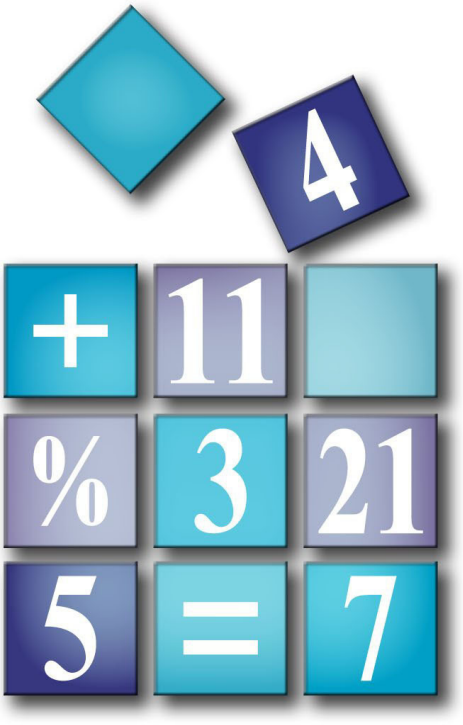
TPLTables®

The Professional Cross Tabulation System

Version 8.0



QQQ Software, Inc.

http://www.qqqsoftware.com
support@qqqsoftware.com

The software described in this document is furnished under a license agreement and may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software except as specifically allowed in the license agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose, without the express written permission of QQQ Software, Inc.

TPL TABLES User Manual Version 8.0

© Copyright 2014 QQQ Software, Inc. All rights reserved.

U.S. GOVERNMENT RESTRICTED RIGHTS. The program and documentation are provided with RESTRICTED RIGHTS. Any use, duplication or disclosure by the U.S. Government or authorized Government contractors is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, FAR 52.227-19 and other applicable agreements. The contractor/manufacturer is QQQ Software, Inc., 302 N. Irving St., Arlington, VA 22201.

TPL TABLES is a registered trademark of QQQ Software, Inc. All other product or company names are used for identification purposes only and may be trademarks of their respective owners.

This manual may be used with licensed copies of TPL Tables and demonstration versions of TPL Tables.

QQQ Software, Inc.
302 N. Irving Street
Arlington, VA 22201 USA
Tel: 703-528-1288
Fax: 703-528-1289
email: support@qqqsoftware.com
web: http://www.qqqsoftware.com

July, 2014

Preface

TPL TABLES is a powerful cross tabulation system that lets you summarize data and present the results in publication quality tables. It is based on the TPL mainframe system developed by the U.S. Bureau of Labor Statistics (BLS). Since 1987, TPL TABLES has been put to the test by professionals in both government and private industry in the U.S. and abroad.

TPL Tables Version 8.0 is available for PCs running Windows XP, VISTA, WINDOWS 7, WINDOWS 8 and various versions of UNIX and LINUX. Unless specifically noted, the information in this manual applies to both Windows and UNIX versions of TPL Tables. ***If you are using the Windows version, please refer to the online Help for information about how to use the interactive features.***

If you are viewing this document with Adobe's Acrobat Reader or a internet browser, you can click on entries in the Table of Contents or Index to be transferred to the pages in the text. Words in blue are hyperlinks to other parts of the manual. If you are using Acrobat Reader you can use ctrl+back arrow to get back to where you were. If you are using a browser, be sure to record the page you jump from if you wish to return. The browser does not provide a command to go back.

All of the table examples in this user manual were produced using TPL TABLES. The tables were then inserted into the text using a desktop publishing system.

With the release of Version 8.0, QQQ Software continues to improve upon the flexibility and functionality that has made TPL TABLES a success. We hope that you will enjoy using TPL TABLES and will write or call if you have questions, comments or suggestions. Your comments are important to us and will guide our selection of features to include in future versions of TPL TABLES.

Historical Notes and Acknowledgments

In the early 1970's, BLS began development of the mainframe Table Producing Language (TPL) system. The goals of the system were:

1. It should produce most, if not all, of the Bureau's statistical tables.

2. It should be driven by a language that did not require the user to be a programmer or computer scientist.

3. It should be flexible and adaptable to changing needs for new tables and formats.

4. It should lead the way to composition of tables for publication.

By the end of 1984, when BLS development ended, TPL had gone through six major releases. It had been distributed to nearly 300 mainframe sites around the world and was one of the most frequently used computer systems at many of these sites, including large national statistical agencies.

The development of the BLS system was made possible by the contributions of many people. Rudolph Mendelssohn, Assistant Commissioner, provided initial encouragement and guidance. Richard Heddinger and Pamela Weeks headed the initial language design and the implementation work under the direction of Peter Stevens, Chief, Division of General Systems. Stephen Weiss later replaced Richard Heddinger as chief programmer. Other major participants in the development of TPL were: Victor G. Stotland, Nancy Byrd, Eugene C. McKay, John D. Sinks, Roxana Kamen, David H. Miller, Jane Powers, and Kenneth Buckley. Early inspiration was also provided by Hugh Brophy during his time at the Australian Bureau of Statistics.

Because of the similarities between the user language of TPL TABLES and that of the BLS system, some parts of the TPL TABLES User Manual are derived from the BLS Version 6 TPL Language Guide. The BLS language guide was prepared by Stephen Levenson. Materials contained in the BLS Version 6 TPL Language Guide are in the public domain.

TPL TABLES is the product of QQQ Software and is primarily the work of Stephen Weiss and Pamela Weeks.

Summary Contents

Contents

Codebook.....90

Run Instructions (UNIX/Linux) 777

CHAPTER 1

Introduction

What Does TPL TABLES Do?

TPL TABLES is a specialized cross tabulation system that lets you summarize data and present the results in tabular form. It can work with data files of many different formats, including hierarchical files. It can process an unlimited amount of data and produce tables that range in size from a few lines to hundreds of pages. Subsets of the data can be selected and new variables can be computed from existing data. Other computational features include percent distributions, maximums, minimums, medians and other quantiles.

Cross tabulation lets you look at your data in new ways by counting or summarizing things into categories. With TPL TABLES, you can request tabulations in an endless variety of ways, and you have complete control over the structure and content of your table output.

TPL TABLES automatically formats your output into quality tabular reports. Optional format commands are available if you want precise control of format details. Tables can be altered by deleting rows and columns, changing labels and titles, and changing the format of numbers. Special format features include footnotes for print labels and data cells. One format command strips the table of everything but the data, thus producing an ASCII data file output that can be used as input to other software packages.

TPL TABLES brings desk-top publishing to data. TPL TABLES makes it easy to create sophisticated, publication-quality tables with choices of type style and size, including proportional fonts. The tables can be printed directly or incorporated in documents that have been created with desk-top publishing software.

TPL TABLES has a wide variety of applications. Typical users are professionals in the fields of finance, economics, statistics, marketing, sales and human resources who are employed by government agencies, corporations,

and universities. However, anyone who needs to get summary information from a data file or prepare tables for publication is a potential user of TPL TABLES.

How Does TPL TABLES Work?

The ingredients needed to create tables are: a data file, a codebook that describes the data file, and a table request that describes the tables. An optional ingredient is a format request that makes changes to the automatic table formats.

The Data File

TPL TABLES can work with data files from a variety of sources. For example, the data can be exported from a database or spreadsheet, downloaded from a mainframe, or prepared using an editor or data entry system. If you have the TPL-SQL database interface, TPL TABLES can also read data directly from a database. TPL TABLES does not prepare the data, import it into a format of its own, or change the data in any way. It simply reads it and extracts the information needed to produce the tables you want.

The Codebook

The first step in creating tables from a particular data file is to prepare a codebook that describes your data. It contains information such as the names of data fields, where they are located within a record and how many character positions (bytes) each occupies within a record. Since TPL TABLES does not require that your data be in a particular format, it needs this information in order to find the data values that you wish to use in your tables.

The codebook is a text file that can be prepared with an editor. In the Windows version of TPL TABLES, you also have the option of preparing the codebook interactively. For the UNIX version, the *tpl conditions* program can assist you in preparing the codebook.

After you have prepared the codebook, TPL TABLES will process it and convert it to a form that it can use to work with the data. When this process is complete, you can use the codebook over and over to create any number of tables from the data file.

The Table Request

The second step in creating tables is to prepare a table request. The table request contains TPL statements that describe the tables you want. You can reference any of the variables in your codebook by name. In addition, you can select subsets of the data file, compute new variables and define new categories for existing variables.

The most important statement in a table request is the one that describes the structure and content of a table. You can request one or many tables in the same table request.

The table request is a text file that can be prepared with an editor. In the Windows version of TPL TABLES, you also have the option of preparing the table request interactively.

Once you have prepared the codebook and the table request, TPL TABLES can read and tabulate your data to produce the tables you have requested. It will automatically format the rows and columns of the tables as directed by the table statements, using names and labels from the codebook and table request.

The Format Request

An optional third step is the preparation of a format request. It contains FORMAT statements that you use to make changes to the table format. The format request can be used with the table request when a table is first produced. It can also be used alone to reformat a table that was prepared earlier.

Like the table request, the format request is a text file that you can prepare with an editor. In the Windows version of TPL TABLES, you also have the option of editing your tables interactively to create a format request.

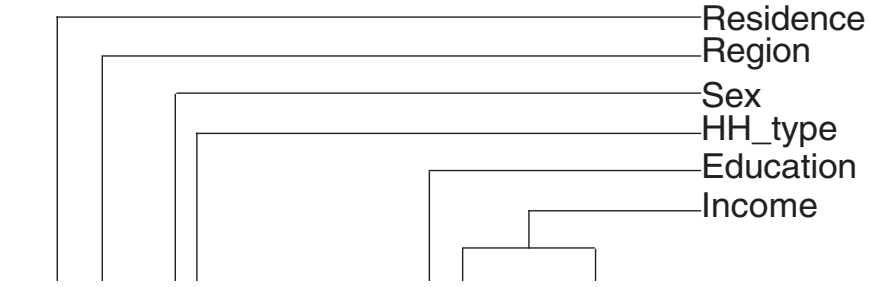
The automatic formats provided by TPL TABLES are usually acceptable for analysis and for some types of publications. However, if your organization publishes tables, you may need to make format adjustments to meet the publication standards of your organization. In other cases, you may find that the defaults for such things as column widths or page size are not appropriate for all of your tables. These table characteristics can be quickly and easily changed with FORMAT statements.

An Example

Following is an example that illustrates how a data file, a codebook and a table request work together in TPL TABLES.

Data

First is a small sample of ten records from a data file that contains information about 58,699 households. Each record in the data file represents one household.




```
90101121134060000241030630019847 2
90101121155030000241030831019492 4
90103121137060000141029200019235 9
90203121333020000262041500018789 9
90202121131030000141030048020328 4
90101121338020000261052000018966 9
90202121151030000241042924019844 4
90202121136040000241033372019187 6
90103121155040000221029000019712 6
90103121122020000241028300019187 6
```

Codebook

Next is the codebook that describes the data items that we plan to use with TPL TABLES. Each data item is described in the order of its occurrence on the data record. FILLER entries account for the parts of the record that we do not plan to use.

```
BEGIN  HH  CODEBOOK

HOUSEHOLDS  'Households'  MASK  99,999  RECORD

FILLER 2

RESIDENCE  'Type of Residence'  CONTROL  1
  (
        'Inside metropolitan areas'       = 1
        'Outside metropolitan areas'      = 2
  )
FILLER  1
REGION  CONTROL  1
 (
        'Northeast' = 1
        'West' = 2
        'South' = 3
  )
FILLER 2

SEX 'Sex of Householder' CONTROL 1
  (
        'Male' = 1
        'Female' = 2
  )
HH_TYPE  'Type of Household'  CONTROL  1
  (
        'Married couple'          = 1
        'Other family'            = 2
        'Nonfamily household'     = 3
  )
FILLER  9

EDUCATION   'Education of Householder'  CONTROL  1
  (
        '8 years or less'                 = 1
        'Some High School'      = 2
        'High School Graduate'  = 3
        'Some College'          = 4
        'College Graduate'      = 5
        'Post Graduate'         = 6
  )
FILLER  1

INCOME   'Income'  OBS   6

FILLER  7

END  HH  CODEBOOK
```

Table Request

The following table request begins with a USE statement that references the name of the codebook to be used with the data. There is one TABLE statement. In this statement, the data items, HOUSEHOLDS, REGION and EDUCATION, are used directly from the data file. The other variables used in the table are defined or computed by other statements. New income categories are defined, and average household income is computed from tabulated values.

```
USE  HH  CODEBOOK;

DEFINE  INCOME_GROUPS  ON INCOME;
        'Household Income Under $30,000'      IF < 30000;
        'Household Income $30,000 and Over'  IF >= 30000;

POST COMPUTE AVERAGE  'Average Income'
        MASK $99,999 =  INCOME / HOUSEHOLDS;

TABLE ONE
        'Table Q1.  Number of Households and Average '
        'Household Income by Geographical Region '
        ' and Education of Householder.':
        HEADING
                (TOTAL THEN INCOME_GROUPS) BY
                        (HOUSEHOLDS THEN AVERAGE);
        STUB
                 (TOTAL THEN REGION) BY EDUCATION;
```

The Table Output

TPL TABLES reads the table request, the data file and the codebook. It uses the codebook to find the required items in the data file. It selects these items from each record, assigns them to the requested categories, then sorts and summarizes them to obtain the tabulated values requested for the table. Finally, it formats the tabulated values into the rows and columns, using labels from the codebook and table request.

