTAL	400	196	204	125	145	113	91	108	107	176
NO ANSWER	48	26	22	16	16	14	11	6	10	32
TOTAL RESPONDING	352	170	182	109	129	99	80	102	97	144
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD	171	102C	69	50	57	61DE	39	51	47	87I
	48.6	60.0	37.9	45.9	44.2	61.6	48.8	50.0	48.5	60.4
VERY GOOD	86	58C	28	29	34	21	19	29	23	48
	24.4	34.1	15.4	26.6	26.4	21.2	23.8	28.4	23.7	33.3
GOOD	85	44	41	21	23	40DE	20	22	24	39
	24.1	25.9	22.5	19.3	17.8	40.4	25.0	21.6	24.7	27.1
FAIR	81	38	43	22	40F	14	16	18	24	30
	23.0	22.4	23.6	20.2	31.0	14.1	20.0	17.6	24.7	20.8
NET POOR	76	18	58B	28	25	19	16	27J	20	19
	21.6	10.6	31.9	25.7	19.4	19.2	20.0	26.5	20.6	13.2
POOR	35	9	26B	11	15	9	7	13	11	10
	9.9	5.3	14.3	10.1	11.6	9.1	8.8	12.7	11.3	6.9



VERY POOR	41	9	32B	17	10	10	9	14J	9	9
	11.6	5.3	17.6	15.6	7.8	10.1	11.2	13.7	9.3	6.2
DON'T KNOW/REFUSED	24	12	12	9	7	5	9	6	6	8
	6.8	7.1	6.6	8.3	5.4	5.1	11.2	5.9	6.2	5.6
MEAN	3.43	3.84C	3.04	3.34	3.46	3.56	3.46	3.41	3.45	3.79HI
STD DEVIATION	1.32	1.15	1.35	1.44	1.25	1.24	1.33	1.42	1.27	1.20
STD ERROR	0.07	0.09	0.10	0.14	0.11	0.13	0.16	0.14	0.13	0.10

-----  
 (sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

## 8.2.2 Changing to Any Row in the Table

When doing significance testing, it is often useful to define a base row so that both you and the System can easily determine which row is the statistical base row. Although the base row is only needed when the statistical base is not the System Total or Any Response row, or if the base changes in the body of the table or when the table is weighted, it is often useful to create it. This will not only reduce possible errors, but also will allow easy modification of the table due to a future client request. Make sure that the stub option `BASE_ROW` is defined on any such row so that the table will be labeled properly.

To create a base row use the keyword `$(BASE)` on a variable definition followed by the definition of the base. Follow that definition with a set of empty brackets (i.e. `$( )`). This forces the program to go back to producing the default frequencies. The base definition actually creates two different categories in the variable. Therefore, the stub has to have two separate labels, one for each of the categories. If the table is not weighted then the two categories will have the same values in each, and one of them may be suppressed. However, if the table is weighted then the first row will have the weighted base and the second row will contain something called the effective base. See “8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES” for a definition of the effective base and why it is needed.

## STATISTICS (SIGNIFICANCE TESTING)

### 8.2 Changing the Statistical Base

In the example below, the System Total row and both categories created by the \$[BASE] would all produce the same numbers, so only the first category created by the \$[BASE] is printed. Notice the VERTICAL\_PERCENT=\* option on the base row to guarantee that it is both the statistical and percentage base.

```
STUB= STUB_TOP_SUP:
[SUPPRESS] TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= { TAB202:
STATISTICS=: I=BC, I=DEF, GHIJ
LOCAL_EDIT=: DO_STATISTICS=.95 }
STUB_PREFACE= STUB_TOP_SUP
HEADER=: TABLE WITH SIGNIFICANCE TESTING AND HAS A BASE
ROW SPECIFIED }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
[BASE_ROW, VERTICAL_PERCENT=*] TOTAL (STAT BASE)
[SUPPRESS] EFFECTIVE BASE
NET GOOD
| VERY GOOD
| GOOD
FAIR
NET POOR
| POOR
| VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
```

```

ROW=: $[BASE] TOTAL $[] [12^4,5/5/4/3/1,2/2/1/X]
$[MEAN,STD,SE] [12]
STORE_TABLES=* }

```

The printed table will look fundamentally the same as Table 201.

### 8.2.3 Changing in the Middle of a Table

If the table you wish to create has a percentage base which changes in the body of the table, then you must define a base row in the variable each time the percentage base changes. Suppose you were trying to do significance testing on a top box and bottom box table with a changing percent base (See “6.1.2 Top Box Tables with a Changing Percentage Base”). For purposes of this example, only two brands are actually shown.

For each top box the percentage base not only has to be defined after a \$[BASE] keyword, but the STUB options BASE\_ROW and VERTICAL\_PERCENTAGE=\* should be specified on each stub.

```

TABLE_SET= { TAB203 :
STATISTICS=: I=BC, I=DEF, GHIJ
LOCAL_EDIT=: DO_STATISTICS=.95 }
STUB_PREFACE= STUB_TOP_SUP
HEADER=:
TABLE WITH SIGNIFICANCE TESTING WITH A CHANGING
PERCENTAGE/STATISTICAL BASE }
TITLE=: SUMMARY OF RATING OF SERVICE FOR BRANDS A AND B }
TITLE_4=: BASE= THOSE WHO RATED THE BRAND }
STUB=:
[BASE_ROW,VERTICAL_PERCENT=*] TOTAL RATED BRAND A (STAT
BASE)
[SUPPRESS] EFFECTIVE BASE

```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.2 Changing the Statistical Base

```
TOP BOX
BOTTOM BOX
[BASE_ROW,VERTICAL_PERCENT=*] TOTAL RATED BRAND B (STAT
BASE)
[SUPPRESS] EFFECTIVE BASE
TOP BOX
BOTTOM BOX }
ROW=: $[BASE] [12^1-5] $[] [12^4,5/1,2] $[BASE]
[21^1-5] $[] [21^4,5/1,2]
STORE_TABLES=* }
```

Here is the table that is printed:

TABLE WITH SIGNIFICANCE TESTING WITH A CHANGING PERCENTAGE/STATISTICAL BASE

TABLE 203

SUMMARY OF RATING OF SERVICE FOR BRANDS A AND B  
BASE= THOSE WHO RATED THE BRAND

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL RATED BRAND A (STAT BASE)	328	170	100	122	94	71	96	91	136	
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	
	%	%	%	%	%	%	%	%	%	
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
TOP BOX	171	102C	69	50	57	61DE	39	51	47	87I
	52.1	64.6	40.6	50.0	46.7	64.9	54.9	53.1	51.6	64.0
BOTTOM BOX	76	18	58B	28	25	19	16	27J	20	19
	23.2	11.4	34.1	28.0	20.5	20.2	22.5	28.1	22.0	14.0
TOTAL RATED BRAND B (STAT BASE)	378	192	117	138	109	83	102	101	170	
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	
	%	%	%	%	%	%	%	%	%	
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
TOP BOX	175	106C	69	53	55	65DE	38	46	48	98H
	46.3	57.0	35.9	45.3	39.9	59.6	45.8	45.1	47.5	57.6
BOTTOM BOX	127	43	84B	39	45	35	28	42J	35J	41
	33.6	23.1	43.8	33.3	32.6	32.1	33.7	41.2	34.7	24.1

(sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

Notice that each base row is marked with percentage signs and the column letters designation, making it clear that the statistical base has changed.

## 8.3 Changing the Statistical Tests

Significance testing can produce two potential errors: marking a test as significant when it is not; or not marking a test as significant when it is. In general, the first type of error is the more dangerous of the two as it may cause decisions to be made based on incorrect assumptions. There are various ways to change the formulas thus reducing the possibility for this type of error.

There are five different sets of formulas that the program can use to do the significance testing.

- 1 The All Possible Pairs Test (default)
- 2 The Newman-Keuls Test
- 3 The ANOVA-Scan Test
- 4 The Fisher Test
- 5 The Kruskal-Wallis Test

The following sections briefly describe each type of test, how each one works, and when it might be appropriate to use a particular test. For more detailed information about each test, consult a statistics textbook. Several of the tests are included in *Appendix A: STATISTICAL FORMULAS*.

### 8.3.1 The All Possible Pairs Test

The All Possible Pairs Test (APP) is the default. When more than two columns are tested at a time, the APP test will act as though each pair in the set of columns is being tested individually. For instance, the same set of tests would be performed whether your STATISTICS statement was specified as ABC or AB,AC,BC. In

general, out of all the tests, the APP test is the most likely to mark cells as significantly different.

For an example that uses this test, see Table 101 in “8.1.4 Standard Significance Testing”.

### 8.3.2 The Newman-Keuls Test Procedure

In general, the Newman-Keuls procedure (N-K) is not as likely to mark a test as significant as the APP test. This is because it uses the additional information supplied by all the columns in the test to produce a more reliable result. When testing more than two columns at a time, the N-K procedure assumes the items are alike and uses this additional information in its calculations.

**NOTE:** Testing items that are not alike (such as males and people from Chicago) using the N-K procedure may cause results to be skewed. You should be careful when testing dissimilar items, especially when using any test other than APP.

The N-K procedure requires larger differences because you are testing like items. You would ordinarily expect a larger difference between the smallest and the largest in the group. As additional columns are added into the test, it will require larger and larger differences between cells in order to see them as significantly different.

For example, suppose you flipped two coins 100 times each. One coin came up heads 42 times and the other came up heads 56 times. If you tested just these two samples against each other, you might guess that the second coin was more likely to come up heads due to the design of the coin. Now suppose you flipped five more coins and got values for heads like 47, 51, 54, 50, and 46. If you now look at all seven coins at one time, you would probably conclude that the coins are not actually different, but that the difference in the values is due to expected randomness.

The N-K procedure is different from the APP test in the following ways:

- 1 The N-K procedure estimates which of the two columns is most likely to be different and tests it first. If they are not different, then it does not test any of the other pairs of columns. If they are different, then it looks for the next pair of columns that are most likely to be different and tests them. It continues this until it finds a pair that is not significantly different or until all the pairs have been tested. This procedure assumes that sample sizes for the columns being tested are similar, but does make corrections if they are not.
- 2 The N-K procedure uses a pooled variance. This pooled variance is calculated by using a formula to combine the variance for each sample in the test. Since this pooled variance is derived from a much larger sample size than any of the individual samples, it is a more reliable estimation of the true variance. The effect of the pooled variance can be eliminated by use of one of the other VARIANCE options on the EDIT statement. See “8.3.4 Changing the Variance”.
- 3 The N-K procedure requires slightly higher t values to mark items as significantly different based on how many columns are being tested.

To use the N-K procedure instead of the APP test, use the option `NEWMAN_KEULS_TEST` on the EDIT statement.

The EDIT statement below shows an example of doing the Newman-Keuls procedure at the 95% confidence level.

**NOTE:** The following set of commands defines a standard front end for the next set of examples

```
>PURGESAME
>PRINT_FILE STAT3
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T301

~DEFINE
```





```
STUB= STUBTOP1:
```

```
TOTAL
```

```
[SUPPRESS] NO ANSWER }
```

```
TABLE_SET= { BAN1:
```

```
EDIT=:
```

```
COLUMN_WIDTH=7,STUB_WIDTH=30,-COLUMN_TNA,STATISTICS_DEC  
IMALS=2,
```

```
-PERCENT_SIGN,RUNNING_LINES=1 }
```

```
STUB_PREFACE= STUBTOP1
```

```
BANNER=:
```

```
|          SEX          AGE          ADVERTISING AWARENESS  
|          <=====>  <=====>  <=====>  
| TOTAL  MALE FEMALE  18-30  31-50  51-70 BRND A BRND B BRND C BRND D  
| -----  -----  -----  -----  -----  -----  -----  -----  ----- }  
COLUMN=: TOTAL WITH [5^1/2] WITH [6^1//3] WITH [7^1//4]  
}
```

### Example:

```
TABLE_SET= { TAB301:
```

```
STATISTICS=: I=BC,I=DEF,GHIJ
```

```
LOCAL_EDIT=: DO_STATISTICS=.95 NEWMAN_KEULS_TEST }
```

```
HEADER=:
```

```
TABLE WITH SIGNIFICANCE TESTING USING THE NEWMAN-KEULS  
TESTING PROCEDURE }
```

```
TITLE=: RATING OF SERVICE }
```

```
TITLE_4=: BASE= TOTAL SAMPLE }
```

```
STUB=:
```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.3 Changing the Statistical Tests

```
NET GOOD
|  VERY GOOD
|  GOOD
FAIR
NET POOR
|  POOR
|  VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: [11^4,5/5/4/3/1,2/2/1/X] $ [MEAN,STD,SE] [11]
STORE_TABLES=* }
```

Here is the table that is printed:

TABLE WITH STATISTICAL TESTING USING THE NEWMAN-KEULS TESTING PROCEDURE

TABLE 301

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	196	204	125	145	113	91	108	107	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD	197	120C	77	57	63	74DE	42	55	54	105
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70C	32	35	38	27	19	32	30	60
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47DE	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28F	44F	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20	19	27J	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30B	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4

VERY POOR	44	9	35B	17	13	10	11	14	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
 (sig=.05) (n\_k) columns tested BC, DEF, GHIJ

If you compare this table with Table 101 you will notice that the VERY POOR row under the column BRND B is marked with a "J" in Table 101, but not in this table. This means that the Newman-Keuls procedure determined that this was an expected difference when testing four separate samples, while the All Possible Pairs Test determined it was a significant difference. Notice that the footnote at the bottom of the page now says (n\_k) instead of (all\_pairs).

### 8.3.3 Other Testing Procedures

There are three other less used tests available. They are the ANOVA-Scan (A/S), Fisher, and Kruskal-Wallis (K/W) tests. These tests only can be performed on independent samples, and Kruskal-Wallis only works on means.

The ANOVA-Scan and Fisher tests are similar in that both first perform an ANOVA (analysis of variance) on the set of columns, and then only test the individual columns if the ANOVA shows significance. The difference between the two is that the Fisher test, like Newman-Keuls, takes into account that multiple columns are being tested and therefore requires a larger t value to show significance. In general, these two tests are less likely than the APP test is to mark items as significant.

The Kruskal-Wallis test only works on means of independent samples created with the EDIT option COLUMN\_MEAN. It is normally done on rating scales and treats the ratings as ordinals instead of values. This means that it treats each point in the scale only as higher than the previous ones. In other words, a rating of 4 is not twice as high as 2, it is just two ratings higher.

For the ANOVA-Scan test use the EDIT option ANOVA\_SCAN, and for the Fisher test the EDIT option FISHER. The Kruskal-Wallis test is invoked by using the stub option DO\_STATISTICS=KRUSKAL\_WALLIS. See “8.5.3 Changing the Type of Test by Row” for examples of the Kruskal-Wallis test.

Here is an example of an ANOVA-Scan test. This example is similar to the one for Table 301, except for the EDIT option ANOVA\_SCAN and the STATISTICS statement that only specifies those tests that are independent.

```
TABLE_SET= { TAB302:
STATISTICS=: I=BC,I=DEF
LOCAL_EDIT=: DO_STATISTICS=.95,ANOVA_SCAN }
HEADER=: TABLE WITH STATISTICAL TESTING USING THE ANOVA
SCAN TEST }
TITLE= TAB301
TITLE_4= TAB301
STUB= TAB301
ROW= TAB301
STORE_TABLES=* }
```

The printed table would look similar to Table 301, except for some of the statistical markings and the footnote. The footnote would be as follows:

```
(sig=.05) (anova_scan) columns tested BC, DEF
```

Here is an example of the Fisher test. It is the same as the one for Table 302, except that the EDIT option FISHER is used instead of ANOVA\_SCAN. As with the ANOVA\_SCAN, only the independent columns are specified on the STATISTICS statement.

```
TABLE_SET= { TAB303:
STATISTICS=: I=BC,I=DEF
LOCAL_EDIT=: DO_STATISTICS=.95,FISHER }
HEADER=: TABLE WITH STATISTICAL TESTING USING THE FISHER
TEST }
TITLE= TAB301
TITLE_4= TAB301
STUB= TAB301
ROW= TAB301
STORE_TABLES=* }
```

The printed table would look similar to Table 301, except for some of the statistical markings and the footnote. The footnote would be as follows:

```
(sig=.05) (Fisher) columns tested BC, DEF
```

### REPEATED MEASURES OPTION

If you have a repeated measures test (everyone is in every banner point) or an incomplete replication (a person is in more than one of the banner points), then you must use the RM feature on the STATISTICS= statement with the ANOVA\_SCAN or Fisher.

#### Example:

```
STATISTICS=: RM=AB,RM=CD,RM=EFGH,RM=IJKL,RM=MNOP
```

This 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. In the case of complete replication, this is equivalent to a properly done repeated\_measures analysis of variance. In the case of incomplete replication, it has the same heuristic justification of that behind the use of the (incompletely based) correlation matrix in the Newman\_Keuls procedure.

Here are the steps in this procedure:

- 1 Both the fisher and ANOVA\_SCAN do an ANOVA first.
- 2 They both test for significance at the level specified.
- 3 If the ANOVA is significant the program goes ahead otherwise it stops.
- 4 If the program goes ahead, then:
  - the ANOVA\_SCAN does an All Possible Pairs test at the same significance level.
  - the Fisher does a more stringent test on the pairs. The Fisher test takes into account that multiple columns are being tested and requires a larger t value to show significance. The significance used is:

the original significance level

-----  
 (number of groups x (number of groups - 1)) / 2

where the number of groups is the number of points being tested in the banner.

In general, the ANOVA\_SCAN and Fisher are less likely to mark items as significant than the all possible pairs test. If you are weighting the data, you must use STATISTICS=RM even if this is an independent sample.

Following are two examples. Example One (stat1.spx) uses the All Possible Pairs test. Example Two (stat2.spx) reproduces these same tables using the ANOVA\_SCAN and Repeated Measures. Example Three (stat3.spx) again reproduces these same tables using the Fisher test and Repeated Measures.

### EXAMPLE ONE - USING THE ALL POSSIBLE PAIRS TEST

```
~comment stat1^spx
```

This job has 4 products but it is a 2 x 2 design. It is set up using multiple 80 column records. General information is on record 1.

```
first product seen data [2/12.8]
```

```
second product seen data [3/12.8]
```

```
third product seen data [4/12.8]
```

```
fourth product seen data [5/12.8]
```

```
first product seen [8/1] \ code 1 = green high salt
```

```
second product seen [8/7] \ code 2 = blue low salt
```

```
third product seen [8/13] / code 3 = blue high salt
```

```
fourth product seen [8/19] / code 4 = green low salt
```

```
~define
```

```
"define a variable to keep track of first pass through the tables
```

```
firstpass[27/59^1]
```

```
"define a variable to keep track of first product seen
```

```
firstseen[27/60^1]
```

```
proc= { proc1:
```

```
modify firstpass = true "this is first pass through with banner1
```

```
modify firstseen = true "this is first product seen
```



```
copy [27/01] = [8/01]
copy [27/12.8] = [2/12.8]
do_tables leave_open
```

```
modify firstseen = false "this is not the first product seen"
copy [8/01] = [8/07]
copy [2/12.8] = [3/12.8]
do_tables leave_open
```

```
copy [8/01] = [8/13]
copy [2/12.8] = [4/12.8]
do_tables leave_open
```

```
copy [8/01] = [8/19]
copy [2/12.8] = [5/12.8]
do_tables
```

```
modify firstpass = false "this is not the first pass through"
copy [8/01] = [27/01]
copy [2/12.8] = [27/12.8]
do_tables leave_open
```

```
copy [8/01] = [8/07]
copy [2/12.8] = [3/12.8]
do_tables leave_open
```

```
copy [8/01] = [8/13]
copy [2/12.8] = [4/12.8]
do_tables leave_open
```

```

copy [8/01] = [8/19]
copy [2/12.8] = [5/12.8]
do_tables
}

banner1: [8/01^2/1/3/4] with &
          ([8/01^3/2/1/4] by (total with firstseen))
banner2: [8/01^3/4/1/2] with &
          ((dud with dud with dud with dud) by (dud with
dud))

stub={summary:
      [freqonly] TOTAL
      [suppress] NO ANSWER
      }

stub={preface:
      [freqonly, -base_row]          TOTAL
      [freqonly]                    DON'T KNOW
      [base_row, print_row=ar, vper=*] STAT BASE
      }

edit={editsum: stub_preface=summary }

tabset=&
{global:
  global_edit={:
    -col_tna
    rank_if_indicated
    rank_column_base=1
  }
}

```



```

        -percent_sign
        stat_decimals=2
        putchars=-z--
        continue=top
        do_statistics=.90 + .95
        all_possible_pairs_test
    }
}

```

```

tabset=&
{ban1:
    edit={:
        col_width=6
        stub_width=24
        stub_preface=preface
    }
    stats=: ab,cd,efgh,ijkl
    ban={:

```

PRODUCT		TOTAL				FIRST POSITION			
PACKAGE	TYPE-TOTAL	<----->				<----->			
TYPE-TOTAL	<----->	BLUE	BLUE	GREEN	GREEN	BLUE	BLUE	GREEN	GREEN
<----->	HIGH LOWER	HIGH	LOWER	HIGH	LOWER	HIGH	LOWER	HIGH	LOWER
BLUE GREEN	SALT SALT	SALT	SALT	SALT	SALT	SALT	SALT	SALT	SALT
=====	=====	=====	=====	=====	=====	=====	=====	=====	=====

```

    }
    col=: banner1 when firstpass otherwise banner2
}

```

```

tabset={q101:

```

```

local_edit=editsum
title={: SUMMARY OF MEANS - DINNER ROLLS ATTRIBUTE
RATINGS
      }
stub={:
      [stats]      OVERALL APPEARANCE OF PRODUCT AND
PACKAGE
      [stats]      NATURAL OR PROCESSED LOOK
      [stats]      LOOKS MOIST OR DRY
      [stats]      FEELINGS ABOUT THE MOISTNESS OR
DRYNESS
      [stats]      OVERALL COLOR
      [stats]      LOOKS LIKE WHAT I EXPECTED
      [stats]      EXPECTED LIKING OF TASTE
      [stats]      ENVIRONMENTAL IMPACT OF PACKAGE
      }
row=: &
      $[mean] [2/12] &
      $[mean] [2/13] &
      $[mean] [2/14] &
      $[mean] [2/15] &
      $[mean] [2/16] &
      $[mean] [2/17] &
      $[mean] [2/18] &
      $[mean] [2/19]
}

tabset={q102:
title={: OVERALL APPEARANCE OF PRODUCT AND PACKAGE }
stub={:

```

```

[]      TOP THREE BOX (NET)
[]      TOP TWO BOX (NET)
[]      (9) LOVE IT
[]      (8)
[]      (7)
[]      (6)
[]      (5)
[]      (4)
[]      (3)
[]      (2)
[]      (1) HATE IT
[]      BOTTOM TWO BOX (NET)
[]      BOTTOM THREE BOX (NET)
[stats] MEAN
[stats] STD DEVIATION
[stats] STD ERROR
}
row=: &
      [2/12^7.9/8,9/9//1/1,2/1.3] $[mean,std,se]
[2/12]
}

~input stats^tr work_length=2070 total_length=2160
>printfile stat1^prt

~set
  statistics_base_ar
  drop_local_edit
''  stat_dump

```

```

~execute
read_proc=proc1

tabset=global
tabset=ban1
  tabset=q101  tab=*
  tabset=q102  tab=*

~end

```

## EXAMPLE TWO - USING THE ANOVA\_SCAN AND REPEATED MEASURES

```
~comment stat2^spx
```

This job has 4 products but it is a 2 x 2 design. It is set up using multiple 80 column records. General information is on record 1.

```

first product seen data [2/12.8]
second product seen data [3/12.8]
third product seen data [4/12.8]
fourth product seen data [5/12.8]

first product seen [8/1] \ code 1 = green high salt
second product seen [8/7] \ code 2 = blue low salt
third product seen [8/13] / code 3 = blue high salt
fourth product seen [8/19] / code 4 = green low salt

~define
'define a variable to keep track of first pass through the tables
firstpass[27/59^1]
'define a variable to keep track of first product seen
firstseen[27/60^1]

proc= { proc1:

```

```
modify firstpass = true 'this is first pass through with banner1
modify firstseen = true 'this is first product seen
copy [27/01] = [8/01]
copy [27/12.8] = [2/12.8]
do_tables leave_open

modify firstseen = false 'this is not the first product seen
copy [8/01] = [8/07]
copy [2/12.8] = [3/12.8]
do_tables leave_open

copy [8/01] = [8/13]
copy [2/12.8] = [4/12.8]
do_tables leave_open

copy [8/01] = [8/19]
copy [2/12.8] = [5/12.8]
do_tables

modify firstpass = false 'this is not the first pass through
copy [8/01] = [27/01]
copy [2/12.8] = [27/12.8]
do_tables leave_open

copy [8/01] = [8/07]
copy [2/12.8] = [3/12.8]
do_tables leave_open

copy [8/01] = [8/13]
copy [2/12.8] = [4/12.8]
do_tables leave_open

copy [8/01] = [8/19]
copy [2/12.8] = [5/12.8]
do_tables
}
```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.3 Changing the Statistical Tests

```
banner1: [8/01^2/1/3/4] with &
          ([8/01^3/2/1/4] by (total with firstseen))
banner2: [8/01^3/4/1/2] with &
          ((dud with dud with dud with dud) by (dud with
dud))

stub={summary:
      [freqonly] TOTAL
      [suppress] NO ANSWER
      }

stub={preface:
      [freqonly, -base_row] TOTAL
      [freqonly] DON'T KNOW
      [base_row, print_row=ar, vper=*] STAT BASE
      }

edit={editsum: stub_preface=summary }

tabset=&
{global:
  global_edit={:
    -col_tna
    rank_if_indicated
    rank_column_base=1
    -percent_sign
    stat_decimals=2
    putchars=-z--
    continue=top
```





```

do_statistics=.90 + .95
anova_scan
}

tabset=&
{ban1:
  edit={:
    col_width=6
    stub_width=24
    stub_preface=preface
  }
  stats=: rm=ab,rm=cd,rm=efgh,i=ijkl

ban={:
|          PRODUCT                TOTAL                FIRST POSITION
| PACKAGE  TYPE-TOTAL <-----> <----->
| TYPE-TOTAL <-----> BLUE  BLUE GREEN GREEN  BLUE  BLUE GREEN GREEN
| <-----> HIGH LOWER  HIGH LOWER  HIGH LOWER  HIGH LOWER  HIGH LOWER
| BLUE GREEN  SALT  SALT  SALT  SALT  SALT  SALT  SALT  SALT  SALT  SALT
| =====
|
}

col=: banner1 when firstpass otherwise banner2
}

```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.3 Changing the Statistical Tests

```
tabset={q101:
  local_edit=editsum
  title={: SUMMARY OF MEANS - DINNER ROLLS ATTRIBUTE
RATINGS
      }
  stub={:
    [stats]      OVERALL APPEARANCE OF PRODUCT AND
PACKAGE
    [stats]      NATURAL OR PROCESSED LOOK
    [stats]      LOOKS MOIST OR DRY
    [stats]      FEELINGS ABOUT THE MOISTNESS OR
DRYNESS
    [stats]      OVERALL COLOR
    [stats]      LOOKS LIKE WHAT I EXPECTED
    [stats]      EXPECTED LIKING OF TASTE
    [stats]      ENVIRONMENTAL IMPACT OF PACKAGE
      }
  row=: &
    $[mean] [2/12] &
    $[mean] [2/13] &
    $[mean] [2/14] &
    $[mean] [2/15] &
    $[mean] [2/16] &
    $[mean] [2/17] &
    $[mean] [2/18] &
    $[mean] [2/19]
  }

tabset={q102:
```

```

title={: OVERALL APPEARANCE OF PRODUCT AND PACKAGE }
stub={:
    []      TOP THREE BOX (NET)
    []      TOP TWO BOX (NET)
    []      (9) LOVE IT
    []      (8)
    []      (7)
    []      (6)
    []      (5)
    []      (4)
    []      (3)
    []      (2)
    []      (1) HATE IT
    []      BOTTOM TWO BOX (NET)
    []      BOTTOM THREE BOX (NET)
    [stats] MEAN
    [stats] STD DEVIATION
    [stats] STD ERROR
    }
row=: &
    [2/12^7.9/8,9/9//1/1,2/1.3] $[mean,std,se]
[2/12]
}

~input stats^tr work_length=2070 total_length=2160
>printfile stat2^prt

~set
    statistics_base_ar
    drop_local_edit

```

```
'' stat_dump
```

```

~execute
read_proc=proc1

tabset=global
tabset=ban1
  tabset=q101  tab=*
  tabset=q102  tab=*

~end

```

### EXAMPLE THREE - USING THE FISHER TEST AND REPEATED MEASURES

```

~comment stat3^spx

```

This job has 4 products but it is a 2 x 2 design. It is set up using multiple 80 column records. General information is on record 1.

```

  first product seen data  [2/12.8]
  second product seen data [3/12.8]
  third product seen data  [4/12.8]
  fourth product seen data [5/12.8]

  first product seen  [8/1]  \  code 1 = green high salt
  second product seen [8/7]  \  code 2 = blue low salt
  third product seen  [8/13] /  code 3 = blue high salt
  fourth product seen [8/19] /  code 4 = green low salt

~define
''define a variable to keep track of first pass through the tables
firstpass[27/59^1]
''define a variable to keep track of first product seen
firstseen[27/60^1]

```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.3 Changing the Statistical Tests

```
proc= { procl:

    modify firstpass = true      ''this is first pass through with banner1
    modify firstseen = true      ''this is first product seen
    copy [27/01] = [8/01]
    copy [27/12.8] = [2/12.8]
    do_tables leave_open

    modify firstseen = false     ''this is not the first product seen
    copy [8/01] = [8/07]
    copy [2/12.8] = [3/12.8]
    do_tables leave_open

    copy [8/01] = [8/13]
    copy [2/12.8] = [4/12.8]
    do_tables leave_open

    copy [8/01] = [8/19]
    copy [2/12.8] = [5/12.8]
    do_tables

    modify firstpass = false     ''this is not the first pass through
    copy [8/01] = [27/01]
    copy [2/12.8] = [27/12.8]
    do_tables leave_open

    copy [8/01] = [8/07]
    copy [2/12.8] = [3/12.8]
    do_tables leave_open

    copy [8/01] = [8/13]
    copy [2/12.8] = [4/12.8]
    do_tables leave_open

    copy [8/01] = [8/19]
    copy [2/12.8] = [5/12.8]
    do_tables

}

banner1: [8/01^2/1/3/4] with &
         ([8/01^3/2/1/4] by (total with firstseen))
banner2: [8/01^3/4/1/2] with &
         ((dud with dud with dud with dud) by (dud with dud))

stub={summary:
      [freqonly] TOTAL
```

```

[suppress] NO ANSWER
}

stub={preface:
      [freqonly, -base_row]          TOTAL
      [freqonly]                   DON'T KNOW
      [base_row, print_row=ar, vper=*] STAT BASE
}

edit={editsum: stub_preface=summary }

tabset=&
{global:
  global_edit={:
    -col_tna
    rank_if_indicated
    rank_column_base=1
    -percent_sign
    stat_decimals=2
    putchar=-z--
    continue=top
    do_statistics=.90 + .95
    fisher
  }
}

tabset=&
{ban1:
  edit={:
    col_width=6
    stub_width=24
    stub_preface=preface
  }
  stats=: rm=ab,rm=cd,rm=efgh,i=ijkl
  ban={:

```

PRODUCT		TOTAL				FIRST POSITION			
PACKAGE	TYPE-TOTAL	<----->				<----->			
TYPE-TOTAL	<----->	BLUE	BLUE	GREEN	GREEN	BLUE	BLUE	GREEN	GREEN
<----->	HIGH LOWER	HIGH	LOWER	HIGH	LOWER	HIGH	LOWER	HIGH	LOWER
BLUE GREEN	SALT SALT	SALT	SALT	SALT	SALT	SALT	SALT	SALT	SALT
=====	=====	=====	=====	=====	=====	=====	=====	=====	=====

```
    }  
    col=: banner1 when firstpass otherwise banner2  
}
```



```

tabset={q101:
  local_edit=editsum
  title={: SUMMARY OF MEANS - DINNER ROLLS ATTRIBUTE
RATINGS
    }
  stub={:
    [stats]    OVERALL APPEARANCE OF PRODUCT AND
PACKAGE
    [stats]    NATURAL OR PROCESSED LOOK
    [stats]    LOOKS MOIST OR DRY
    [stats]    FEELINGS ABOUT THE MOISTNESS OR
DRYNESS
    [stats]    OVERALL COLOR
    [stats]    LOOKS LIKE WHAT I EXPECTED
    [stats]    EXPECTED LIKING OF TASTE
    [stats]    ENVIRONMENTAL IMPACT OF PACKAGE
    }
  row=: &
    $[mean] [2/12] &
    $[mean] [2/13] &
    $[mean] [2/14] &
    $[mean] [2/15] &
    $[mean] [2/16] &
    $[mean] [2/17] &
    $[mean] [2/18] &
    $[mean] [2/19]
  }

tabset={q102:

```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.3 Changing the Statistical Tests

```
title={: OVERALL APPEARANCE OF PRODUCT AND PACKAGE }
stub={:
    []      TOP THREE BOX (NET)
    []      TOP TWO BOX (NET)
    []      (9) LOVE IT
    []      (8)
    []      (7)
    []      (6)
    []      (5)
    []      (4)
    []      (3)
    []      (2)
    []      (1) HATE IT
    []      BOTTOM TWO BOX (NET)
    []      BOTTOM THREE BOX (NET)
    [stats] MEAN
    [stats] STD DEVIATION
    [stats] STD ERROR
    }
row=: &
    [2/12^7.9/8,9/9//1/1,2/1.3] $[mean,std,se]
[2/12]
}

~input stats^tr work_length=2070 total_length=2160
>printfile stat3^prt

~set
    statistics_base_ar
    drop_local_edit
```

```

''      stat_dump

~execute
read_proc=proc1

tabset=global
tabset=ban1
  tabset=q101  tab=*
  tabset=q102  tab=*

~end
-

```

### 8.3.4 Changing the Variance

When determining significance the Mentor program can calculate the variance of a multiple column test in three different ways. It can use a pooled variance, a paired variance, or a separate variance. Pooled variance uses a formula that combines the variance of all the columns in a test. Paired variance combines the variance of each pair in a test. Separate variance calculates the variance for each item separately. The default variance for the All Possible Pairs Test is a paired variance, while the default variance for the Newman-Keuls test is a pooled variance.