Table Q1. Number of Households and Average Household Income by Geographical Region and Education of Householder.

	Total		Household Income Under $30,000		Household Income $30,000 and Over	
	Households	Average Income	Households	Average Income	Households	Average Income
Total						
Education of Householder						
8 years or less	7,846	$16,613	6,685	$11,630	1,161	$45,305
Some High School	7,153	21,234	5,433	13,110	1,720	46,896
High School Graduate	21,200	28,959	12,644	15,796	8,556	48,411
Some College	10,013	34,357	4,979	16,658	5,034	51,862
College Graduate	6,859	45,539	2,345	18,403	4,514	59,636
Post Graduate	5,628	56,120	1,408	18,532	4,220	68,661
Northeast						
Education of Householder						
8 years or less	1,765	17,698	1,473	11,562	292	48,653
Some High School	1,694	22,102	1,264	13,194	430	48,288
High School Graduate	5,138	31,727	2,818	16,049	2,320	50,771
Some College	1,940	37,098	852	16,809	1,088	52,986
College Graduate	1,801	46,717	575	18,850	1,226	59,787
Post Graduate	1,489	61,189	317	17,758	1,172	72,937
West						
Education of Householder						
8 years or less	3,014	17,510	2,533	12,404	481	44,400
Some High School	2,968	21,585	2,219	13,192	749	46,452
High School Graduate	10,028	28,456	6,028	15,795	4,000	47,538
Some College	5,123	33,819	2,620	16,430	2,503	52,020
College Graduate	3,011	44,698	1,037	18,344	1,974	58,543
Post Graduate	2,569	55,272	667	18,744	1,902	68,082
South						
Education of Householder						
8 years or less	3,067	15,107	2,679	10,936	388	43,908
Some High School	2,491	20,225	1,950	12,961	541	46,406
High School Graduate	6,034	27,438	3,798	15,611	2,236	47,526
Some College	2,950	33,489	1,507	16,971	1,443	50,739
College Graduate	2,047	45,738	733	18,137	1,314	61,136
Post Graduate	1,570	52,698	424	18,776	1,146	65,248

CHAPTER 2

Overview

AN OVERVIEW OF TPL TABLES FEATURES

This chapter provides a brief introduction to the basic TPL TABLES features. The features are described in the approximate order of the chapters and appendixes of the User Manual.

Defining the Structure and Content of a Table

The TABLE statement allows you to define both the structure and content of a table by specifying the columns (HEADING), the rows (STUB) and the optional repetitions (WAFERS) of the basic column and row structure. Within each of these three components of the TABLE statement, variables can appear next to each other as independent tabulations, or be combined to represent tabulations satisfying the conditions of multiple variables.

This simple, yet powerful, statement is the key to TPL TABLES' flexibility in designing and producing tables. Any discussion of the TABLE statement is truly a case of "a picture is worth a thousand words". The TABLE statement chapter contains many examples and illustrations that show how you can design tables in endless variety.

Data Files

TPL TABLES reads data from sequential data files. File structures can be either "flat", containing only one type of record, or hierarchical. Hierarchical files contain a variable number of related records of increasing detail. Data can be stored as ASCII characters or as binary or floating point numbers. Only one data file format, described by a single codebook, can be processed at one time by TPL TABLES. Multiple data files with the same file format can be processed in one job. TPL TABLES can also read CSV and other types of delimited data files. With the TPL-SQL option, TPL TABLES can read data from a variety of databases.

Describing the Data

The input data file is described to TPL TABLES by means of a codebook. The codebook describes the file structure, naming each record that makes up a processing unit. Each data item of the record is assigned a name and an indication of whether the variable represents a classifying variable or contains values to be aggregated. Each entry for a classifying variable includes a list of all of its possible values.

The codebook is created as a separate step before tables can be produced. Once the codebook is created, it can be referenced any number of times.

Selecting Subsets of the Data

A SELECT statement can be used to tabulate only a subset of the data file. Data can be selected based on data values, or certain sections or percentages of the data can be selected. The SELECT statement can contain combinations of logical and arithmetic tests. Multiple tests on several variables can be strung together with AND's and OR's.

Reclassifying Data

TPL TABLES provides a very powerful DEFINE statement for regrouping, reordering and deleting values for a variable. For example, income amounts can be classified by ranges, with certain incomes eliminated from the tabulation.

Computing New Values and Weighting

COMPUTE statements can be used to create new variables by combining variables from the data file with arithmetic operations. Weighting is a common application whereby each record contains a weighting factor which is applied to one or more other variables in that record. The weighted values from each record can then be tabulated. Alternate computations can also be requested, depending on whether specified conditions are met. Arithmetic operations allowed in computations include addition, subtraction, multiplication, division, and exponentiation, plus the absolute value and square root functions.

Computing New Values from Final Tabulations

Arithmetic operations can be performed on summarized table values to produce averages, percentages, standard deviations and other calculations. The statement for computations involving summarized values is called POST

COMPUTE. As with COMPUTE, alternate computations can be specified, depending of whether specified conditions are met.

Percentages

The PERCENT feature allows great flexibility in calculating and displaying percents. You can specify whether only percents or both original cell values and percents are to be displayed. More than one type of percent distribution can be calculated within the same table.

Statistics

Maximum and minimum values, medians, and quantiles such as deciles and percentiles can be calculated and displayed in a table with complete flexibility. Other statistics include means, variances, standard deviations, and standard errors.

Ranking

Table rows can be ranked (sorted) in descending or ascending order based on the values in a selected data column. Optionally, a rank column can be added to the table to display the rank number for each row. Another option lets you keep only the top (or bottom) **n** rows for a particular ranking. With this option, you can request a row to display the residual.

Labels

Descriptive print labels can be assigned as table titles, variable and value labels, and footnote texts. Labels can include spaces, upper and lower case letters, and special characters. Break points can be chosen for multiline labels, and alignment can be specified. Labels can also contain references to footnotes. You can vary the type styles within labels.

Masks

Masks can be used to control the format of values printed in table cells. With a mask, you can format data to show decimal places, include special characters such as dollar signs and percent symbols, and specify the alignment of data within a column. Masks can also reference footnotes. You can also choose the type style for table cells.

Footnotes

Footnotes can be referenced in labels and masks for automatic inclusion in printed tables. The footnotes can be described and referenced in codebooks, table requests, format requests and in the TPL TABLES profile. Printing of footnotes can also be conditional. In other words, you can specify that a table cell value should be footnoted, or replaced by a footnote, in certain circumstances, including situations in which the data is confidential. A separate chapter describes footnote features in general; techniques for conditional footnoting are described in the Post Compute chapter.

Table Formatting

Tables can be formatted automatically, but, in addition, many details of table format can be adjusted with FORMAT statements. Column widths can be altered, footnotes can be added, tables can be split into sections on the same page, separate tables can be combined onto the same page, and extensive relabeling can be done. Tables can be prepared for publication allowing type size, style, and boldness to be specified. You can also request that a table be turned into a data file, or you can export the table in web page format (HTML), delimited(CSV) format, spreadsheet format (XLS or ODS) or text table format.

After a table has been produced initially, extensive reformatting can be done without reprocessing the data file.

INSTALLING AND RUNNING TPL TABLES

Complete instructions for installing and running TPL TABLES are contained in appendixes to this User Manual. The following is a quick summary of the steps to produce tables.

1. Write the codebook statements necessary to describe the data file. Run the codebook processor to create a codebook "object". If you want to make changes to your codebook after it has been processed, you can make the changes and rerun the codebook processor. Otherwise, you only need to do it once. Any number of TPL TABLES jobs can be run using the same codebook object.

2. Write TPL statements to describe the tables you want. Run the procedure to produce the tables. This procedure uses your codebook

object and your TPL TABLES statements to read the required data and produce the tables you have described. You can request that the system print the tables immediately, or you can print them later. The tables will always be saved until you decide to remove them.

3. If the automatic table format is not acceptable for a table, you can reformat it using FORMAT statements in a format request. Write the FORMAT statements to make the desired changes and rerun the table formatting procedure.

CHAPTER 3

Entering Statements

RULES AND NOTATIONS FOR CODEBOOKS AND REQUESTS

Statement Rules

Codebook, TPL and FORMAT statements are free format; that is, there are no requirements to begin entries at fixed column positions.

When you enter words using lower case letters, TPL TABLES treats them the same as upper case letters unless you enclose them in single or double quote marks (' or ").

For ease of reading, it is best to structure statements so that entries of the same type are neatly aligned.

Identifiers

Identifiers are names that you create to refer to items such as tables and variables. They can be up to 30 characters long and can contain letters, numbers, and the special characters # and underscore (_). An identifier cannot begin with a number and cannot contain embedded blanks. An identifier is terminated by any character from the TPL TABLES character set other than a number, a letter, # or _. Letters can be upper or lower case. When TPL TABLES reads a lower case letter in an identifier, it converts it to upper case.

Values

Numeric values used as constants can contain embedded decimal points. Optional zeros can be added to the left of the values. For example, 053 is the same as 53. Alphabetic values must be enclosed in quote marks. To enter alphabetic values that contain quotes or the backslash character (\), see the instructions below under Print Labels.

Keywords

TPL TABLES uses many words which identify certain functions and must not be used as names. These keywords are shown in an appendix. Keywords can be entered in upper or lower case.

Print Labels

Data names can be given extensive labels which appear automatically on printed output. These labels are bounded with quote marks and are not limited in length. All characters can be used in labels, although the characters ' " and \ require special treatment. Tabs and carriage returns (typed with the **<Enter>** key) should not be used in labels. Tabs are replaced with blanks, and carriage returns are removed when labels are printed.

If you are using single quotes to enclose a label string and need to include a single quote within the string, use two single quotes where you want the single quote to print. An example is **'Inside MSA''s'**, which would print as **Inside MSA's** if used in a table. Similar instructions apply to the use of double quotes.

Backslash

To include the backslash (\) character in a string, enter a double backslash (\\) at the point where you want the backslash to print. This special treatment is necessary because the backslash is used to enter characters that are not on the keyboard.

There are many other label options, all of which are described in a separate chapter called Creating and Formatting Print Labels.

Entering Characters that Are Not on the Keyboard

You may have some characters available on your printer that cannot be entered directly from your keyboard. This is especially true if you want to use special characters as footnote symbols or if you need to enter non-English language characters with an editor that doesn't support these characters.

TPL TABLES provides two ways to enter characters not on your keyboard. One way is to use character names and the other is to use character codes.

You can enter the character as a code by typing **\nnn** (that is backslash followed by 3 decimal digits) to represent the character.

Three digits are always required. If the character can be represented by fewer than 3 digits, add leading zeros. For example, for a character represented by the code 65, enter \065.

The value **nnn** must be the DECIMAL code for the character. The character code tables in some software and printer manuals show the octal or hexidecimal codes for the characters. If you are referring to such a table, you must convert the code to its decimal equivalent. There are tables in the **Character Sets** appendix that show the decimal codes used by TPL for characters.

The other way to enter characters not on your keyboard is to use the character name preceded by **&** and followed by **;**. **É** is the character name for the letter **E** with an acute accent. Character names are case-sensitive. **é** is **e** with an acute accent. See the Appendix **Interational** for more on entering characters and the **Character Sets** appendix for a list of supported names.

Dashes in TPL TABLES

There are three sizes of "dash" characters available in TPL TABLES. A short dash is used for hyphenation. This dash is the hyphen character on your keyboard. A medium dash (**endash**) is used as the footnote symbol for the EMPTY built-in footnote ("Data not available."), and a long dash (**emdash**) is used in title continuations. The choice of dash can be changed for the title continuation with the FORMAT statement REPLACE TITLE CONTINUATION; and the footnote symbol for EMPTY can be changed with a SET FOOTNOTE statement.

The medium and long dashes are special characters that are not on your keyboard but can be entered with **&endash;** and **&emdash;**.

Suppose we wish to change the symbol for the EMPTY footnote. The default symbol for the EMPTY footnote is the medium dash . This statement will replace it with the long dash character.

Example

```
SET FOOTNOTE EMPTY SYMBOL '&emdash;';
```

Mathematical Operators

Statements which involve computations use the mathematical symbols of +(addition), -(subtraction), *(multiplication), /(division), **(exponentiation) and =(equals). Mathematical symbols need not be separated from other elements by spaces.

Comment Entries

You may add your own comments anywhere in a codebook, table request or format request. Comments allow you to include documentation with your statements.

A comment must begin with **/*** and end with ***/.** For example,

```
/* This is a comment. */
```

All characters can be used in comments. The only exception is the pair of characters */, since this pair of characters ends a comment.

Notation Used in Presenting Statement Formats

In this manual, the syntax for TPL TABLES statements is described using a symbolic notation.

- Entries surrounded by [ ] are optional.
- Vertically stacked entries indicate that one entry must be chosen.
- Keywords are presented in UPPER CASE.
- Entries presented in lower case are to be replaced with the proper elements.
- The special delimiters **= ( ) ; > < ^** are presented as they should appear in the statement specification.

The "INCLUDE" Feature

If you have a set of statements or other information that you would like to store in a separate file and then use in multiple codebooks, table requests or format requests, you can use the following notation to get this file included in your codebook or request:

%INCLUDE filename

TPL TABLES will include the contents of the named file during processing of the codebook or request. You can also use %INCLUDE in a profile.

Some common uses of %INCLUDE are:

1. inclusion of a long list of condition values and labels that apply to more than one control variable, either within the same codebook or in multiple codebooks;

2. inclusion of a long DEFINE statement in several table requests;

3. inclusion of a set of FORMAT statements in format requests that apply to a particular group of tables.

The include notation must be on a line by itself and must begin in the leftmost position on the line, i.e. there should be no blanks or other characters preceding the % sign. ***NOTE*** *that the include notation must not be followed by a semicolon.*

A codebook, request or profile can have multiple "includes".

"Nesting" is allowed. This means that an included file can include other files. Ten levels of nesting are allowed.

If you have an error in an included file and choose to review your output to find the error message, the review will display the included file as if it is part of the main codebook, request or profile file. Comments will show where the included file begins and ends. If you then wish to edit, you must keep track of which file has the error, because the main file will be transferred to your editor. Do not edit this file, but, instead, bring the appropriate included file into your editor and correct the errors in that file before returning to codebook or request processing.

Example Assume that we have three variables in the same codebook that all use the same long list of country values and labels. The list can be stored in a file called COUNTRY.LST as follows:

```
(
        'Australia'                     =  01
        / 'Northern Europe' /
        'United Kingdom and Ireland'    =  02
        ' Austria'                      =  03
        ' Belgium'                      =  04
        ' France'                       =  05
                .
                .
                .
        'New Zealand'                   =  38
        'Other Oceania'                 =  39
        / 'N/S'                         =  40
)
```

Then in the codebook, we can reference the list for each country variable:

```
BPF 'Birthplace of father'     CON 2
%INCLUDE COUNTRY.LST

BPM 'Birthplace of mother'     CON 2
%INCLUDE COUNTRY.LST

CIT 'Country of citizenship'   CON 2
%INCLUDE COUNTRY.LST
```

Substitutions for Names, Labels and Numbers

You can make substitutions for identifiers (names), strings and numbers in a codebook, table request, format request or profile. To do this, you assign a name to the item you wish to replace and precede the name with the character **%** (no blanks between). You fill in the specific name, string or number with a **REPLACE** statement.

For example, you might have a set of tables that you produce from time to time and the only thing that you need to change in your table request is the date appearing in the table titles. Rather than looking through your table request to find and change all of the dates to the current date, you would like to make the change in date just once.

To do this type of substitution, you can give the date a name and use this name in all of the table titles with the character **%** in front of the date name. Assuming the date is called MO_YR, you can write a table title such as:

```
'Latest information as of ' %MO_YR '.'
```

Somewhere preceding the first use of MO_YR, you must provide the information to replace MO_YR. For example:

```
REPLACE MO_YR WITH 'January, 1996';
```

For this replacement, the table title will be:

Latest information as of January, 1996.

You will probably want to put your REPLACE statements at the beginning of the request, codebook or profile in which they are used so that they will be easy to find. The only rule with respect to placement is that the **REPLACE** statement for **%name** must always precede the use of **%name**. For example, you cannot use **%name** in a table request and replace it in a format request.

REPLACE statements can be entered at the very beginning or between other statements in requests and in the profile. In a codebook, they can be at the beginning or between entries such as variables or fillers.

To replace a string, you must use quotes in the REPLACE statement. To replace a name or number, just provide the replacement name or number without quotes.

Examples of number replacement

```
REPLACE MASK_TYPE WITH 99,999.99;

COMPUTE INCOME MASK $%MASK_TYPE =
        WEEKLY_INCOME * 52;
```

or

```
REPLACE THIS_MONTH WITH 3;
REPLACE LAST_MONTH WITH 2;

DEFINE SELECTED_MONTHS ON MONTH_CODE;
'This month'     IF %THIS_MONTH;
'Last month'     IF %LAST_MONTH;
```

Example of name and label replacement

```
REPLACE SALES_ITEM WITH AUTOS;
REPLACE SALES_LABEL WITH 'Automobiles';

TABLE R1 'Monthly sales figures for ' %SALES_LABEL :
STUB  DEALERSHIPS;
HEADING  MONTH BY %SALES_ITEM;
```

Putting REPLACE Statements in %INCLUDE Files

If you wish, you can put your REPLACE statements in one or more %INCLUDE files. This means, for example, that you can change entries for a table request without changing the request itself. The %INCLUDE must be entered in the request at a point that precedes the first use of any of the substitutions in the %INCLUDE file.

Example We can redo the preceding example as follows. In a %INCLUDE file called ITEM, we can enter the REPLACE statements:

```
REPLACE SALES_ITEM WITH AUTOS;
REPLACE SALES_LABEL WITH 'Automobiles';
```

In the table request, we can have a %INCLUDE entry:

```
%INCLUDE  ITEM

TABLE R1 'Monthly sales figures for ' %SALES_LABEL :
STUB  DEALERSHIPS;
```

```
HEADING  MONTH BY %SALES_ITEM;
```

With this approach, we can generate tables for different sales items by changing only the file called ITEM to substitute the desired item into the table heading and substitute its description into the table title.

Using Substitutions with Formulas in %INCLUDE Files

If you have a long, complex series of statements, such as formulas specified with Computes and Post Computes, and you need to apply these to many variables, you may wish to enter these statements in a %INCLUDE file with %name in each place where a specific variable is to be used and %label to assign appropriate labels.

In the following example, cell values for all tabulated observation variables are to be displayed in different ways or suppressed, depending on a variety of conditions. If there were only one variable that needed to be tested, we would enter the statements directly in the table request. However, if there are many variables that need to be tested in the same way, the table request will become quite long, and with many repetitions of the same statements, there is increased chance of making an error.

To simplify the process, we can put the following statements in a %INCLUDE file and save the file with the name FORMULA.

```
compute %CNT  =
1        if    abs (%VBL) > 0;
null     if    other;

compute %WT =
WT_1  if    abs (%VBL) > 0;
null     if    other;

post compute %VAL %LBL =
%VBL                              if      %VBL = 0;
mask text right '-0'              if      abs (%VBL) < 700 and
                                          max (%WT) < 2.5;
mask text right '*-0'             if      abs (%VBL) < 700 and
                                          %CNT < 12 and
                                          max (%WT) >= 2.5;
%VBL mask right '*'999,999        if      abs (%VBL) >= 700 and
                                          %CNT < 12 and
                                          max (%WT) >= 2.5;
%VBL                              if      other;
```

In the table request, we can replace the % entries with a series of simple REPLACE statements and include the formula for each variable we need to test. Only the first two are shown below, but we could repeat this pattern for as many variables as needed.

```
replace VBL with SALES;
replace CNT with SALES_CNT;
replace WT with SALES_WT;
replace VAL with SALES_VAL;
replace LBL with 'Sales';
%include FORMULA

replace VBL with EXPENSE;
replace CNT with EXPENSE_CNT;
replace WT with EXPENSE_WT;
replace VAL with EXPENSE_VAL;
replace LBL with 'Expenses';
%include FORMULA
```

Now we can use the new variables in a Table statement. For example:

```
TABLE R1:   HEADING  INDUSTRY,
            STUB  SALES_VAL THEN EXPENSE_VAL;
```

CHAPTER 4

Tables

DEFINING THE STRUCTURE AND CONTENT OF A TABLE: THE TABLE STATEMENT

Before requesting tables, you must describe your data file in a codebook so that TPL TABLES will know the names and locations for the data values that you want to use in tables. The two chapters following this one provide details on the types of data files that can be used and how to describe them in a codebook.

TABLE statements are prepared using an editor and saved in a file called a TPL table request. The smallest table request contains just a USE statement, telling TPL TABLES which codebook to use, and a TABLE statement, but you can request as many tables as you want by including additional TABLE statements.

The TABLE statement allows you to define both the structure and the content of a desired output table. This one comprehensive statement specifies the types of tabulations to be performed and the arrangement of the tabulated data for output. TPL TABLES takes care of all formatting details, using names and labels from the codebook or from other types of statements included in the table request.

Please note that if you are working with a hierarchical (multi-level) data file, you can tabulate information from different levels of the file. Multi-level tabulations are described in the chapter on processing hierarchical files.

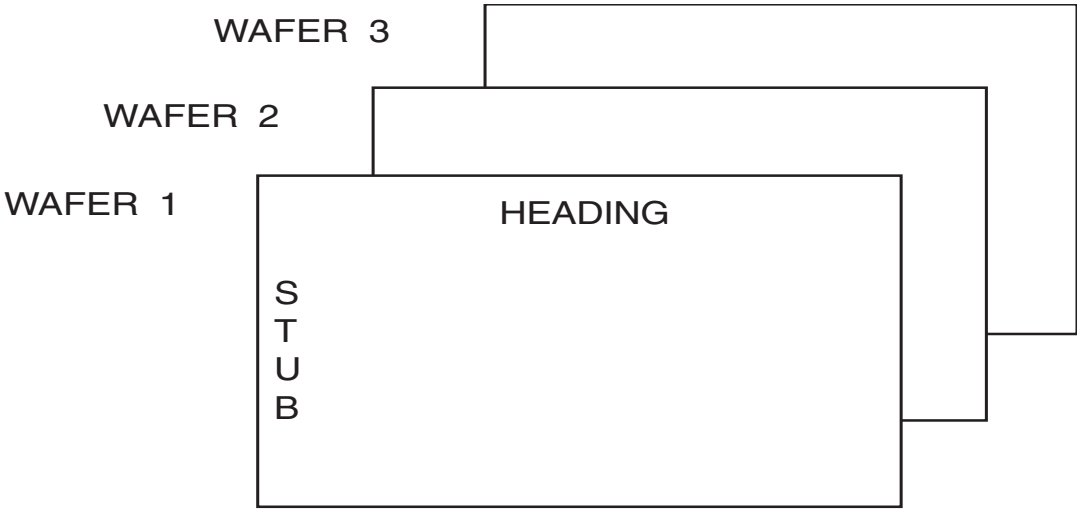
Specifying Column, Row, and Wafer Dimensions

Format The general format of the TABLE statement is:

```
TABLE name ['table title'] :  [ Ew, ]  Es,  Eh;
```

where **name** identifies the table being specified. The name can be followed by an optional **table title**. If you do not include a title in the TABLE statement, the table name will be used as the title. Table titles can include any of the options allowed in other labels. These are described in detail in the chapter on labels.