You can change the default variance used by the program with one of these EDIT options: USUAL\_VARIANCE (default), POOLED\_VARIANCE, PAIRED\_VARIANCE, or SEPARATE\_VARIANCE.

The following is a table of the type of variance that is used when testing means using a STATISTICS statement that looks like STATISTICS= STAT1: ABCDE.

EDIT OPTION	All Possible Pairs	Newman-Keuls
-------------	--------------------	--------------

USUAL_VARIANCE	$\text{VAR}(a,b)$	$\text{VAR}(a,b,c,d,e)$
POOLED_VARIANCE	$\text{VAR}(a,b,c,d,e)$	$\text{VAR}(a,b,c,d,e)$
PAIRED_VARIANCE	$\text{VAR}(a,b)$	$\text{VAR}(a,b)$
SEPARATE_VARIANCE	$\text{VAR}(a)+\text{VAR}(b)$	$\text{VAR}(a)+\text{VAR}(b)$

Separate variance may be useful when trying to duplicate a formula from a textbook or some other program, but it only makes sense for independent or inclusive tests. Also, any overlap in the sample (after taking into account any inclusive test) will force separate variance to be ignored and the paired variance to be used.

None of this affects the testing done on percentages. The variance for percentage tests is always defined as the square root of (p times (1-p)), where p is the sum of the all frequencies in the test divided by all the sum of all the percentage bases in the test.

## 8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES

When doing significance testing with weighted data, it is recommended that you create the effective base row, even when the percentage base is the System Weighted Total or the System Weighted Any Response row. The effective base row is needed to verify any tests on weighted data.

The effective base is an estimation of how the weighting is affecting the test. It is the actual base number that is used when determining whether two samples are significantly different. The effective base will never be higher than the original unweighted base and will usually be slightly less. As the variance in the weights increases, the effective base decreases in order to compensate for the likely change in the percentages that will occur. Without this correction, some weighting factor could always be applied, which would make any item significantly greater than any other. Since the effective base is so integral to the test, it is recommended that it be printed on the table so that it can be determined how the weighting might be affecting the significance testing.



There are two different ways to create the effective base row. You can use either the \$[BASE] or \$[EFFECTIVE\_N] keywords. The \$[BASE] keyword creates two different rows, both of which are needed for the significance tests: the weighted total (which is needed to properly calculate all the percentages) and the effective base (which is used as the base for the significance test). The \$[EFFECTIVE\_N] keyword can be used to create the effective base when the percentage base is either the System Total or Any Response rows. In this case you do not need to again specify the percentage base because the system has already calculated it.

In the example below, the \$[BASE] keyword is used to create the effective base. A new STUB\_PREFACE is defined because the SET UNWEIGHTED\_TOP option is used to also produce an unweighted total row.

**NOTE:** The following set of commands defines a standard front end for the next set of examples

```
>PURGESAME
>PRINT_FILE STAT4
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T401

~DEFINE

TABLE_SET= { BAN1 :
EDIT=:
COLUMN_WIDTH=7,STUB_WIDTH=30,-COLUMN_TNA,STATISTICS_DECIMALS=2,
-PERCENT_SIGN,DO_STATISTICS=.95,RUNNING_LINES=1 }
BANNER=:
```

	SEX		AGE			ADVERTISING AWARENESS			
	<=====>		<=====>			<=====>			
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D
TOTAL									

```
| ----- }
COLUMN=: TOTAL WITH [5^1/2] WITH [6^1//3] WITH [7^1//4]
}
```

**Example:**

```
STUB= STUB_TOP_UNWGT:
[-VERTICAL_PERCENT] UNWEIGHTED TOTAL
[SUPPRESS] UNWEIGHTED NO ANSWER
[SUPPRESS] WEIGHTED TOTAL
[SUPPRESS] WEIGHTED NO ANSWER }

TABLE_SET= { TAB401:
WEIGHT=:
SELECT_VALUE([6^1//3/X],VALUES(1.021,.880,1.130,1))
SET UNWEIGHTED_TOP
STATISTICS=: I=BC,I=DEF,GHIJ;
STUB_PREFACE= STUB_TOP_UNWGT
HEADER=: WEIGHTED TABLE WITH STATISTICAL TESTING
(USING BASE ROW) }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
[BASE_ROW,VERTICAL_PERCENT=*] WEIGHTED TOTAL (% BASE)
[-VERTICAL_PERCENT] EFFECTIVE BASE (STAT BASE)
NET GOOD
| VERY GOOD
| GOOD
FAIR
NET POOR
```



```
| POOR
| VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: $[BASE] TOTAL $[] [11^4,5/5/4/3/1,2/2/1/X]
$[MEAN,STD,SE] [11]
STORE_TABLES=* }
```

STATISTICS (SIGNIFICANCE TESTING)  
 8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES

Here is the table that is printed:

WEIGHTED TABLE WITH STATISTICAL TESTING (USING BASE ROW)

TABLE 401

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
UNWEIGHTED TOTAL	400	196	204	125	145	113	91	108	107	176
WEIGHTED TOTAL (% BASE)	400	194	206	128	128	128	91	108	106	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
EFFECTIVE BASE (STAT BASE)	396	194	202	125	145	113	90	107	106	174
NET GOOD	200	120C	80	58	55	84DE	43	55	54	106G
	50.1	62.0	38.9	45.6	43.4	65.5	47.5	51.3	51.0	60.6
VERY GOOD	102	69C	33	36	33	31	19	32	30	60G
	25.4	35.7	15.8	28.0	26.2	23.9	21.1	29.5	27.8	33.9
GOOD	99	51	48	22	22	53DE	24	24	25	47
	24.6	26.3	23.1	17.6	17.2	41.6	26.3	21.8	23.2	26.7
FAIR	89	42	47	29F	39F	16	19	20	23	35
	22.3	21.8	22.8	22.4	30.3	12.4	20.7	18.3	21.7	19.9
NET POOR	83	17	65B	30	26	23	19	27J	22	24



STATISTICS (SIGNIFICANCE TESTING)  
8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES

	20.7	8.9	31.7	23.2	20.7	17.7	20.4	24.9	20.8	13.4
POOR	39	8	30B	12	15	11	8	13	13	12
	9.6	4.2	14.7	9.6	11.7	8.8	8.8	11.8	12.0	7.0
VERY POOR	44	9	35B	17	11	11	11	14J	9	11
	11.0	4.7	17.0	13.6	9.0	8.8	11.6	13.1	8.8	6.4
DON'T KNOW/REFUSED	28	14	14	11	7	6	10	6	7	11
	7.0	7.3	6.7	8.8	5.5	4.4	11.4	5.5	6.5	6.1
MEAN	3.47	3.91C	3.07	3.40	3.42	3.66	3.41	3.45	3.53	3.79GH
										I
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.31	1.40	1.30	1.20
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
(sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

In the preceding table, the unweighted total, weighted total, and effective base are all printed. Compare the three numbers and notice that the effective base is usually a little less than the unweighted total. This is because all the weights were close together (near 1.00), so the weighting did not substantially change the percentages on the table. Compare these percentages with those that were printed on Table 101. If the weights had a greater variance (for example, respondents were assigned weights between 5 and .2), the effective base would have been much less than the unweighted total.

To see how the effective base works, look at the TOTAL column in the above table. Notice that the weighted and unweighted totals are both 400, because weights were chosen to weight the sample back to its original size. Also notice that the effective base is only 396, which is due to the minor variation in the weights that were applied to this table. The formula for the effective base is as follows:

STATISTICS (SIGNIFICANCE TESTING)  
 8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES

EB= WEIGHTED TOTAL SQUARED DIVIDED BY THE SUM OF THE SQUARE OF EACH WEIGHT

Reproduce the number 396 from above by plugging in all the appropriate numbers from TABLE 401.

$$EB = (WT)^2 / (Fn * (Wn^2))$$

$$EB = ((400)^2) / ((125 * (1.021^2)) + (145 * (.880^2)) + (113 * (1.13^2)) + (17 * (1^2)))$$

$$EB = 160000 / (130.30 + 112.29 + 144.29 + 17)$$

$$EB = 160000 / 403.88$$

$$EB = 396.16$$

An important characteristic of the effective base is demonstrated in the 31-50 AGE column, where the unweighted total and the effective base are both 145 while the weighted total is only 128. Since the weighting on this table was based on AGE and everyone in that column was weighted by the same factor of 0.880, the weighted total drops to 128. However, since there is no variance in the weighting, the effective base remains unchanged. Furthermore, the percentages in that column are exactly the same as those in Table 101.

Exactly the same table could be produced by using the \$[EFFECTIVE\_N] keyword instead of the \$[BASE] keyword and a different STUB\_PREFACE.

```
STUB= STUB_TOP_WGT:
[-VERTICAL_PERCENT] UNWEIGHTED TOTAL
[SUPPRESS] UNWEIGHTED NO ANSWER
[BASE_ROW] WEIGHTED TOTAL
```

```
[SUPPRESS] WEIGHTED NO ANSWER }

TABLE_SET= { TAB402:
STUB_PREFACE= STUB_TOP_WGT
HEADER=: WEIGHTED TABLE WITH STATISTICAL TESTING (USING
EFFECTIVE_N) }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
[-VERTICAL_PERCENT] EFFECTIVE BASE (STAT BASE)
NET GOOD
|  VERY GOOD
|  GOOD
FAIR
NET POOR
|  POOR
|  VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: $ [EFFECTIVE_N] TOTAL $ [] [11^4,5/5/4/3/1,2/2/1/X]
$ [MEAN,STD,SE] [11]
STORE_TABLES=* }
```

The printed table would be basically the same as Table 401.

### 8.4.1 Weighted Tables with Different Weights

When performing significance testing in conjunction with applying different weights to different columns, use the SET option

MULTIPLE\_WEIGHT\_STATISTICS. This option allows significance testing on similarly weighted columns when the table has columns with varying weights. However, it does not allow a given respondent to have a different weight in the same test. You can test independent columns with different weights, but dependent columns must have the same weights applied to them. For instance, MULTIPLE\_WEIGHT\_STATISTICS allows significance testing on a table where both a weighted and unweighted total column have been created, but it does not allow the unweighted total to be tested against any of the weighted columns. If this statement is not used, then the program will print an error message if a STATISTICS statement is used in conjunction with a COLUMN\_SHORT\_WEIGHT or COLUMN\_WEIGHT table element.

The example below shows how to produce an unweighted total column and still do significance testing on the rest of the table. Notice in the STATISTICS statement that all the letters are one lower in the alphabet than previous statements because an additional category has been added to the column variable.

```
TABLE_SET= { TAB403 :
HEADER=: TABLE WITH STATISTICAL TESTING AND DIFFERENT
WEIGHTS APPLIED
TO DIFFERENT COLUMNS OF THE TABLE }
SET MULTIPLE_WEIGHT_STATISTICS
COLUMN_SHORT_WEIGHT=: TOTAL WITH &

SELECT_VALUE( [6^1//3/X] , VALUES(1.021, .880, 1.130, 1))
STATISTICS=: I=CD, I=EFG, HIJK;
BANNER=:

|
|          SEX                AGE                ADVERTISING AWARENESS
| UNWGHT  WGHT  <=====>  <=====>  <=====>
|  TOTAL  TOTAL  MALE FEMALE  18-30  31-50  51-70  BRND A BRND B BRND C BRND D
|  -----  -----  -----  -----  -----  -----  -----  -----  -----  -----}
COLUMN=:    TOTAL WITH TOTAL WITH [5^1/2] WITH [6^1//3] WITH [7^1//4]
TITLE= TAB402
TITLE_4= TAB402
```



```
STUB= TAB402  
ROW= TAB402  
STORE_TABLES=* }
```

STATISTICS (SIGNIFICANCE TESTING)  
 8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES

Here is the table that is printed:

TABLE WITH STATISTICAL TESTING AND DIFFERENT WEIGHTS  
 APPLIED TO DIFFERENT COLUMNS OF THE TABLE

TABLE 403

RATING OF SERVICE

BASE= TOTAL SAMPLE

	UNWGHT TOTAL	WGHT TOTAL	SEX		AGE			ADVERTISING AWARENESS			
			MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D
			(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)	(K)
UNWEIGHTED TOTAL	400	400	196	204	125	145	113	91	108	107	176
WEIGHTED TOTAL	400	400	194	206	128	128	128	91	108	106	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%	%
EFFECTIVE BASE (STAT BASE)	400	396	194	202	125	145	113	90	107	106	174
NET GOOD	197	200	120D	80	58	55	84EF	43	55	54	106H
	49.2	50.1	62.0	38.9	45.6	43.4	65.5	47.5	51.3	51.0	60.6
VERY GOOD	102	102	69D	33	36	33	31	19	32	30	60H
	25.5	25.4	35.7	15.8	28.0	26.2	23.9	21.1	29.5	27.8	33.9
GOOD	95	99	51	48	22	22	53EF	24	24	25	47
	23.8	24.6	26.3	23.1	17.6	17.2	41.6	26.3	21.8	23.2	26.7
FAIR	92	89	42	47	29G	39G	16	19	20	23	35
	23.0	22.3	21.8	22.8	22.4	30.3	12.4	20.7	18.3	21.7	19.9
NET POOR	83	83	17	65C	30	26	23	19	27K	22	24
	20.8	20.7	8.9	31.7	23.2	20.7	17.7	20.4	24.9	20.8	13.4
POOR	39	39	8	30C	12	15	11	8	13	13	12
	9.8	9.6	4.2	14.7	9.6	11.7	8.8	8.8	11.8	12.0	7.0

VERY POOR	44	44	9	35C	17	11	11	11	14K	9	11
	11.0	11.0	4.7	17.0	13.6	9.0	8.8	11.6	13.1	8.8	6.4
DON'T KNOW/REFUSED	28	28	14	14	11	7	6	10	6	7	11
	7.0	7.0	7.3	6.7	8.8	5.5	4.4	11.4	5.5	6.5	6.1
MEAN	3.46	3.47	3.91D	3.07	3.40	3.42	3.66	3.41	3.45	3.53	3.79HI
											J
STD DEVIATION	1.31	1.31	1.12	1.35	1.41	1.28	1.22	1.31	1.40	1.30	1.20
STD ERROR	0.07	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
 (sig=.05) (all\_pairs) columns tested CD, EFG, HIJK

## 8.5 PRINT PHASE STATISTICAL TESTING

Print phase statistical testing is calculated from the numbers that are printed on the table. This means that this

has several advantages. But it also has several drawbacks.

The advantages:

- Table processing time will be much faster.
- Tables can be loaded into table manipulation, altered, and still be tested.
- A different type of test can be performed on each row in the table, including a Kruskal-Wallis test on the COLUMN\_MEAN.
- The columns being tested can be changed without rereading any data.

The limitations:

- The columns must be independent or inclusive.
- The data cannot be weighted.

- Means created during the data reading phase with \$[MEAN] will not be tested. Only Means created using the EDIT option COLUMN\_MEAN can be tested.
- If testing errors are made, such as dependent or weighted columns are tested or inclusive tests are not marked as Inclusive, no error messages will be generated and possibly incorrect statistical markings will be printed on the table.

**NOTE:** Both print phase and table building phase significance testing can be performed on the same table, although it is recommended that each test different sets of columns.

### 8.5.1 EDIT Options

To produce print phase significance testing, you need to use the EDIT options DO\_PRINTER\_STATISTICS and DO\_STATISTICS\_TESTS instead of the STATISTICS statement that is used for table building phase tests. The DO\_PRINTER\_STATISTICS option lets the program know you are going to be doing print phase testing and the DO\_STATISTICS\_TESTS specifies the columns to be tested.

Like the STATISTICS statement, the DO\_STATISTICS\_TESTS option uses letters to designate which columns are being tested and a comma to separate multiple tests. Unlike the STATISTICS statement, all tests must be independent, (there is no I= option), however tests may be inclusive and must be marked as such by using T=. T values may be printed as with the STATISTICS statement, but the PRINTABLE\_T option cannot be used to error check the tests. See “8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES” for more information about printing T values for print phase based tests and an example of using the T= option.

To test the first three banner points against each other, the EDIT statement would be look something like:

```
EDIT= EDIT1: DO_PRINTER_STATISTICS,DO_STATISTICS_TEST=ABC }
```



The other EDIT options, such as DO\_STATISTICS= to set the confidence level, and NEWMAN\_KEULS\_TEST to perform a Newman Keuls test, work the same way with the same defaults.

**NOTE:** The following set of commands defines a standard front end for the next set of examples

```

>PURGESAME
>PRINT_FILE STAT5
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T501

~DEFINE

STUB= STUBTOP1:
[BASE_ROW] TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= { BAN1:
EDIT=:
COLUMN_WIDTH=7,STUB_WIDTH=30,-COLUMN_TNA,STATISTICS_DEC
IMALS=2,
      -PERCENT_SIGN,RUNNING_LINES=1 }
STUB_PREFACE= STUBTOP1
BANNER=:
|           SEX                AGE                ADVERTISING AWARENESS
|           <-----> <-----> <----->
| TOTAL   MALE FEMALE  18-30  31-50  51-70 BRND A BRND B BRND C BRND D
| -----}
COLUMN=: TOTAL WITH [5^1/2] WITH [6^1//3] WITH [7^1//4]
      }
  
```

**Example:**

```

TABLE_SET= { TAB501:
LOCAL_EDIT=:
DO_PRINTER_STATISTICS,DO_STATISTICS_TESTS=BC,DEF
          ALL_POSSIBLE_PAIRS_TEST DO_STATISTICS=.95

COLUMN_STATISTICS_VALUES=VALUES(,5,4,3,,2,1)
          COLUMN_MEAN,COLUMN_STD COLUMN_SE }
HEADER=: TABLE WITH STATISTICAL TESTING DONE DURING THE
PRINT PHASE }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
NET GOOD
|  VERY GOOD
|  GOOD
FAIR
NET POOR
|  POOR
|  VERY POOR
DON'T KNOW/REFUSED
[PRINT_ROW=MEAN] MEAN
[PRINT_ROW=STD] STD DEV
[PRINT_ROW=SE] STD ERR }
ROW=: [11^4,5/5/4/3/1,2/2/1/X]
STORE_TABLES=* }

```



Here is the table that is printed:

TABLE WITH STATISTICAL TESTING DONE ON THE PRINTED NUMBERS

TABLE 501

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	204	125	145	113	91	108	107	176	
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	
	%	%	%	%	%	%	%	%	%	
		(B)	(C)	(D)	(E)	(F)				
NET GOOD	197	120C	77	57	63	74DE	42	55	54	105
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70C	32	35	38	27	19	32	30	60
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47DE	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28F	44F	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20	19	27	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30B	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4

VERY POOR	44	9	35B	17	13	10	11	14	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79
STD DEV	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
 (sig=.05) (all\_pairs) columns tested BC, DEF

This table is very similar to Table 101 at the beginning of this chapter. The only difference is that columns G, H, I, and J are not tested in this table because they are not independent. Notice that the footnote has nothing in it to differentiate between tests done during the print phase or table building phase.

## 8.5.2 Changing the Confidence Level and the Type of Test

Changing the default significance levels or type of test procedure is done in exactly the same way as with the table building phase tests. For example, if you wanted to test at the 90% confidence level using the N-K test procedure you would add the options `DO_STATISTICS=.90` and `NEWMAN_KEULS_TEST` onto your EDIT statement. Bi-level testing and using the approximation formula are also specified the same way. See 8.1.3 through 8.1.7 for more information on the EDIT option `DO_STATISTICS`. See “8.4 SIGNIFICANCE TESTING ON WEIGHTED TABLES” for more information about changing the type of test.

TABLE\_SET= { TAB502 :

```

LOCAL_EDIT=:
DO_PRINTER_STATISTICS,DO_STATISTICS_TESTS=BC,DEF
      NEWMAN_KEULS_TEST,DO_STATISTICS=.90

COLUMN_STATISTICS_VALUES=VALUES(,5,4,3,,2,1)
      COLUMN_MEAN,COLUMN_STD,COLUMN_SE }

HEADER=: TABLE WITH STATISTICAL TESTING DONE DURING THE
PRINT PHASE

CHANGING THE CONFIDENCE LEVEL AND USING THE NEWMAN-KEULS
PROCEDURE }

TITLE= TAB501
TITLE_4= TAB501
STUB= TAB501
ROW= TAB501
STORE_TABLES=* }

```

The printed table would similar to Table 501 except for some of the statistical markings and the footnote. The footnote would be as follows:

```
(sig=.10) (n_k) columns tested BC, DEF
```

### 8.5.3 Changing the Type of Test by Row

When doing print phase statistical testing you can use the STUB option DO\_STATISTICS to change the type of test being performed on that row or to exclude that row entirely from testing. This means that you can test some of the rows using the APP test, test other rows using the N-K test, and not test other rows. DO\_STATISTICS can be set to any of ALL\_POSSIBLE\_PAIRS\_TEST, NEWMAN\_KEULS\_TEST, ANOVA\_SCAN, FISHER, or KRUSKAL\_WALLIS\_TEST. If -DO\_STATISTICS is used, then that row will be excluded from the test.

## STATISTICS (SIGNIFICANCE TESTING)

### 8.5 PRINT PHASE STATISTICAL TESTING

In the example below the NET GOOD, FAIR, and NET POOR are tested with the APP test as specified on the EDIT statement. The COLUMN\_MEAN is tested using the Kruskal-Wallis test. The rest of the rows are not tested.

```
TABLE_SET= { TAB503 :
LOCAL_EDIT=:
DO_PRINTER_STATISTICS,DO_STATISTICS_TESTS=BC,DEF
          ALL_POSSIBLE_PAIRS_TEST,DO_STATISTICS=.95

COLUMN_STATISTICS_VALUES=VALUES(,5,4,3,,2,1)
          COLUMN_MEAN,COLUMN_STD,COLUMN_SE }
HEADER=: TABLE WITH STATISTICAL TESTING DONE DURING THE
PRINT PHASE
USING THE KRUSKAL WALLIS TEST ON THE MEAN }
TITLE= TAB501
TITLE_4= TAB501
TITLE_5=:\2NNETS AND FAIR MENTIONS ARE TESTED USING ALL
PAIRS TEST
MEAN IS TESTED USING KRUSKAL WALLIS TEST }
STUB=:
NET GOOD
[-DO_STATISTICS] | VERY GOOD
[-DO_STATISTICS] | GOOD
FAIR
NET POOR
[-DO_STATISTICS] | POOR
[-DO_STATISTICS] | VERY POOR
[-DO_STATISTICS] DON'T KNOW/REFUSED
[PRINT_ROW=MEAN,DO_STATISTICS=KRUSKAL_WALLIS_TEST] MEAN
[PRINT_ROW=STD] STD DEV
[PRINT_ROW=SE] STD ERR }
```



```
ROW= TAB501  
STORE_TABLES=* }
```

STATISTICS (SIGNIFICANCE TESTING)

8.5 PRINT PHASE STATISTICAL TESTING

Here is the table that is printed:

TABLE WITH STATISTICAL TESTING DONE ON THE PRINTED NUMBERS

USING THE KRUSKAL WALLIS TEST ON THE MEAN

TABLE 503

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	204	125	145	113	91	108	107	176	
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	
	%	%	%	%	%	%	%	%	%	
		(B)	(C)	(D)	(E)	(F)				
NET GOOD	197	120C	77	57	63	74DE	42	55	54	105
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70	32	35	38	27	19	32	30	60
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28F	44F	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20	19	27	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4



**STATISTICS (SIGNIFICANCE TESTING)**  
**8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING**

VERY POOR	44	9	35	17	13	10	11	14	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79
STD DEV	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

NETS AND FAIR MENTIONS ARE TESTED USING ALL PAIRS TEST  
 MEAN IS TESTED USING KRUSKAL WALLIS TEST

-----  
 (sig=.05) (all\_pairs) (k\_w) columns tested BC, DEF

Notice that the standard footnote mentions that both the APP and Kruskal-Wallis tests were used, but does not say what rows were tested with which test. As in this example, you may want to use the TITLE\_5 keyword to create a customized footnote.

## 8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING

Often when doing significance testing there will be only a particular row or couple of rows in the table that are of interest. The processing time and the number of undesirable letters that print on the table can be greatly reduced by only performing the statistical tests on the specific rows.

The Mentor program can be instructed to test mean rows only, test only specified rows, or to drop columns with low sample sizes from the testing.

### 8.6.1 Testing Mean Rows Only

To perform statistical testing on mean rows only you can use the SET option MEAN\_STATISTICS\_ONLY. This option causes the program to ignore all testing for percentages. Return to testing both means and percentages by turning the option off with -MEAN\_STATISTICS\_ONLY.

**NOTE:** The following set of commands defines a standard front end for the next set of examples.

```

>PURGESAME
>PRINT_FILE STAT6
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T601

~DEFINE

STUB= STUB_TOP_TOT:
TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= { BAN1:
EDIT=: COLUMN_WIDTH=7,STUB_WIDTH=30,-
COLUMN_TNA,STATISTICS_DECIMALS=2,
        -PERCENT_SIGN,RUNNING_LINES=1 }
STATISTICS=: I=BC,I=DEF,GHIJ;
BANNER=:

|                SEX                AGE                ADVERTISING AWARENESS
|                <=====>  <=====>  <=====>
|  TOTAL  MALE FEMALE  18-30  31-50  51-70 BRND A BRND B BRND C BRND D
|  -----  -----  -----  -----  -----  -----  -----  -----  -----  -----  }

```



```
COLUMN=:    TOTAL WITH [5^1/2] WITH [6^1//3] WITH [7^1//4]
}
```

**Example:**

```
TABLE_SET= { TAB601:
STUB_PREFACE= STUB_TOP_TOT
SET MEAN_STATISTICS_ONLY
LOCAL_EDIT=: DO_STATISTICS=.95 }
HEADER=: TABLE WITH STATISTICAL TESTS PERFORMED ON MEANS
ONLY }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
NET GOOD
|  VERY GOOD
|  GOOD
FAIR
NET POOR
|  POOR
|  VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: [11^4,5/5/4/3/1,2/2/1/X] $ [MEAN,STD,SE] [11]
STORE_TABLES=* }
```

STATISTICS (SIGNIFICANCE TESTING)

8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING

Here is the table that is printed:

TABLE WITH STATISTICAL TESTS PERFORMED ON MEANS ONLY

TABLE 601

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	204	125	145	113	91	108	107	176	
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	
	%	%	%	%	%	%	%	%	%	
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD	197	77	57	63	74	42	55	54	105	
	49.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7	
VERY GOOD	102	32	35	38	27	19	32	30	60	
	25.5	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1	
GOOD	95	45	22	25	47	23	23	24	45	
	23.8	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6	
FAIR	92	48	28	44	14	20	20	24	36	
	23.0	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5	
NET POOR	83	65	29	30	20	19	27	22	24	
	20.8	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6	
POOR	39	30	12	17	10	8	13	13	13	
	9.8	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4	
VERY POOR	44	35	17	13	10	11	14	9	11	

**STATISTICS (SIGNIFICANCE TESTING)**  
*8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING*

	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79GH
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

-----  
 (sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

Compare this table with Table 101 and notice that only the MEAN row is marked with any letter because the tests on all the other rows were suppressed.

**NOTE:** There is no change in the footnote on the table, so a customized notation may want to be included somewhere on the table pointing out which rows were tested.

### 8.6.2 Excluding any Row from Statistical Testing

If only specific rows in the table are to be tested, you can either mark the rows that are to be tested or the rows that are to be excluded. If a simple variable is being defined the keyword STATISTICS may be used inside parentheses in front of the code as part of the data definition. If the variable definition is complex (it uses joiners or functions), then the keyword \$[DO\_STATISTICS] must be used to mark the parts of the table that will be tested and the keyword \$[-DO\_STATISTICS] to mark which parts of the table will not be tested.

## STATISTICS (SIGNIFICANCE TESTING)

### 8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING

For example, suppose you only wanted to test the top two box (codes 5 and 4) and the bottom two box (codes 1 and 2) in a 5 point scale stored in data position 21. Using the STATISTICS keyword method the variable would look like:

```
[21^(STATISTICS) 4, 5/5/4/3/ (STATISTICS) 1, 2/2/1]
```

This would cause only those categories marked with the STATISTICS keyword to be tested, while all other categories would not be tested.

Using the \$[DO\_STATISTICS] keyword method the variable would look like:

```
[21^4, 5] $[-DO_STATISTICS] [21^5/4/3] $[DO_STATISTICS]  
[21^1, 2] &  
$[-DO_STATISTICS] [21^2/1]
```

The default is that categories are tested so the net of 4 and 5 will be tested. All categories after \$[-DO\_STATISTICS] are not tested, while \$[DO\_STATISTICS] turns testing back on.

If table printing phase statistical testing is being done, you can exclude a row from the test by using the STUB option -DO\_STATISTICS. See “8.5.3 Changing the Type of Test by Row” for more information.

**NOTE:** The \$[DO\_STATISTICS] keyword should not be confused with either the EDIT statement option DO\_STATISTICS or the STUB option DO\_STATISTICS.