The expressions Ew, Es, and Eh represent the three components of the table. An expression can be a variable name or a combination of variable names. The expression Eh is called the **heading** expression and defines the column structure of the table. The expression Es is called the **stub** expression and defines the row structure of the table. The two expressions Eh (heading) and Es (stub) define a two-dimensional array called the **wafer**. The optional expression, Ew, then defines repetitions of that wafer.



Format An alternate format for the TABLE statement is:

```
TABLE name ['table title'] :
HEADING expression,
STUB    expression,
[WAFER   expression;]
```

This format allows expressions to be specified in any order if each expression is identified by the word HEADING, STUB, or WAFER.

The TABLE statement must end with a semicolon. A semicolon or comma is required between expressions. Each TABLE statement must have at least a stub and a heading expression.

To visualize the description of the TABLE statement, consider the following table which counts individuals according to their age and income category:

TABLE A1: STUB AGE, HEADING INCOME;

	Income		
	1	2	3
Age 1	*X	X	X
Age 2	X	X	X
Age 3	X	X	X

* X shows placement of tabulated data.

Assume that AGE is coded by age group as:

AGE = 1 for ages 16-29
AGE = 2 for ages 30-49
AGE = 3 for ages 50+

and INCOME is coded by income group as:

INCOME = 1 for incomes 0-$9,999
INCOME = 2 for incomes $10,000-$19,999
INCOME = 3 for incomes $20,000 and over

The table structure is defined by the AGE code (stub expression) and INCOME code (heading expression) to give a count of occurrences for all combinations of AGE and INCOME code. In this case, the stub expression and heading expression produce nine combinations of values. Each of these combinations defines a table cell that contains one item of information. The table in this example consists of only one wafer.

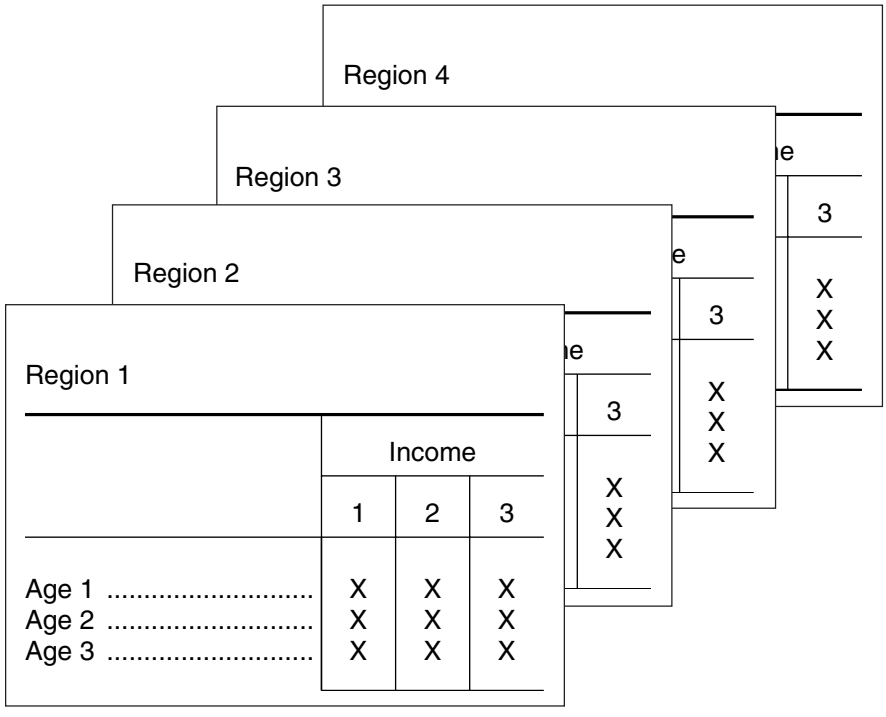
Now we will add a wafer expression to the TABLE statement:

TABLE A2: REGION, AGE, INCOME;

where REGION is coded as:

REGION = 1 for NE
REGION = 2 for NW
REGION = 3 for SE
REGION = 4 for SW

The addition of this expression will cause the AGE, INCOME wafer shown as TABLE A1 to be generated for each of the four REGION codes. Wafers for each REGION will begin on a new page as shown below:



In this example, each combination of AGE code and INCOME code is also combined with each REGION code.

If a particular wafer contains no data, then that wafer will not be printed.

In the examples of table A1 and table A2, each expression in the TABLE statement consisted of only one variable name. However, many tabulations will require a more complex type of expression. To satisfy this need, two operators are provided: a nesting operator called BY and a concatenate

operator called THEN. These operators can be used singly or in combination with any number of variables to create an expression of the TABLE statement.

Note

The expression "**nested with**" is used frequently in this manual. When we say that one variable or a collection of variables is nested with another, we mean that they are "crossed" for tabulation purposes. All variables in a table heading are nested with all variables in the table stub. If there is a wafer expression in the TABLE statement, all variables in the wafer are nested with all variables in the stub and heading. In addition, variables can be nested within a heading, stub or wafer using the BY operator as described in the next section.

The Nesting Operator: BY

Placement of the nesting operator between two variables causes their values to be paired in all possible combinations. Using the AGE and INCOME variables of the previous examples to create a "nested" heading expression, we would get AGE BY INCOME. In table form, the result of the nesting would be:

Age 1			Age 2			Age 3		
Income 1	Income 2	Income 3	Income 1	Income 2	Income 3	Income 1	Income 2	Income 3
X	X	X	X	X	X	X	X	X

Each value for AGE is paired with each INCOME value. This nesting operation causes the same action as the comma which separates expressions in the TABLE statement. The nesting operator, however, is used to define a structure within an expression of the table definition. Each variable used with the nesting operator will define a level of nesting. Thus the number of cells defined can be calculated by multiplying the number of values for each variable.

There is no limit to the number of levels allowed within one expression. For example, another level could be added to the AGE and INCOME expression for a total of three levels in the following TABLE statement which shows only a heading expression:

```
TABLE HEADER:  SEX BY AGE BY INCOME;
```

where SEX has possible values of 1 and 2. This expression would define the heading structure:

Sex 1									Sex 2								
Age 1			Age 2			Age 3			Age 1			Age 2			Age 3		
Income			Income			Income			Income			Income			Income		
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

This heading expression defines the same information as would the TABLE statement:

```
TABLE SAME:  SEX, AGE, INCOME;
```

The basic difference is in the structure of the output. The first TABLE statement defines 18 items of information arrayed in one line whereas the second TABLE statement defines the same 18 items of information arrayed as two wafers, each with three rows and three columns.

The Concatenate Operator: THEN

Placement of the concatenate operator between two variables causes the values for the two variables to be displayed in sequence. In other words, all values of the first variable are followed by all values of the second. For illustration, consider the heading expression:

```
AGE THEN SEX
```

where AGE and SEX have the values used in previous examples. The expression defines a heading structured as follows:

Age			Sex	
1	2	3	1	2
X	X	X	X	X

In this case, the occurrence of an AGE-SEX pair of values in an input record does not result in the tabulation of occurrences of the combination of values as in the nesting expression, AGE BY SEX. Instead it results in a separate tabulation for each variable.

You may find it useful to think of the concatenate operator as defining an additional tabulation while the nesting operator defines an additional level of the same tabulation. When two variables are joined by the concatenate operator, each value of each variable defines a cell. Thus the number of cells defined can be calculated by adding the number of values for each variable.

Combining the Nesting and Concatenate Operators

In the initial description of the nest and concatenate operators, we said that these operators could be used in combination with any number of variables in any expression of the TABLE statement. It is mainly this combination of operations that allows the specification of the basic structure of a table in one concise and clear statement.

When the two operators are used in combination within an expression, nesting takes precedence over the concatenate operation. If a different order of action is desired, parentheses can be used to specify a different precedence. To illustrate this ordering of operations, some examples will be shown of various combinations of the same variables in a heading expression.

1. AGE THEN INCOME BY SEX

Age 1	Age 2	Age 3	Income 1		Income 2		Income 3	
			Sex		Sex		Sex	
			1	2	1	2	1	2
X	X	X	X	X	X	X	X	X

In example (1), since nesting takes precedence, SEX is nested with INCOME, and the result of that nesting operation is then concatenated with AGE.

Now suppose that we wish to have SEX nested under both AGE and INCOME. We can use the same variables in the same order, but we need parentheses to cause the concatenate operator to be applied first.

2. (AGE THEN INCOME) BY SEX

Age 1		Age 2		Age 3		Income 1		Income 2		Income 3	
Sex 1	Sex 2	Sex 1	Sex 2	Sex 1	Sex 2	Sex 1	Sex 2	Sex 1	Sex 2	Sex 1	Sex 2
X	X	X	X	X	X	X	X	X	X	X	X

We can define two more headings using the same three variables but reversing the placement of the operators. These examples do not add to the concepts already discussed, but they do provide an idea of the variety of table structures that can be defined using only three variables with the combination of operators.

Compare the next heading with example 1.

3. AGE BY INCOME THEN SEX

Age 1			Age 2			Age 3			Sex	
Income			Income			Income			1	2
1	2	3	1	2	3	1	2	3		
X	X	X	X	X	X	X	X	X	X	X

Compare the following heading with example 2.

4. AGE BY (INCOME THEN SEX)

Age 1					Age 2					Age 3				
Income			Sex		Income			Sex		Income			Sex	
1	2	3	1	2	1	2	3	1	2	1	2	3	1	2
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

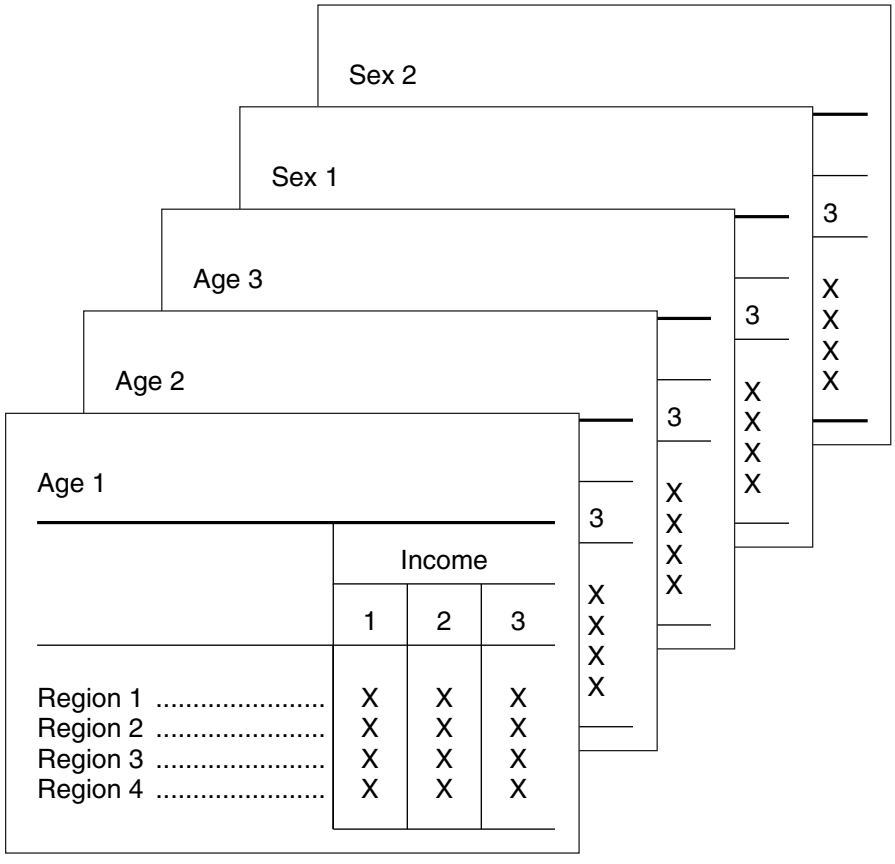
Before going on to new concepts, we look at some examples which show the interaction of ideas discussed up to this point. The following TABLE statement shows how BY and THEN can be used in both the stub and the heading.

TABLE B: REGION BY YEAR, AGE BY (SEX THEN INCOME);

	Age 1					Age 2					Age 3				
	Sex		Income			Sex		Income			Sex		Income		
	1	2	1	2	3	1	2	1	2	3	1	2	1	2	3
Region 1															
Year 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Year 2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Region 2															
Year 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Year 2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Region 3															
Year 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Year 2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Region 4															
Year 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Year 2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

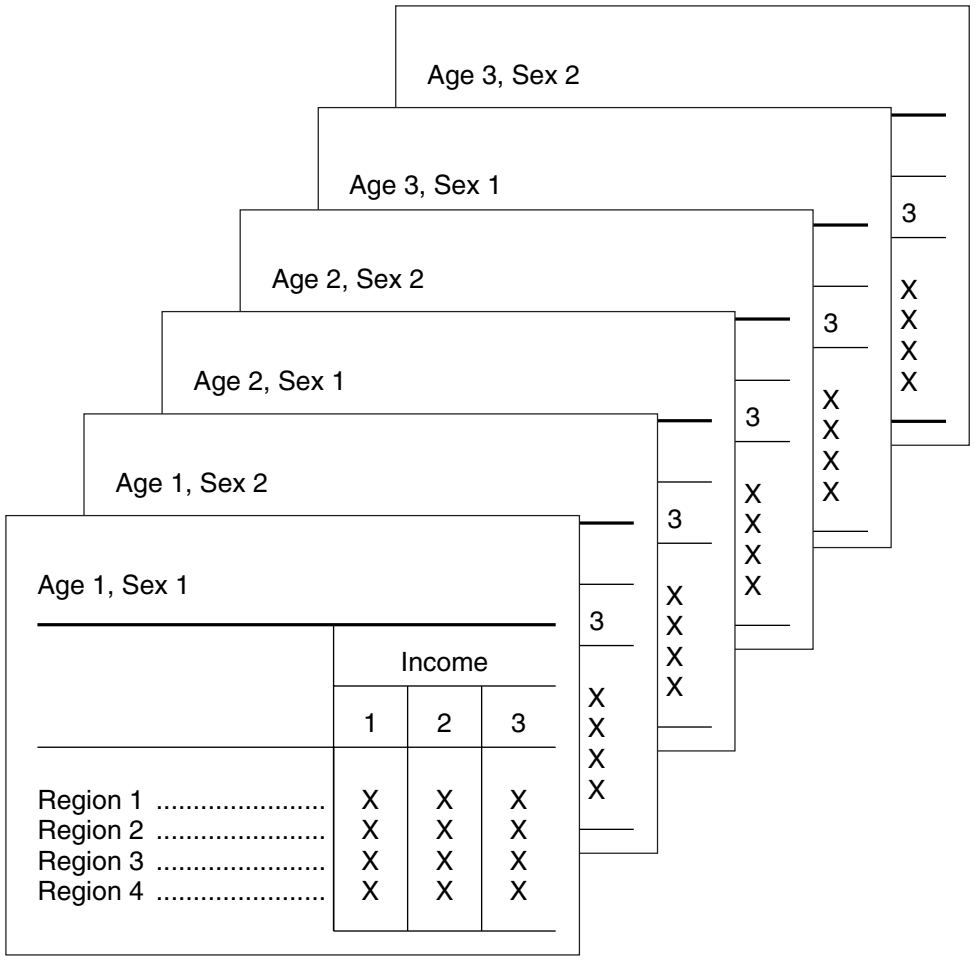
The next examples illustrate the use of BY and THEN in the wafer expression.

1. TABLE WAFER1: AGE THEN SEX, REGION, INCOME;



We can see from this table that the use of THEN in the wafer expression causes the wafer to be repeated for each value of the first variable and then for each value of the second variable.

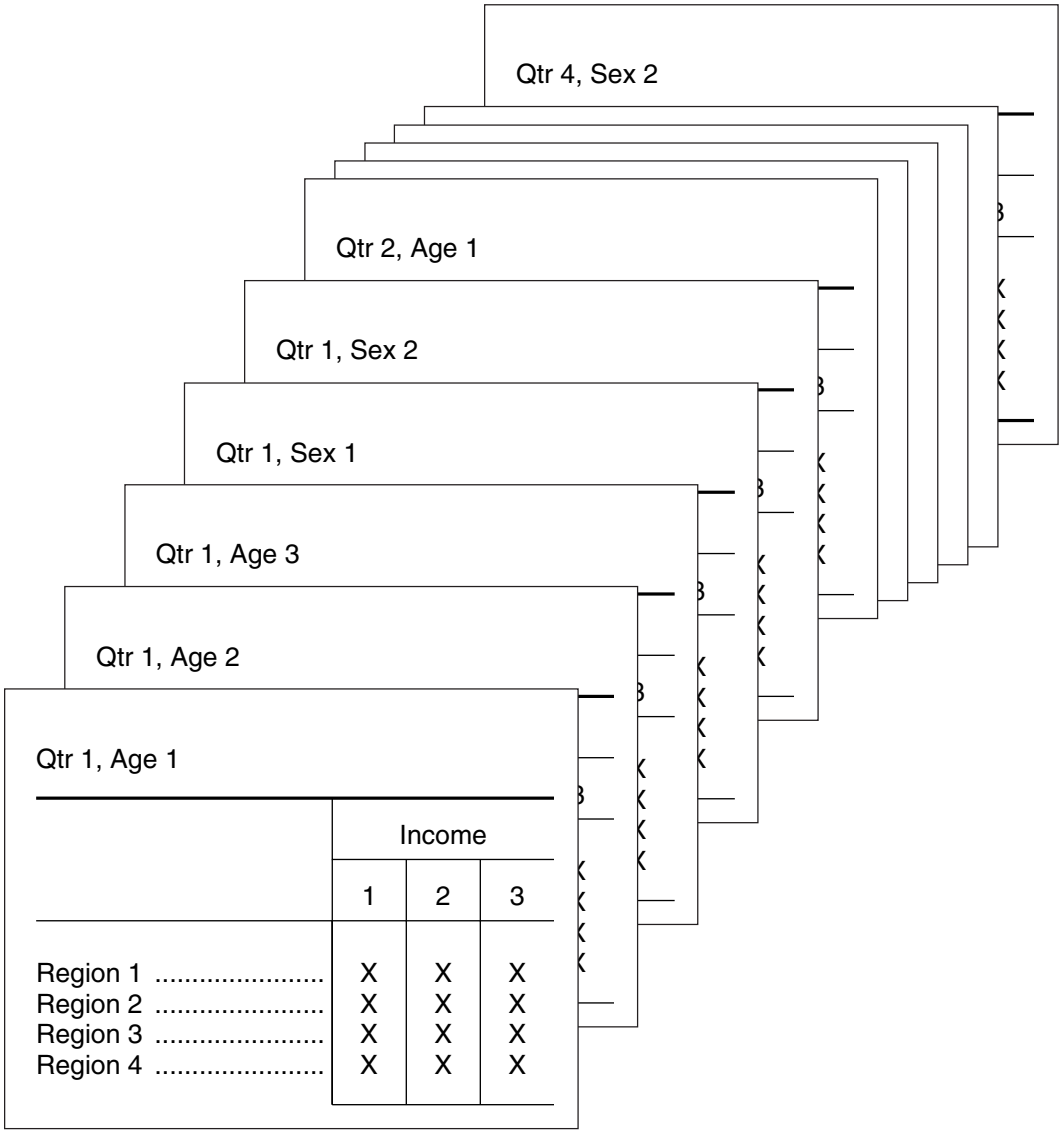
2. TABLE WAFER2: AGE BY SEX, REGION, INCOME;



The nesting of AGE BY SEX in the wafer expression causes repetitions of the wafer for each combination of the values of AGE and SEX.

3. TABLE WAFER3:
QTR BY (AGE THEN SEX), REGION, INCOME;

where QTR refers to the quarter of the year for which the data is collected.



We can see from this example that it is possible to define many repetitions of the same wafer. By simply nesting the variable QTR in front of the wafer expression shown in TABLE WAFER1, we have caused four times as many wafers to be generated and have further qualified the content of the output to provide quarterly tabulations.

CONTROL and OBSERVATION Variables

So far we have concentrated on the structure of tables. The cells of the examples would have contained only simple counts. The TABLE statement allows aggregation of other values within the cells. To describe this feature of TPL TABLES, we must distinguish between variables which control the table structure and those which specify the cell content.

When a variable is described in a codebook, it can be assigned either a CONTROL or an OBSERVATION attribute. This attribute determines the treatment of the variable in a TABLE statement.

CONTROL variables are classifying variables such as REGION, AGE or SEX. The values of a control variable are treated like codes, where each code value represents a classification for which we want to produce summaries in a table. The values can contain letters, numbers and other characters.

Control variables control a tabulation by providing the classifications for each table cell. If only control variables are used in a TABLE statement, a value of 1 will be added to the cell for each record that matches the cell classification. To tabulate values other than 1, we can add OBSERVATION variables.

OBSERVATION variables contain numeric values that can be added to table cells. Typical examples of observation variables are income, salary and number of persons in a family.

Most variables can logically be used in only one of the two ways, either OBSERVATION or CONTROL, but some variables in a data file can be reasonably used in either way. For example, a variable giving the number of persons for each family could be used as a control variable when each family is to be categorized by the number of persons in the family. In other cases, we might wish to actually tabulate the number of persons.

You can assign a dual usage to this type of variable by giving it two names when you describe it in the codebook. For example, you could have an observation variable called PERSONS_IN_FAMILY and redefine it as a control variable called PERSONS. The treatment of the variable in a TABLE statement would depend on which name you used.

Alternatively, the variable can be described as observation in the codebook and then reclassified as a control variable with a DEFINE statement. This approach is especially useful for variables that have a large number of

numeric values out of which only a few are to be selected or a few large groupings are to be made; for example, industry or commodity codes.

Adding Observation Variables to TABLE Statements

If no observation variables are used in a TABLE statement, the table cells will contain only simple counts. If an observation variable is nested into the table, values for that variable will be added into the table cells in place of counts.

Suppose that we have a file of information on retail stores, where each record contains information about one store: the region where the store is located (REGION), the number of employees (EMPLOYEES), the amount of sales in dollars (SALES) and a code that identifies the store as a clothing store or a food store (TYPE).

We can count each type of store by region with a TABLE statement that contains no observation variables:

```
TABLE STORE_COUNT: HEADING TYPE, STUB REGION;
```

	Clothing	Food
Region		
1	6	10
2	8	12
3	5	9
4	9	14

If, instead, we want to tabulate sales for each store type and region, we can do so by simply nesting SALES into the table with the word BY.

```
TABLE SALES_SUMMARY:    HEADING  SALES BY TYPE,
                        STUB  REGION;
```

	Sales	
	Clothing	Food
Region		
1	653,978	246,865
2	727,234	365,898
3	432,154	497,953
4	842,656	522,444

If we were to nest SALES into the table stub instead of the heading, we would not change either the structure or content of the table. The table would be exactly the same except that the label for SALES would be printed in the stub rather than the heading.