The following example shows how to test only the top box, bottom box, and mean on a rating scale:

```
TABLE_SET= { TAB602 :  
LOCAL_EDIT=: DO_STATISTICS=.95 }
```

STATISTICS (SIGNIFICANCE TESTING)  
8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING

```
HEADER=: TABLE WITH STATISTICAL TESTS PERFORMED ON
SELECTED ROWS ONLY }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
TITLE_5=:\2N ONLY ROWS WITH (*) ARE TESTED }
STUB=:
NET GOOD  (*)
|  VERY GOOD
|  GOOD
FAIR
NET POOR  (*)
|  POOR
|  VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN (*)
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: [11^(STATISTICS) 4,5/5/4/3/(STATISTICS) 1,2/2/1/X]
$ [MEAN,STD,SE] [11]
STORE_TABLES=* }
```

**STATISTICS (SIGNIFICANCE TESTING)**

*8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING*

Here is an alternate way to write the row variable:

```
ROW602A: [11^4, 5] $ [-DO_STATISTICS] [11^5/4/3] &
$ [DO_STATISTICS] [11^1, 2] $ [-DO_STATISTICS] [11^2/1/X]
&
$ [DO_STATISTICS, MEAN, STD, SE] [11]
```

Here is the table that is printed:

TABLE WITH STATISTICAL TESTS PERFORMED ON SELECTED ROWS ONLY

TABLE 602

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	196	204	125	145	113	91	108	107	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD (*)	197	120C	77	57	63	74DE	42	55	54	105G
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70	32	35	38	27	19	32	30	60
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28	44	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5



**STATISTICS (SIGNIFICANCE TESTING)**  
*8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING*

NET POOR (*)	83	18	65B	29	30	20	19	27J	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4
VERY POOR	44	9	35	17	13	10	11	14	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN (*)	3.46	3.90C	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79GH
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09

ONLY ROWS WITH (\*) ARE TESTED

-----  
 (sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

If you compare this table with Table 601 note that only the NET GOOD, NET POOR, and MEAN row in this table have statistical markings. In addition, a TITLE\_5 variable was defined to create a customized footnote.

### 8.6.3 Excluding Columns with Low Bases from Statistical Testing

If some of the columns in the test could have low bases you might want to exclude them from the testing. You may want to do this because the small bases might skew the tests, or because the sample is such that you do not want to report on any small base. The EDIT option MINIMUM\_BASE= can be used to suppress not only statistical testing but also all the other values in that column. If MINIMUM\_BASE is set to some value like 50, then any column that has a base less than 50, will print an asterisk (\*) under the base row where the letter usually prints, and the rest of the

## STATISTICS (SIGNIFICANCE TESTING)

### 8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING

column will be blank. The FLAG\_MINIMUM\_BASE option can be used in conjunction with MINIMUM\_BASE. Instead of blanking the column, the program will print all the numbers in that column followed by an asterisk where the statistical markings would normally print.

```
TABLE_SET= { TAB603 :
STATISTICS=: I=BC, I=DEF, GHIJ ;
LOCAL_EDIT=: MINIMUM_BASE=100, DO_STATISTICS=.95 }
HEADER=: USING MINIMUM BASE OPTION TO SUPPRESS A COLUMN
WITH A LOW BASE }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
NET GOOD
| VERY GOOD
| GOOD
FAIR
NET POOR
| POOR
| VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR }
ROW=: [11^4, 5/5/4/3/1, 2/2/1/X] $ [MEAN, STD, SE] [11]
STORE_TABLES=* }
```

**STATISTICS (SIGNIFICANCE TESTING)**  
**8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING**

Here is the table that is printed:

USING MINIMUM BASE OPTION TO SUPPRESS A COLUMN WITH A  
 LOW BASE

TABLE 603

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	196	204	125	145	113	91	108	107	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(*)	(H)	(I)	(J)
NET GOOD	197	120C	77	57	63	74DE		55	54	105
	49.2	61.2	37.7	45.6	43.4	65.5		50.9	50.5	59.7
VERY GOOD	102	70C	32	35	38	27		32	30	60
	25.5	35.7	15.7	28.0	26.2	23.9		29.6	28.0	34.1
GOOD	95	50	45	22	25	47DE		23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6		21.3	22.4	25.6
FAIR	92	44	48	28F	44F	14		20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4		18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20		27J	22	24
	20.8	9.2	31.9	23.2	20.7	17.7		25.0	20.6	13.6
POOR	39	9	30B	12	17	10		13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8		12.0	12.1	7.4

**STATISTICS (SIGNIFICANCE TESTING)**

*8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING*

VERY POOR	44	9	35B	17	13	10	14J	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.45	3.53	3.79H
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.14	0.13	0.09

-----

(sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

\* - small base

Notice that the column BRND A is blank except for the base value. Also notice that the footnote includes a note that the asterisk denotes a small base.

You can suppress only the statistical testing instead of the entire column by also using the EDIT option FLAG\_MINIMUM\_BASE. In the example below the only difference from Table 603 is this option.

```
TABLE_SET= { TAB604:
HEADER=: USING MINIMUM BASE OPTION TO FLAG A COLUMN
WITH A LOW BASE }
LOCAL_EDIT=:
MINIMUM_BASE=100, FLAG_MINIMUM_BASE, DO_STATISTICS=.95 }
TITLE= TAB603
TITLE_4= TAB603
STUB= TAB603
ROW= TAB603
STORE_TABLES=* }
```

**STATISTICS (SIGNIFICANCE TESTING)**  
*8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING*



STATISTICS (SIGNIFICANCE TESTING)

8.6 EXCLUDING ROWS/COLUMNS FROM SIGNIFICANCE TESTING

Here is the table that is printed:

USING MINIMUM BASE OPTION TO FLAG A COLUMN WITH A LOW BASE

TABLE 604

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	TOTAL	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D
TOTAL	400	196	204	125	145	113	91	108	107	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
		(B)	(C)	(D)	(E)	(F)	(*)	(H)	(I)	(J)
NET GOOD	197	120C	77	57	63	74DE	42*	55	54	105
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
VERY GOOD	102	70C	32	35	38	27	19*	32	30	60
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95	50	45	22	25	47DE	23*	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92	44	48	28F	44F	14	20*	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83	18	65B	29	30	20	19*	27J	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
POOR	39	9	30B	12	17	10	8*	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4

**STATISTICS (SIGNIFICANCE TESTING)**  
*8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES*

VERY POOR	44	9	35B	17	13	10	11*	14J	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10*	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46	3.90C	3.05	3.40	3.42	3.66	3.38*	3.45	3.53	3.79H
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32*	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15*	0.14	0.13	0.09

-----

(sig=.05) (all\_pairs) columns tested BC, DEF, GHIJ

\* - small base

## 8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES

Both t values and the significance of the t value can be printed on the table either in addition to or instead of the statistical letter markings. The STUB options DO\_T\_TEST and DO\_SIG\_T are used to print these values. If you are printing values from a STATISTICS statement you can also use PRINTABLE\_T for error checking. PRINTABLE\_T checks that the STAT= tests are only in pairs and that no column is the second column in any pair more than once (you can only print one T value per column).

The STUB options DO\_T\_TEST and DO\_SIG\_T can be set to any of the following:

- DO\_T\_TEST=\*Print the t value for the last data row seen
- DO\_T\_TEST=n Print the t value for the Nth row in the table
- DO\_T\_TEST=-nPrint the t value for the Nth row above this row in the table
- DO\_T\_TEST=PRINT\_MEANPrint the t value for the COLUMN\_MEAN

## STATISTICS (SIGNIFICANCE TESTING)

### 8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES

In the example below the t values and their significance are printed for the top box, bottom box, and mean rows in the table.

**NOTE:** The following set of commands defines a standard front end for the next set of examples

```
>PURGESAME
>PRINT_FILE STAT7
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T701

~DEFINE

STUB= STUB_TOP_TOT:
TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= { BAN1:
EDIT=:
COLUMN_WIDTH=7,STUB_WIDTH=30,-COLUMN_TNA,STATISTICS_DEC
IMALS=2,
        -PERCENT_SIGN }
BANNER=:
|
|          SEX          AGE          ADVERTISING AWARENESS
|
|  <----->  <----->  <----->
|  TOTAL  MALE FEMALE  18-30  31-50  51-70 BRND A BRND B BRND C BRND D
|  -----}
COLUMN=: TOTAL WITH [5^1/2] WITH [6^1/3] WITH [7^1/4]
}
```





**Example:**

```
TABLE_SET= { TAB701:
STUB_PREFACE= STUB_TOP_TOT
STATISTICS=: PRINTABLE_T
T=AB, T=AC, T=AD, T=AE, T=AF, T=AG, T=AH, T=AI, T=AJ
LOCAL_EDIT=: DO_STATISTICS=.95 }
HEADER=: TABLE WITH T AND SIGNIFICANCE VALUES PRINTED ON
THE TABLE }
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
NET GOOD
[DO_T_TEST=*,SKIP_LINES=0] T-VALUE
[DO_SIG_T=*,SKIP_LINES=0] SIGNIFICANCE
| VERY GOOD
| GOOD
FAIR
NET POOR
[DO_T_TEST=*,SKIP_LINES=0] T-VALUE
[DO_SIG_T=*,SKIP_LINES=0] SIGNIFICANCE
| POOR
| VERY POOR
DON'T KNOW/REFUSED
[STATISTICS_ROW] MEAN
[STATISTICS_ROW] STD DEVIATION
[STATISTICS_ROW] STD ERROR
[DO_T_TEST=9] T-VALUE
[DO_SIG_T=9] SIGNIFICANCE OF T }
ROW=: [11^4,5/5/4/3/1,2/2/1/X] $ [MEAN,STD,SE] [11]
```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES

```
STORE_TABLES=* }
```

**STATISTICS (SIGNIFICANCE TESTING)**  
*8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES*

Here is the table that is printed:

TABLE WITH T AND SIGNIFICANCE VALUES PRINTED ON THE TABLE

TABLE 701

RATING OF SERVICE

BASE= TOTAL SAMPLE

	SEX		AGE			ADVERTISING AWARENESS				
	MALE	FEMALE	18-30	31-50	51-70	BRND A	BRND B	BRND C	BRND D	
TOTAL	400	196	204	125	145	113	91	108	107	176
	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%	%	%	%	%
	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
NET GOOD	197C	120A	77	57	63	74A	42	55	54	105A
	49.2	61.2	37.7	45.6	43.4	65.5	46.2	50.9	50.5	59.7
T-VALUE		-4.69	4.69	0.98	1.75	-4.07	0.67	-0.41	-0.29	-3.69
SIGNIFICANCE		0.00	0.00	0.33	0.08	0.00	0.51	0.69	0.77	0.00
VERY GOOD	102C	70A	32	35	38	27	19	32	30	60A
	25.5	35.7	15.7	28.0	26.2	23.9	20.9	29.6	28.0	34.1
GOOD	95E	50	45	22	25	47A	23	23	24	45
	23.8	25.5	22.1	17.6	17.2	41.6	25.3	21.3	22.4	25.6
FAIR	92F	44	48	28	44A	14	20	20	24	36
	23.0	22.4	23.5	22.4	30.3	12.4	22.0	18.5	22.4	20.5
NET POOR	83BJ	18	65A	29	30	20	19	27	22	24
	20.8	9.2	31.9	23.2	20.7	17.7	20.9	25.0	20.6	13.6
T-VALUE		5.58	-5.58	-0.81	0.02	0.94	-0.03	-1.27	0.06	3.11
SIGNIFICANCE		0.00	0.00	0.42	0.98	0.35	0.97	0.20	0.95	0.00
POOR	39B	9	30A	12	17	10	8	13	13	13
	9.8	4.6	14.7	9.6	11.7	8.8	8.8	12.0	12.1	7.4

**STATISTICS (SIGNIFICANCE TESTING)**

*8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES*

VERY POOR	44BJ	9	35A	17	13	10	11	14	9	11
	11.0	4.6	17.2	13.6	9.0	8.8	12.1	13.0	8.4	6.2
DON'T KNOW/REFUSED	28	14	14	11	8	5	10	6	7	11
	7.0	7.1	6.9	8.8	5.5	4.4	11.0	5.6	6.5	6.2
MEAN	3.46C	3.90A	3.05	3.40	3.42	3.66	3.38	3.45	3.53	3.79A
STD DEVIATION	1.31	1.12	1.35	1.41	1.28	1.22	1.32	1.40	1.29	1.21
STD ERROR	0.07	0.08	0.10	0.13	0.11	0.12	0.15	0.14	0.13	0.09
T-VALUE		-6.58	6.58	0.57	0.44	-1.84	0.62	0.10	-0.60	-4.38
SIGNIFICANCE OF T		0.00	0.00	0.57	0.67	0.06	0.54	0.91	0.55	0.00

-----  
 (sig=.05) (all\_pairs) columns tested T= AB, T= AC, T= AD, T= AE, T= AF,  
 T= AG, T= AH, T= AI, T= AJ

Notice that t values are positive when the first item in the cell is greater than the second item, and negative when the opposite is true. Also notice that any cell with a significance of 0.05 or less is either marked with the letter A (negative t value) or the Total column is marked with its letter (positive t value). Further notice that the t values for males and females are opposites of each other. This is because each is being tested inclusively against the total which is actually the same as testing them against each other.

Basically the same table could be produced using the print phase tests. For more information on print phase tests see “8.5 PRINT PHASE STATISTICAL TESTING”. Here is an example of printing the t values when doing print phase tests.

**STATISTICS (SIGNIFICANCE TESTING)**  
*8.7 PRINTING THE ACTUAL T AND SIGNIFICANCE VALUES*



```
TABLE_SET= { TAB702:
LOCAL_EDIT=:
DO_PRINTER_STATISTICS,ALL_POSSIBLE_PAIRS_TEST,
    DO_STATISTICS=.95,

DO_STATISTICS_TESTS=T=AB,T=AC,T=AD,T=AE,T=AF,T=AG,T=AH,
T=AI,T=AJ

    COLUMN_STATISTICS_VALUES=VALUES(,5,4,3,,2,1)
    COLUMN_MEAN,COLUMN_STD,COLUMN_SE }
HEADER=: TABLE WITH T AND SIGNIFICANCE VALUES PRINTED ON
THE TABLE
FOR TESTS PERFORMED ON THE NUMBERS ON THE PRINTED TABLE
}
TITLE=: RATING OF SERVICE }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
NET GOOD
[DO_T_TEST=*,SKIP_LINES=0] T-VALUE
[DO_SIG_T=*,SKIP_LINES=0] SIGNIFICANCE
| VERY GOOD
| GOOD
FAIR
NET POOR
[DO_T_TEST=*,SKIP_LINES=0] T-VALUE
[DO_SIG_T=*,SKIP_LINES=0] SIGNIFICANCE
| POOR
| VERY POOR
DON'T KNOW/REFUSED
[PRINT_ROW=MEAN] MEAN
```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)

```
[PRINT_ROW=STD] STANDARD DEVIATION
[PRINT_ROW=SE] STANDARD ERROR
[DO_T_TEST=PRINT_MEAN] T-VALUE
[DO_SIG_T=PRINT_MEAN] SIGNIFICANCE OF T }
ROW=: [11^4, 5/5/4/3/1, 2/2/1/X]
STORE_TABLES=* }
```

The printed table will look basically the same as Table 701.

## 8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)

Significance testing on rows can be performed in two ways: a direct comparison test or a distributed preference test. In both cases only two rows may be compared at one time, although multiple pairs of rows may be compared in a single table. As with column testing, the STATISTICS statement and the DO\_STATISTICS option on the EDIT statement control the tests.

**NOTE:** Row testing cannot be performed during the table-printing phase.

On the STATISTICS statement, rows are designated numerically rather than alphabetically as are the columns. The first data row is assigned the number "1", the second data row the number "2", and so on. Every data row is included in this count even if it is not printed. To do a direct comparison of rows use the letter D followed by an equal sign (=) before the two row numbers. Separate the two row numbers with a comma. Separate different pairs of rows with a space. To do a distributed preference test use P instead of D. Rows can only be tested sequentially and a given row may only be in one test on the table. For example, if rows 1 and 4 are being compared, then rows 5 and 6 could also be compared, but row 3 could not be compared with row 7 (row 4 being in between), nor could row 1 be compared to row 2 (row 1 is already being compared to row 4).



### 8.8.1 Direct Comparison Testing

A direct comparison of two rows is similar to the test that is performed on columns, except that the letter D must be specified to indicate that it is a direct test. To do a direct comparison of rows 1 and 2 use the following STATISTICS statement:

```
STATISTICS= ROWSTAT1: D=1,2
```

The following statement would test row 1 versus row 2, row 3 versus row 4, and row 5 versus row 6.

```
STATISTICS= ROWSTAT2: D=1,2 D=3,4 D=5,6
```

Row testing can be combined with column testing by specifying both the column and row tests on the same STATISTICS statement. The following statement would do column testing on columns B, C, and D, in addition to testing row 4 versus row 6.

```
STATISTICS= ROWSTAT3: BCD, D=4,6
```

The DO\_STATISTICS option on the EDIT statement is again used to set the confidence level. The same setting is used for both the row and column tests. As with column testing, a footnote will be printed to indicate which rows were tested and the significance level used. If the difference is significant, a lower case "s" will print under the second row tested.

## STATISTICS (SIGNIFICANCE TESTING)

### 8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)

This is an example of a direct comparison of rows.

**NOTE:** The following set of commands defines a standard front end for the next set of examples

```
>PURGESAME
>PRINT_FILE STAT8
~INPUT DATA
~SET DROP_LOCAL_EDIT,BEGIN_TABLE_NAME=T801

~DEFINE

STUB= STUBTOP1:
TOTAL
[SUPPRESS] NO ANSWER }

TABLE_SET= { BAN1:
EDIT=:
COLUMN_WIDTH=7, STUB_WIDTH=30, -COLUMN_TNA, STUB_PREFACE=S
TUBTOP1,

STATISTICS_DECIMALS=2, -PERCENT_SIGN, DO_STATISTICS=1 }
BANNER=:
|
|          SEX          AGE
|          <=====>  <=====>
|  TOTAL  MALE FEMALE  18-30  31-50  51-70
|  -----  -----  -----  -----  ----- }
COLUMN=:  TOTAL WITH [5^1/2] WITH [6^1/3]
}
```

And here is this example:



**STATISTICS (SIGNIFICANCE TESTING)**  
*8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)*



```
TABLE_SET= { TAB801:
STATISTICS=: D=1,2 D=4,5 D=7,8
HEADER=:
TABLE WITH DIRECT STATISTICAL TESTING OF ROWS AT THE 95%
CONFIDENCE LEVEL}
TITLE=: PREFERENCE OF PRODUCTS }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
[COMMENT,UNDERLINE]  FIRST TEST
[STUB_INDENT=2]  PREFER BRAND A
[STUB_INDENT=2]  PREFER BRAND B
[STUB_INDENT=2]  NO PREFERENCE A VS B
[COMMENT,UNDERLINE]  SECOND TEST
[STUB_INDENT=2]  PREFER BRAND C
[STUB_INDENT=2]  PREFER BRAND D
[STUB_INDENT=2]  NO PREFERENCE C VS D
[COMMENT,UNDERLINE]  THIRD TEST
[STUB_INDENT=2]  PREFER BRAND E
[STUB_INDENT=2]  PREFER BRAND F
[STUB_INDENT=2]  NO PREFERENCE E VS F
}
ROW=: [7,8,9^1/2/X]
STORE_TABLES=* }
```

STATISTICS (SIGNIFICANCE TESTING)

8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)

Here is the table that is printed:

TABLE WITH DIRECT STATISTICAL TESTING OF ROWS AT THE  
95% CONFIDENCE LEVEL

TABLE 801

PREFERENCE OF PRODUCTS

TITLE\_4=: BASE= TOTAL SAMPLE

	TOTAL	SEX		AGE		
		MALE	FEMALE	18-30	31-50	51-70
TOTAL	500	251	249	140	223	101
	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%

FIRST TEST

-----

(a) PREFER BRAND A	236	111	125	88	98	30
	47.2	44.2	50.2	62.9	43.9	29.7
(b) PREFER BRAND B	214	113	101	43	101	59
	42.8	45.0	40.6	30.7	45.3	58.4

S

S

NO PREFERENCE A VS B	50	27	23	9	24	12
	10.0	10.8	9.2	6.4	10.8	11.9

SECOND TEST

-----

STATISTICS (SIGNIFICANCE TESTING)  
8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)

(d)	PREFER BRAND C	266	125	141	78	121	48
		53.2	49.8	56.6	55.7	54.3	47.5
(e)	PREFER BRAND D	190	108	82	49	84	43
		38.0	43.0	32.9	35.0	37.7	42.6
			s		s	s	s
	NO PREFERENCE C VS D	44	18	26	13	18	10
		8.8	7.2	10.4	9.3	8.1	9.9
THIRD TEST							
-----							
(g)	PREFER BRAND E	248	132	116	87	93	44
		49.6	52.6	46.6	62.1	41.7	43.6
(h)	PREFER BRAND F	187	87	100	40	96	42
		37.4	34.7	40.2	28.6	43.0	41.6
			s	s		s	
	NO PREFERENCE E VS F	65	32	33	13	34	15
		13.0	12.7	13.3	9.3	15.2	14.9

-----  
(sig=.05) (all\_pairs) rows tested a/b, d/e, g/h

Notice the "s" in the FEMALE column underneath the PREFER BRAND D row. This indicates that there is a significant difference between PREFER BRAND C and PREFER BRAND D for females. The blank under the MALE column in that row indicates that there is no significant difference for males. Also notice the additional lower case letter assigned to each row that was tested. This allows easy identification of which rows were tested against each other when compared to the footnote that prints at the bottom of the page.

## 8.8.2 Distributed Preference Testing

A distributed preference test allows a "No Preference" (or similar neutral third category) to be distributed between the two original categories while ensuring the integrity of the underlying statistical test. This is usually done for cosmetic purposes so that the percentages of the two preference categories add up to 100 percent.

The rules for the STATISTICS statement are the same as for the direct comparison, except the letter P is used instead of a D. To do a distributed preference test on rows 1 and 2, use the following STATISTICS statement:

```
STATISTICS= ROWSTAT4: P=1, 2
```

In a distributed preference test, the SELECT\_VALUE function is used to define the row variable. This ensures that the "No preference" response is evenly divided between the two categories (see "9.3.2 Functions" for more information on the SELECT\_VALUE function). A typical row definition might look like this:

```
ROW= : SELECT_VALUE ( [7^1/X] , VALUES (1 , .5) ) WITH &  
      SELECT_VALUE ( [7^2/X] , VALUES (1 , .5) )
```

This causes the X punch ("No preference") to have a value of .5 for both categories, splitting it evenly between the two.

As with the direct comparison, significant differences are marked with an "s" underneath the second row being tested. However, unlike the direct comparison test, small (not significant) differences are marked with a lower case "ns" and statistically equal rows are marked with a lower case "e".

The following example uses a distributed preference test to compare the same rows used in Table 801. Note the difference in the row variable definition.

**STATISTICS (SIGNIFICANCE TESTING)**  
*8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)*



```
TABLE_SET= { TAB802:
STATISTICS=: P=1,2 P=3,4 P=5,6
HEADER=:
TABLE WITH DISTRIBUTED PREFERENCE TESTING OF ROWS AT THE
95% CONFIDENCE LEVEL}
TITLE=: PREFERENCE OF PRODUCTS }
TITLE_4=: BASE= TOTAL SAMPLE }
STUB=:
[COMMENT,UNDERLINE]  FIRST TEST
[STUB_INDENT=2]  PREFER BRAND A
[STUB_INDENT=2]  PREFER BRAND B
[COMMENT,UNDERLINE]  SECOND TEST
[STUB_INDENT=2]  PREFER BRAND C
[STUB_INDENT=2]  PREFER BRAND D
[COMMENT,UNDERLINE]  THIRD TEST
[STUB_INDENT=2]  PREFER BRAND E
[STUB_INDENT=2]  PREFER BRAND F
}

ROW=: SELECT_VALUE ([7^1/X],VALUES (1, .5)) WITH &
      SELECT_VALUE ([7^2/X],VALUES (1, .5)) WITH &
      SELECT_VALUE ([8^1/X],VALUES (1, .5)) WITH &
      SELECT_VALUE ([8^2/X],VALUES (1, .5)) WITH &
      SELECT_VALUE ([9^1/X],VALUES (1, .5)) WITH &
      SELECT_VALUE ([9^2/X],VALUES (1, .5))
STORE_TABLES=* }
```

Here is the table that is printed:

STATISTICS (SIGNIFICANCE TESTING)

8.8 SIGNIFICANCE TESTING ON ROWS (PREFERENCE TESTING)

TABLE WITH DISTRIBUTED PREFERENCE TESTING OF ROWS AT THE 95% CONFIDENCE LEVEL

TABLE 802

PREFERENCE OF PRODUCTS

TITLE\_4=: BASE= TOTAL SAMPLE

	TOTAL	SEX		AGE		
		MALE	FEMALE	18-30	31-50	51-70
TOTAL	500	251	249	140	223	101
	100.0	100.0	100.0	100.0	100.0	100.0
	%	%	%	%	%	%

FIRST TEST

-----

(a)	PREFER BRAND A	261	124	136	92	110	36
		52.2	49.6	54.8	66.1	49.3	35.6
(b)	PREFER BRAND B	239	126	112	48	113	65
		47.8	50.4	45.2	33.9	50.7	64.4
		e	e	ns	s	e	s

SECOND TEST

-----

(c)	PREFER BRAND C	288	134	154	84	130	53
		57.6	53.4	61.8	60.4	58.3	52.5

(d)	PREFER BRAND D	212	117	95	56	93	48
		42.4	46.6	38.2	39.6	41.7	47.5
		s	ns	s	s	s	e

THIRD TEST

-----

(e)	PREFER BRAND E	280	148	132	94	110	52
		56.1	59.0	53.2	66.8	49.3	51.0
(f)	PREFER BRAND F	220	103	116	46	113	50
		43.9	41.0	46.8	33.2	50.7	49.0
		s	s	ns	s	e	e

-----

(sig=.05) (all\_pairs) rows tested a/b, c/d, e/f

Compare this table with Table 801 and notice how the frequency and percentages have changed. The numbers in this table equal the sum of the numbers in Table 801 plus half of the numbers that were in the NO PREFERENCE row. Also notice that "s" appears in the same place, but that cells that were previously blank now contain either an "ns" or "e", depending upon the difference of the two cells. One additional thing to notice is that the footnote for this table is exactly the same as the one from Table 801, so the only way to tell which test was done is by looking to see if there are any of the "ns" or "e" markings on the table.

## 8.9 CHI-SQUARE AND ANOVA TESTS

Statistical significance testing is often desirable as a part of cross-tabulation reporting. Such testing is used to determine whether or not a statistically significant relationship exists between two or more tabulated factors. Tests commonly used for

this are chi-square and analysis of variance (ANOVA). ANOVA tests for significance between means. So, if significance testing is desired on any question for which means are calculated, ANOVA would be the likely choice. The chi-square test is for significance between parts of the column axis (banner) and the entire row axis (stub). It would be chosen for questions where calculation of means is not applicable. For both ANOVA and chi-square testing, row categories must be mutually exclusive, as must column categories within tested parts of the banner.

The ANOVA and chi-square tests discussed here are invoked by EDIT= statements rather than by variable/axis definition expressions and are performed by Mentor at the time of printing rather than at the time of numeric calculation. Run times are shortened by this approach thus improving overall efficiency. There are occasions when having statistics computed at the time of numerical calculation is required. A discussion of invoking statistical testing with variable/axis definition expressions can be found in the *Appendix B: TILDE COMMANDS*.

The following are some relevant keywords for creating tables with ANOVA and chi-square tests.

### EDIT

Used in the ~DEFINE block this controls numerous printing and percentage options. Each table can have its own EDIT statement, so options can be changed as required by varying question types. Some options that pertain to ANOVA and chi-square testing are:

TABLE\_TESTS=<region> is used to specify table regions to be tested. Text labeling for the test is included in the region definition. The EDIT statement must include a separate TABLE\_TESTS= command for each region tested.

-TABLE\_TESTS causes statistical testing not to be performed. This option should be included in an EDIT statement separate from that which includes TABLE\_TESTS=<region> options, and invoked for tables that do not require statistical testing.



COLUMN\_STATISTICS\_VALUES=VALUES(<values>) assigns response weights to the row categories. These weights are used in the calculation of statistics such as mean and standard deviation, and for ANOVA testing.

MEAN, STD, ANOVA, and CHI\_SQUARE cause the corresponding statistical calculations to be performed and printed as part of the table. (STD is the abbreviation for standard deviation.)

-CHI\_SQUARE\_ANOVA\_FORMAT places chi-square and ANOVA statistics in list form after the corresponding table. If this option is not used, these statistics will be printed directly under the table regions tested, provided the regions are wide enough. ANOVA and chi-square statistics for table regions with narrow column widths, such as yes/no questions in the banner, will not print and will cause Mentor to generate error messages.

SHOW\_SIGNIFICANCE\_ONLY causes only the significance to show under the table regions tested. This will not work with -CHI\_SQUARE\_ANOVA\_FORMAT, so it cannot be done in list form.

MARK\_CHI\_SQUARE 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.

The syntax is: EDIT={edit1: MARK\_CHI\_SQUARE=abcde } See *Mentor, Volume II, ~DEFINE EDIT* for more details and examples.

### **LOCAL\_EDIT=<name>**

Used in the ~EXECUTE block, this invokes a previously defined EDIT statement. Options specified in the EDIT statement and also named in a LOCAL\_EDIT command will take precedent over the same options in any other EDIT statements previously invoked. Options that are in a previously invoked EDIT statement, but not in the LOCAL\_EDIT command will stay in affect. If ~SET DROP\_LOCAL\_EDIT is used, a LOCAL\_EDIT command is in effect only for the first table following it.

**STUB**

Text that will label each vertical category on the printed tables is defined using this statement. Control is offered over various options, but one is of particular interest:

`[-COLUMN_STATISTICS_VALUES]` excludes that category from statistical calculations. It is often used to exclude the "Don't know" category.

More detailed descriptions of the capabilities and syntax of these keywords can be found in *Appendix B: TILDE COMMANDS, STUB=*.