We can use more than one observation variable in the same table. For example, to get a tabulation of both sales and employees in the same table, we could specify:

```
TABLE SALES_AND_EMPLOYEES:
        STUB  REGION,
        HEADING  TYPE BY (SALES THEN EMPLOYEES);
```

	Clothing		Food	
	Sales	Employees	Sales	Employees
Region				
1	653,978	104	246,865	98
2	727,234	86	365,898	83
3	432,154	93	497,953	45
4	842,656	77	522,444	123

We could get alternate rows of sales and employees by requesting them in the table stub instead of the heading. Likewise, we could ask for alternate wafers of sales and employees:

```
TABLE ALTERNATE_WAFERS:
        WAFER  SALES THEN EMPLOYEES,
        STUB  REGION,
        HEADING  TYPE;
```

Counts can be combined with tabulations of other observation variables in the same table. For example:

```
TABLE COUNT_AND_SALES:
        HEADING  TYPE THEN SALES BY TYPE,
        STUB  REGION;
```

	Clothing	Food	Sales	
			Clothing	Food
Region				
1	6	10	653,978	246,865
2	8	12	727,234	365,898
3	5	9	432,154	497,953
4	9	14	842,656	522,444

The first two columns represent counts of each type of store in each region. The second two columns represent tabulations of sales amounts for each type of store in each region. See also the section on record names and COUNT for other examples showing the combination of counts with other observations.

The only restriction on the use of observation variables in the TABLE statement is that if they are nested with each other, they cannot produce meaningful results. If some part of a table contains nested observations, no tabulations will be done for that part of the table. The cells will be empty and a dash will be printed in each cell.

If an entire table statement is specified with nested observation variables, no table can be produced for the statement. No table would be produced for the following statement:

```
TABLE INVALID:
        HEADING  SALES BY REGION,
        STUB  EMPLOYEES BY TYPE;
```

The stub specifies SALES for all cells of the table and the heading specifies TYPE for all cells, but a table cell can contain only one value.

The following rules can be used as guidelines for TABLE statements in which at least one observation variable is explicitly named.

1. Observation variables are normally used in only one expression of the same TABLE statement: wafer, stub or heading. If they are used in more than one expression, there is a conflict of cell content and no data will be printed for the cells where the observations intersect.

The remaining rules apply to the use of observation variables in one expression.

2. Observation variables can be nested with control variables using the word BY, but they should not be nested with other observation variables.

3. Observation variables can be joined with any number of control variables or other observation variables using the word THEN.

Using Record Names and COUNT

The record name is used in the codebook to describe a collection of control and observation variables which make up a record in the data file. The record name is an implied observation variable with a value of 1. It can be used just the same as any other observation variable.

TPL TABLES also has a special built-in observation variable called COUNT that can be used like any other observation variable. In a flat (non-hierarchical) data file, COUNT is equivalent to the record name. For hierarchical files, the meaning of COUNT depends on whether or not the codebook contains repeating group variables. See the chapters on hierarchies and repeating groups for details.

The examples in this section assume a non-hierarchical data file. They show the record name RETAIL_STORES being used in the TABLE statement. In all cases, if we substituted the word COUNT for RETAIL_STORES, the tables would be the same except for the label.

Suppose that we would like to see a table with alternate columns of counts and tabulations of the observation EMPLOYEES, where each record of the input file represents one retail store. If the data record is named RETAIL_ STORES in the codebook, we can specify:

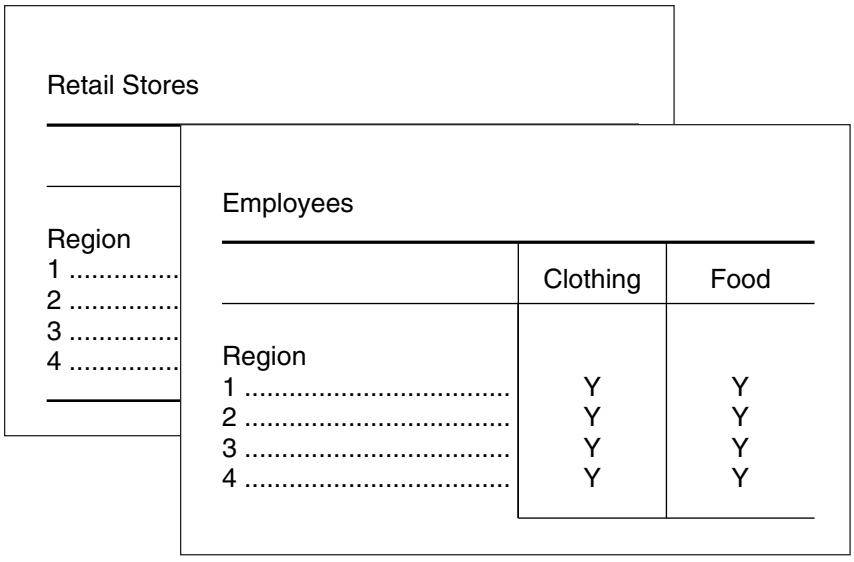
```
TABLE RETAIL:
     STUB  REGION,
HEADING  TYPE BY (EMPLOYEES THEN RETAIL_STORES);
```

	Clothing		Food	
	Employees	Retail Stores	Employees	Retail Stores
Region				
1	(1)	(2)	Y	X
2	Y	X	Y	X
3	Y	X	Y	X
4	Y	X	Y	X

where TYPE is a control variable with a code for each type of store. The cell numbered (1) represents the sum of the number of employees for each clothing store in REGION 1. The cell numbered (2) represents a count of the records with that combination of characteristics, in other words, the number of stores, since the data file contains one record for each store.

Now suppose that the same information content is desired, but that we would like to see one wafer for EMPLOYEES and another for RETAIL_ STORE count rather than alternating the two in the heading. We could then specify:

```
TABLE RETAIL2:
        WAFER  EMPLOYEES THEN RETAIL_STORES,
        HEADING  TYPE,
        STUB  REGION;
```



One further example illustrates the explicit use of both EMPLOYEES and RETAIL_STORES in the stub expression:

```
TABLE RETAIL3:
        HEADING  TYPE,
STUB  REGION BY (EMPLOYEES THEN RETAIL_STORES);
```

	Clothing	Food
Region 1		
Employees	Y	Y
Retail Stores	X	X
Region 2		
Employees	Y	Y
Retail Stores	X	X

Weight Variables

A common need in statistical processing is to weight various observed values in a data record which represent a sampling. Typically, each processing unit contains a weight to be applied during tabulation, so that the final table values represent a larger universe.

The simplest example of weighting gives weighted frequency counts. The weight is simply an observation variable like any other. It can be nested in the TABLE statement so that the weight is tabulated instead of the default observation variable which has a value of one. In the following example, the weight variable WGT is nested with all cells of the table and tabulated for each cell.

```
TABLE W1 'Weighted tabulation of population' :
        HEADING  WGT BY REGION;
        STUB  MARITAL_STATUS BY EDUCATION;
```

To obtain a weighted tabulation other than a weighted frequency count, a COMPUTE statement can be used. See the "Compute" chapter for details. See also the WEIGHTING statement to create a variable that contains multipliers for use in a TABLE statement. It is especially useful in tables where there are many observation variables that need to be multiplied by one or more weights or other variables within the same table.

The TOTAL Control Variable

Many tables require that row, column, or wafer totals be displayed. For example, following the count of persons in each AGE classification we may wish to see the total count of persons. The built-in variable TOTAL provides a convenient way to get these totals.

Note that a variable created with the LABEL statement will give exactly the same result as the built-in variable TOTAL but with a label of your choice. If you understand the concepts described here for TOTAL, you can apply them with LABEL variables as described in the "Labels" chapter.

TOTAL can be thought of as a single-valued control variable that is added to each data record by TPL TABLES. Since TOTAL is a single-valued control variable, all records are included in the TOTAL classification. TOTAL can be used in the wafer, stub and heading expressions, and in any position within these expressions. The print label for TOTAL is "Total".

Consider these examples:

TABLE T1: TOTAL, TOTAL;

	Total
Total	500

Since TOTAL has the same value for each record, all records fit in the TOTAL category. The single table cell produced is the count of records read, in this case 500.

TABLE T2: TOTAL, TOTAL THEN SEX;

	Total	Sex	
		Male	Female
Total	500	343	157

SEX is a control variable with two classifications concatenated with TOTAL. Each record will contribute a count of 1 into both the TOTAL category and one of the SEX categories, so that the first column will equal the sum of columns two and three.

TABLE T3: TOTAL, INCOME THEN TOTAL;

	Income	Total
Total	1,813,380	500

Since INCOME is an observation variable concatenated with the control variable TOTAL, the first column is a tabulation of INCOME values for all records, while the second column is the count of records.

If TOTAL is simply nested with one other variable within an expression of the TABLE statement, it has no effect on either the structure or content of the table (except to cause an additional label to be printed). In the following heading expression, TOTAL is redundant since the categories of AGE are not further qualified by TOTAL.

```
TOTAL BY AGE THEN SEX
```

Total			Sex	
Age1	Age 2	Age 3	Male	Female

However, when used in conjunction with the concatenate operator or with the nest and concatenate operators together, it can be very useful. Consider another heading expression where we use the observation variable EARNINGS and the control variables AGE and SEX:

```
EARNINGS BY AGE BY (TOTAL THEN SEX)
```

EARNINGS								
Age1			Age 2			Age 3		
Total	Sex		Total	Sex		Total	Sex	
	Male	Female		Male	Female		Male	Female

Each TOTAL column will contain the total EARNINGS for each AGE group. Each record will contribute an EARNINGS amount to a TOTAL column depending on the AGE code. Since TOTAL is concatenated with SEX, EARNINGS from each record will also be added to one of the SEX classifications. As a result, for each classification of AGE, the TOTAL column will equal the sum of earnings for that AGE, distributed among both sexes.

If TOTAL is nested under one or more variables, it provides totals for the categories above it. If it is only used alone or concatenated with other variables, it counts records. TOTAL is independent of any variable it is concatenated with. Thus TOTAL will not always equal the sum of variables it is concatenated with.

Some additional examples of TOTAL in TABLE statements follow. RENT and INCOME are observation variables in the examples. In the first two examples, the totals are the sums of the other rows and columns. In the second two examples, they are not.

TABLE A: TOTAL THEN AGE, TOTAL THEN REGION;

	Total	Region 1	Region 2
Total	500	127	373
Age 1	113	37	76
Age 2	194	42	152
Age 3	193	48	145

TABLE B: RENT BY (TOTAL THEN AGE), TOTAL THEN REGION;

	Total	Region 1	Region 2
Rent			
Total	666,625	106,888	559,737
Age 1	147,144	35,902	111,242
Age 2	258,159	31,503	226,656
Age 3	261,322	39,483	221,839

In tables A and B above, TOTAL happens to equal the sum of the variable entries concatenated with it, both vertically and horizontally. In Table A, this is because each record will be counted into the upper left cell (TOTAL, TOTAL) as well as into one of the Region categories and into one of the Age categories.

In Table B, each RENT value will be aggregated into four cells: two cells in the TOTAL row and two cells in one of the AGE rows.

TABLE C: AGE, RENT THEN INCOME THEN TOTAL;

	Rent	Income	Total
Age 1	147,144	407,783	113
Age 2	258,159	841,557	194
Age 3	261,322	564,040	193

TABLE D: REGION THEN TOTAL, RENT THEN AGE THEN TOTAL;

	Rent	Age 1	Age 2	Age 3	Total
Region					
1	45,948	19	19	21	59
2	59,125	18	21	24	63
3	219,884	41	73	64	178
4	341,668	35	81	84	200
Total	666,625	113	194	193	500

In table C, the TOTAL column does not contain sums of the values in the other columns. It counts persons in each AGE category.

In table D, the TOTAL row is the sum of the rows above it. However, the TOTAL column contains the sum of only the AGE columns, because the tabulations of RENT, AGE and TOTAL are done independently.

Interaction of TOTAL and DEFINE

Some other aspects of TOTAL are worth noting here, although they may not be meaningful until you have read about the DEFINE statement:

First, if you define new categories for a control variable such as REGION, so that the categories either do not include all of the original REGION values or use some values in more than one category, TOTAL will not equal the sum of the new categories.