The example that follows illustrates how to create tables with ANOVA and chi-square tests. Also demonstrated are the following design characteristics:

- Separate definition of table regions for testing makes the specs more readable and easier to understand should maintenance be required in the future.
- Region definitions (`$R`) are created to be banner (column) specific, but not stub (row) specific by always typing "1 to LAST" for the row part of the definition. This way, it is necessary to type the column parts of the region definitions for a banner only once since the same banner regions are tested each time a specific banner is used. Testing of row categories is then controlled on a question-by-question basis through use of `[-COLUMN_STATISTICS_VALUES]`.
- Separate EDIT statements invoked by `LOCAL_EDIT` commands control which statistical tests are to be performed on each question.
- For banner 1, the `-CHI_SQUARE_ANOVA_FORMAT` option is used to print ANOVA and chi-square statistics in list form following their corresponding tables.

```
~INPUT DATACLN
```

```
~SET AUTOMATIC_TABLES, DROP_LOCAL_EDIT
```



~DEFINE

EDIT={STATS\_OFF: -TABLE\_TESTS }

'Banner 1 definitions

```
'
                                     column      row
''           Stat tests must be labeled   region      region
''           since they will appear in list form  in banner   in stub
''           -----
BAN1_REG1:   [$T="STAT TEST FOR SERVICE TYPE      " $R  2 TO 3  BY 1 TO LAST]
BAN1_REG2:   [$T="STAT TEST FOR NEW SERVICE       " $R  4 TO 5  BY 1 TO LAST]
BAN1_REG3:   [$T="STAT TEST FOR TAX PREPARATION   " $R  6 TO 7  BY 1 TO
LAST]
BAN1_REG4:   [$T="STAT TEST FOR FREQUENCY OF USE " $R  8 TO 10 BY 1 TO
LAST]
BAN1_REG5:   [$T="STAT TEST FOR SALES            " $R 11 TO 14 BY 1 TO LAST]
```

```
EDIT={ BAN1_EDIT: -COLUMN_TNA,PERCENT_DECIMALS=0,
        COLUMN_WIDTH=5,STUB_WIDTH=25,
        -CHI_SQUARE_ANOVA_FORMAT,    ''puts stat tests in
                                     ''list following table
        TABLE_TESTS=BAN1_REG1,
        TABLE_TESTS=BAN1_REG2,
        TABLE_TESTS=BAN1_REG3,
        TABLE_TESTS=BAN1_REG4,
        TABLE_TESTS=BAN1_REG5
      }
```

BANNER={BAN1\_BANNER:

```
'
      REG1      REG2      REG3      REG4      REG5
''   <-----> <-----> <-----> <-----> <----->
|
|           NEW      TAX      FREQUENCY      SALES
|   SERVICE  SERVICE  PREPARA-    OF USE    =====
|   TYPE    =====  TION    =====                    500-
|   =====  SW  NW  =====  MED-    <500  <1  4.9  5+
|TOTAL NEW  OLD AREA AREA  YES  NO  LOW  IUM HIGH MIL. BIL. BIL. BIL.
|-----
```

```
BAN1_COL: TOTAL WITH &
          [199^1/2] WITH &
          ([217#S] AND [15^1,2]) WITH &
          ([199^1] AND (([217#S] AND &
            [15^N1,2]) OR [217#L])) WITH &
          [78^1/2] WITH &
          [147^1,2,6/3/4,5] WITH &
[182.8#1-499999/500000-999999/1000000-4999999/5000000-99999999]
```

''End of banner 1 definitions

''Banner 2 definitions

```
''          column          column
''          region          region
''          in banner      in stub
''          -----      -----
BAN2_REG1:  [$R  2 TO 4   BY 1 TO LAST]
BAN2_REG2:  [$R  5 TO 8   BY 1 TO LAST]
```

'Since -CHI-SQUARE-ANOVA-FORMAT is not used in this edit  
 'statement, statistics tests for BAN2 will appear under their  
 'corresponding table regions.

```
EDIT={BAN2_EDIT:-COLUMN_TNA,PERCENT_DECIMALS=0,
      COLUMN_WIDTH=5,STUB_WIDTH=25,
      TABLE_TESTS=BAN2_REG1,TABLE_TESTS=BAN2_REG2
    }
```

```
BANNER={BAN2_BANNER:
  '          REG1                REG2
  '    <-----> <----->
  |          FREQUENCY          SALES
  |          OF USE          =====
  |          =====          500-
  |          MED-          <500  <1  4.9  5+
  |TOTAL  LOW  IUM HIGH MIL.  BIL.  BIL.  BIL.
  |-----
  }
```

```
BAN2_COL: TOTAL WITH [147^1,2,6/3/4,5] WITH &
[182.8#1-499999/500000-999999/1000000-4999999/5000000-9
999999]
```

'End of banner 2 definitions

'Stub definitions

'Question 4

## STATISTICS (SIGNIFICANCE TESTING)

### 8.9 CHI-SQUARE AND ANOVA TESTS

```
TITLE={Q4_TITLE:
```

```
    Q4. Please indicate on a scale from 1 to 4 how
    satisfied you are with & your overall relationship with
    this company.
```

```
    }
```

```
STUB={Q4_STUB:
```

```
    4 - Very satisfied
```

```
    3 - Satisfied
```

```
    2 - Dissatisfied
```

```
    1 - Very dissatisfied
```

```
[-COLUMN_STATISTICS_VALUES] Don't know/not sure
'excluded from
```

```
''statistics
```

```
    }
```

```
Q4_ROW: [163^4//1/Y]
```

```
EDIT={Q4_EDIT:
```

```
COLUMN_STATISTICS_VALUES=VALUES(4,3,2,1),
```

```
    MEAN, STD, ANOVA
```

```
    }
```



```
'Question 19
```

```
TITLE={Q19_TITLE:
```

```
    Q19. Does this company prepare taxes?
```

```
    }
```

```
STUB={Q19_STUB:
```

```
    Yes
```

```
    No
```

```
[-COLUMN_STATISTICS_VALUES] Don't know/not sure
```

```
'excluded from
```

```
'statistics
```

```
    }
```

```
Q19_ROW: [78^1//3]
```

```
EDIT={Q19_EDIT: CHI_SQUARE }
```

```
EDIT={Q19SIG_EDIT: CHI_SQUARE, SHOW_SIGNIFICANCE_ONLY
```

```
    }
```

```
>CREATE_DB TABLES
```

```
>PRINT_FILE TABLES
```

```
~EXECUTE
```

```
'BAN1's EDIT statement causes statistics tests to be  
printed as
```

```
'lists after the corresponding tables.
```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.9 CHI-SQUARE AND ANOVA TESTS

```
BANNER=ban1_banner,EDIT=ban1_edit,COLUMN=ban1_col
```

```
'Question 4
```

```
LOCAL_EDIT=q4_edit,TITLE=q4_title,STUB=q4_stub,ROW=q4_row
```

```
'Question 19 with statistics test
```

```
LOCAL_EDIT=q19_edit,TITLE=q19_title,STUB=q19_stub,ROW=q19_row
```

```
'BAN2's EDIT statement allows statistics tests to be printed
```

```
'under their corresponding table regions (default).
```

```
BANNER=ban2_banner,EDIT=ban2_edit,COLUMN=ban2_col
```

```
'Question 19 with statistics test
```

```
LOCAL_EDIT=q19_edit,TITLE=q19_title,STUB=Q19_STUB,ROW=Q19_ROW
```

```
'Question 19 with statistics test showing significance only
```

```
LOCAL_EDIT=q19sig_edit,TITLE=q19_title,STUB=q19_stub,ROW=q19_row
```

```
'Question 19 without statistics test
```

```
LOCAL_EDIT=stats_off,TITLE=q19_title,STUB=q19_stub,ROW=q19_row
```

```
RESET,PRINT_ALL
```





~END

Here are the tables that are printed:

TABLE 001

Q4. Please indicate on a scale from 1 to 4 how satisfied you are with your overall relationship with this company.

	SERVICE TYPE		NEW SERVICE AREA		TAX PREPARATION		FREQUENCY OF USE			SALES				
	NEW	OLD	SW	NW	YES	NO	LOW	MED-	HIGH	<500 MIL.	<1 BIL.	4.9 BIL.	5+ BIL.	
	TOTAL	NEW	OLD	AREA	AREA	YES	NO	LOW	IUM	HIGH	MIL.	BIL.	BIL.	BIL.
Total	151	96	55	12	84	64	79	56	30	41	37	18	49	25
	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
N/A	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4 - Very satisfied	31	22	9	-	22	12	19	9	7	9	12	3	7	3
	21%	23%	16%		26%	19%	24%	16%	23%	22%	32%	17%	14%	12%
3 - Satisfied	68	39	29	6	33	34	30	25	16	17	12	12	22	12
	45%	41%	53%	50%	39%	53%	38%	45%	53%	41%	32%	67%	45%	48%
2 - Dissatisfied	38	28	10	5	23	11	25	16	5	11	12	1	14	6
	25%	29%	18%	42%	27%	17%	32%	29%	17%	27%	32%	6%	29%	24%
1 - Very dissatisfied	8	4	4	1	3	5	2	4	1	2	-	1	4	2
	5%	4%	7%	8%	4%	8%	3%	7%	3%	5%		6%	8%	8%
Don't know/not sure	6	3	3	-	3	2	3	2	1	2	1	1	2	2
	4%	3%	5%		4%	3%	4%	4%	3%	5%	3%	6%	4%	8%
Mean	2.8	2.8	2.8	2.4	2.9	2.9	2.9	2.7	3.0	2.8	3.0	3.0	2.7	2.7
Standard Deviation	0.8	0.8	0.8	0.7	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.7	0.8	0.8

```
STAT TEST FOR SERVICE TYPE
anova = 0.02, df1,df2 = (1,143) prob = 0.8694
STAT TEST FOR NEW SERVICE
anova = 3.83, df1,df2 = (1,91) prob = 0.0504
STAT TEST FOR TAX PREPARATION
anova = 0.01, df1,df2 = (1,136) prob = 0.9204
STAT TEST FOR FREQUENCY OF USE
anova = 1.10, df1,df2 = (2,119) prob = 0.3374
STAT TEST FOR SALES
anova = 1.50, df1,df2 = (3,119) prob = 0.2178
```



TABLE 002

Q19. Does this company prepare taxes?

	SERVICE TYPE		NEW SERVICE AREA		TAX PREPARATION		FREQUENCY OF USE			SALES				
	NEW	OLD	SW	NW	YES	NO	LOW	MED-	HIGH	<500 MIL.	<1 BIL.	4.9 BIL.	5+	
	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	
	TOTAL	NEW	OLD	AREA	AREA	YES	NO	LOW	IUM	HIGH	MIL.	BIL.	BIL.	BIL.
Total	151	96	55	12	84	64	79	56	30	41	37	18	49	25
	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
N/A	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Yes	64	23	41	4	19	64	-	25	14	22	15	11	18	14
	42%	24%	75%	33%	23%	100%		45%	47%	54%	41%	61%	37%	56%
No	79	69	10	8	61	-	79	31	15	15	20	5	27	11
	52%	72%	18%	67%	73%		100%	55%	50%	37%	54%	28%	55%	44%
Don't know/not sure	8	4	4	-	4	-	-	-	1	4	2	2	4	-
	5%	4%	7%		5%				3%	10%	5%	11%	8%	

STAT TEST FOR SERVICE TYPE  
 chi\_square = 38.51, d\_f = 1, prob = 0.0000  
 STAT TEST FOR NEW SERVICE  
 chi\_square = 0.13, d\_f = 1, prob E<5  
 STAT TEST FOR TAX PREPARATION  
 chi\_square = 138.98, d\_f = 1, prob = 0.0000  
 STAT TEST FOR FREQUENCY OF USE  
 chi\_square = 2.00, d\_f = 2, prob = 0.3694  
 STAT TEST FOR SALES  
 chi\_square = 4.93, d\_f = 3, prob = 0.1764

TABLE 003

Q19. Does this company prepare taxes?

STATISTICS (SIGNIFICANCE TESTING)

8.9 CHI-SQUARE AND ANOVA TESTS

	FREQUENCY OF USE				SALES =====			
	LOW	IUM	HIGH	MIL.	<500	<1 BIL.	4.9 BIL.	5+ BIL.
Total	151	56	30	41	37	18	49	25
	100%	100%	100%	100%	100%	100%	100%	100%
N/A	-	-	-	-	-	-	-	-
Yes	64	25	14	22	15	11	18	14
	42%	45%	47%	54%	41%	61%	37%	56%
No	79	31	15	15	20	5	27	11
	52%	55%	50%	37%	54%	28%	55%	44%
Don't know/not sure	8	-	1	4	2	2	4	-
	5%		3%	10%	5%	11%	8%	
CHI-SQUARE:	<---	2.00	-->	<---		4.93		-->
D.F.:		2				3		
SIG:		0.3694				0.1764		



TABLE 004

Q19. Does this company prepare taxes?

	FREQUENCY OF USE				SALES =====			
	TOTAL	LOW	MED- IUM	HIGH	<500 MIL.	<1 BIL.	4.9 BIL.	5+ BIL.
	-----	-----	-----	-----	-----	-----	-----	-----
Total	151	56	30	41	37	18	49	25
	100%	100%	100%	100%	100%	100%	100%	100%
N/A	-	-	-	-	-	-	-	-
Yes	64	25	14	22	15	11	18	14
	42%	45%	47%	54%	41%	61%	37%	56%
No	79	31	15	15	20	5	27	11
	52%	55%	50%	37%	54%	28%	55%	44%
Don't know/not sure	8	-	1	4	2	2	4	-
	5%		3%	10%	5%	11%	8%	
CHI-SQUARE (SIG) :	<--	0.3694	-->	<--	0.1764	-->		

TABLE 005

Q19. Does this company prepare taxes?

**STATISTICS (SIGNIFICANCE TESTING)**

*8.9 CHI-SQUARE AND ANOVA TESTS*

	FREQUENCY OF USE				SALES =====			
	LOW	IUM	HIGH	MIL.	<500	<1 BIL.	4.9 BIL.	5+
Total	151	56	30	41	37	18	49	25
	100%	100%	100%	100%	100%	100%	100%	100%
N/A	-	-	-	-	-	-	-	-
Yes	64	25	14	22	15	11	18	14
	42%	45%	47%	54%	41%	61%	37%	56%
No	79	31	15	15	20	5	27	11
	52%	55%	50%	37%	54%	28%	55%	44%
Don't know/not sure	8	-	1	4	2	2	4	-
	5%		3%	10%	5%	11%	8%	

**DISCUSSION OF OUTPUT**

While interpretation of these statistical tests for specific reports is beyond the scope of this manual, some explanation of the results might be helpful.

For table 001, output from the ANOVA was as follows:

STAT TEST FOR SERVICE TYPE

```
anova = 0.02, df1,df2 = (1,143) prob = 0.8694
STAT TEST FOR NEW SERVICE
anova = 3.83, df1,df2 = (1,91) prob = 0.0504
STAT TEST FOR TAX PREPARATION
anova = 0.01, df1,df2 = (1,136) prob = 0.9204
STAT TEST FOR FREQUENCY OF USE
anova = 1.10, df1,df2 = (2,119) prob = 0.3374
STAT TEST FOR SALES
anova = 1.50, df1,df2 = (3,119) prob = 0.2178
```

The number following "ANOVA=" is the value of the F statistic for the region tested. "df1,df2" are the degrees of freedom for the F statistic's numerator and denominator respectively. "prob=" is the probability of there not being a more than coincidental relationship between the factors tested.

To check whether the desired row and banner categories are in fact being used in the ANOVA calculation, the following equations can be used:

$df1 = (\text{the number of banner points included in the ANOVA}) - 1$

$df2 = (\text{the sum of frequencies of all cells included in the ANOVA}) - (\text{the number of banner points included in the ANOVA})$

If a column is blank, it is not considered as being included in the test.

Checking the "STAT TEST FOR SALES" ANOVA above, there are four banner points included in the test, resulting in

$df1 = 4 - 1 = 3$

Adding together all cells included in this test ("DON'T KNOW/NOT SURE" is excluded), "<500 MIL."=36, "<1 BIL."=17, "500-4.9 BIL."=47, and "5+ BIL."=23. The number of banner points is four, resulting in

$$df2 = (36 + 17 + 47 + 23) - 4 = 119.$$

For table 002, the chi-square test produced the following output:

```

STAT TEST FOR SERVICE TYPE
chi_square = 38.51, d_f = 1, prob = 0.0000
STAT TEST FOR NEW SERVICE
chi_square = 0.13, d_f = 1, prob E<5
STAT TEST FOR TAX PREPARATION
chi_square = 138.98, d_f = 1, prob = 0.0000
STAT TEST FOR FREQUENCY OF USE
chi_square = 2.00, d_f = 2, prob = 0.3694
STAT TEST FOR SALES
chi_square = 4.93, d_f = 3, prob = 0.1764

```

Similar to the ANOVA output, the number following "chi\_square=" is the value of the chi-square statistic for the region tested. "d\_f" represents the degrees of freedom for this statistic. The same facts apply to "prob=" as did for ANOVA. "E<5" means that the expected value of the frequency for 25% or more of the cells in the tested region is less than 5, possibly making probability calculations for the region invalid.

Checking that the desired categories are being tested is easier for the chi-square test is easier than for the ANOVA. For degrees of freedom, the following equation applies:

$$d\_f = (\text{number of stub points included in the test} - 1) *$$



(number of banner points included in the test - 1).

If a row or column is blank, it is not considered as being included in the test.

Using "STAT TEST FOR SALES" as an example, "number of stub points included in the test"=2 since "DON'T KNOW/NOT SURE" is excluded, and "number of banner points included in the test"=4. Putting these numbers in the equation:

$$d_f = (2 - 1) * (4 - 1) = 1 * 3 = 3.$$

## OTHER ANOVA AND CHI-SQUARE OPTIONS

It is possible to mix ANOVA and chi-square tests on the same table. This is true only when the -CHI\_SQUARE\_ANOVA\_FORMAT option is used to output the results in list format. Here is an example of an EDIT statement that would do this. The regions have been defined as in the above example.

```
EDIT={Q47_EDIT2 :
      CHI_SQUARE=BAN1_REG1
      CHI_SQUARE=BAN1_REG2
      CHI_SQUARE=BAN1_REG3
      CHI_SQUARE=BAN1_REG4
      ANOVA=BAN1_REG5
}
```

**NOTE:** Alternative way of defining regions and invoking statistical tests is as follows:

```
EDIT={EDIT1 :
      TABLE_TESTS=[$R 1 TO 2 BY 1 TO LAST]
      TABLE_TESTS=[$R 3 TO 4 BY 1 TO LAST]
```

## STATISTICS (SIGNIFICANCE TESTING)

### 8.9 CHI-SQUARE AND ANOVA TESTS

```
TABLE_TESTS= [$R 5 TO 6 BY 1 TO LAST]  
ANOVA  
}
```

If this method is used, Mentor will not allow both ANOVA and chi-square on the same table.

Two other interesting possibilities are testing overlapping ranges and testing columns that are not next to each other. Again, -CHI\_SQUARE\_ANOVA\_FORMAT must be used. Also, doing this might require some creative labeling of the horizontal axis. Example specs follow.

Here are some example specs:

```
BAN3_REG1:  [$T="GENDER                                " $R
2 TO 3  BY 1 TO LAST]

BAN3_REG2:  [$T="DRIVE CAR                                " $R
4 TO 5  BY 1 TO LAST]

BAN3_REG3:  [$T="1-10, 11-49, & 50+ YRS OF DRIVING " $R
6,7,9  BY 1 TO LAST]

BAN3_REG4:  [$T="<50 & 50+ YRS OF DRIVING            " $R
8,9  BY 1 TO LAST]
```

'BAN3\_REG3 defines columns which are not next to each other.

'BAN3\_REG4 defines a region that overlaps BAN3\_REG3.

```
EDIT={BAN3_EDIT: -COLUMN_TNA,PERCENT_DECIMALS=0
        COLUMN_WIDTH=7,STUB_WIDTH=25
        -CHI_SQUARE_ANOVA_FORMAT
        TABLE_TESTS=BAN1_REG1
        TABLE_TESTS=BAN1_REG2
        TABLE_TESTS=BAN1_REG3
        TABLE_TESTS=BAN1_REG4
    }
```

BANNER={BAN3\_BANNER:

	GENDER		DRIVE CAR		YEARS OF DRIVING			
	=====		=====		=====			
TOTAL	MALE	FEMALE	YES	NO	1-10	11-49	<50	50+
-----	-----	-----	-----	-----	-----	-----	-----	-----

}

```
BAN3_COL: TOTAL WITH &
          [5^1/2] WITH &
          [10^1/2] WITH &
          [30.2*P#1-10/11-49/1-49/50-99]
```

```
EDIT={Q49_EDIT: ''Used as a LOCAL_EDIT to invoke tests for a
          ''specific question.
          CHI_SQUARE=BAN3_REG1
          CHI_SQUARE=BAN3_REG2
          CHI_SQUARE=BAN3_REG3
          CHI_SQUARE=BAN3_REG4
        }
```

## 8.10 NOTES ON SIGNIFICANCE TESTING

This section contains some additional notes on significance testing in order to help you understand some of the problems or errors that might occur.

When using other sources to verify significance testing accuracy, be aware that the majority of other programs or textbooks do not take into consideration the consequences of dependent or inclusive tests, the Newman-Keuls procedure, and pooled variances. Mentor uses the additional information available to produce more reliable results in general.

### 8.10.1 What Can and Cannot Be Tested

Significance testing can only be performed on means or percentages produced from simple frequencies. As a general rule, a "simple" frequency counts **respondents**, not **responses**, nor does it include any mathematical calculations. Basically each data case must return either a 1 (it is there) or a 0 (it is not there) for the cells being tested.

The only exceptions to this are either a Mean or a weighted table. You can test cells with different weighting schemes, but only if a given data case has the same weight everywhere in that test.

The following constructions will not produce simple frequencies and therefore cannot be tested for significance:

- \*L modifier
- all summary statistics other than means
- any arithmetic operation
- sigma
- sums
- NUMBER\_OF\_ITEMS or any other number returning function

These rules apply to both ROW and COLUMN definitions. In addition, if you have any kind of summary statistic such as a \$[MEAN] in the COLUMN variable you cannot test any part of the table. If you have any of the above constructions in the ROW other than summary statistics, you will not be able to test any part of the table, unless you turn off the testing around that item using \$[-DO\_STATISTICS] (See Section 8.6 EXCLUDING ROWS/COLUMNS FROM THE SIGNIFICANCE TESTING).

Also the AXIS commands \$[BREAK] and \$[BASE] are acceptable in ROW definitions but not in COLUMN definitions. \$[OVERLAY] and \$[NETOVERLAY] tables can be tested. If you wish to do a test on responses, you must either use a \$[OVERLAY] or a READPROCEDURE to read each data case multiple times.

Tables that are created or modified by table manipulation cannot be tested for significance. Market share tables which use sums cannot be tested. If you are trying to test items based on market share we recommend that you test the mean of the items. Testing a straight market share is very dangerous as a single outlier can easily skew the results.

Any construction can be tested using print phase tests, but remember that Mentor does not check or complain if your testing is illogical. For example, you should never use Mentor to test two numbers that were created as a calculation.

### 8.10.2 Degrees of Freedom

Degrees of freedom (df) is a measure of sample size for use in statistical tests. The higher the degrees of freedom, the more reliable the resulting t values are. The degrees of freedom should be calculated as follows:

Let's assume a sample of size n (either simple count or effective\_n) with n1 in group 1, n2 in group 2, and n\_both in the overlap.

The degrees of freedom for the All Possible Pairs Test procedure, regardless of the variance specified, or for the Newman-Keuls procedure when only testing two groups (all simple t-tests), is calculated as follows:

no overlap, means	$df = n1 + n2 - 2$
no overlap, percents	$df = n1 + n2 - 1$
overlap, means or percents	$df = n1 + n2 - n\_both - 1$

When you use the T= (for inclusive tests) option to take out the overlap, the program utilizes the no overlap formula which results in n - 2 for means and n - 1 for percents. If one group is completely contained in the other it always results in n - 1.

The degrees of freedom for the Newman-Keuls procedure when testing more than two groups would be:

no overlap	$df = n1 + n2 - 2$
overlap	$df = n1 + n2 - n\_both - 1$

### 8.10.3 Verifying Statistical Tests

It may be desirable to verify the results on tables that utilize Mentor's statistical tests. We could use the formulas shown in *Appendix A: STATISTICAL FORMULAS*, but more often a cursory check that the correct columns and rows were tested is all that is needed.

The ~SET keyword STATISTICS\_DUMP does this by sending a printout to the list file showing key elements of the statistical procedure.

For example, if the test performed was an independent test of the means which included a test of column A against column B, then a portion of the list file pertaining to this statistical test will look similar to this:

```
test 1 (216 len, 2 groups, err=0, base_row=0):I=AB row/col=(15,15; 1,2) means
ncases=28
    group 1  12, 76, 564, 0,
    group 2  16, 106, 778, 0,
effn,mean,std:  12 6.33333 2.74138 -- sumsq,sumsqadj,effn: 82.6667 1 12
effn,mean,std:  16 6.625 2.24722 -- sumsq,sumsqadj,effn: 75.75 1 16
tags:  1, 2,
getpoolv: 6.09295,2:26=158.417/26
doqs (6.09295,26) 0-1 tags: 1, 2,
-->0: (-0.437582,26)
SIGFAREA(26,-0.437582->0.05 for 1) returns 0.7572 from -0.309417 -> 0,0.7572
differences[ in AB: ]
```

This information can be used in numerous ways to check the statistical test performed. The first line tells us this is a test of two groups, the base row for statistics is the System Total row (base\_row=0), it is an independent test of columns A and B (:I=AB), it is using row 15 and columns 1 and 2, and it is a test of means. The number of cases in the statistics base is 28 (or 16 + 12). The line with -->0: (-0.437582,26) shows 0:(q-value, df). In the next line down, the "returns 0.7572 from -0.309417 -> 0,0.7572" means "returns <significance> from <t-value>".

## STATISTICS (SIGNIFICANCE TESTING)

### 8.10 NOTES ON SIGNIFICANCE TESTING

A printout like the one shown above would be generated for each pair of columns tested.

If our test was a dependent test of columns A, B, C, D, E, and F for all percent rows and the mean row, then a portion of the list file pertaining to this statistical test would look similar to this:

```
test 1 (1232 len, 6 groups, err=0, base_row=0):ABCDEF
row/col=(3,3; 1,6) percents ncases=166
group 1 12, 4, 4, 0,
group 2 16, 6, 6, 0,
group 3 14, 2, 2, 0,
group 4 27, 13, 13, 0,
group 5 69, 9, 9, 0,
group 6 28, 9, 9, 0,
sxy matrix all zero
effn,mean,std: 12 0.333333 0.492366 --
sumsq,sumsqadj,effn: 2.66667 1 12
effn,mean,std: 16 0.375 0.5 -- sumsq,sumsqadj,effn: 3.75
1 16
effn,mean,std: 14 0.142857 0.363137 --
sumsq,sumsqadj,effn: 1.71429 1 14
effn,mean,std: 27 0.481481 0.509175 --
sumsq,sumsqadj,effn: 6.74074 1 27
effn,mean,std: 69 0.130435 0.339248 --
sumsq,sumsqadj,effn: 7.82609 1 69
effn,mean,std: 28 0.321429 0.475595 --
sumsq,sumsqadj,effn: 6.10714 1 28
tags: 1, 2, 3, 4, 5, 6, multiple comparisons
getpoolv: 0.1931,2:165=43/166
doq (0.1931,165) 0-5 tags: 1, 2, 3, 4, 5, 6,
-->0: (-0.351143,26) (1.55823,24) (-1.37423,37)
(2.08774,79) (0.111041,38)
```



```

-->1: (2.04147,28) (-1.08619,41) (2.83657,83) (0.550136,42)
-->2: (-3.309,39) (0.136389,81) (-1.75572,40)
-->3: (4.97691,94) (1.90971,53)
-->4: (-2.74322,95)
doqs (0.1931,165) 0-5 tags: 1, 2, 3, 4, 5, 6,
-->0: (-0.316228,27) (1.59364,25) (-1.2021,38) (2.48386,80)
(0.102869,39)
-->1: (1.99451,29) (-0.94989,42) (3.25042,84) (0.50417,43)
-->2: (-2.98179,40) (0.175692,82) (-1.73374,41)
-->3: (5.17632,95) (1.69734,54)
-->4: (-3.08478,96)
differences[ in ABCDEF: D vs E; ]
differences[ D vs E; ]

```

The first line tells us this is a test of six groups (ABCDEF), the base row is the System Total row (base\_row=0), it is a dependent test of columns A, B, C, D, E, and F. The test is using row 3 and columns 1 to 6, and it is a test of percents. The line beginning with "doqs (0.1931,165)" shows:

```

(q-value,df) for
-->0: (A vs B) (A vs C) (A vs D) (A vs E) (A vs
F)
-->1: (B vs C) (B vs D) (B vs E) (B vs F)
-->2: (C vs D) (C vs E) (C vs F)
-->3: (D vs E) (D vs F)
-->4: (E vs F)

```

A printout like this would be available for each row tested.

### 8.10.4 Error and Warning Messages

The following is a list of error and warning messages that can occur while doing statistical testing. Each message has a brief description of why it might occur and how to fix it.

```
(ERROR #603) printable_t collision with code C
```

This error is caused when the STATISTICS= PRINTABLE\_T option is used, but the column list is not printable. Column "C" is probably used multiple times. Either do not print t values or check your column list.

```
(ERROR #5055) table T002, test 3 (=GHIJ) is an i= test  
but has cells 4 and 2
```

Test of columns GHIJ was marked as independent, but is dependent. Data is either dirty or you are using the wrong test.

```
(ERROR #5091) Newman_Keuls test not ok with approximate  
significance 0.75
```

This means you cannot use the DO\_STATISTICS APPROXIMATELY option in conjunction with the Newman-Keuls test.

```
(ERROR #5520) tables with COL/ROW_WEIGHT= cannot have  
STATS= without SET MULTIWGT
```

COLUMN\_SHORT\_WEIGHT or ROW\_SHORT\_WEIGHT has been used in conjunction with a STATISTICS statement. Use the SET option MULTIPLE\_WEIGHT\_STATISTICS to override.

```
(ERROR #5524) stat test 3 (=GHIJ) had an error during  
construction
```

Test of columns GHIJ was marked as inclusive, but is not. Data is either dirty or you are using the wrong test.

```
(ERROR #6736) table T011, test 1 (at 2) has conflicting
weight values 1 vs 0.88
```

Cannot have a data case with multiple weights in a single test even if SET MULTIPLE\_WEIGHTS\_STATISTICS is used. Table 011 has a data case with weight of 1.00 and 0.88.

```
(ERROR #6831) col A has stats percent 102/372 different
from vertical percent 102/295
```

There is a cell in column A in which the statistical base does not match the vertical percentage base. In particular the cell has a frequency of 102, the statistical base a value of 372, and the percentage base a value of 295.

```
(WARN #1141) to install this option, we have to clear
out the existing tables first at (18):
```

SET option STATISTICS\_BASE\_AR or -STATISTICS\_BASE\_AR has been used in the middle of a run. No override possible.

Error And Warning Messages (*continued*)

(WARN #5157) we will use all\_pairs tests as the default statistical test

No test was specified on the EDIT statement so the default All Possible Pairs Test will be used. Suppress this warning by specifying ALL\_POSSIBLE\_PAIRS\_TEST on the main EDIT.

(WARN #5385) table T006 with STATS= TAB6\_st but no stat tests to do or report

This warning can appear for multiple reasons. One is that there is a non-simple frequency in the table that was not tested.

(WARN #5621) MULTIPLE\_WEIGHT override for stats testing: be careful!

Set option MULTIPLE\_WEIGHT\_STATS has been used. No override possible.

### 8.10.5 Commands Summary

The following is a list of all statements/keywords/options that affect statistical testing. Information about these keywords can be found in either *Appendix B: TILDE COMMANDS* or in the section reference mentioned on the right.

STATEMENT/KEYWORD/OPTION	SECTION
--------------------------	---------

#### AXIS

\$[BASE]	8.2.2 and 8.4
\$[EFFECTIVE_N]	8.4

\$_[DO_STATISTICS]	8.6.2
\$_[MEDIAN]	7.2.2
\$_[INTERPOLATED_MEDIAN]	7.2.2
[col.wid^code/(STATISTICS)/code/code]	7.2

**EDIT=**

ALL_POSSIBLE_PAIRS_TEST	8.3.1
ANOVA_SCAN	8.3.3
DO_PRINTER_STATISTICS	8.5.1
DO_STATISTICS	8.1.3, 8.1.5, 8.1.6, and 8.1.7
DO_STATISTICS_TEST	8.5.1
FISHER	8.3.3
FLAG_MINIMUM_BASE	8.6.3
MARK_CHI_SQUARE	8.9
MINIMUM_BASE	8.6.3
NEWMAN_KEULS_TEST	8.3.2
PAIRED_VARIANCE	8.3.4
POOLED_VARIANCE	8.3.4
SEPARATE_VARIANCE	8.3.4
USUAL_VARIANCE	8.3.4

**STATISTICS=**

ABCD	8.1.1
I=ABCD	8.1.1
T=ABCD	8.1.1 and 8.1.8
PRINTABLE_T	8.1.1 and 8.7.1
D=1,2	8.8.1

## STATISTICS (SIGNIFICANCE TESTING)

### 8.10 NOTES ON SIGNIFICANCE TESTING

P=1,2	8.8.2
RM=ABCD	8.3.3

STATEMENT/KEYWORD/OPTION	SECTION
--------------------------	---------

#### STUB=

BASE_ROW	8.2.1
DO_SIG_T=	8.7.1
DO_STATISTICS=	8.5.3
DO_T_TEST=	8.7.1

#### SET OPTIONS

MEAN_STATISTICS_ONLY	8.6.1
MULTIPLE_WEIGHT_STATISTICS	8.4.1
STATISTICS_BASE_AR	8.2.1
STATISTICS_DUMP	8.9.2

# SPECIALIZED FUNCTIONS

## INTRODUCTION

**T**his chapter describes generating and printing specialized reports, table manipulation, and other functions. It also covers how to partition data files.

### 9.1 GENERATING SPECIALIZED REPORTS

The normal report that Mentor creates is a summary of responses in a table format. If you would rather see how each respondent answered a specific question, then you can write a specification file that will create a specialized report. You can produce these case-by-case reports by running a procedure on the data and printing information about the selected records, much like the cleaning specs you created in Chapter 2. All you need to do is add a few commands to control the how the report is formatted. You can also add headers, footers and a summary at the end of a report.

You can create specialized reports which are lists of respondents who meet certain criteria, or lists of comments written on surveys. You can also create specialized reports to fill out existing forms your company has. These reports can include data accumulated across cases, calculated values, or text. Examples of each of these are provided below.

The specialized reports you create using Mentor can be simple or complex. In general, you will use the following commands in a spec file to create reports:

```
~SET AUTOMATIC_NEW_LINE
>PRINT_FILE <filename> <page size, other options>
~DEFINE
PROCEDURE={ <proc name>:
```

WHEN TOP/BOTTOM	control printing at the top/bottom of a page
IF/THEN/ELSE/ENDIF	controls program logic, printing,
or GOTO	data modification, or other commands
EXECUTE_EOF	executes commands at the end of a run (used
	to print text at bottom of report)

}

~INPUT &lt;datafile&gt;,&lt;SELECT=specific cases or other options&gt;

~EXECUTE PROCEDURE=&lt;proc name&gt;

~END

**~SET AUTOMATIC\_NEW\_LINE**

Says to go to a new line before printing a new case's information; this is the default. Use ~SET -AUTOMATIC\_NEW\_LINE to stay on the same line across cases.

**>PRINT\_FILE** (a meta command)

Specifies the name of the file to print to, its size and other options. (Refer to your *UTILITIES* manual under *Appendix A: META COMMANDS*).

**~DEFINE PROCEDURE=name: commands }**

Defines what commands to execute across cases.

**~INPUT <datafile>,<options>**

Says which data file to use, and which cases of that file, as well as other options. The SELECT= option may have any valid variable description to subset the records to use.

**~EXECUTE PROCEDURE=<procedure name**

Executes the procedure on the ~INPUT file, writing to the print file specified on >PRINT\_FILE.





**~END**

Ends the program.

Most of the syntax is specified within the procedure defined with ~DEFINE PROCEDURE=name:. Use IF/ELSE/THEN/ENDIF blocks or GOTO commands to control whether to execute the printing commands within the procedure. Any other procedure command can be used in a printing procedure as well, such as data modification commands. Here are the major keywords used for report printing in a procedure:

**PRINT\_LINES # "text \codes text "&&**

**"more text" variables &**

**more variables**

This is the main printing command. Various backslash (\) codes are specified within the text to control printing. PRINT\_LINES formats the text line(s) and fills data text controls with the responses to the variables. Text printing controls are described below.

The number (#) in the command line allows you to specify which print file to print to if more than one is opened. A double ampersand (&&) after the text allows you to continue defining the text on the next line. An ampersand (&) at the end of a line allows you to continue a PRINT\_LINES statement to the next syntax line. (**NOTE:** This command can be abbreviated to PRT)

**SAY "text" variable "text" variable**

This allows you to combine text and the responses to a variable, but has none of the special print controls. Use it for quick but simple listings.

**SKIP\_TO +-#**

This is used to skip forward some number of lines, or skip to a position some number of lines from the bottom of the page.

**WHEN TOP #/END\_WHEN**

## SPECIALIZED FUNCTIONS

### 9.1 GENERATING SPECIALIZED REPORTS

This is generally used for page headings. PRINT\_LINES and other commands appearing between the WHEN TOP and END\_WHEN will be executed at the top of each page.

If both WHEN TOP and WHEN BOTTOM are used together, WHEN BOTTOM must precede other regular print statements.

#### **WHEN BOTTOM #/END\_WHEN**

Similar to WHEN TOP, this executes commands at the bottom of each page. It is commonly used to print a footer.

#### **EXECUTE\_ANY**

Execute the commands following both for each data case and at the end of the file.

#### **EXECUTE\_DATA**

Execute the commands following for each data case, but not at the end of the file. This is the default.

#### **EXECUTE\_EOF**

Execute the following command only at the end of the run; for instance, if you wish to print a summary at the bottom of a report.

Specific syntax and options for these keywords can be found in *Appendix B: TILDE COMMANDS*.

The >PRINT\_FILE controls relevant to reporting are:

#### **>PRINT\_FILE filename <options>**

This command names the file which will contain the finished report. The name will have a PRT extension unless preceded by a dollar sign (\$) or the meta command >-CFMC\_EXTENSION has been invoked, in which case the name you specify will be the actual name.



**Options:**

**-FORM\_FEED**

Says not to include form feeds in the report. Do this if you are unsure about the number of lines per page your printer prints and wish to fill the pages as much as possible.

**PAGE\_LENGTH=#**

Controls how many lines each page of the report will have. If unspecified, there will be 66 lines per page. The first three lines and the last three lines are reserved as default top and bottom margins respectively. Adjusting PAGE\_LENGTH can prevent a new page in the middle of a respondent's information. *Example 2: A Conditional List of Client Information* later in this section illustrates how to do this.

**PAGE\_WIDTH=#**

Controls the page width. The default page width is 132.

**LASER\_CONTROL=<file name**

Says that the printing will be controlled by the printer standards set up in the file \CFMC\CONTROL\UTILITIES manual, refer to *Appendix D: CFMC CONVENTIONS* under *Command Line Keywords*, *LISTFILE* for more information.

The following options are only available on the MPE operating system:

**>PRINT\_FILE LP**

Says to print directly to the line printer, labelled 'LP' in the devices list.

## SPECIALIZED FUNCTIONS

### 9.1 GENERATING SPECIALIZED REPORTS

#### **LASER\_NUMBER=#**

Says which laser printer to print directly to. The devices must be labelled LJET1, LJET2, etc.

#### **COPIES=#**

Says how many copies to print, if printing directly to a printer.

#### **FORMS="message"**

Says to send the message to the printer, stop the printer, and wait until the operator replies to the message before continuing printing.

## **THE PRINT\_LINES COMMAND**

The PRINT\_LINES command is the main command used to generate reports. Here is a short description of the text controls and variable descriptions. For more information refer to *~CLEANER PRINT\_LINES* in *Appendix B: TILDE COMMANDS*. The syntax for PRINT\_LINES is:

```
PRINT_LINES "text and \text controls" variable1 variable2 & more variables
```

There are codes to control how much information is printed on a line, how the data is formatted, and codes to print specific characters. The most common codes are described below.

## **LINE PRINTING CONTROL CODES**

**Syntax:** \#<code>

#### **Options:**

#repeat the operation X number of times (optional)



- G** go to print position on line (must be to right of current position)
- N** skip line(s)
- P** starts a new page
- T** generate actual TAB character(s)
- X** skip print position(s)

**Example:**PRINT\_LINES "\10GThis is my name:\N\10G\S" Name

**This will print:** 'This is my name:  
FRED SMITH '

## CODES USED TO PRINT INFORMATION FROM THE DATA OR A VARIABLE

**Syntax:** \

**Options:** <modifiers>:

- < left-justify (default)
- = center
- > right-justify
- # Indent second and subsequent lines by a # number of spaces.

<width> the number of print positions the item is to print in. The default is the actual width of the item.

<.numdecs> the number of decimals a numeric item is to print with.

<|maxitems> the max number of items to include in a category.

## SPECIALIZED FUNCTIONS

### 9.1 GENERATING SPECIALIZED REPORTS

#### <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.
- \L Prints exactly what is in the variable location.
- \S Prints response text from a data location or category variable; trailing blanks are dropped. Multiple responses are separated by commas.
- \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.]

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

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

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

This will print the title text and response for the variable AGE.

```
PRINT_LINES "Company Name: \20s, No. Employees:  
\>3S" &  
[10.20] [numempl]  
PRINT_LINES "Question One: \V1s \V2s" QN1 QN1
```



```
PRINT_LINES "First 3 Responses: \|3*" QN1
PRINT_LINES "                Text: \|3S" QN1
```

Would print an entry for each case in the following format:

```
Company Name: ABC GADGET CO., No. Employees: 23
Question One: Cards, Number of Credit Cards Used
First 3 Responses: 01, 02, 05
                Text: American Express, Mastercard, Visa
Gold
```

## CODES TO PRINT SPECIFIC CHARACTERS

**Syntax:** \

### Options:

\	prints a backslash
'	prints an apostrophe
[	start ignoring ALL backslash codes except \] (Print the backslashes)
]	stop ignoring ALL backslash codes
^_	followed by <hexdigit><hexdigit> to print special characters from the ASCII character set (see <i>Appendix G: GRAPHICS CHARACTERS</i> in your <i>UTILITIES</i> manual). \letter will print control characters.

**Example:** PRINT\_LINES "This file is of type \'cfmc\', with name" &&  
"\[\cfmc\data\myfile.tr\"]"



## SPECIALIZED FUNCTIONS

### 9.1 GENERATING SPECIALIZED REPORTS

**will print:** 'This file is of type 'CfMC', with name  
\cfmc\data\myfile.tr'





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

**Example:** PRINT\_LINES "Here is a ""quoted"" string."

**Will print:** Here is a "quoted" string.

## VARIABLE REFERENCES

The variables in the variable list can be predefined items from a DB file, data location references, or items defined on the PRINT\_LINES command itself, including mathematical expressions.

### *Predefined Variables*

Predefined variables already have a data type. If they are categorical, the responses to the categories are printed; if they are string or text type, the text is printed. If they are numeric, the number is printed. All you need to do is reference them by name, but you can also use the name in data location type references. If the items are stored in a DB file, it must be opened with the >USE\_DB meta command in order to access them.

### **Example:**

PRINT_LINES "\S" Gender	Prints 'Male' or 'Female'
PRINT_LINES "\S" Income	Prints a number for income
PRINT_LINES "\S" Whyliked	Prints the text recorded under 'Whyliked'
PRINT_LINES "\S" [(Name) 1.10\$]	Prints the first 10 characters of 'Name'
PRINT_LINES "\S" Age * 5	Prints a number that is 'Age' times 5
PRINT_LINES "\S" [Cards\$P]	Prints all the punches in 'Cards'.

***Data Location Variables***

Print what is in the data depending on the data type. You cannot use multiple location data variables, but you may use a single location loop variable to print multiple items from the data.

**Example:**

<code>PRINT_LINES "\S" [10.5]</code>	Prints number in location 10 with length of 5
<code>PRINT_LINES "\S" [10.5\$]</code>	String in location 10 with length of 5
<code>PRINT_LINES "\S" [10.5\$P]</code>	Punches in [10.5] (i.e., "1/23/B/1.5XY/7")
<code>PRINT_LINES "\S" [10.1\$T]</code>	Text pointer to text variable (for long text)
<code>PRINT_LINES "\S" [(8,2)10.2\$]</code>	Print the eight strings starting at 10.2

For categorical variables, you can specify the text for each code:

```
PRINT_LINES "\S" [10.5#"Less than 500":0-500/501-5000/ &&
"> 5000":5001-99999]
```

This would print "Less than 500" if the data contained a 0-500 and "> 5000" if greater than 5000. If the data was 501-5000, the text "501-5000" would print.



```
PRINT_LINES "\S: [10^Yes:1/No:2/"Don't know":3-Y,B]
```

This would print "Yes" if there was a 1 punch in column 10, "No" if a 2 punch, and "Don't know" if anything else including no punch (B). Notice that you do not need quotes around the text unless there are special characters in the text such as spaces or apostrophes.

For calculations, any valid math calculation may be used, but you must follow it with a semi-colon before specifying the next variable to be printed.

```
PRINT_LINES "Factor: \5.2S Age: \S" &  
(Income*Age/NUMBER_OF_ITEMS(Cards)) + [10.5] ; Age
```

This would print the result of the calculation as a 5 wide, 2 decimal number then the age of the person for each case.

There are certain variables, called system variables or constants, that contain information that may be useful in your listings. Especially useful are:

**DATE\_TIME** Specifies the current date and time in the format:

"NOV 02, 1994 10:23 AM"

See ~CLEANER PRINT\_LINES in *Appendix B: TILDE COMMANDS* for examples of how to get just the month, day, etc. from the date/time.

**LINE\_NUMBER**

Keeps track of the current line number. Line numbers are seldom printed on reports, but can be used in conditional statements to print something at a specific place on the page. This is illustrated in example 2.

**PAGE\_NUMBER**

## SPECIALIZED FUNCTIONS

### 9.1 GENERATING SPECIALIZED REPORTS

Similar to `LINE_NUMBER`, this variable automatically keeps track of the current page number.

```
WHEN TOP
PRINT_LINES "Community access report on date: \S, page
\S" &
DATE_TIME PAGE_NUMBER
END_WHEN
```

See `~DEFINE VARIABLE=` in *Appendix B: TILDE COMMANDS* for more information on defining variables to print.

Use the following guidelines for your specs:

- Remember that print commands are executed sequentially from left to right and from top to bottom of the page.
- Remember that Mentor automatically goes to a new line at the end of a case.
- Use separate `PRINT_LINE` commands for headings and data. This makes it easier to make changes to your specs.

#### *Example Reports*

##### EXAMPLE 1: A LISTING OF DATA FOR CLEANING PURPOSES

This example illustrates a simple case-by-case data list. This example also illustrates some uses of the `PRINT_LINES` command with spacing control and different variable types. Refer to *Appendix B: TILDE COMMANDS* for a detailed explanation of `PRINT_LINES` syntax.

#### **Example 1 SPECFILE**

```

>USE_DB Sample ``this file contains the predefined
      variables
``referenced by name in this spec file
>PRINT_FILE Listdata

~DEFINE
  PROCEDURE={List:
    WHEN TOP
      PRINT_LINES "ID
\TNAME\TDAY\TSIBLINGS\TNUMSIBS\TOTHERS"
      PRINT_LINES
"----\T----\T---\T-----\T-----\T-----"
      END_WHEN
      PRINT_LINES "\S\T\S\T\S\T\S\T\>2S\T\S" CASE_ID NAME
[DAY$] [SIBLINGS$P] &
      [NUMSIBS] [OTHERS$P]
    }    ``end of procedure list

~INPUT Dataraw
~EXECUTE PROCEDURE=List
~END

```

### Example 1 OUTPUT

ID	NAME	DAY	SIBLINGS	NUMSIBS	OTHERS
---	----	---	-----	-----	-----
0001	MICKEY	MON	1	12	1
0002	OLIVER	TUE		0	2
0003	ZAZU	WED	3	1	123
0004		THU	2	?	8

## SPECIALIZED FUNCTIONS

### 9.1 GENERATING SPECIALIZED REPORTS

0005	CICI		1	1	45
0006	DELI	FRI		?	45
0007	CARMINA	SAT	1	?	8
0008	NORM	SUN	1	7	B
0009	PEANUT	MON	1	3	28
0010	CHI - CHI	WEE	1	2	6



## EXAMPLE 2: A CONDITIONAL LIST OF CLIENT INFORMATION

The next example is a list of a group of people who meet the condition -- "all tours cancelled." This example illustrates some of the keywords and design characteristics discussed before:

- 1) ",PAGE\_LENGTH=#" is used on the >PRINT\_FILE command to insure that no respondent's information will be divided between two pages. Blank lines must be included in all line counts. A page length of 66 (default) allows up to 60 lines to be printed (since there are three lines in the top and bottom margin).

For the following example, the heading takes four lines and the trailer takes one line. With 60 usable lines per page, this leaves 55 lines available for the data list (60-4-1=55). Each person's data requires eight lines to list. Dividing the number of lines available for the data list, which is 55, by 8 yields 6.875. Since it is undesirable to divide one person's data between two pages, each page can accommodate a maximum of six people's data. So the actual data listing can be as big as 48 lines on a single page (6x8=48). This gives

```
# LINES OF GENERATED TEXT PER PAGE = 4 lines in heading + 48 lines  
in data list + 1 lines in trailer = 53,
```

and

```
PAGE_LENGTH = 3 lines in top margin + 3 lines in bottom margin +  
53 lines of generated text per page = 59.
```

- 2) As in the previous example, specs for the heading were divided to resemble the final report. It was also practical to do this with the block that prints from the data ("SURVEY #", "NAME", etc.). Again, the results are more readable specs which are more likely to produce correct output in fewer runs.
- 3) LINE\_NUMBER is used in an IF-THEN conditional statement to insure that the final trailer will appear at the end of the last page.

## SPECIALIZED FUNCTIONS

### 9.1 GENERATING SPECIALIZED REPORTS

#### Example 2 SPECFILE

```
>USE_DB Tours
~INPUT data,SELECT=Tours(C) ``Only use those whose tours
      were cancelled
>PRINT_FILE REPORT,PAGE_LENGTH=59

~DEFINE ``Variable names were not assigned previously.
      ``They are assigned here to make the specs more
readable      ``and for ease of maintenance.
Survey: [1.3$] ``Use dollar sign ($) to print data as a
string instead
      ``of a number
First: [12.12$]
Last: [24.15$]
Company: [39.30$]
Address: [69.30$]
City: [99.20$]
State: [209.3$]
Zip: [212.7$]
Phone: [126.10$]
Tour_name: [140.50$]

PROCEDURE={print_it:
  WHEN TOP
    PRINT_LINES "                SAMPLE TOURISM BOARD TOUR
STUDY"
    PRINT_LINES "                RESPONDENTS WHO HAD ANY OR ALL
TOURS &&    CANCELLED"
    PRINT_LINES "                APR 1991"
```



```

PRINT_LINES "\N"
END_WHEN
WHEN BOTTOM
PRINT_LINES "\19X----- Page \S -----" PAGE_NUMBER
END_WHEN
PRINT_LINES " SURVEY #: \S" Survey
PRINT_LINES " NAME: \S \S" First Last
PRINT_LINES " COMPANY: \S" Company
PRINT_LINES " ADDRESS: \S" Address
PRINT_LINES " \20S \3S \7S" City State
Zip
PRINT_LINES "TELEPHONE #: (\S) \S-S" [(Phone)1.3]
[(Phone)4.3] &
[(Phone)7.4]
PRINT_LINES " TOUR NAME: \S" Tour_name
PRINT_LINES "\N"'throws an extra line since it's the
end of the case
EXECUTE_EOF''goes to end of last page as determined by
LINE_NUMBER ''and prints footer
IF LINE_NUMBER < 56 THEN ''56=# lines of
generated
''text/page +
SKIP_TO -1 ''# lines in top margin
PRINT_LINES "\19X----- Page \S -----" PAGE_NUMBER
ENDIF
} ''end of procedure Print_it

~EXECUTE PROCEDURE=Print_it
~END

```

**SPECIALIZED FUNCTIONS**

*9.1 GENERATING SPECIALIZED REPORTS*

**Example 2 OUTPUT**

SAMPLE TOURISM BOARD TOUR STUDY  
RESPONDENTS WHO HAD ANY OR ALL TOURS CANCELLED  
APR 1991

SURVEY #: 001

NAME:

COMPANY: JOE SMITH & SONS

ADDRESS: 1111 MAIN #111

DENVER

NY 11111

TELEPHONE #: (888) 888-8888

TOUR NAME: TREASURES OF THE MIDWEST

SURVEY #: 002

NAME:

COMPANY: HARKER TOURS INC

ADDRESS:

TX

TELEPHONE #:

TOUR NAME: MIDWESTERN PANORAMA

SURVEY #: 003

NAME:

COMPANY: MIDAMERICAN TOURS

ADDRESS: BOX 222

FT COLLINS

KS 22222

TELEPHONE #: (111) 111-1111

TOUR NAME: FLAT LANDS PLUS



SURVEY #: 004

NAME:

COMPANY: FLATLAND TOURS INC

ADDRESS: 333 MAIN ST

COLORADO SPRINGS VT 33333

TELEPHONE #: (222) 222-2222

TOUR NAME: FLATLANDS

SURVEY #: 005

NAME:

COMPANY: JED & WALLY FERRIS TOURS

ADDRESS: 444 GRAHAM RD

CENTRAL CITY GA 44444

TELEPHONE #: (333) 333-3333

TOUR NAME: KANSAS WHEAT FIELDS

SURVEY #: 006

NAME:

COMPANY: ELVIS BUS LINES

ADDRESS: PO BOX 5555

AURORA NB 55555

TELEPHONE #: (444) 444-4444

TOUR NAME: MIDWESTWARD HO

**SPECIALIZED FUNCTIONS**

*9.1 GENERATING SPECIALIZED REPORTS*

----- Page 1 -----

SAMPLE TOURISM BOARD TOUR STUDY  
RESPONDENTS WHO HAD ANY OR ALL TOURS CANCELLED  
APR 1991

SURVEY #: 007

NAME:

COMPANY: SHELAC TOURS, INC

ADDRESS: 6666 U.S. HWY 66, #6

                  SIOUX CITY

  OK      66666

TELEPHONE #: (555) 555-5555

TOUR NAME: MIDWEST & THE CORN FIELDS

SURVEY #: 008

NAME: JOHN SMITH

COMPANY: WHEAT CLUB TOURS

ADDRESS: PO BOX 777-7777 DEL NORTE DR

                  BOULDER

  LA      77777

TELEPHONE #: (666) 666-6666

TOUR NAME: NEBRASKA PLAINS

SURVEY #: 009

NAME: JOHN DOE

COMPANY: MIME TOURS, INC

ADDRESS: PO BOX 888-8888 ST ANDREWS DR

                  GREELY

  AK      88888

TELEPHONE #: (777) 843-1211

TOUR NAME: FLATLAND RAIL ADVENTURE



SURVEY #: 010  
NAME: DRAKE WHITE  
COMPANY: JIM MAGEE TOURS  
ADDRESS: PO BOX 999-9999 W FORREST LN  
SILVERTHORN NM 99999  
TELEPHONE #: (888) 843-1211  
TOUR NAME: MIDWEST INDIAN LANDS

### **9.1.1 Printing a Report Footer using WHEN BOTTOM**

In order to print a footer or WHEN BOTTOM block at exactly the bottom of every page of your specialized report run, use the following method:

- 1 Turn off the Mentor's new line default with `~SET -AUTOMATIC_NEW_LINE`.
- 2 Count the exact number of lines that should appear at the bottom of the page. This number should include all blank lines, and should account for any `\N` commands.
- 3 Use the number calculated above on the `WHEN BOTTOM #` statement. To print three lines at the bottom of each page, the `WHEN BOTTOM` statement would be: `WHEN BOTTOM 3`.
- 4 The first line of the `WHEN BOTTOM` block should be `"SKIP_TO -#"`, where `#` is the same number specified on the `WHEN BOTTOM` statement. Once again, if there are three lines in the `WHEN BOTTOM` block, then the first statement of the `WHEN BOTTOM` block should be: `SKIP_TO -3`.
- 5 If you use both `WHEN TOP` and `WHEN BOTTOM` together, the `WHEN BOTTOM` statements must precede other regular print statements.
- 6 The `WHEN BOTTOM` block will not print on the bottom of the last page of the run unless you include instructions for `EXECUTE_EOF`. These instructions should be exactly those found inside of the `WHEN BOTTOM` block.

Here is an example of a three line `WHEN BOTTOM` block, including the `EXECUTE_EOF` instructions:

```

                WHEN BOTTOM 3

SKIP_TO -3
PRINT_LINES "\NLine 1 of WHEN BOTTOM block\n"
PRINT_LINES "Line 2 of WHEN BOTTOM block\n"
PRINT_LINES "Line 3 of WHEN BOTTOM block"

END_WHEN
EXECUTE_EOF
SKIP_TO -3
PRINT_LINES "\NLine 1 of WHEN BOTTOM block\n"
PRINT_LINES "Line 2 of WHEN BOTTOM block\n"
PRINT_LINES "Line 3 of WHEN BOTTOM 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.

Because the WHEN BOTTOM block and the EXECUTE\_EOF instructions are exactly the same, a more efficient way to write the above example is to save the PRINT\_LINES statements as an item in a DB file and then read in the db item (i.e., &&dbitem):

```

>FILE_TO_DB print_bot #
SKIP_TO -3
PRINT_LINES "\nLine 1 of WHEN BOTTOM block\n"
PRINT_LINES "Line 2 of WHEN BOTTOM block\n"
PRINT_LINES "Line 3 of WHEN BOTTOM block"
>END_OF_FILE
WHEN BOTTOM 3
    &&print_bot
END_WHEN
EXECUTE_EOF
&&print_bot

```

Although both examples contain the same number of lines, by using the second method, you would only need to alter the PRINT\_LINES instructions in one location if an alteration or correction was necessary.

## 9.2 TABLE MANIPULATION

Table manipulation is useful if you have numbers you want to combine from one or more tables, if you have information in a data file that you want to display on a table, or if you want to transfer numbers from a table to a data file. Tables can be manipulated in either the ~CLEANER block or a procedure can be written in the ~DEFINE block which is later processed in the ~EXECUTE block. (In the SPL software, this was called TPROG.)

Use the ~CLEANER block if you have all the tables you want to manipulate in a DB file and the names of the tables are readily available. Write a procedure to manipulate tables in a ~DEFINE block if you:

- want to save the procedure in a DB file so you can run it again at a later time
- need to use IF/THEN/ELSE/ENDIF statements to evaluate table cells
- find it easier to maintain the list of tables and cells you want to use in a data file rather than in a >REPEAT.

A table is made up of columns and rows. In addition to columns and rows that are defined, there are system-generated Total and No Answer columns and rows.

The following terms may be used to describe columns and rows on a table:

T	system-generated Total column or Total row
NA	system-generated No Answer column or No Answer row
ALL	all columns or all rows (including system-generated)
LAST	last column or last row on a table

Other examples of describing columns (or rows) are:

1 TO 5	columns 1,2,3,4, and 5
1,...,5	columns 1,2,3,4, and 5
1,3,...,5	columns 1 and 3 and 5





Other examples of describing a region on a table are:

T001(1 TO 3 BY 3 TO 5)the region of table T001 which includes columns 1, 2 and 3 by rows 3, 4 and 5

T002(ALL BY 1,...,5)the region of table T002 which includes all columns (including the system-generated Total and No Answer column) by rows 1 to 5

T003(1 TO LAST BY 4)the region of table T003 which includes all columns first to last (excluding the system-generated Total and No Answer column) by row 4

If you are using anything other than simple column ranges (eg, 1 to last by 1,3,...,last) then a semicolon may be required in place of BY. This is also true if a previously defined name is used in place of a range.

We will concentrate on ~CLEANER command table manipulation in this section and follow with writing a procedure in the next section. In the ~CLEANER block the following table manipulation commands are available for creating a new table or changing an existing table:

### **CREATE\_TABLES**

Creates a new table in memory of specified dimensions and fills the cells of the table with a numeric value or MISSING. Can also create a new table in memory by copying or combining other tables.

### **MODIFY**

Alters or combines tables.

A table with 3 columns and 5 rows would be 5 by 7 when the System columns and rows are added. The following statement would create such a table and fill all the cells with zero.



```
CREATE_TABLES T001 (=5, =7) =0
```

We can create a new table that was exactly the same as an existing table:

```
CREATE_TABLES T002 = T001
```

We can create a table that is the sum of one table added to another:

```
CREATE_TABLES T003 = T001 + T005
```

We can create a new table T004 that has the same number of columns as table T001 and four more rows than table T005 and fill the table with zeros:

```
CREATE_TABLES T004 (= NUMCOLS (T001) , = NUMROWS (T005) +  
4) = 0
```

The following arithmetic operations are available in both the ~CLEANER and ~DEFINE blocks:

= copies table cells

!= copies table and all table elements (banner, stub, etc.)

+= adds table cells

-= subtracts table cells

\*= multiplies table cells

## SPECIALIZED FUNCTIONS

### 9.2 TABLE MANIPULATION

`/=` divides table cells

`%=` percentages table cells

We can change a cell in column 1 and row 1 to the number 5:

```
MODIFY T001 (1 BY 1) = 5
```

We can change all the cells in column 3 to a 7:

```
MODIFY T001 (3 BY ALL) = 7
```

We can add the number 3 to all the cells in row 5:

```
MODIFY T001 (ALL BY 5) += 3
```

The following commands are available in both the `~CLEANER` and `~DEFINE` blocks for loading and unloading tables from memory and storing tables in a DB file.

**LOAD\_TABLES tablename** Loads the table from a DB file into memory

**STORE\_TABLES tablename** Stores the table in a DB file that has write access

**UNLOAD\_TABLES tablename** Unloads the table from memory

**UNLOAD\_TABLES \*** Unloads automatically loaded tables from memory

**UNLOAD\_TABLES !** Unloads every table from memory

In the following example we will define, execute and store a table called T001 in a DB file. If you are not familiar with these steps, see *Chapter 4: "Basic Tables"*. We will then create a new table called T101 that is exactly the same as T001. We will multiply every cell in T101 by 2 and then print our two tables (T001 and T101).

```
>CREATE_DB TEST1
>PRINT_FILE TEST1

~DEFINE
COLDEF: TOTAL WITH [6^1//3]
ROWDEF: [4^1/2]

EDIT=EDIT_BASIC:
      COLUMN_WIDTH=10, STUB_WIDTH=0, -VERTICAL_PERCENT,
      -COLUMN_TNA, -ROW_TNA }

BANNER=BAN1:
      |      TOTAL          A          B          C
      |      -----          --          --          --  }

STUB=STUB1:
      FIRST
      SECOND  }

~SET AUTOMATIC_TABLES

~INPUT DATA

~EXECUTE
COLUMN=COLDEF,   ROW=ROWDEF
RUN_CHAIN
```

## SPECIALIZED FUNCTIONS

### 9.2 TABLE MANIPULATION

~CLEANER

```
CREATE_TABLES T101=T001
```

```
MODIFY T101 *= 2
```

```
STORE_TABLE T101
```

~EXECUTE

```
EDIT=EDIT_BASIC, BANNER=BAN1, STUB=STUB1
```

```
LOAD_TABLE=T001, PRINT_TABLE
```

```
LOAD_TABLE=T101, PRINT_TABLE
```

~END



The resulting two tables from the above setup will look like this:

TABLE 001

BANNER: TOTAL WITH [6^1//3]

STUB: ROWDEF

	TOTAL	A	B	C
	-----	--	--	--
FIRST	6	2	1	3
SECOND	4	2	1	1

TABLE 101

	TOTAL	A	B	C
	-----	--	--	--
FIRST	12	4	2	6
SECOND	8	4	2	2

The following ~CLEANER commands are useful for examining tables that are either in a DB file or in memory:

**PRINT\_TABLES tablename** Prints the table

**SHOW\_TABLES** Lists the tables that are loaded in memory

**SHOW tablename** Shows the table and all the elements (banner, stub, etc.) that are stored with the table.

If we use the DB file that we just created we could examine table T001 in the following ways:

```
>USE_DB TEST1

~CLEANER
LOAD_TABLES T001
SHOW_TABLES
PRINT_TABLES T001
SHOW T001
~END
```

The results (which have been edited for readability) would be:

```
SHOW_TABLES
dump of all 1 tables in in_core chain
#1: test1^T001 (6 by 4)
end of dump
```

```
PRINT_TABLES T001
TABLE test1^T001
BANNER: TOTAL WITH [6^1//3]
STUB: ROWDEF

Total      N/A      TOTAL      6^1      6^2      6^3
```



Total	10	-	10	4	2	4
	100.0		100.0%	100.0%	100.0%	100.0%
N/A	-	-	-	-	-	-
4^1	6	-	6	2	1	3
	60.0%		60.0	50.0%	50.0%	75.0%
4^2	4	-	4	2	1	1
	40.0%		40.0%	50.0%	50.0%	25.0%

SHOW T001

```

title= T001_t (size 2, user_said 0)
..column= COLDEF_c (size 874, user_said 1)
..banner= COLDEF_bn (size 1, user_said 0)
..row= ROWDEF_r (size 410, user_said 1)
..stub= ROWDEF_s (size 2, user_said 0)
table:
    row (-1): 10 0 10 4 2 4
    row (0): 0 0 0 0 0 0
    row (1): 6 0 6 2 1 3
    row (2): 4 0 4 2 1 1

```

You will notice in the previous table that the Total row is shown as 'row (-1)' and the No Answer row is shown as 'row (0)'. The defined rows are numbered (1) and (2). The columns are described in the same manner. This means the first defined column will be referred to as column 1.

## SPECIALIZED FUNCTIONS

### 9.2 TABLE MANIPULATION

We will use the same table T001 that we just created and we will create a new table called T102 that is 6 columns wide (includes T and NA columns) and 4 rows long (includes T and NA rows) and every cell is set to missing. We will modify table T102 so that it will be the same table T001 with rows 1 and 2 switched. We will then print both tables.