Second, you can group values into totals and subtotals using a DEFINE statement. For example, REGION values could be tabulated for individual regions, for groups of two regions and for the total of all regions with the DEFINE statement:

```
DEFINE REGION_GROUPS ON REGION;
        'Northeast'       IF  1;
        'Northwest'       IF  2;
        'North Total'     IF  1:2;
        'Southeast'       IF  3;
        'Southwest'       IF  4;
        'South Total'     IF  3:4;
        'All Regions'     IF  ALL;
```

Third, you can define other variables that behave in the same way as TOTAL when used in TABLE statements. This technique is sometimes used to get a total variable with a label other than "Total". To use this technique, you define a new variable on some other variable that already exists and include all values. The following DEFINE statement creates a total variable with the label "All Regions".

```
DEFINE ANOTHER_TOTAL ON REGION;
        'All Regions'  IF  ALL;
```

You can also use this technique to suppress the TOTAL label in the table stub, when you want the TOTAL label to be replaced by the label for the variable at the next higher level of nesting. Suppose the stub expression is CITY BY (TOTAL THEN SEX), where CITY and SEX are both control variables. The normal format for the beginning of the stub is as follows:

```
BOSTON
        TOTAL......(total values)
        MALE.......(sex values)
        FEMALE....(sex values)
```

A DEFINE statement can be used to create a total variable with no label (a null label):

```
DEFINE NEW_TOTAL ON SEX;
        "  IF  ALL;
```

If we then describe the stub as CITY BY (NEW_TOTAL THEN SEX), the line for NEW_TOTAL will not have a label and the city label will "collapse" down to the total line, giving:

```
BOSTON.........(total values)
        MALE.........(sex values)
        FEMALE.... (sex values)
```

What is a Cross Tabulation?

A Single **Cross Tabulation** is all of the cells of a table determined by the same (instances of) control and observation variables.

One Cross Tabulation

	Race			
	White		Black	
	Age			
	Young	Old	Young	Old
Average Income	21,572	33,496	14,149	22,525

In this table there is one cross tabulation because all of the cells are determined by the control variables RACE and AGE and the observartion variable AVERAGE INCOME.

Two Cross Tabulations

	Race							
	White		Black		White		Black	
	Age				Sex of Householder			
	Young	Old	Young	Old	Male	Female	Male	Female
Average Income	21,572	33,496	14,149	22,525	37,146	21,488	27,420	16,144

This table has two cross tabulations. The shaded cells are determined by RACE, AGE, and AVERAGE INCOME while the non-shaded cells are determined by RACE, SEX, and AVERAGE INCOME.

Two Cross Tabulations Interleaved

	Race							
	White				Black			
	Age		Sex of Householder		Age		Sex of Householder	
	Young	Old	Male	Female	Young	Old	Male	Female
Average Income	21,572	33,496	37,146	21,488	14,149	22,525	27,420	16,144

This table also has two cross tabulations but they are interleaved.

Most statistical packages, tabling systems and management information systems work with a single data cube or cross tabulation. With the use of THEN, TPL TABLES can produce a table with multiple cross tabulations.

Some TPL TABLES functions such as certain statistical tests are only meaningful for single cross tablulations. In these cases, TPL TABLES allows you to perform the function on a single cross tabulation within the table.

Table Formatting

TPL TABLES automatically formats your tables using default settings for sizes such as column and stub width. Print labels are taken from the codebook or from other TPL statements that create new variables in a table request. The following chapters contain details on all aspects of table format.

Automatic Formatting. This chapter describes default settings for the overall table format.

Format. Many aspects of table format can be changed using FORMAT statements. This chapter tells how to use FORMAT statements and describes each one in detail. Note that a special FORMAT statement called DATA TABLES can be used to produce a data file rather than a table formatted for printing.

Labels. The simplest print label is a string of text characters enclosed in quotes. The text can include spaces, upper and lower case letters, and special characters. The many other formatting options for print labels are described in the Labels chapter.

Masks. This chapter describes the options available for controlling the format of the numbers printed in table cells.

Footnotes. Footnotes can be printed automatically at the end of a table. The footnote chapter tells how to create and reference footnotes.

Color and Grey. This chapter tells you how to use color for labels, masks and lines. In addition, you can request color or grey shading for an entire table or for selected parts of a table. You need a color printer to print color, but you can use grey shading effectively on a monochrome printer.

CHAPTER 5

Data

ORGANIZATION OF INPUT DATA FILES

TPL TABLES can process many different kinds of data files. The data can be exported from a data base or spread sheet, downloaded from a main-frame, entered using a data entry system or editor, or prepared using a custom program. TPL TABLES does not "import" your data or change it in any way. Before you can use your data with TPL TABLES, you need to prepare a data description, called a codebook, that TPL TABLES can use to find the data items that you want to use in your tables.

Codebooks are discussed in the next chapter. This chapter describes the types of data that TPL TABLES can read, the treatment of data errors and the way in which multiple data files can be used together.

If you have the TPL-SQL database interface, TPL TABLES can also read data directly from a database. For more information, see the TPL-SQL chapter or TPL Help for the Windows version of TPL TABLES.

Types of Files and Data

Data Records

Data files can be either fixed width or delimited.

In **fixed width** files, the records must all be of the same length, except in the case where more than one type of record describes a unit of processing. In this case, the records of different types can be of different lengths, but all records of a particular type must be of the same length. The data fields must be in columns of fixed width.

Delimited files can have records of different widths. The records have values separated by commas or some other delimiter character such as tab or semicolon. This means that fields are identified by their order in a row of data and that data values for a particular field may have different widths. The first row of the file can optionally have names for each field. Delimited files that have comma as the delimiter are sometimes called CSV (comma separated values) files.

Each record can be thousands of bytes in length. See the Appendix called "Limits" for current information on the maximum record size.

Flat File Structure

Fixed width data files can be organized in two ways. They can be single level (flat) or multi-level (hierarchical) files. The single level file is the simplest form of file organization. It consists of a fixed number of records which represent a processing unit. In the most common case, there is just one type of record, and each record represents a processing unit. Delimited files are always single level (flat) files.

Hierarchical File Structure

Several data records in succession may be required to describe a processing unit. These records may be related in such a way that for records of a given type, a variable number of a more detailed type follow. Consider a file of family survey information. Each family record may contain general information such as city, employment status, income, and number of children. Following each family record there could be a record for each member of the family. The family member records could each contain information on occupation, salary, sex, and age for the family member. An indefinite number of member records may follow each family record, but there must be at least one.

In TPL TABLES, one record from each level of detail is read, starting with the most general, and together they form a processing unit. This type of relationship between records is known as a hierarchical structure. Records at any level of detail may be followed by subordinate records of greater detail. Records at any level need not repeat information already present at higher levels.

Processing hierarchical files with TPL TABLES is described in detail in a later chapter.

Data Types

Control and **Char** variable values must be stored as characters (ASCII). **Observation** variables can be character (ASCII), binary, or machine-generated floating point. Binary fields can be 1, 2, or 4 bytes wide. Floating point fields can be 4 or 8 bytes wide (single or double precision). TPL TABLES will process floating point data according to the standard floating point representation for your computer. Since floating point data representations vary on different types of computers, floating point data to be used with TPL TABLES should be generated on the type of computer you are using to do your TPL TABLES work. In delimited files, all values must be ASCII text characters. Optionally, the values can be enclosed in single or double quotes.

Treatment of Data Errors

TPL TABLES has been designed to work with files that do not have data errors. When an error is detected in a data record, the default treatment is to report the error and discard the entire data record containing the error. An error message will display the record number, the variable name and the invalid value. If the data file is hierarchical, then any records below the record containing the error are also discarded.

In recognition of the fact that real data files often do have errors, several options are available to help you control the treatment of errors. These options depend on whether the data field containing the error is a control variable or an observation variable.

CONTROL Variables

If TPL TABLES finds a control variable value that has not been listed in the codebook, an error is reported and the record is discarded as described above.

Note In the Windows version of TPL TABLES, you can prepare the codebook interactively. For the UNIX version, the *tpl conditions* program can assist you in preparing the codebook. Both of these tools read the data and create or update the lists of condition values so that you will not get data errors for control variables.

If you do not want TPL TABLES to discard automatically data records with control variable errors, you should add the error values to the codebook list of values for the variable. You can then use SELECT, DEFINE

or Conditional Compute statements in your table requests to control how the error values should be treated. Please refer to the chapters on SELECT, DEFINE and COMPUTE statements for details.

In particular, note the COPY option of the DEFINE statement. Using a DEFINE statement with COPY, you can select the valid values for a control variable, copy their labels from the codebook, and eliminate invalid values or group them into a separate category without discarding data records.

OBSERVATION Variables

Character (ASCII) Observations

For a character (ASCII) observation variable, TPL TABLES will detect and report as errors values that are completely blank and values that contain other non-numeric characters. Records containing data errors are discarded.

You can specify different treatment for errors when you describe variables in the codebook. Error values can be set to NULL, so that no records will be discarded because of errors. NULL values are not used in tabulations. Optionally, the error reporting can also be turned off.

An alternate option can be used for blank values. If you choose this option, blank values will be replaced with the value 0, and no errors will be reported for blanks.

These options are described in the codebook chapter.

Binary and Floating Point Observations

TPL TABLES cannot detect errors in binary or floating point observation variables, since any values are possibly valid. In some cases, a "floating point overflow" error may occur, but usually the only evidence of errors is unexpected numbers in the table output.

Using File Lists to Process Multiple Data Files and Merge Outputs

In some cases, the data you wish to process to form a set of tables may be contained in several data files. These data files may exist in one or more directories on your hard disk, on multiple CDs or diskettes, or perhaps even on different types of computers. In other cases the data files may

be so large that you do not wish to process all of the data at one time but you would like to have the option of combining the outputs from several smaller jobs to get tables that include all of the data.

TPL TABLES can handle all of these cases with its multifile and merge capabilities. The basic way these facilities are activated is to create a file list that contains the names of the data files to be combined, then to substitute the file list name for the data file name when running your TPL TABLES job.

We will discuss the file list first in the context of multifile inputs. Second, we will extend the idea of the file list to show how outputs from multiple jobs can be combined.

Processing Data from Multiple Files

If your data is located in several files, you can create a **file list** containing the names of the data files to be processed. Then, instead of giving TPL TABLES the name of a single data file, you can give it the name of the file list.

Windows. If you are running a Table Request from the Windows Run menu, enter the name of the file list in the **Data File** blank and check the box next to **File List?**

If you are running from a command in Start then Run, from the command line, from a .bat file, or from a TPL script, use the -l (**lower case** letter L) argument to provide the name of the file list. See **Scripts** in TPL Help for more information.

Examples

To run a table request using a command, if your file list is called **DATALIST**, TPL TABLES is installed in c:\qqq\table, the table request is c:\myjobs\my.req, and you want the output to go in TPL123, enter:

```
c:\qqq\table\wtpl.exe table -r c:\myjobs\my.req -l DATALIST -O 123
```

To run the same job from a script executed from a .bat file, if the script file is c:\myjobs\runtable.lst, the .bat file would contain:

```
c:\qqq\table\wtpl.exe -A c:\myjobs\runtable.lst
```

The script file would contain:

```
table -r c:\myjobs\my.req -l DATALIST -O 123
```

UNIX. When entering the data file name, a % symbol should be placed directly in front of the file list name to tell TPL TABLES that the file contains a list of names of data files rather than the data itself.

File names are "case sensitive". For example, the name **SURVEY.DAT** is different from the name **survey.dat**. **PAUSE** entries can be used before the names of files that you need to mount. Forward slashes (**/**) should be used in path names.

If the file list has a name that begins with a **%** symbol (we do not recommend this), you must precede the name with an additional **%** symbol. Otherwise, the file list will be used as the data file.

Windows and UNIX. Optional **PAUSE** entries can be included in the list to allow you to insert CDs or diskettes. A **PAUSE** entry applies to the file name that follows it.

TPL TABLES will process each of the specified files during its CELLGEN step and combine the data to produce a set of tables identical to the those that would be produced if all of the data were in a single file.