```
>USE_DB TEST1, READ_WRITE, DUPLICATE=WARN
>PRINT_FILE TEST2

~CLEANER
CREATE_TABLES T102 (=6, =4) =MISSING
MODIFY T102 (ALL BY 1 TO 2) =T001 (ALL BY 2 TO 1)
STORE_TABLES T102

~EXECUTE
EDIT=EDIT_BASIC, BANNER=BAN1, STUB=STUB1

LOAD_TABLE=T001, PRINT_TABLE
LOAD_TABLE=T102, PRINT_TABLE

~END
```

The results will look like:

```
TABLE 001
BANNER: TOTAL WITH [6^1//3]
STUB: ROWDEF

          TOTAL          A          B          C
-----  --          --          --          --
```



FIRST	6	2	1	3
SECOND	4	2	1	1

TABLE 102

	TOTAL	A	B	C
----	--	--	--	--
FIRST	4	2	1	1
SECOND	6	2	1	3

Functions are also available that operate on table cells. Some examples are:

**ABSOLUTE\_VALUE** Return the absolute value of a cell

**MAKE\_NUMBER** Treat the result of another function as a number

**NUMBERS\_FROM\_TABLE**(#,tablename) Make a region with # numbers by copying numbers from the table that is specified

**TABLE\_FROM\_NUMBERS**(VALUES(##),2,1) Fill a table region with numbers (#)

**SQUARE\_ROOT** Return the square root of a number

**SUM** Sum a range of table cells

**X**            Return zero if the cell is missing

(See “9.3.2 Functions” for more information on these and other functions.)

We can take the absolute value of a number on a table:

```
MODIFY T002 (4 BY 4) = &
ABSOLUTE (NUMBERS_FROM_TABLE (1, T001 (3 BY 3)))
```

We can modify a table cell to equal a square root or an absolute value:

```
MODIFY T001 (1 BY 1) = SQUARE_ROOT (14)
MODIFY T001 (1 BY 1) = ABSOLUTE (-6)
```

We can modify a table region to equal a square root or an absolute value:

```
MODIFY T001 (ALL BY ALL) =
MAKE_NUMBER (SQUARE_ROOT (14))

MODIFY T001 (ALL BY ALL) = MAKE_NUMBER (ABSOLUTE (-6))
```

We can fill a table region with different values. The syntax for this function is:

TABLE\_FROM\_NUMBERS (VALUES (numbers desired starting in upper left-most corner, ending with lower right-most corner) number of columns, number of rows)

```
MODIFY T001 (1 TO 2 BY 1 TO 2) =
TABLE_FROM_NUMBERS (VALUES (2, 3, 4, 5), 2, 2)
```



We can take the square root of a number on a table:

```
MODIFY T002 (2 BY 2) =  
SQUARE_ROOT (NUMBERS_FROM_TABLE (1, T001 (1 BY 1)))
```

or, for a two cell region:

```
MODIFY T002 (2 BY 2 TO 3) =  
SQUARE_ROOT (NUMBERS_FROM_TABLE (2, T001 (3 BY 3 &  
TO 4)))
```

or, for a four cell region:

```
MODIFY T002 (2 TO 3 BY 2 TO 3) = &
    SQUARE_ROOT (NUMBERS_FROM_TABLE (4, T001 (3 TO 4 BY 3
TO 4)))
```

or, to a region from a single cell:

```
MODIFY T002 (2 TO 3 BY 2 TO 3) = &
    MAKE_NUMBER (SQUARE_ROOT (NUMBERS_FROM_TABLE (1, T001 (3
BY 4))))
```

We can sum a range of table cells and place the result into another cell: (the number after the NUMBERS\_FROM\_TABLES function, 6 in this case, is the number of cells in the range to be summed)

```
MODIFY T001 (7 BY 1) = &
    SUM (NUMBERS_FROM_TABLES (6, T001 (1 TO 6 BY 1)))
```

We can make a procedure to return a zero if a table cell is missing:

```
PROCEDURE={ PROC1 :
    IF X (MAKE_NUMBER (NUMBER_FROM_TABLE (1, T001 (4 BY
4)))) = 0
        THEN MODIFY T002 (3 BY 3) = 100
    ENDIF
}
```

Functions are also available to combine or evaluate tables.

**FLIP**      Turns columns into rows and the reverse



**JOIN\_COLUMNS** Joins tables side by side

**JOIN\_ROWS** Joins tables by appending one to the other

**NUMBER\_OF\_COLUMNS** Returns how many columns are in a table

**NUMBER\_OF\_ROWS** Returns how many rows are in a table

**REPLICATED** Duplicates blocks of columns or rows on a table

We could flip an entire table:

```
MODIFY T001=FLIP(T001)
```

or just a region:

```
MODIFY T001(1 TO 3 BY 2 TO 4) = FLIP(T001(1 TO 3 BY 2  
TO 4))
```

We could make a new table by joining two other complete tables column wise:

```
CREATE_TABLES T002=JOIN_COLUMNS(T001, T001)
```

or row wise:

```
CREATE_TABLES T002=JOIN_ROWS(T001, T001)
```

or by joining two other table regions column wise:

```
CREATE_TABLES T002=JOIN_COLUMNS(T001(1 TO 3 BY
ALL) , &
T001(1 TO 3 BY ALL))
```

or row wise:

```
CREATE_TABLES T002=JOIN_ROWS(T001(ALL BY 1 TO 3) , &
T001(ALL BY 1 TO 3))
```

We could fill a table cell with the number of columns in a table:

```
MODIFY T002(1 BY 1) = NUMBER_OF_COLUMNS(T001)
```

or the number of rows:

```
MODIFY T002(1 BY 1) = NUMBER_OF_ROWS(T001)
```

We can replicate regions of one table onto another table. The syntax for this command is:

```
REPLICATE(tablename(region), column multiplier, row
multiplier)
```

For this function the receiving region must be the appropriate size to receive the sending region as specified by the multiplying factors:

```
MODIFY T002(3 TO 4 BY 2 TO 3) = &
REPLICATE(T001(1 BY 3 TO 4) , 2, 1)
```



In a procedure created in the ~DEFINE block, table regions are described by \$R. To create a new table with 3 detail columns and 5 detail rows we would say:

```
MODIFY [$R=T001 T TO 5 BY T TO 7]=0
```

In the following example we will write a procedure to create a new table, copy data from an old table to the new table and store the new table. A new table called NEW1 is created and it is exactly the same as the old table named GB009.

```
~DEFINE
  PROCEDURE={ PROC:
    MODIFY [$R=?=NEW1]=GB009  ''? means get size of new
table
''from old table
    STORE_TABLES NEW1
  }
```

Tables with the same name that are stored in different DB files can be combined and stored in a third DB file using the following statement:

```
MODIFY COMBINE^T010 = WAVE1^T010 + WAVE2^T010
```

Name^ in the above example means get the table from that DB file, which must have been opened with a previous DB command.

The following example uses table manipulation to create a table of means called TAB\_MEANS from three tables, called TAB\_QUALITY, TAB\_CORRECT and TAB\_DELIVERY that have been stored in a DB file called TABS1. We will define our banner and stub for our table of means first. Then our table manipulation step will occur in a ~CLEANER block where we will make row 1 of our table of means equal to the mean row (row 7) of the table TAB\_QUALITY, row 2 of our table of means equal to the mean row (again row 7) of the table TAB\_CORRECT, and row

3 of our table of means equal to the mean row (still row 7) of the table TAB\_DELIVERY. We will then print the table TAB\_MEANS.

```
>USE_DB TABS1, READ_WRITE, DUPLICATE=WARN
>PRINT_FILE PRTMEANS

~DEFINE
BANNER={BAN_AGE:
  |  TOTAL  18-24  25-34  35-44  45-54  55-64  65-99
  |  -----  -----  -----  -----  -----  -----  ----- }
STUB={STUB_MEANS:
  QUALITY
  CORRECT
  DELIVERY }
EDIT={EDIT_MEANS:
  COLUMN_WIDTH=7, STUB_WIDTH=20, -VERTICAL_PERCENT,
  -COLUMN_TNA, -ROW_TNA, FREQUENCY_DECIMALS=2,
  PRINT_ALPHA_TABLE_NAMES }
LINES={TITLE_MEANS: SUMMARY TABLES OF MEAN SCORES }

~CLEANER
CREATE TAB_MEANS (=9, =5) = 0
MODIFY TAB_MEANS (1 TO 7 BY 1) = TAB_QUALITY (1 TO 7 BY 7)
MODIFY TAB_MEANS (1 TO 7 BY 2) = TAB_CORRECT (1 TO 7 BY 7)
MODIFY TAB_MEANS (1 TO 7 BY 3) = TAB_DELIVERY (1 TO 7 BY 7)
STORE TAB_MEANS

~EXECUTE
BANNER=BAN_AGE, EDIT=EDIT_MEANS, STUB=STUB_MEANS
```



```
TITLE=TITLE_MEANS  
LOAD_TABLE=TAB_MEANS, PRINT_TABLE  
  
~END
```

This will produce output which looks like this:

```
TABLE TAB_MEANS
SUMMARY TABLES OF MEAN SCORES
```

	TOTAL	18-24	25-34	35-44	45-54	55-64	65-99
	-----	-----	-----	-----	-----	-----	-----
QUALITY	3.39	3.25	3.26	3.65	3.43	3.56	3.50
CORRECT	3.61	3.42	3.47	3.85	4.00	3.61	3.57
DELIVERY	3.88	3.83	3.80	3.80	4.14	4.00	3.75

This table could then be utilized in a specialized report run to form an ASCII file for input into a spreadsheet or graphics program. In the following example \$TAB is the label we will have on each line, \$ROW is the row in our table of means (TAB\_MEANS) that we will associate with each label, and \$NEW is the report command to give us a new line in the output file which we will call GRAPH1. The first PRINT command prints the \$TAB variable in the first 8 spaces of the line. This first PRINT command is repeated three times, once for each line of our table of means. The second PRINT command says to skip a space then print a number from a cell on our table of means (TAB\_MEANS) into a four column location with two decimal places. This second PRINT command is repeated seven times for each line, once for each cell of each line in our table of means. (For more information on print control, see "9.1 GENERATING SPECIALIZED REPORTS").

```
>USE_DB TABS1
>PRINT_FILE GRAPH1

~CLEANER
>REPEAT $TAB=QUALITY, CORRECT, DELIVERY ; &
```

```

$ROW=1,2,3;&
$NEW=" ", "\N", "\N"
PRINT_LINES "$NEW\8S" "$TAB"
>REPEAT $COL=1, . . . , 7
PRINT_LINES "\1X\4.2S"
NUMBERS_FROM_TABLE(1, TAB_MEANS($COL BY $ROW))
>END_REPEAT
>END_REPEAT

~END

```

The output from this run will be named GRAPH1.PRT and will look like:

```

QUALITY  3.39 3.25 3.26 3.65 3.43 3.56 3.50
CORRECT  3.61 3.42 3.47 3.85 4.00 3.61 3.57
DELIVERY 3.88 3.83 3.80 3.80 4.14 4.00 3.75

```

Normally, tables saved into db files have only frequencies and variable statistics saved, not percents or any edit statistics. If our original tables, TAB\_QUALITY, TAB\_CORRECT and TAB\_DELIVERY, had edit means as opposed to variable means, these mean rows would not normally be stored with the table into the db file TABS1. However, through the use of the ~SET command, SAVE\_TABLE="\_X" ("\_X" is a user defined tablename suffix), we can save the tables as printed (including all percent and edit statistics rows).

```

>CREATE_DB TABS1
>PRINT_FILE TABS1

~DEFINE
AGE: TOTAL WITH
[40.2#18-24/25-34/35-44/45-54/55-64/65-99]
QUALITYP: [98^5//1/0]

```

## SPECIALIZED FUNCTIONS

### 9.2 TABLE MANIPULATION

```
CORRECTP: [99^5//1/0]
DELIVERYP: [100^5//1/0]
BANNER={BAN_AGE:
  | TOTAL  18-24  25-34  35-44  45-54  55-64  65-99
  | -----  -----  -----  -----  -----  -----  ----- }
STUB={STUB_RATINGP:
  5- EXCELLENT
  4- GOOD
  3- FAIR
  2- POOR
  1- VERY POOR
  NO OPINION
[PRINT_ROW=MEAN]  EDIT MEAN
[PRINT_ROW=STD]   EDIT STD      }
EDIT={EDIT_PRTST: COLUMN_WIDTH=7,STUB_WIDTH=20,
      VERTICAL_PERCENT=T, -COLUMN_TNA,
      PRINT_ALPHA_TABLE_NAMES,
      COLUMN_STATISTICS_VALUES=VALUES(5,4,3,2,1),
      COLUMN_MEAN,COLUMN_STD,STATISTICS_DECIMALS=2
      }
LINES={TITLE_QUALITY: RATING QUALITY OF FOOD  }
LINES={TITLE_CORRECT: RATING CORRECT FOOD ITEMS
RECEIVED  }
LINES={TITLE_DELIVERY: RATING PROMPT DELIVERY OF FOOD
}

~INPUT DATACLN

~SET SAVE_TABLE="_X"
```



```

~EXECUTE
  COLUMN=AGE, BANNER=BAN_AGE, EDIT=EDIT_RATING
  STUB=STUB_RATINGP

  TITLE=TITLE_QUALITY, LOCAL_EDIT=EDIT_PRTST, ROW=QUALITYP
  TABLE=TAB_QUALITYP

  TITLE=TITLE_CORRECT, LOCAL_EDIT=EDIT_PRTST, ROW=CORRECTP
  TABLE=TAB_CORRECTP

  TITLE=TITLE_DELIVERY, LOCAL_EDIT=EDIT_PRTST, ROW=DELIVERY
  P
  TABLE=TAB_DELIVRYP

~END
  
```

We can find out what saved printed table row corresponds to the mean row (or a percentage row if desired) by printing the saved printed tables using a small spec file such as:

```

      >USE_DB TABS1
>PRINT_FILE PRTTABS

~DEFINE EDIT={FREQ_EDIT: FREQUENCY_ONLY, FREQUENCY_DECIMALS=2 }

~EXECUTE
  EDIT = FREQ_EDIT
  LOAD=TAB_QUALITYP_X, PRINT_TABLE
  LOAD=TAB_CORRECTP_X, PRINT_TABLE
  LOAD=TAB_DELIVRYP_X, PRINT_TABLE
  
```

~END

Once again, we will define our banner and stub for our table of means first. Then our table manipulation step will occur in a CLEAN block where we will make row 1 of our table of means equal to the mean row (row 17) of the table TAB\_QUALITYP\_X, row 2 of our table of means equal to the mean row (again row 17) of the table TAB\_CORRECTP\_X and row 3 of our table of means equal to the mean row (still row 17) of the table TAB\_DELIVRYP\_X. We will then print the table TAB\_EMEANS.

```

>USE_DB TABS1, READ_WRITE, DUPLICATE=WARN
>PRINT_FILE PRTMEANS

~DEFINE
BANNER={BAN_AGE:
  |  TOTAL  18-24  25-34  35-44  45-54  55-64  65-99
  |  -----  -----  -----  -----  -----  -----  ----- }
STUB={STUB_MEANS:
  QUALITY
  CORRECT
  DELIVERY }
EDIT={EDIT_MEANS: COLUMN_WIDTH=7, STUB_WIDTH=20,
  -VERTICAL_PERCENT, -COLUMN_TNA, -ROW_TNA,
  FREQUENCY_DECIMALS=2, PRINT_ALPHA_TABLE_NAMES }
LINES={TITLE_EMEANS: SUMMARY TABLE OF EDIT MEAN SCORES }

~CLEANER
CREATE TAB_EMEANS (=9, =5) = 0

MODIFY TAB_EMEANS(1 TO 7 BY 1) = &
TAB_QUALITYP_X(1 TO 7 BY 17)

```



```
MODIFY TAB_EMEANS(1 TO 7 BY 2) = &  
TAB_CORRECTP_X(1 TO 7 BY 17)
```

```
MODIFY TAB_EMEANS(1 TO 7 BY 3) = &  
TAB_DELIVRYP_X(1 TO 7 BY 17)
```

```
STORE TAB_EMEANS
```

```
~EXECUTE
```

```
BANNER=BAN_AGE, EDIT=EDIT_MEANS, STUB=STUB_MEANS
```

```
TITLE=TITLE_EMEANS
```

```
LOAD=TAB_EMEANS, PRINT_TABLE
```

```
~END
```

This table could then be utilized in the same formatted report run as described above to form an ASCII file for input into a spreadsheet or graphics program.

You should be familiar with the following ~SET options that are useful in manipulating tables and are described in *Appendix B: TILDE COMMANDS*.

**TABLE\_DROP\_MODE=#**Specifies when tables or regions are unloaded from memory

**TABLE\_DROP\_WARN=#**Specifies how modified tables will react when unloaded from memory

**TABLE\_MISSING MODE=#**Specifies how the program reacts when tables are not found

**TABLE\_MODIFY\_MODE=#**Specifies how the program reacts when tables do not fit together

**TABLE\_STORE\_MODE=#**Specifies when tables are stored in a DB file

In the following example, we will utilize the `RANK_TABLE_COLUMNS` function to add a row to the bottom of a table which shows the rank of the mean for each column in the banner. Since the `RANK_TABLE_COLUMNS` function is designed to rank row cells and show the ranks as a column, we must use the `FLIP` function to rank column cells.

```

1  ~INPUT  DATACLN
2  >CREATE_DB  TRANK
3  >PRINT_FILE TRANK
4
5  ~DEFINE
6  TABLE_SET={BANCNTRS:
7      EDIT=: COLUMN_WIDTH=6, STUB_WIDTH=20, -COLUMN_TNA
8          -ROW_NA, CALL_TABLE="", VERTICAL_PERCENT=T,
9          PERCENT_DECIMALS=0, STATISTICS_DECIMALS=2  }
10 BANNER=:
11     |
12     |          STORE STORE STORE STORE STORE STORE STORE
13     | TOTAL      1      2      3      4      5      6      7
14     | -----
15     }
16     COLUMN=: NET([23.2#00,01,02/03,05/06,38,39,07,08/&
17                10,31,35/09,13/37,19,12,30/36,21,32,34])
18     }
19
20 TABLE_SET={Q4A:
21 LOCAL_EDIT=: EXTRA_STUBS_OK }
22 TITLE=: Q4. HOW CONSISTENT IS THE CONCEPT:\N
23         A. GOOD PLACE TO SHOP  }
24 STUB=:
25     7-VERY CONSISTENT
26     6
27     5
28     4-NEITHER CONSISTENT NOR INCONSISTENT
29     3

```

## SPECIALIZED FUNCTIONS

### 9.2 TABLE MANIPULATION

```
28      2
29      1-VERY INCONSISTENT
30      [STATISTICS] MEAN
31      [FREQUENCY_ONLY] RANK MEANS }
32      ROW=: [35^7//1] $ [MEAN] [35] }
33
34 ~EXECUTE
35 TABLE_SET=BANCNTRS
36 TABLE_SET=Q4A TAB=TQ4A
37
38 ~CLEANER
39 CREATE TEMP (+1, *) =FLIP (TQ4A)
40 MODIFY TEMP (LAST BY 2 TO 8) = &
41 RANK_TABLE_COLUMNS (HIGH, LOW_TIES, TEMP (LAST-1 BY 2
TO 8))
42 CREATE RANKQ4A=FLIP (TEMP)
43 STORE RANKQ4A
44
45 ~EXECUTE
46 TABLE_SET=BANCNTRS, STUB=Q4A_S, TITLE=Q4A_T
47 LOAD=RANKQ4A, PRINT_TABLE
48
49 ~END
```

To add the row of ranks, we create a temporary table in line 39 of the above spec file which is one row bigger in size than our existing table (TEMP(+1,\*)) and into this temporary table we place the existing table, FLIPped. Then in line 40 of the above spec file we say to modify the last row (the RANK MEANS row) to reflect the rank of the cells in the next to last row (the MEAN row) but only for columns 2 to eight (we don't want to rank the TOTAL column). Then the temporary file is FLIPped back to its proper orientation for final printing. When the ranking is accomplished in line 41, we say to rank high to low (HIGH) and show any ties in



ranking with the lower rank number for all ties (LOW\_TIES). (See “9.3.2 Functions”, *Table Related Functions* for a more complete discussion of the RANK\_TABLE\_COLUMNS function.)

Here is the final table:

Q4. HOW CONSISTENT IS THE CONCEPT:

A. GOOD PLACE TO SHOP

	STORE	STORE	STORE	STORE	STORE	STORE	STORE	STORE
	TOTAL	1	2	3	4	5	6	7
	-----	-----	-----	-----	-----	-----	-----	-----
Total	87	14	14	14	7	15	12	11
	100%	100%	100%	100%	100%	100%	100%	100%
7-VERY CONSISTENT	4	-	-	2	-	2	-	-
	5%			14%		13%		
6	17	5	3	2	1	2	3	1
	20%	36%	21%	14%	14%	13%	25%	9%
5	15	2	-	3	2	5	1	2
	17%	14%		21%	29%	33%	8%	18%
4-NEITHER CONSISTENT NOR INCONSISTENT	26	5	7	3	2	4	2	3
	30%	36%	50%	21%	29%	27%	17%	27%

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

3	14	-	2	1	1	2	4	4
	16%		14%	7%	14%	13%	33%	36%
2	5	-	1	2	-	-	1	1
	6%		7%	14%			8%	9%
1-VERY INCONSISTENT	6	2	1	1	1	-	1	-
	7%	14%	7%	7%	14%		8%	
MEAN	4.22	4.43	3.93	4.36	4.00	4.87	3.83	3.82
RANK MEANS	-	2	5	3	4	1	6	7

## 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

Mentor has some special keywords to give you more power to describe and manipulate your data. These are divided into System constants and functions.

System constants allow you to get information on current System values or use System-defined items. Functions allow you to modify the meaning of variables. Both have special uses which you will need at times when cleaning data, running procedures, or building tables.

Functions and System constants can be used anywhere standard variables are used in Mentor. Remember that most of the names can be abbreviated. See *Appendix X: ALLOWED ABBREVIATIONS* for the allowed abbreviations.

**NOTE:** Cross-case operations (also called functions) are special features used for row and column creation, and are not related to the functions described here. For more information, see “5.2 Axis Commands/Cross-Case Operations” and *Appendix B: TILDE COMMANDS, ~DEFINE VARIABLE=*.

### 9.3.1 System Constants

System constants can be accessed at any time, but cannot be modified. They contain information such as the current date and time, information about the data case being worked on, and other current system values.

The System constants can be classified as follows:

- Variable constants refer to constants that are used with or in creation of variables or vectors when running procedures or building tables.
- Case reading constants hold information about the data case being read.
- System information constants contain other general information.

You can specify the name of certain system constants inside parentheses causing the data to come from the value of the system variable instead of the case.

**Example:**

```
[(DATE_TIME) $] will say the date/time
[(DATE_TIME) 1.6 $] will be the day and month only
[(DATE_TIME) 13.2 # 00//23] will evaluate the hour
[(DATE_TIME) 16.2] will be the minutes as a number
[(DATE_TIME) 13.2 #
"Morning":6-11/"Afternoon":12-16/"Eve":0-5,17-23]
[(LINE_NUMBER) # 1-30] > 39
```

You can subset the following System constants:

- CASE\_ID
- CASE\_NUMBER
- DATE\_TIME

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

- JULIAN\_DATE
- LINE\_NUMBER
- PAGE\_NUMBER
- TABLE\_NAME
- TEXT\_AREA\_STATUS

The following constants can be referenced for a specific data file (when you have multiple data files open) by using a caret (^) after the data file name:

ALTER\_FLAG  
CASE\_ID  
CASE\_NUMBER  
CASE\_WRITTEN  
DELETE\_FLAG  
EOF\_DATA  
ERROR\_FLAG  
FIRST\_CASE  
TEXT\_AREA\_STATUS

For instance you could compare the value of CASE\_ID in two different data files.

```
data1^CASE_ID EQ data2^CASE_ID
```

The constants LINE\_NUMBER and PAGE\_NUMBER can be used (in an IF block) to control printing when you have multiple print files open. See the meta command >PRINT\_FILE in your *Utilities* manual for information on using multiple print files.

Here are the descriptions of the system constants. The examples given highlight cases where the System constant would be particularly useful. As with other keywords throughout the manual, many of these can be abbreviated. Those that



can be shortened will show the allowed abbreviation in the syntax or example for that keyword or enclosed in parentheses at the end of the description.

## VARIABLE CONSTANTS

### CATEGORIES(#,#)

Specifies which categories are turned ON using numbers; the numbers are the category numbers to be treated as being ON.

**Syntax:** CATS ( #, #, #-#, #, . . . , # )

You can use ranges or ellipses to describe a CATEGORIES list.

CATS ( 1, 3, . . . , 23 ) Says every other category from 1 to 23 is ON.

CATS ( 1-5, 9 ) Says that categories 1 through 5 and 9 are ON.

This is used for particular data modifications. For instance, to add a category to a multi-category variable without affecting the other categories:

```
TRANSFER [10^1/3/5/7/9] += CATS (3)
```

This would add a 5 punch to column 10 since it is the third category.

CATEGORIES is used with the RANDOM\_CATEGORY function to return a random category to a variable:

```
TRANSFER [10^1/3/5/7/9] = RANDOM_CATEGORY (CATS (1-5))
```

This would randomly assign one of the 5 categories (1, 3, 5, 7, or 9).

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

Another special use of category assignment is when reading a file under the control of a `~MAKE_READ_CONTROL` variable, which speeds up processing when trying to read only particular cases:

```
~INPUT MYFILE
~MAKE_READ_CONTROL, STORE = [5.3#1//100]
~INPUT
MYFILE, READ_CONTROL=STORE(1, 5-10, 81, 90, 92, . . . , 98)
```

The items in parentheses are the categories from the store variable to be used when determining which cases to read in the file.

#### DUD

A category variable with one category, that one being FALSE. Can be used to describe an empty cell in a table. Especially useful to have an empty cell in `$(OVERLAY)` or `$(BREAK)` tables where the number of categories must be the same for each section, but you want to combine different numbers of categories in the different sections.

```
ROW=Items_ate: [10^1//5] $(BREAK) [11^1//4] WITH DUD
```

The DUD category would line up with the 5 of column 10 and put a blank cell in the table for that category.

#### ERRORS

Returns the number of errors (i.e., (ERROR # text)) for a particular run. This constant can be used to control execution in a specification file.

```
~GO_TO (done) ERRORS >0
```

#### FALSE

A boolean which has the value of FALSE. Same as DUD.

**MISSING**

A numeric category which has no value.

**TOTAL**

A category variable, with one category being TRUE. This is useful when you want a Total category in a table, such as a Total column.

```
COLUMN=Banner: TOTAL WITH Sex WITH Age
```

**TRUE**

A boolean which has the value of TRUE. Same as TOTAL.

**VALUES(values)**

Returns a vector of a set of numbers, where the numbers are the value for a given category. If you have no number, just a comma, then that category does not have a value. `VALUES ( , , 5.56 , 1.2 , , -3 )` has 6 categories with values 5.56 in category 3, 1.2 in category 4, and -3 in category 6.

In table building, this is used with the `SELECT_VALUE` function to describe the values for weights or when using assigned values for mean or percentile calculations.

```
~DEFINE
Weight1:
SELECT ( [10^1//5] , VALS (1.2 , .85 , 1.275 , 0.654 , .999)
~EXECUTE WEIGHT=Weight1
COLUMN=... , ROW=... , TABLE=...
...
```

This would assign 1.2 as a weight for those with a 1 in col 10, .85 for those with a 2, and so on.

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

On the EDIT statement, VALUES is used to describe the weights used for the rows or columns when calculating print time statistics.

```
~DEFINE EDIT=Domean:
```

```
COLUMN_MEAN, COLUMN_STATISTICS_VALUES=VALS(, , 10, 20, . . . , 90) }
```

This says to do a mean on the table skipping the first two rows (,) then using the values 10, 20, etc. through 90 for the rows 2 through 10.

## CASE READING CONSTANTS

### ALTER\_FLAG

This is TRUE or FALSE depending on whether the case has ever been changed. When the case is written to a new file, this is set back to FALSE. You can see the current value of the ALTER\_FLAG for the case you are on in the ~CLEANER block with the >STATUS command.

### CASE\_ID

A string whose value is the case ID of the current case being read. This is usually used in procedures when trying to find a particular case, or just to print the case ID.

```
IF [5^1] SAY "CASE" CASEID "IS A MALE" ENDIF
```

This is not the data in the columns that are said to be the case ID, but the value the system has for the case ID. You can reload a new case ID with the PUT\_ID command.

The CASE\_ID value is displayed whenever you move to a new case in the ~CLEANER block. It is also used by the ~CLEANER NEXT command when looking for a particular case.

### CASE\_NUMBER

This is the relative position of the data case in the file, not to be confused with CASE\_ID, which is the assigned identifier for the case.

The case number is used by the ~CLEANER NEXT command using the syntax:

```
NEXT ###,
```

For example, NEXT 256. This would find the 256th case in the file.

### CASE\_WRITTEN

This returns TRUE if the case has been written to an output file during the current procedure. It is most useful to gather all cases that haven't been written to a prior file into a separate file, i.e., write out the exceptions.

```
IF [5^1] THEN
WRITE_CASE #1
ENDIF
IF [5^2] THEN
WRITE_CASE #2
ENDIF
IF CASE_WRITTEN
ELSE
WRITE_CASE #3
ENDIF
```

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

#### **CHECK\_ERROR**

This is a boolean that returns either TRUE or FALSE for the last ~CLEANER CHECK statement executed. Use this constant in an IF or GOTO block to control the execution of a data cleaning procedure.

#### **DELETE\_FLAG**

This says whether the case has been marked as deleted. Note that to read previously deleted cases, the file must be opened with the USE\_DELETED option on, i.e., ~INPUT oldfile,USE\_DELETED. DELETE\_FLAG is turned on when a case is written after the ~CLEANER ASSIGN\_DELETE\_FLAG or DROP commands have been issued. It can be turned off with the ~CLEANER UNDELETE command.

```
IF DELETE_FLAG
PRINT_LINES "CASE \S WAS DELETED\N" CASE_ID
ENDIF
```

When working in ~CLEANER interactively, the >STATUS command will also tell you whether the delete flag is set for the case you are looking at.

#### **EOF\_DATA**

Says whether you are at the end of the data file. Useful in procedures when you wish to do something after you are done processing the data, such as print summary information.

```
IF EOF_DATA
PRINT_LINES "Total exceptions: \S\N" Exctot
           "Total errors: \S\N" Errtot
ENDIF
```



## **ERROR\_FLAG**

Tells whether the case has been marked as having an error. The error flag is turned on by the ~CLEANER commands CLEAN, CHECK, EDIT, and ERROR.

The error flag is removed after modifications are made to the case. The ~CLEANER FIND\_FLAGGED command finds the next case with the error flag turned on.

The >STATUS command will tell you if the error flag is turned on for the case, or use the System constant ERROR\_FLAG in an IF block.

```

    IF ERROR_FLAG
command(s)
.
.
.
ENDIF

```

## **FIRST\_CASE**

Is TRUE whenever you are on the first case of the data file. This is useful in procedures when you want to do something special at the beginning of the run.

```

IF FIRST_CASE
TRANSFER COUNTER [2/1 . 5] =1
ENDIF

```

This sets a counter to 1 at the beginning of a procedure.

## **LAST\_CASE**

Is true when Mentor reaches the last case of the data file. This is useful in procedures when you want to do something at the end of a run.

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

#### TEXT\_AREA\_STATUS

Checks the status of the text area and returns one of following numbers:

1	text area empty
2	text area is okay
3	some front pointers do not point to text because they are blank
4	text area not blank after last text answer
5	some front pointers do not point to text, and are not blank
6	some other problem with the text area, bad back pointers
10	not the input file specified
11	no data case in hand
12	no text location specified

TEXT\_AREA\_STATUS evaluates to 1-6 for the values 1-6 described above, to -1 for 10 or 11. It is a fatal error if value 12 is returned.

Use <studyname>^TEXT\_AREA\_STATUS or <studyname>!TEXT\_AREA\_STATUS to control which input file to check, when multiple files are open. You may also use this system variable inside brackets [ ], e.g., [(TEXT\_AREA\_STATUS) # -1/1/2//6]

Related commands are the ~ADJUST options INPUT\_TEXT\_LOCATION and OUTPUT\_TEXT\_LOCATION.

#### SYSTEM INFORMATION CONSTANTS

##### DATE\_TIME

Returns the current system date/time in the form:



DD MMM YYYY HH:MM (day month year hours minutes)

12 OCT 1994 15:23

This is useful to print the date on reports made with procedures.

**Example:** ~CLEANER PRINT "The current date and time are: \S" DATETIME

### **DATE\_TIME\_DIFF**

Here is an example of the syntax:

**Example:** datetimediff(string,string,datepart)

Where the strings are YYYYMMMDHMMSS and, where the datepart can be:

1 for seconds

2 for minutes

3 for hours

4 for days

5 for months

6 for years

7 for weeks

Examples:

datetimediff("20050504030201","20040504030201",4)

datetimediff([11.14\$],[31.14\$],4)

datetimediff(str1,str2,4)

### **OFFSETDATE**

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

Here is an example syntax:

**Example:** `offsetdate(string,increment,datepart)`

Where the strings are YYYYMMDDHHMMSS and where increment is how much to offset the datepart

Thee datepart can be:

1 for seconds

2 for minutes

3 for hours

4 for days

5 for months

6 for years

7 for weeks

**Example:** say `offsetdate("20010101010101",1,1)`

### **JULIAN\_DATE**

Returns the current system date and time in the form:

YYMMDDHHMMSSHHWJJJ

(year/month/day/hour/minutes/seconds/hundredths/day of week/julian date)

941112155145606316

Day of the week begins with Monday as day 1.

**LINE\_NUMBER**

Returns the current line number you are positioned on in the opened print file (see the meta command >PRINT\_FILE). This can be used to print something on the same line of every page. See “9.1 GENERATING SPECIALIZED REPORTS” for an example of this constant.

```

        IF LINE_NUMBER = 60 THEN
PRINT_LINES "Values for America Report, End of page \S"
&
PAGE_NUMBER
        ENDIF

```

**MATH\_VALUES**

A 17 category vector defined by the constant (seven of which have values as indicated):

```
VALUES(1,0,-1,,1.41,,,,,3.14,,2.72,,,,,1.62)
```

These are special values often used in mathematical calculations, i.e., the 11th value is Pi.

**PAGE\_NUMBER**

Returns the page number of the currently opened print file (see the >PRINT\_FILE command). This is usually used to print the page number on the page during printing procedures. See “9.1 GENERATING SPECIALIZED REPORTS” for an example of this constant.

```

        WHEN TOP
            PRINT_LINES "Values for America Report - page
\S \2N" PAGE_NUMBER
        END_WHEN

```

**RANDOM\_VALUE**

This returns a 14 digit random number between 0 and 1. It can then be used to make decisions about random samples or assigning random categories.

```

IF RANDOM_VALUE > .5
    PRINT_LINES "This should get about half the
cases\n"
    WRITE_CASE
ENDIF

```

The RANDOM\_VALUE system variable can also be used to assign a number in a range of 0 to some high value, then making decisions based on the number returned. This example writes out a 10% random sample to a new data file.

```

~CLEANER
TRANSFER Pick[2/35.3] = RANDOM_VALUE * 100
IF Pick <= 10 THEN
    WRITE_CASE
ENDIF

```

**TABLE\_NAME**

The name of the table currently loaded.

**9.3.2 Functions**

Functions are used to get special values or translate one type of element to another. They can be divided into several groups. You can use these functions interchangeably wherever functions can be used. The function types are:

- Arithmetic functions for mathematical computation
- Vector functions for table axis creation

- Number returning functions returning special numbers
- Logical functions
- Table related functions
- Integer functions
- String functions

The general syntax for all functions is:

**Syntax:** function name( argument1, argument2, ..., argumentn )

The function name must be immediately before the opening parenthesis. Items inside the parentheses can have spaces separating them from the parentheses as well as each other. A comma is required between arguments, and a closing parenthesis must follow the arguments. Most functions have only one argument.

Here are the functions within each group. The examples include likely uses for that function. Note that items within functions may be defined earlier, then referenced by name. Also, a vector is a complex variable description; you may also use simple data variables or numbers wherever a vector is mentioned. Allowed abbreviation of function names are shown in the syntax or example, or indicated in parentheses at the end of the description.

## ARITHMETIC FUNCTIONS

### ABSOLUTE\_VALUE( vector )

Returns the absolute (positive) values of the numbers in the vector. See “9.2 TABLE MANIPULATION” for an example of this function.

```
TRANSFER [10.2] = ABSOLUTE_VALUE (Age-20000)
```

### AVERAGE( vector1, vector2, vectorn, region )

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

Returns the average of all of the numbers present. The average is the sum of the values divided by the number of values present. **AVERAGE** can also operate on a table region (see **SUM** for an example).

```
$ [MEAN] AVERAGE ( [10, . . . , 15] )
```

On a table, this returns the mean of the average of columns 10 to 15.

#### **EXPONENT(vector)**

Returns the exponents of the numbers in the vector.

```
TRANSFER [20.5, 25, 30*F3] =EXPONENT(1 WITH 2 WITH 3)
```

This returns the exponents of 1, 2, and 3 to the specified columns with three decimal places of significance.

#### **LOGARITHM(vector)**

Returns the natural logs ( $e^{\text{sub } n}$ ) of the numbers in the vector.

```
TRANSFER [20.5, 25, 30*F3] =LOGARITHM(1 WITH 2 WITH 3)
```

This returns the logarithm of 1, 2, and 3 to the specified columns with three decimal places of significance.

#### **MAKE\_NUMBER(function)**

Treats the result of another function as a number.

```
MODIFY T001 (ALL BY ALL) =  
MAKE_NUMBER (SQUARE_ROOT (25) )
```

See “9.2 *TABLE MANIPULATION*” for an example of this function.

**SQUARE\_ROOT( vector )**

Returns the square roots of the numbers in the vector.

```
TRANSFER [35.5*F4] = SQUARE_ROOT (Age)
```

This returns the square root of the variable AGE. See “9.2 *TABLE MANIPULATION*” for an example of this function.

**STANDARD\_DEVIATION( vector1, vector2, vectorn )**

Performs a standard deviation on a list of numeric fields

```
STD ( [1.2] , [3.2] , [5.2] )
```

**SUM( vector1, vector2, vectorn, region )**

Returns the sum of all of the numbers and categories present.

```
Tab1: SUM ( [2/1, ..., 2/5] ) WITH [2/1, ..., 2/5]
```

This returns the sum of the five fields followed by each of the fields. SUM can also operate on a table region.

```
SAY SUM(tab1(T to last by 1) )
```

**X( Numeric variable or Math equation )**

Returns a 0 if the numeric variable or equation is blank or is not a valid number. This is usually used to make sure that a good value gets used even if part of the equation is missing. By default, if an item is missing in an equation, the equation returns MISSING.

```
TRANSFER [45.2] = X ([50.2]) + 5
```

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

This returns the sum of the data in location 50-51 plus 5. If columns 50-51 are not a valid number, a 0 is used, and 5 is the result. See “9.2 TABLE MANIPULATION” for an example of this function.

## VECTOR FUNCTIONS

### **BALANCE( vector )**

Returns the vector, followed by the No Answer category for the vector. This is the same as `CATEGORY_FUNCTION(-2,vector)`, only easier to say.

```
Row23: BALANCE ( [10.2#1/20] )
```

This returns 21 categories, the twenty numeric categories plus one category containing anyone not included in the other categories.

`BALANCE` must be used with a category variable and not a boolean variable.

### **Example:**

```
a: BALANCE ( ( [1^1] BY [1^2] ) )
```

```
b: BALANCE ( MAKE_CATEGORIES ( [1^1/2] AND [1^2] ) )
```

```
c: BALANCE ( MAKE_CATEGORIES ( [1^1] AND [1^2] ) )
```

```
d: BALANCE ( CATS ( 1, 2 ) )
```

### **CATEGORY\_FUNCTION( -#, vector )**

Returns the vector with different combinations of Total, No Answer, and Net of the vector added 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; i.e., if you want the Total and No Answer before and Net after the vector, use:

$$(-32) + (-16) + (-1) = -49$$

Row19: CATEGORY\_FUNCTION(-49, [10^1] WITH [11^1//5])

This returns the Total and No Answer, followed by the six categories of the vector, followed by the Net of the vector.

### **NET( vector )**

Returns a vector which is the Net of the vector followed by the original vector. This is the same as CATEGORY\_FUNCTION(-8,vector), but easier to say.

NET(Age WITH Income)

This returns the Net of AGE WITH INCOME followed by AGE, followed by INCOME.

VECTOR FUNCTIONS (*continued*)**NUMBERS\_FROM\_TABLE( maximum # of cells to get, table description )**

Makes a vector of # numbers and assigns the numbers from a table region there.

A standard table region description is:

TABLENAME(<first col> TO <lastcol> BY <first row> TO <last row>)

```
TRANSFER [10.2, 12, . . . , 24] = &
        NUMBERS_FROM_TABLE(8, T001(1 TO 4 BY 1 TO 2))
```

This moves the first 4 columns and 2 rows of table T001 into the data in 2 column wide fields from 10 through 24. The first cell will be in 10-11, the second column, first row will be in 12-13, and so on. See “9.2 TABLE MANIPULATION” for an example of this function.

**RANDOM\_CATEGORY( category vector )**

Randomly picks one of a set of categories that are TRUE.

This is used to pick one of a set of categories that have been previously chosen. A typical example is picking one of two codes that have been chosen to rate an item; i.e., the respondent picked 3 and 4 on a 5 point rating scale, and you wish to just pick one of the codes and use it for analysis.

To pick a random category among categories chosen, and put it back in the same location:

```
TRANSFER [12^1//10] = RANDOM_CATEGORY([12^1//10])
```

RANDOM\_CATEGORY can also be used to just pick a number out of a set of numbers. Do this by using numeric categories.



To get a random integer returned between 1 and 100:

```
TRANSFER [10.3#1//100] =
RANDOM_CATEGORY(CATS(1, . . . , 100))
```

See the RANDOM System constant to get a random number between 0 and 1. This could be multiplied by any number to get a random number in the range from 0 to that number.

## NUMBER RETURNING FUNCTIONS

### **FIRST\_SUBSCRIPT( category vector )**

Returns the subscript (or number) of the first category seen in the vector. For example, if the third and fifth values of the category vector are present, FIRST\_SUBSCRIPT returns a 3.

### **FIRST\_VALUE( vector )**

Returns the first numeric value present in the vector. Note that the vector can include categorical data, in which case a 1 will be returned if the category is the first thing present.

This would be used if you allowed responses in different locations, but only wanted to tabulate the first location answered.

```
Tab1: FIRST_VALUE([10, . . . , 20])
```

This would return the first number seen in columns 10 through 20.

**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.

**LAST\_SUBSCRIPT( categorical vector )**

Like FIRST\_SUBSCRIPT, except it returns the subscript (or number) of the last category seen in the vector.

**LAST\_VALUE( vector )**

Like FIRST\_VALUE, except it returns the last value present in the vector.

**MAXIMUM\_VALUE( vector1, vector2, vectorn, region )**

Returns the highest number present in the vectors. MAXIMUM\_VALUE can also operate on a table region (see SUM for an example).

**MAXIMUM\_VALUE\_SUBSCRIPT( vector1, vector2, vectorn )**

Returns the subscript of the item with the highest value in the vector. If the second item is the highest, it returns a 2.

This is useful when checking against a set of values, then using the highest value in later calculations.

```

IF MAXIMUM_VALUE_SUBSCRIPT( [10, ..., 15] ) = 1 THEN
    TRANSFER [20/5] = [1] * AGE
ELSE
    IF MAXIMUM_VALUE_SUBSCRIPT( [10, ..., 15] ) = 2
THEN
        TRANSFER [20/5] = [2] * AGE
        ...
ENDIF

```

```
ENDIF
```

This finds the high value to use for the ~CLEANER TRANSFER calculation.

**MINIMUM\_VALUE( vector1, vector2, vectorn, region )**

Returns the smallest number present in the list. MINIMUM\_VALUE can also operate on a table region (see SUM for an example).

**MINIMUM\_VALUE\_SUBSCRIPT( vector1, vector2, vectorn )**

Returns the subscript of the item with the lowest number seen in the vector. If the third value is 12, and the fifth is 34, it returns a 3.

**NUMBER\_OF\_ITEMS( vector )**

Returns the total number of categories present. If there is a data location in the vector ([col.wid]), this returns the number of binary punches in the columns, in addition to the other categories in the vector. The NUMBER\_OF\_ITEMS function can be used to count ASCII responses as well as punches.

NUMITEMS creates the equivalent of the Total Responses in a category set, which is often useful as a percentage base in tables. A zero is returned if there are no responses, never MISSING.

```
Q12open_end: NUMBER_OF_ITEMS ( [10.3] ) WITH [10.3^1//36]
```

This produces the sum of the punches in columns 10 to 12, then each of the punches as a separate category.

```
Q13open: NUMBER_OF_ITEMS ( [13.2^1//5/18//24] ) WITH  
[13.2^1//5/18//24]
```

This produces the sum of the categories described in columns 13 to 14, then the separate categories.

To get a table of the number of responses to a set of questions, you could do the following:

In a procedure . . .

```
TRANSFER [5/78.2] =
NUMBER_OF_ITEMS ( [2/10.3, 2/13, 2/16^1/27] )
```

In a table definition ...

```
Q23Responses: [5/78.2#0//12/13-99] $ [MEAN, STD]
[5/78.2]
```

This would produce a table of the number of responses to question in columns 2/10 to 2/18, along with the mean number and standard deviation of responses.

### **RANDOM\_SEQUENCE(vector variable,(seed))**

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.

Its most common use is maintaining a user-controlled random chain with a specification such as ~CLEANER MODIFY

```
var1[1/11.10,1/21.10]=RANDOM_SEQUENCE(var2), where the user can supply
the first seed, or if var2(2)=MISSING, then the program will generate a random
start or the user can supply one with the meta command >RANDOM_SEED=.
```

### **SELECT\_VALUE( vector, <vector or VALUES( #,...,#> )**

Returns a number which is the number in the second vector or values list corresponding to the category seen in the first vector. Can have only one active category, or an ERROR is produced. You must have the same number of categories or values on both sides of the comma.

SELECT\_VALUE is usually used to assign weights for data or statistical calculations (such as mean, standard deviation, percentile, etc.).

```
~DEFINE Weight1:

SELECT_VALUE ( [10^1//5] , VALUES (1.2 , .85 , 1.275 , 0.654 , .999)
)

~EXECUTE WEIGHT=Weight1
      COLUMN=... , ROW=... , TABLE=...
```

This would assign 1.2 as a weight for those with a 1 in col 10, .85 for those with a 2, etc.

### **STRING\_LENGTH( <"string", \$, \$T, or \$P string variable> )**

Returns the number of characters in the string variable, starting at the first position and going to the last non-blank character.

This is often used when listing open-end responses:

```
IF STRING_LENGTH(a[2/23.20$]) > 5
      SAY "Question had response" A
ENDIF
```

This checks to see the length of the response in columns 2/23 to 2/42, then prints the response if the length is greater than 5.

### **SUBSCRIPT( vector )**

Returns the subscript (position of the category) of the category found. Used to assign ordered values. If more than one value is found, returns MISSING.

SUBSCRIPT is often used to assign values when you want continuous numeric categories; for instance, changing a 0 punch to a 10 for purposes of statistical calculation.

```
$(MEAN) SUBSCRIPT([2/13^1//0])
```

If column 2/13 was a 0, this would return a 10. If there was more than one category present, the value would be MISSING, and would not be used in the MEAN calculation.

See also FIRST\_SUBSCRIPT and LAST\_SUBSCRIPT.

### **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.

### **VARIABLE\_EXISTS( string variable )**

Evaluates the string variable as a variable name and checks for its existence in any open data base (DB) files. Returns a number indicating the type of DB entry if the variable named is found in an open DB file, otherwise it returns MISSING. The number returned is the same number found for that variable type in the >LIST\_DB\_CONTENTS meta command.

The string variable can be a name in quotes (i.e., "thisvar"), or a string in the data (i.e., [20.5\$]).

```
IF VARIABLE_EXISTS("myvar") TRANSFER Myvar = 10 ENDIF
```

This would put the number 10 in the variable MYVAR if the variable is in an open DB file.

### **WORD\_MATCHES( string variable, string variable )**



Returns the number of ASCII character matches in the string variable to the string variable. This is a word search; the string must not be contained within another string. This is a non-case-sensitive search.

The comparison is word oriented, thus `WORD_MATCHES("b", "abc")` will return zero (no matches), but `WORD_MATCHES("b", "a b c")` will return one, as will `WORD_MATCHES("B", "a_b_c")`.

The string variable can be a specific string in quotes ("string"), or an ASCII data string ([2/23.10\$]), a punch string ([2/23.10\$P]), or a text string (probably from Survent) ([4/10.1\$T]).

This is useful when recoding open-ended responses to codes. You can search for meaningful keywords to help put the text into categories.

```

IF WORD_MATCHES (food [20.20$] , "candy" ) >= 1 THEN
    SAY "CASE " CASE_ID "HAS " food "Maybe code
as '1'?"
ENDIF

```

This looks for all occurrences of the word "candy" in data locations 20 through 39, and, if one or more, lists the response to possibly be recoded as a 1 in the response list. To find all words starting with a string, see the `WORD_STARTS` function.

### **WORD\_STARTS( string variable 1, string variable 2 )**

Returns the number of times the second string starts a word in the first string. This is a non-case-sensitive search.

The string variable can be a specific string in quotes ("string"), or an ASCII data string ([2/23.10\$]), a punch string ([2/23.10\$P]), or a text string (probably from Survent) ([4/10.1\$T]).

```
WORD_STARTS ( [20.1$T] , "HE" )
```

This returns the number of times "HE" starts a word in the text variable with a pointer in column 20. If it contained "hello there Henry", STARTS would return the value 2.

## **LOGICAL FUNCTIONS**

### **CASCADE( vector )**

Returns TRUE if all categories are true starting at the first category until no more are true. This is useful when checking a set of items where you are supposed to mark the top ones in a larger list, without forcing the user to mark all the items.

```
IF CASCADE( [2/1.2,2/3, ...,2/19*F#1//10] )
    ELSE
        ERROR "Ranking should be continuous from 1 on"
ENDIF
```

If there was just a 1 ranking, this would be OK. If there were no ranking at all, it would be OK. If there was a 1, 2, 4, and 5 ranking, this would not be OK. If there were a 1, 2, and 3 ranking, it would be OK.

### **COMPLETE( vector )**

This returns TRUE if all categories in the vector are true. This is useful when checking a set of items that are supposed to be ranked, say from 1 to 10, with no skips.

```
IF COMPLETE( [2/1.2,2/3, ...,2/19*F#1//10] )
    ELSE
        ERROR "Ranking should be continuous from 1 to
10"
ENDIF
```

If there was just a 1 ranking, this would not be OK. If there were no ranking at all, it would not be OK. If there was a 1, 2, 4, and 5 ranking, this would not be OK. It is only OK if it were ranked 1 through 10.

COMPLETE is also useful to make sure that all items in a set are present.

### **MAKE\_BOOLEAN( vector )**

Returns TRUE if any category in the vector is true. This is useful to get a net category from a complex vector.

```
Mytable: MAKE_BOOLEAN( [5^1] WITH [8.2#1//10] ) WITH &
                [5^1] WITH [8.2#1//10]
```

This returns a 12 category vector; the first category is anyone having a 1 punch in column 5 or a number 1 to 10 in columns 8-9, then each of the separate categories is laid out.

MAKE\_BOOLEAN is particularly useful when you have previously defined an item, and now just want a net of the answers in the item.

```
table23: MAKE_BOOLEAN(likeit) with likeit
```

This would provide the net of LIKEIT followed by LIKEIT.

MAKE\_BOOLEAN is also useful when you need to collapse any vector to a single category for other purposes.

**NOTE:** IF statements always collapse all categories into one category like MAKE\_BOOLEAN.

**TABLE RELATED FUNCTIONS****FLIP( table region )**

This flips the columns and rows in a region of a table.

```
CREATE Tab001=FLIP(t001)
```

This would create the new table TAB001 which is a flipped version of the original table T001.

**JOIN\_COLUMNS( table region 1, table region 2 )**

This extends a table by columns. It is used to put two tables or table regions side-by-side on a page; i.e., first wave vs. second wave numbers. The tables must have the same number of rows to be joined.

```
CREATE t301 = JOIN_COLUMNS(t001,t201)
```

This would make a new table T301 which would combine tables T001 and T201 such that the columns of the tables would be side-by-side. Table T001's columns would be first, followed by table T201's columns.

**JOIN\_ROWS( table region 1, table region 2 )**

This extends tables by rows. It is used to put two tables or table regions one above the other on a page; i.e., to combine two product lists. The tables must have the same number of columns to be joined.

```
CREATE t301 = JOIN_ROWS(t001,t201)
```

This would make a new table T301 which would combine tables T001 and T201 such that the rows of the tables would be combined. Table T001's rows would be first, followed by table T201's rows.

**LOADED( table region )**

This returns TRUE or FALSE depending on whether the table region is currently loaded in core.

**NUMBER\_OF\_COLUMNS( table region )**

This counts the number of columns in a region. It is used to calculate the number of columns to use in future tables.

For instance, if you want to combine two tables you can say:

```
CREATE t003 (=NUMBER_OF_COLUMNS (t001) +
NUMBER_OF_COLUMNS (t002) ,=5 ) = 0
```

You can create tables with twice as many columns, etc.

**NUMBER\_OF\_ROWS( table region )**

This counts the number of rows in a region. It is used to calculate the number of rows to use in future tables. For instance, if you want to combine two tables you can say:

```
CREATE t003 (=10 ,=NUMBER_OF_ROWS (t001) +
NUMBER_OF_ROWS (t002)) = 0
```

You can create tables with twice as many rows, etc.

**RANK\_TABLE\_COLUMNS(HIGH/LOW,LOW\_TIES/MEDIAN\_TIES,<region>)**

Reads a region of a table and returns the rank value of each item in a column.

**Options:**

HIGH        rank high to low.

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

LOW rank low to high.

LOW\_TIES return the low rank value where rank value is the same (tied) for all ties.

MEDIAN\_TIE return the midpoint where rank value is the same (tied) for all ties.

region> defines the region of the table to rank

If columns and rows are included, it ranks down the columns, one column at a time.

You cannot rank across rows; if you want to do this, set things up in columns, then flip the table. See section "9.2 TABLE MANIPULATION" for an example table.

#### Example:

```
MODIFY T001=RANK_TABLE_COLUMNS(HIGH, MEDIAN_TIES,
T000)
```

```
MODIFY T007(5 TO 6 BY 6 TO
9)=RANK_TABLE_COLUMNS(HIGH, MEDIAN_TIES, &
T000(5 TO 6 BY 1 TO 4))
```

#### REPLICATE( table region, # of col reps, # of row reps )

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).

```
TRANSFER T001(1 TO 10 BY 1) += REPLICATE(t002(1 BY
1), 10, 1)
```

This would add the cell from column 1, row 1 of table T002 to the corresponding cells of table T001 (columns 1 through 10, row 1).

See the ~CLEANER FILL command; it also does replicates. If you are using only numbers, then the table automatically fills in a replicated manner.

```
TRANSFER T001 (1 by all) = 5
```

This would put the number 5 in all rows of column 1 of table T001.

#### **TABLE\_FROM\_NUMBERS( vector, # columns, # rows )**

This takes data from the vector and fills a region of a table.

```
TRANSFER T001 (1 TO 5 BY 1) =
TABLE_FROM_NUMBERS ( [5.2, 7, . . . , 13] , 5, 1)
```

This takes data from columns 5-6, 7-8, etc. and moves it into columns 1 through 5, row 1 of table T001. See “9.2 TABLE MANIPULATION” for an example of this function.

## **INTEGER FUNCTIONS**

#### **FILE\_COMPARE("file1", "file2", n, filter options, # lines until resynch)**

Compares two files, and produces an error for every instance where they are not identical. File1 is the master file and file2 is the compare file. You can stop the comparison after n number of errors. This function returns the number of errors found between the two files or one of the following.

- 1 If one of the files can't be opened (for example, if one of the files does not exist)
- 2 If the resulting line would be too long. For example, if you are comparing two files with a PAGE\_WIDTH=132 and writing the FILE\_COMPARE results to a printfile, the printfile must be 18 columns wider or PAGE\_WIDTH=150. If it's not at least that wide, FILE\_COMPARE will return a -2 and a program WARNING message.
- 3 If one of the file names is a bad file name

**Options:**

**n** Stop after n errors. 0 means don't print any errors, and return a 1 if there are any errors. -1 means print all the errors that occur and return the number of errors. Otherwise, the function prints up to the number of errors you specify and returns the number of errors.

**Filter Options:** Will filter out items as follows:

0	no filter
1	filter out blank lines
2	filter out text marked between /* and */
4	filter out temporary file names (i.e. TE001001)
8	trim trailing blanks off lines before compare

You can combine filters by adding them together:

3	1 and 2
5	1 and 4
9	1 and 8
6	2 and 4
10	2 and 8
7	1, 2, and 4
15	all filters (1, 2, 4 and 8)

Number (#) of lines until files are re-synchronized controls how many lines are read, when the files are out of synchronization, before attempting another comparison. The default is 5, used if a number less than 0 is specified. If you set this to "0" then no attempt is made to re-synchronize. You can set this to any other positive number to specify the number of lines to be read before a re-synchronization is attempted. If you only want to confirm that two files are exactly the same, set this parameter to 0. If you are comparing two print files in which one



file could have whole pages that the other did not, use a value of at least 66 (the number of lines on a page).

Line 1 is read from the master file and compared to line 1 in the compare file. If there is a match then line 2 in both files is read. If the two lines do not match, FILE\_COMPARE reads up to the "# lines until resynch" (default is 5) in the compare file looking for a match, FILE\_COMPARE also reads the next"# lines until resynch" from the master file.

FILE\_COMPARE sees if any of these lines in the compare file match any of these lines in the master file. If any match then the file that needs to skip forward the shortest number of lines is resynched to the other file and the lines skipped are listed. If no match is found in these lines then the unmatched line from the master file is printed. Another line is then read from the master file and the compare process begins again, without advancing in the compare file.

```
>PRINT_FILE keywords
~DEFINE
  DIFF: FILE_COMPARE("sk.old", "sk.new", -1, 7, 5)
~CLEANER
  PRINT_LINES "\S" "Checking new msgfile for
keyword changes"
  PRINT_LINES "\S" diff
~END
```

Here is a sample of the print file output. The total number of errors found would be listed at the end of the file. The master file is listed first, followed by the line number and text of the line that differs from the compare file. In this example lines 1 and 2 in the master file differ from the same line in the compare file. 24 is the total errors found comparing these two files.

```
Checking new msgfile for keyword changes
```

## SPECIALIZED FUNCTIONS

### 9.3 USING Mentor's SYSTEM CONSTANTS AND SPECIAL FUNCTIONS

```
File: 1 ( 1) postrelease Mentor
13Jul93(showkey,sk.new) ... Watcom (C) CfMC 1978 -
1993
File: 1 ( 2) System versions: 714 ... 0 ... 0 ...
9305
File: 2 ( 1) postrelease Mentor
21Jul93(showkey,sk.new) ... Watcom (C) CfMC 1978 - 1993
File: 2 ( 2) System versions: 720 ... 0 ... 0 ...
9305
.
.
.
24
```

## STRING FUNCTIONS

### **FIND\_STRING( string variable1, string variable2)**

Reports the number times string variable1 appears in string variable2. This is a string search, text may be inside other words. The search is not case sensitive.

**Example:** FIND\_STRING("IF", "ENDIF")

Will return a 1. You can use variables in your search.

**Example:** INPUT \$  
~CLEAN  
MODIFY str1 [1.10\$]="123456"  
PRINT "Number of matches: \s"  
FIND\_STRING("5", str1)  
~END

This example would return: `Number of matches: 1`

For a search that match entire strings only, use the `WORD_MATCHES` function.

### **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. For example, you can use the system constant `CASE_NUMBER` for the numeric argument, for example:

**Example:** `PUTID STRING_FROM_NUM(CASE_NUMBER, -4, 0)`

The example below assigns sequential case IDs to a data file that has no ID field.

### **Example:**

```
~DEFINE
PROC={mkid:
    IF FIRST_CASE then
        CREATE tmp(=1, =1)=0
    ENDIF
    MODIFY [$r=tmp T by T] += 1

    PUTID STRING_FROM_NUM([$r=tmp T by T], 4, 0)
    ERROR "Table cell" CASEID

WRITECASE
}

~INPUT data.asc ascii=80:11.4
```

```
~OUTPUT string
~EXC PROC=mkid
~END
```

**STRIP(string variable)**

Strips leading and/or trailing blanks from a string variable or data location containing string information.

```
~DEFINE str1[$S=" abc de "]
MODIFY [10-16$]=STRIP(str1)
```

This would strip the blanks off the front and back of the variable str1, and return "abc de" to columns 10 - 16.

**SUBSTITUTE(string variable, "original string", "replacement string" )**

Allows you to substitute one string of characters for another in a string variable or data location containing string information.

This function is useful when making changes to "forms" being printed.

```
info[$s="Beaver Cleaver, 4 Primrose Ln, Upper
Kirkwood MO"]
SUBSTITUTE(info, ", ", "\n")
```

This would change each ", " in the string variable INFO to a new line character, resulting in:

```
Beaver Cleaver
      4 Primrose Ln
      Upper Kirkwood MO
```

This function is case-sensitive, thus in the fourth example below, "apples" is not a match with "Apples". No substitution is done. The SUBSTITUTE function can be a part of a variable definition.

```

~DEFINE
  str_1: [$("#Apples")]
  str_2: [$("#Oranges")]
  str_orig: [$("#Apples for sale!")]
  sub_def: SUBSTITUTE("Apples and lemons say the
bells of &
  St. Clemons.", "Apples", "Oranges")

~CLEANER
  SAY SUBSTITUTE(str_orig, str_1, str_2)
  SAY SUBSTITUTE(str_orig, "Apples", "Oranges")
  SAY SUBSTITUTE("Apples for sale!", "Apples",
"Oranges")
  SAY SUBSTITUTE("Apples for sale! Get your apples
here!" &
  , "Apples", "Oranges")
  SAY sub_def

~END

```

### UPSHIFT() and DOWNSHIFT ()

Changes the case of a string.

#### Example:

```

~clean
m test [1.12$]="ABcdEFghIJk1"

```

```
print "Upshifted= \s" upshift(test)
print "Downshifted= \s" downshift(test)
```

**would print the lines:**

```
Upshifted= ABCDEFGHIJKL
Downshifted= abcdefghijkl
```

The upshift and downshift functions can also be used in conjunction with \$U, \$D and \$N variable types. For example, the DOWNSHIFT() function can return a lowercase version of an \$U variable (see *Chapter 3 Changing Case*).

## 9.4 PARTITIONING DATA FILES

If multiple table runs are to be performed on a single data file, with each run based on a subset of the whole, we can partition or index the data utilizing a user-supplied criteria. For instance, we may want to run multiple sets of tables using a particular region or store as a base for each run. We could accomplish this using a SELECT option on the ~INPUT statement or using a FILTER for a given run. However, Mentor gives us another, faster way to accomplish this goal.

### MAKE\_READ\_CONTROL

This command is used to define the variable that controls the reading of the data file in future runs. The variable must have unique categories so that no case falls into more than one category. CAT or NUM type variables may be used. Categories must be specified in order of sort. Table suffixes will differentiate the sets of tables and will be based on the controlling categories.

```
MAKE_READ_CONTROL =MARITAL[359#M/S]
```

In this example, the table suffixes will be "\_M" and "\_S".

### READ\_CONTROL

Used as an option to the ~INPUT command, this command allows the run to read only categories previously specified by the MAKE\_READ\_CONTROL command, thereby speeding up data processing. Categories must be specified in ascending order.

Specific syntax for these keywords can be found in *Appendix B: TILDE COMMANDS*.

### ***Example Reports***

1) An open-ended opinion table run by a combination of responses to a prior scale question.

This example illustrates a simple use of MAKE\_READ\_CONTROL and READ\_CONTROL to get two sets of tables, each based on different responses to a scale question; one run on those very or somewhat satisfied and one run on those very or somewhat dissatisfied.

```
>CREATE_DB  HARDWARE
~DEFINE
  LINES= {HEADSAT: =SATISFACTION LEVEL: VERY/SOMEWHAT
SATISFIED\N
          }
  LINES= {HEADDISSAT: =SATISFACTION LEVEL: VERY/SOMEWHAT
&
          DISSATISFIED\N
          }

TABLE_SET={BAN1 :
  EDIT=:
          COLUMN_WIDTH=8
          STUB_WIDTH=22
```

## SPECIALIZED FUNCTIONS

### 9.4 PARTITIONING DATA FILES

```
PERCENT_DECIMALS=1
-COLUMN_TNA
-PERCENT_SIGN
STATISTICS_DECIMALS=2
RUNNING_LINES=1
}
```





```
BANNER={ :
    |
    |                                     AGE
    |                                     =====
    |                                     UNDER                               45 OR
    |   TOTAL           25    25-34    35-44    OLDER
    |   -----  -----  -----  -----  }
COLUMN=: TOTAL WITH &
        [24^1,2//5,6] 'AGE
    }
```

```
TABLE_SET={Q3 :
    TITLE=: Q3.  HOW SATISFIED ARE YOU WITH HARRY'S
    HARDWARE?\N
    }
```

```
STUB=:
    VERY SATISFIED (1)
    SOMEWHAT SATISFIED (2)
    SOMEWHAT DISSATISFIED (3)
    VERY DISSATISFIED (4)
    [STATROW] MEAN
    [STATROW] STD    }
ROW=: [6^1//4] $ [MEAN,STD] [6]    }
```

```
TABLE_SET={Q3A:
    TITLE=: Q3A.  WHY ARE YOU SATISFIED OR DISSATISFIED
    WITH &
    HARRY'S HARDWARE?\N    }
LOCAL_EDIT=: RANK_LEVEL=1,MINIMUM_FREQUENCY=1    }
STUB=:
```

## SPECIALIZED FUNCTIONS

### 9.4 PARTITIONING DATA FILES

```
[COMMENT, UNDER_LINE] POSITIVE RESPONSES
[R=1, STUB_INDENT=2] GOOD QUALITY TOOLS/HARDWARE
[R=1, STUB_INDENT=2] PAYMENT CHOICES GOOD
[R=1, STUB_INDENT=2] RESPONSIVE/COOPERATIVE
[R=1, STUB_INDENT=2] HARRY'S IS GOOD HARDWARE
[R=1, STUB_INDENT=2] GOOD PROJECT MANAGEMENT ADVICE
[R=1, STUB_INDENT=2] EDUCATION/INFORMATION
[R=0, STUB_INDENT=2] OTHER POSITIVE
[COMMENT, UNDER_LINE] NEGATIVE RESPONSES
[R=1, STUB_INDENT=2] POOR QUALITY TOOLS/HARDWARE
[R=1, STUB_INDENT=2] INSUFFICIENT PAYMENT OPTIONS
[R=1, STUB_INDENT=2] POOR MANAGEMENT/HELP
[R=1, STUB_INDENT=2] TOO CROWDED
[R=1, STUB_INDENT=2] HIGH PRICES
[R=1, STUB_INDENT=2] INSUFFICIENT PROJECT HELP
[R=1, STUB_INDENT=2] LACK OF LUMBER CHOICES
[R=0, STUB_INDENT=2] OTHER NEGATIVE
[R=1L]                NO RESPONSE
}
ROW=: [07.2, ..., 15.2*F#1//6/9/10//16/19/20]  }

~INPUT DATA CLN
~MAKE_READ_CONTROL    Q3READCTRL = Q3RC [6^1, 2/3, 4]

>PRINT_FILE HHSAT

~INPUT  DATA CLN, READ_CONTROL=Q3READCTRL (1)
```

```
~EXECUTE
HEADER=HEADSAT
TABLE_SET=BAN1
TABLE_SET=Q3, TABLE=*
TABLE_SET=Q3A, TABLE=*
RESET

>PRINT_FILE HHDISSAT

~INPUT  DATACLN, READ_CONTROL=Q3READCTRL(2)

~EXECUTE
HEADER=HEADDISSAT
TABLE_SET=BAN1
TABLE_SET=Q3, TABLE=*
TABLE_SET=Q3A, TABLE=*

~END
```

This run stores the READ\_CONTROL item Q3READCTRL in the open DB file HARDWARE and can therefore be called up in other runs without re-specifying.

A previously defined variable, i.e., SEX could be used in a ~MAKE\_READ\_CONTROL statement, but only if the responses were 1,2 or F,M because M,F isn't in sorted order.

The list file created by this run has a summary of the READ\_CONTROL item included. This summary looks like:

```
~MAKE_READ_CONTROL    Q3READCTRL = Q3RC[6^1,2/3,4]
Number of cases read not fitting into any category: 5
```

Number of cases fitting into category 1: 38

Number of cases fitting into category 2: 25

The print file HHSAT.PRT, the first table of which follows, has only those respondents who qualified for category 1 of the READ\_CONTROL item named Q3READCTRL. This category is those respondents who had a 1 or a 2 punch in column 6. The HEADER was created by the user and the table name was created automatically due to the use of TABLE=\*

EXAMPLE 1 TABLE:

SATISFACTION LEVEL: VERY/SOMEWHAT SATISFIED

TABLE 001

Q3. HOW SATISFIED ARE YOU WITH HARRY'S HARDWARE?

	AGE				
	TOTAL	UNDER 25	25-34	35-44	45 OR OLDER
TOTAL	38 100.0	7 100.0	15 100.0	10 100.0	6 100.0
N/A	-	-	-	-	-
VERY SATISFIED (1)	12 31.6	5 71.4	4 26.7	1 10.0	2 33.3
SOMEWHAT SATISFIED (2)	26 68.4	2 28.6	11 73.3	9 90.0	4 66.7
SOMEWHAT DISSATISFIED (3)	-	-	-	-	-
VERY DISSATISFIED (4)	-	-	-	-	-
MEAN	1.68	1.29	1.73	1.90	1.67

STD 0.47 0.49 0.46 0.32 0.52

**Related keywords:****TABLE\_FIELD**

Allows automatic table names to be suffixed with the categories specified by the READ\_CONTROL command. Causes a separate set of tables to be run for each category of the MAKE\_READ\_CONTROL variable. If this command is used then each category in the MAKE\_READ\_CONTROL variable can have only one value.

**#VARIABLE=<varname>#**

A System variable similar to #DATE# or #PAGE# that allows substitution of variable category labels into text strings.

**TABLE\_NAME**

A System constant which is the name of the last table the program has dealt with.

**JOIN**

A function used to join two text type variables or a text string with a text variable.

2) Automatic switching between four bases which are the responses to a prior scale question.

This example uses MAKE\_READ\_CONTROL and READ\_CONTROL to get four runs through the data, each based on a different response to a scale question. The tables are numbered and titled automatically using TABLE\_FIELD.

```
>CREATE_DB HARDWARE
```



```

~DEFINE
TABLE_SET= {BAN1:
  EDIT=: COLUMN_WIDTH=8
          STUB_WIDTH=22
          -COLUMN_TNA
          PERCENT_DECIMALS=1
          -PERCENT_SIGN
          STATISTICS_DECIMALS=2
          RUNNING_LINES=1
        }

BANNER=:
|
|                                     AGE
|          =====
|          UNDER                      45 OR
| TOTAL      25    25-34    35-44    OLDER
|  -----  }
COLUMN=: TOTAL WITH &
        [24^1,2/3/4/5,6] 'AGE
  }
  
```

```

~SPEC_FILES HHSPC ``this is required before the DEFINE
block
  
```

```

~DEFINE
TABLE_SET= {Q3:
  TITLE=:
          Q3.  HOW SATISFIED ARE YOU WITH HARRY'S
HARDWARE?\N  }
  STUB=:
  
```

## SPECIALIZED FUNCTIONS

### 9.4 PARTITIONING DATA FILES

```

                                VERY SATISFIED (1)
                                SOMEWHAT SATISFIED (2)
                                SOMEWHAT DISSATISFIED (3)
                                VERY DISSATISFIED (4)

[STATROW]  MEAN
[STATROW]  STD          }
ROW=: [6^1//4] $[MEAN,STD] [6]
}

TABLE_SET= {Q3A:
  TITLE=: Q3A.  WHY ARE YOU SATISFIED OR DISSATISFIED
WITH &
      HARRY'S HARDWARE?\N  }
  LOCAL_EDIT={:  RANK_LEVEL=1, MINIMUM_FREQUENCY=1 }
  STUB=:
[COMMENT, UNDERLINE] POSITIVE RESPONSES
[R=1, STUB_INDENT=2]    GOOD QUALITY TOOLS/HARDWARE
[R=1, STUB_INDENT=2]    PAYMENT CHOICES GOOD
[R=1, STUB_INDENT=2]    RESPONSIVE/COOPERATIVE
[R=1, STUB_INDENT=2]    HARRY'S IS GOOD HARDWARE
[R=1, STUB_INDENT=2]    GOOD PROJECT MANAGEMENT ADVICE
[R=1, STUB_INDENT=2]    EDUCATION/INFORMATION
[R=0, STUB_INDENT=2]    OTHER POSITIVE
[COMMENT, UNDERLINE] NEGATIVE RESPONSES
[R=1, STUB_INDENT=2]    POOR QUALITY TOOLS/HARDWARE
[R=1, STUB_INDENT=2]    INSUFFICIENT PAYMENT OPTIONS
[R=1, STUB_INDENT=2]    POOR MANAGEMENT/HELP
[R=1, STUB_INDENT=2]    TOO CROWDED
[R=1, STUB_INDENT=2]    HIGH PRICES
[R=1, STUB_INDENT=2]    INSUFFICIENT PROJECT HELP
```





```
[R=1, STUB_INDENT=2]      LACK OF LUMBER CHOICES
[R=0, STUB_INDENT=2]      OTHER NEGATIVE
[R=1L]                     NO RESPONSE          }
      ROW=: [07.2, ..., 15.2*F#1//6/9/10//16/19/20]
    }
```

```
CTRLNAME [ (TABLE_NAME) 6.1 # "VERY SATISFIED":1 /&
          "SOMEWHAT SATISFIED":2/&
          "SOMEWHAT DISSATISFIED":3/&
          "VERY DISSATISFIED":4 ]
```

```
LINES= {HEADSAT: =SATISFACTION LEVEL:
#VARIABLE=CTRLNAME#\N }
```

```
~INPUT DATACLN
```

```
~MAKE_READ_CONTROL      Q3READCTRL = Q3RC [6^1/2/3/4]
```

```
>PRINT_FILE HHALL
```

```
~INPUT DATACLN, DOTs=1, READ_CONTROL=Q3READCTRL (1, 2, 3, 4)
```

```
~SET   TABLE_FIELD="_" JOIN [ Q3RC $]
      AUTOMATIC_TABLES
```

```
~EXECUTE
```

```
HEADER=HEADSAT
```

```
TABLE_SET=BAN1
```

```
TABLE_SET=Q3
```

```
TABLE_SET=Q3A
```

```
RESET, PRINT_RUN
```

```
~END
```

**NOTE:** There must be one label for each category specified for the tables to be labeled properly. Punches don't have to be in the same order, but they must match.

The list file created by this run has a summary of the READ\_CONTROL item included. This summary looks like:

```
~MAKE_READ_CONTROL      Q3READCTRL = Q3RC[6^1/2/3/4]
Number of cases read not fitting into any category: 5
Number of cases fitting into category   1: 12
Number of cases fitting into category   2: 26
Number of cases fitting into category   3: 18
Number of cases fitting into category   4: 7
```

The print file HHALL.PRT, the first table of which follows, has only those respondents who qualified for category 1 of the READ\_CONTROL item named Q3READCTRL. This category is those respondents who had a 1 punch in column 6. The HEADER was created by using the CTRLNAME variable which is the sixth column of the table name with associated text. The table names, which because of AUTOMATIC\_TABLES being set would normally be T001, in this run have been JOINed to an "\_" and the four separate categories of the variable Q3RC made into a text string.

The pertinent lines in the spec file above are:

```
CTRLNAME [ (TABLE_NAME) 6.1 # "VERY SATISFIED":1/&
                                "SOMEWHAT SATISFIED":2/&
```



```

                                "SOMEWHAT
DISSATISFIED":3/&
                                "VERY DISSATISFIED":4 ]

    LINES= {HEADSAT: =SATISFACTION LEVEL:
#VARIABLE=CTRLNAME#\N
          }

~MAKE_READ_CONTROL      Q3READCTRL = Q3RC[6^1/2/3/4]

~INPUT,DATACLN,DOTS=1,READ_CONTROL=Q3READCTRL(1,2,3,4)

~SET  TABLE_FIELD="_",JOIN [ Q3RC $]
      AUTOMATIC_TABLES
  
```

Four different sets of tables were made (one for each category of the MAKE\_READ\_CONTROL variable Q3READCTRL and each with an automatically created HEADER), with a table names that look like T001\_1, T002\_1, etc. for category 1 and T001\_2, T002\_2, etc. for category 2 and so on, the tables are printed in the order of all category 1 tables in numerical order, then all category 2 tables in numerical order, etc.

## EXAMPLE 2 TABLE:

SATISFACTION LEVEL: VERY SATISFIED

TABLE 001\_1

Q3. HOW SATISFIED ARE YOU WITH HARRY'S HARDWARE?

	AGE				
	TOTAL	UNDER 25	25-34	35-44	45 OR OLDER
Total	12	5	4	1	2
	100.0	100.0	100.0	100.0	100.0
N/A	-	-	-	-	-
VERY SATISFIED (1)	12	5	4	1	2
	100.0	100.0	100.0	100.0	100.0
SOMEWHAT SATISFIED (2)	-	-	-	-	-
SOMEWHAT DISSATISFIED (3)	-	-	-	-	-
VERY DISSATISFIED (4)	-	-	-	-	-
MEAN	1.00	1.00	1.00	1.00	1.00



$$\begin{aligned}
\text{Sums:} & S_{km} = \sum W_{kjm} X_{kjm} \\
\text{Sums of Squares:} & Z_{km} = \sum W_{kjm} X_{kjm}^2 \\
\text{Squares of Weights:} & Y_{km} = \sum W_{kjm}^2 \\
\text{Cross Products:} & T_{km} = \sum W_{kjm} X_{kjm} X_{mjk}
\end{aligned}$$

**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  $j = 1, \dots, I$ ) then all the formulas reduce to their usual unweighted versions.

$$\begin{aligned}
\text{Mean:} & M_k = S_k/F_k \\
\text{Adjusted sum of squares:} & A_k = (Z_k - S_k^2/F_k)/(\sum_j W_j^2/\sum_j W) = (Z_k - S_k^2/F_k)/(Y_k/F_k) \\
\text{Effective sample size*} & E_k = F_k^2/Y_k \\
\text{Variance:} & V_k = A_k/(E_k - 1) \\
\text{Standard deviation:} & SD_K = \sqrt{V_K} \\
\text{Standard error:} & SE_k = SD_k/\sqrt{E_K}
\end{aligned}$$

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

Bivariate effective

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

## SETS OF VARIABLES: NEWMAN-KEULS PRELIMINARIES

Suppose  $\beta$  is a subset of the indices  $\{1, 2, \dots, I\}$ . We will need:



$V_{\beta}$ , the pooled variance and

$DF_{\beta}$ , the associated degrees of freedom.

Let  $E_{\beta} = \sum_{\beta} E_k$  then,

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

Let  $P_{\beta} = \sum_{\beta} S_k / \sum_{\beta} F_k$

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

$$DF_{\beta} = E_{\beta} - 1$$

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

Let  $G_{\beta}$  = the number of indices in  $\beta$

$$V_{\beta} = V_{\beta} = \frac{\sum_{\beta} A_k \cdot Y_k / F_k}{\sum_{\beta} (E_k - 1) Y_k / F_k}$$

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

\*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  $Q_{km}$   $k, m \in \beta$  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:

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

with all variables independent except:

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

and define:

$$M_1 = \frac{\sum X_1 + \sum X_2}{N_1 + N_c}$$



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

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

$$M_1 \sim N \left( b_1 = \frac{N_1 a_1 + N_c a_2}{N_1 + N_c}, S_1 = \frac{N_1 \sigma_1^2 + N_c \sigma_2^2}{(N_1 + N_c)^2} \right)$$

$$M_2 \sim N \left( b_4 = \frac{N_c a_3 + N_4 a_4}{N_c + N_4}, S_4 = \frac{N_c \sigma_3^2 + N_4 \sigma_4^2}{(N_c + N_4)^2} \right)$$

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

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

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

$$M_1 - M_4 \sim N \left( a_1 - a_4, \frac{\sigma_1^2}{N_1} + \frac{\sigma_4^2}{N_4} \right)$$

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

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

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

$$M_1 - M_4 \sim \frac{N_4}{N_4 + N_c} \cdot N \left( a_2 - a_4, \frac{\sigma_2^2}{N_c} + \frac{\sigma_4^2}{N_4} \right)$$

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

### 9.4.1 The Newman-Keuls Procedure

Given a set of indices  $\beta$ , do the following with all the indices  $k, m \in \beta$ .

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

If all variables are computationally independent, i.e.,  $E_{km} = 0$  for all  $k, m \in \beta$ ,  $k$  (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.,  $\text{Sum}_k = \sum_{m \in \beta, m \neq k} Q_{km}$

$l \in \beta$

$m \neq k$

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  $V_\beta$ .

**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  $X$ s are 0 - 1 (proportions), recalculate  $V_\beta$  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  $(DF_\beta, N_g)$  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.

### 9.4.2 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  $k, m \in \beta$  or

**1.2** The user specified separate tests.

In case 1.1, calculate  $V_\beta$  and  $Q_{km}$   $k, m \in \beta$  and test each  $Q$  separately using  $DF_{km}$ ,  $G_\beta$  as degrees of freedom.

In case 1.2, calculate  $V_\beta$ ,  $Q_{km}$  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  $D = e_1x_1 + e_2x_2 \dots + e_mx_m$  with  $\left(\sum_j\right)\left(\sum_m\right)\frac{E_{km}e_k e_m}{E_k E_m}$

$$e_k = 1 \text{ or } -1 \text{ for } k = 1, \dots, I.$$

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

then let  $D_1 = \sum e_k m_k$

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

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

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

**9.4.3 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  $(X_1, Y_1), \dots, (X_n, Y_n)$  are independent  $\sim N(\mu, \Phi)$

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

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

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

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

Define  $F = \sum W_i$

$$S_x = \sum W_i X_i$$

$$S_y = \sum W_i Y_i$$

$$V_x = \sum W_i X_i^2 - \frac{S_x^2}{F}$$

$$V_y = \sum W_i Y_i^2 - \frac{S_y^2}{F}$$

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

then  $S_x \sim N(F\mu_x, \sum W_i^2 \sigma_x^2)$

$S_y \sim N(F\mu_y, \sum W_i^2 \sigma_y^2)$

so  $E[S_x] = F\mu_x$

$E[S_y] = F\mu_y$

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

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

### ESTIMATE FOR MEANS, STANDARD DEVIATIONS, STANDARD ERRORS AND CORRELATIONS

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

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

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

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

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

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



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

then 
$$\hat{s}e_x = \sqrt{\frac{\sum W_i^2}{F^2} \sigma_x^2} \quad \text{so,}$$

$$\hat{s}e_x = \sqrt{\frac{\sigma_x^2}{\left(\frac{F^2}{\sum W_i^2}\right)}} \quad \hat{s}e_y = \sqrt{\frac{\sigma_x^2}{\left(\frac{F^2}{\sum W_i^2}\right)}}$$

(Note: As with  $std$ ,  $\hat{s}e$  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 
$$\hat{t} = \frac{\hat{\mu}_x - \hat{\mu}_y}{\sqrt{\hat{s}e_x^2 - 2\hat{v}se_x se_y + \hat{s}e_x^2}}$$

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

### FORMULAS FOR STATISTICS CREATED DURING TABLE BUILDING (ROW=)

Suppose  $X_i, W_i, i = 1, \dots, N$  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  $F = \sum W_i$

The SUM  $S = \sum W_i X_i$

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

$$N = \frac{F^2}{\sum W_i^2} \quad N - 2 = \frac{F^2 - \sum W_i^2}{\sum W_i^2}$$

The STANDARD DEVIATION  $sd = \sqrt{\frac{\sum X_i^2 W_i - \frac{S^2}{F}}{F - \frac{\sum W_i^2}{F}}}$

The STANDARD ERROR  $se = \frac{std}{\sqrt{\frac{F^2}{\sum W_i^2}}}$

**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  $\bar{X}$ ,  $sd$ ,  $se$  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.

	<b>Group 1</b>	<b>Group 2</b>
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).

	<b>Group X</b>	<b>Group Y</b>
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:  $Z_{1i} = \left(\sum_j\right) N_{ij} \cdot W_j$

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

OVERALL:

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

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

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

$$DF_N = C - 1$$

$$DF_D = N_T - DF_N$$

$$SS_W = SS_T - SS_B$$

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

$$MS_N = SS_B/DF_N$$

$$MS_D = SS_W/DF_N$$

## 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

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

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

**WALKER T-TEST**

Calculate  $F = \left( \frac{\text{STD}_1}{\text{STD}_2} \right)^2$  with  $DF = N_1 - 1, N_2 - 2$

If F is significant at .05 use  $D = (\text{SE}_1)^2 + (\text{SE}_2)^2$

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

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

**RANK SUM/WILCOXEN TEST**

- 1 Rank the cells in the cumulative sample. Call the ranks  $R_1, R_2, \dots$ , etc.
- 2 Call the sample size of the smaller group  $N_1$ . For this group, let:

$$T = \sum N_1 R_1$$

- 3 Call the size of the bigger group  $N_2$ .  $N = N_1 + N_2$
- 4 Normalize T with

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

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



- 5 Correction for ties: Call the ties  $T_1, T_2, \dots$ , etc.

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

**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 Edition, 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.



# INDEX

AND

See joiners

## Symbols

~SET TABLE\_NAME=

command 370

~SET TABLE\_NAME=name 368

## A

access DB file 374

Add Ranking To A Table 246

adding a base to a table 247

adding data to data file 123

adding statistics rows to tables 377

adding summary statistics to a table 249

adding/removing punches 123

adding/removing responses 140

additional data cleaning

commands 109

advanced functions

generating specialized reports 767

how to partition data files 767

table manipulation 767

advanced tables 381

All Possible Pairs

example 664

See significance testing

ALLOW\_INDENT meta 196

ampersand (&)

&filename 254

for DB items 196

ANOVA test

table-building 890

ANOVA\_SCAN

example 670

arithmetic calculations 142

ASCII data (#) 204

assigning variable names 178

asterisk (\*)

in cleaning 41

Survent variable modifiers 85

when reformatting data 142

with STORE 190

at sign (@)

with DEFINE meta 195

auto-fixing data 71

AXIS commands 326

## B

back to defaults

on AXIS commands 649

on EDIT commands 184

banner

banner points 205

defining 215

defining a procedure for

complex banner 266

definition 169

editing 227

formatting 223

formatting text 218

make\_banner 237

weighting banner points 481

banner\_title 227

banners

printing multiple banners 363

basic steps of report process 19

blanking data 114

## C

CALL\_TABLE 369

CfMC

getting Tech Support 24

Changing Table Element Defaults

(The DEFINE EDIT Statement) 181

Changing Table Processing Defaults

(The SET Statement) 185

Cleaning 43

cleaning data 28

cleaning data for error listing 91

cleaning data in batch mode 94

cleaning examples 41

cleaning process overview 27

cleaning punch data 43

cleaning specifications 32

cleaning with Survent variables 77

Codes

Print Specific Characters 775

column medians 433

on range type variables 436

commands

SET DROP 186

commands for statistical testing 764

correcting error messages 66

correcting errors 91

create banner

using make\_banner 237

create On-Demand tables 609

CREATE\_DB command 196

creating variables 143

cross-case operations 326

custom cleaning specifications 82

customizing specs 565

## D

data

auto-fixing 71

data cleaning

reasons why 29

data definitions 197

Data Location Variables 778

data manipulation commands 111

data manipulation for pre-defined

variables 133

data manipulation for punch, string,

and numeric variables 118

Data Manipulation in the

~CLEANER Block 158

## INDEX

Data Types 203  
database capabilities 16  
Database commands  
    CREATE\_DB 196  
database commands  
    USE\_DB 196  
DB commands 196  
DB file 196  
    accessing 374  
decimal points in data 153  
Default Varname Generation 180  
DEFINE command 195  
Defining a Procedure for Complex Banners 266  
defining data 197  
defining individual tables 245  
defining table elements 174  
defining the banner 215  
definition of Mentor 13  
drop\_banner\_title 227  
Dynamic table options 567

**E**

Edit  
    banner 227  
EDIT command 184  
edit option  
    banner\_title,  
        drop\_banner\_title 227  
error messages  
    correcting 66  
    generating a list 87  
    program-generated 40  
    sending to a print file 107  
erros  
    correcting 91  
E-Tabs 305  
expressions  
    basic 310  
    See also joiners 310

**F**

feature summary of Mentor 14  
Formatting a banner 223  
formatting banner text 218

formatting data elements 151  
Formulas  
    for table building 888  
formulas  
    Newman-Keuls 882  
function types 836  
functions  
    arithmetic 837  
    integer 855  
    logical 850  
    number returning 843  
    string 858  
    table related 852

## G

generating specialized reports 767  
getting started with Mentor, overview  
of using Mentor 13

## H

Holecount Table with a Varying  
Percentage Base 501

## I

installation requirements for On-  
Demand 589  
integration of Survent/Mentor 17

## J

joiner  
    definition 207  
joiners  
    BY 315  
    INTERSECT 317  
    JOIN 319  
    logical 312  
    NET 317  
    OTHERWISE 318  
    vector 315  
    WHEN 316  
    WITH 315

## L

logical joiners 310

## M

make\_banner format 237  
Making Several Tables 193  
mathematical joiners  
    logical 312  
    See also joiners 311  
medians  
    formula for column  
        medians 437  
    lost 439  
Mentor  
    benefits 13  
    definition 13  
    features 14  
Mentor EQUIVALENTS TO  
SPL 165  
Mentor features  
    database capabilities 16  
Mentor output files  
    post processing steps 612  
Mentor specs  
    customize to produce HTML  
        tables 565  
meta commands 195  
minus sign (-)  
    in cleaning 41

## N

Newman-Keuls (N-K)  
    See significance testing 655  
Newman-Keuls Procedure 882

## O

On-Demand  
    installation 589  
On-Demand tables 589  
    creating 609  
    sample output 612  
operators  
    See also joiners 325

## P

partitioning data files 862  
percentiles 443  
post processing



- prepare Mentor output files 612
  - preference testing
    - distributed 732
  - print
    - weighted/unweighted total
      - row 476
  - print specific characters
    - codes 775
  - PRINT\_FILE meta 195
  - PRINT\_LINES command 772
  - printing
    - a report footer 789
    - subtotal rows 549
  - Printing Individual Tables 189
  - printing text and data fields 114
  - Punch Data 203
  - punch data
    - cleaning 43
  - punch data punctuation 212
  - punches
    - adding 123
    - removing 123
  - Punctuation
    - for punch data 212
    - used in defining ASCII and
      - numeric data 214
    - used In referencing data field
      - locations 210
  - PURGE\_SAME meta. See also
    - Utilities manual, Appendix A
- Q**
- questionnaire
    - understanding framework 30
- R**
- range of numbers
    - creating stats for 407
    - mean summary tables 454
  - RANK SUM/WILCOXEN
  - TEST 892
  - ranking
    - nets and subnets 529
    - tables 246
    - top box/bottom box 381
  - Recoding 10-Point Scales 156
  - Recoding To Exclude Selected
    - Responses 157
  - Recoding To Reverse A Scale
    - Question 157
  - References
    - table-building
      - tests/formulas 893
  - relational operators 146
  - Report footer 789
  - rules for manipulating data 112
- S**
- sample basic table 264
  - sample error messages 37
  - sample specification files 251
  - Sample Specifications And
    - Table 220
  - See also minus sign
  - SET DROP 186
  - significance testing
    - ANOVA tests 735
    - ANOVA-Scan (A/S) 660
    - bi-level testing 640
    - changing confidence level 637
    - changing statistical base 644
    - chi-square 735
    - Fisher 660
    - inclusive T test 643
    - Kruskal-Wallis 660
    - Newman-Keuls (N-K) 655
    - on rows (preference testing) 726
    - reducing error results 654
    - repeated measures option 662
    - sets of formulas 654
    - setting confidence level 631
    - using nonstandard confidence
      - levels 642
  - single response string data 49
  - specific characters
    - codes to print 775
  - specifications
    - custom cleaning 82
  - spreading multi-punched data 154
  - Statistical Testing 883
  - statistical testing
    - commands summary 764
    - error messages 761
    - excluding any row 709
    - print phase 695
    - verifying 759
  - Statistics
    - for table building 888
  - statistics
    - changing default print
      - options 430
    - statistics, significance testing 627
  - Storing Tabsets in the DB file 190
  - Storing the Weight in the Data 479
  - string functions 858
  - STUB TABLE\_SET 221
  - SUFFIX= and PREFIX= EDIT 369
  - summary of chapters 22
  - Summary of Rules for Defining
    - Data 208
  - Support contact information 24
  - Survent
    - cleaning with Survent
      - variables 77
  - system constants 823
  - System Information Constants
    - datetimediff 833
    - juliandate 834
    - linenumber 835
    - mathvalues 835
    - offsetdate 833
    - randomvalue 836
    - tablename 836
- T**
- T and Z tests 891
  - table
    - add ranking, keep\_rank 246
    - parts of, definition 169
  - Table Building
    - INPUT and EXECUTE
      - statements 188
  - table building basics 172
  - table names 366
  - table-related functions 852
  - tables
    - adding statistics rows 377

## INDEX

- advanced 381
  - bottom box 381
  - break table with multi-level
    - banner 504
  - categories within data
    - variable 205
  - customizing for internet use 565
  - defining new edit statement 374
  - excluding respondents 319
  - expressions and joiners 309
  - holecount 501
  - Holecount table with rating
    - scales 498
  - intermediate 309
  - long brand lists 526
  - Master-Trailer processing 554
  - multiple location 507
  - On-Demand 589
  - overlayed banner and row 523
  - overlayed row and base 517
  - printing different names 372
  - printing name with
    - prefix/suffix 369
  - punctuation to create
    - categories 205
  - replacing 'TABLE' text before
    - table name 369
  - reprinting 372, 375
  - simple multiple location 509
  - specify starting name 368
  - specify unique names 370
  - suppressing blank rows 538
  - top box 381
  - weighted 470
  - Tech Support contacts 24
  - three levels of EDIT 184
  - transforming numbers into strings 156
- U**
- USE\_DB command 196
  - using cleaning screens 70
  - using E-Tabs 305
  - Using Survent to generate Mentor files 268
  - using Survent-type cleaning screens 91
  - using the DB file 251
- V**
- variable modifiers 85
  - variable references 777
  - variables
    - constants 825
    - data location 778
    - predefined 777
  - vector joiners 310
- W**
- WALKER T-Test 892
  - WebTables 565
- Z**
- zero-filled numeric data 49