If there is a problem such as a missing file or a mistyped file name in the list, TPL TABLES will not abruptly end your job but, instead, will prompt you for assistance or corrections.

The data files can be read from any available device. For example, for one job, you could have sections of your data on hard disk, CD, or diskette. There are two important restrictions. First, all of the data files must be in exactly the same format. Second, each file must end with a complete data record. In the case of hierarchical files, a hierarchical unit cannot span across data files.

Example

Assume that we are running on a Windows PC with survey data for last year saved on hard disk in the subdirectory **C:\LASTYEAR**, the first three months of the same survey in the current working directory on the hard disk, and additional smaller files with the last two months of data on CD. We can process all four files with the following data

```
C:\LASTYEAR\SURVEY.DAT
QUARTER1.DAT
D:\APRIL.DAT
D:\MAY.DAT
```

There is no required format for the file list, except that the entries must be separated by at least one blank, and extra characters, such as semicolons, are not allowed.

Notice that in the above example, two of the files are on CD drive D. If these files are on different CDs, it will be necessary to stop TPL TABLES processing to allow you to change CDs. You can accomplish this by inserting **PAUSE** before the file name.

```
C:\LASTYEAR\SURVEY.DAT
QUARTER1.DAT
PAUSE
D:\APRIL.DAT
PAUSE
D:\MAY.DAT
```

When the **PAUSE** is encountered, TPL TABLES will stop and prompt you to perform whatever action is necessary to make the next file available. **PAUSE** can be inserted in front of as many file names as desired.

Treatment of Data Errors

Counts of data errors are restarted for each file, and the first 50 errors are shown for each file. Thus, if some files are correct and others have data errors, you will be able to tell which files have errors. A cumulative total is also reported for the entire job.

Merging Output from Multiple Runs to Create a Single Output

A file list can also be used if you want to process parts of your data at different times or on different computers and then merge the processed data into one set of tables. There are no restrictions on the tables that can be produced in this way. The contents of a single cell of the output table may contain contributions from all of the input files. However, all of the data should be processed using the same codebook and table request. If you attempt to combine outputs produced with different codebooks and table requests, we cannot predict the results.

The files that are actually merged are not the finished table files but rather the **cellfile** outputs created by the CELLGEN step of a TPL TABLES run. Normally, to avoid wasting disk space, this file is automatically deleted before the end of a TPL TABLES run since it is not needed for printing of output tables or RERUN jobs. However, if you wish to merge a table run with a future run, you must tell TPL TABLES to preserve the cellfile.

To preserve a cellfile, add the statement

```
RETAIN  CELLFILE;
```

to your profile (profile.tpl) or FORMAT request. If you frequently merge tables, we recommend that you put the **RETAIN CELLFILE;** statement in your profile so that you won't forget to insert it when you need it. The cellfile is about the size of a tables file. Since it is saved in the TPL subdirectory for the job, it will be deleted along with the other files if you delete its TPL subdirectory.

Suppose in our previous example for multifile processing, the data file **C:\LASTYEAR\SURVEY.DAT** was tabulated some time ago and **RETAIN CELLFILE;** was specified. If its TPL subdirectory was **TPL92** in the current directory, then there exists a file called **TPL92\cellfile**. You can now substitute **MERGE TPL92\cellfile** or **MERGE TPL92** or even just **MERGE 92** for **C:\LASTYEAR\SURVEY.DAT** in your file list. The new **DATALIST** file will contain:

```
MERGE  TPL92
QUARTER1.DAT
PAUSE
D:\APRIL.DAT
PAUSE
D:\MAY.DAT
```

Now, instead of reprocessing the entire **C:\LASTYEAR\SURVEY.DAT** file, the new job will quickly merge in the already processed data.

Note that both **PAUSE** and **MERGE** can precede a file name. Their order doesn't matter.

Suppose we have three large data files, **SURVEY1**, **SURVEY2** and **SURVEY3**. If we wish to process these separately and combine their outputs, there are several sequences we could use. One is to process each of them separately and then combine them in a fourth job using a file list. IF **SURVEY1**'s cellfile is in **TPL1**, **SURVEY2**'s cellfile in **TPL2**, and **SURVEY3**'s cellfile in **TPL3**, we can then run a job using the same codebook and table request that produced these cellfiles, but use a file list instead of a complete data file. The file list would be:

```
MERGE  TPL1
MERGE  TPL2
MERGE  TPL3
```

Another approach would be to accumulate the data as you proceed. First process the **SURVEY1** data file. Assume that the output of this job is retained in the subdirectory **TPL1**.

Next process **SURVEY2** and merge in **SURVEY1** results using the file list:

```
MERGE  TPL1
SURVEY2
```

Assuming the the merged output has been retained in the subdirectory **TPL2**, we can next process **SURVEY3** and merge in the combined results of the previous two jobs using the file list

```
MERGE  TPL2
SURVEY3
```

Note that we should not include **MERGE TPL1** in this file list since the **SURVEY1** data has already been merged into the cellfile stored in **TPL2**.

Cellfiles need not be left in their TPL subdirectories. They can be copied out and given unique names and then subdirectories can be deleted. Suppose our three cellfiles were created in separate jobs and then copied out of their subdirectories into files called **CELLS1**, **CELLS2** and **CELLS3**. These files could be combined in a final TPL TABLES job using the file list:

```
MERGE  CELLS1
MERGE  CELLS2
MERGE  CELLS3
```

Combining Cellfiles from Jobs Run on Different Types of Computers

The instructions for using the following vary slightly between Windows and UNIX systems. See the notes at the end of this section for information on the differences.

Cellfiles are not character files. Consequently, if you are saving cellfiles from jobs run on different types of computers, you may need to convert them to character files before you can move them from one type of computer to another. For example, a cellfile produced on a PC running Windows cannot be directly merged into a table job running under UNIX on a different computer. Likewise, cellfiles produced on two different types of UNIX computers may not be directly compatible. Two programs, **CEL2CHAR** and **CHAR2CEL** have been provided to allow cellfiles to be

merged on a computer that differs from the computer where the cellfiles were produced.

After the cellfile is produced on the first computer, run **CEL2CHAR** to convert the cellfile to a character file. Then copy the character file to the target computer using your standard character file transfer procedure. Run **CHAR2CEL** on the target computer to convert the file back into the appropriate cellfile format for that computer.

CEL2CHAR takes two arguments. The first is the name of the existing cellfile you wish to move. The second argument is the name of the character file you wish to create. This second file will be transferred to the target computer.

CHAR2CEL takes two arguments. The first is the name of the character version of the cellfile after it has been transferred to the target computer. The second argument is the new name for the cellfile. This second argument is the name that will appear in your file list when you merge the data.

Example Assume that we have run a job on the first computer and that the output, including the retained cellfile, is in the subdirectory TPL3135. To convert the cellfile from the job and store it in a file called CHARFIL1, type the following on the command line:

```
CEL2CHAR  TPL3135\CELLFILE  CHARFIL1    <enter>
```

CHARFIL1 can be transferred to a second computer and converted back to TPL TABLES format using the program **CHAR2CEL**. For this conversion, type:

```
CHAR2CEL  CHARFIL1  CELLFIL1    <enter>
```

Now the cellfile called **CELLFIL1** can be entered in a file list to be merged with other jobs on the second computer.

Note If the directory where TPL TABLES is installed is not in your path, you must provide the path information in the command. For example:

```
C:\QQQ\TABLE\CEL2CHAR  TPL3135\CELLFILE  CHARFIL1  <enter>
```

UNIX Note Both the cellfile and the program names should be entered with lower case letters. You can choose either upper or lower case letters for the other file names. Use forward slashes (/) in path names. Assuming we have chosen

lower case letters for file names, the above example when executed on a UNIX system would be as follows:

```
cel2char  TPL3135/cellfile  charfil1    <enter>

char2cel  charfil1  cellfil1    <enter>
```

Piping Data to TPL TABLES (UNIX only)

TPL TABLES running under UNIX supports standard piping of data into a request and also supports the more flexible named pipes. For complete instructions, see "Piping Data to TPL TABLES" in the appendix called "Run Instructions (UNIX)".

CHAPTER 6

Codebook

DESCRIBING AN INPUT DATA FILE

Introduction

A codebook describes a data file to be used with TPL TABLES. Information provided in a codebook includes names of data items, where they are located within a record, how many character positions each occupies within a record, and valid entries for control variables. Since TPL TABLES does not require that your data be in a particular format, it needs this information in order to find the data items that you wish to use in your tables.

The codebook is prepared and processed as a separate step before tables can be produced. This makes the description of data independent of the procedure which produces tables.

A codebook can be prepared with an editor and then submitted to TPL TABLES for processing. In the Windows version of TPL TABLES, you also have the option of preparing the codebook interactively. See Help in the Codebook Builder for instructions. For the UNIX version, the *tpl conditions* program can assist you in preparing the codebook. This program is described in an Appendix.

If you have the TPL-SQL database interface, TPL TABLES can also read data directly from a database. For more information about database codebooks, see the TPL-SQL chapter or TPL Help for the Windows version of TPL TABLES.

When a codebook is processed, it is checked for errors and a new file is created that contains the codebook information stored in a form that can be used by TPL TABLES to read your data. For your convenience, TPL TABLES prepares a codebook abstract when processing a codebook. The

abstract lists the data items in alphabetical order, along with characteristics such as type, size and location. Once a codebook has been processed, it can be used over and over to produce tables.

The version of the codebook that you prepare is called the codebook source. The version created by TPL TABLES during codebook processing is called the codebook object. The source is not changed during codebook processing. At any time, you can edit the source to make changes, then reprocess it to replace the previous codebook object. The TPL TABLES "Run Instructions" explain this procedure in greater detail.

General Format of the Codebook

The first entry within a codebook names the codebook. It is followed by entries defining all data items within each type of record. When you specify "USE codebookname;" as the first statement in a TPL table request, all data items within the codebook become available for use in following TPL statements.

Following is an example of a small codebook which describes a simple file with only one record type. Each record contains information about one family. The data items are described in the order that they appear in the record, and the parts of the record that are not to be used are described with FILLER entries.

```
BEGIN FAMILIES CODEBOOK
  FAMILY RECORD
       FILLER 7
       REGION  CONTROL 1
         (
              NORTHEAST          =     1
              NORTH_CENTRAL      =     2
              SOUTH              =     3
              WEST               =     4
         )
       LIVING_QRT  CONTROL 1
         (
              OWNED              =     1
              CONDOMINIUM        =     2
              RENTED             =     3
              UNKNOWN            =     ' '
         )
```

```
        MORTGAGE  CONTROL 1
          (
                MORTGAGED          =       'A'
                NOT_MORTGAGED      =       'B'
                NOT_AVAILABLE      =       'C'
          )
        HEADS_NAME  CHAR 30
        FILLER 20
        HEADS_CLASS_OF_WORK  CON 1
           (
                WHITE_COLLAR       =       1
                BLUE_COLLAR        =       2
                FARM_WORKERS       =       3
                SERVICE_WORKERS    =       4
                ARMED_SERVICES     =       5
                NOT_REPORTED       =       6
          )
        PERSONS_IN_FAMILY  OBS 2
        NO_EARNERS  OBS 2
        FILLER 1
        GROSS_INCOME_OF_HEAD  OBS 7
        GROSS_INCOME_OF_SPOUSE  OBS 7
END FAMILIES CODEBOOK
```

An Example Using Start Position

You can also describe data items by giving the START position and length. This technique would most often be used (1) in place of a FILLER entry for skipping over parts of a record or (2) in a REDEFINES entry where part of a record is to be used two different ways. Both of these are described in more detail in later sections of this chapter.

Following is a second version of the FAMILIES codebook in which all data items are described using START position. Note that you can use a mix of the two styles. For example, you might want to use START position only to skip over unused sections of the record. Comments show where this takes place in the version below.

```
BEGIN FAMILIES CODEBOOK
   FAMILY RECORD
/* first 7 positions are skipped; start at 8 */
         REGION  START 8 CONTROL 1
           (
                NORTHEAST          =      1
                NORTH_CENTRAL      =      2
                SOUTH              =      3
                WEST               =      4
           )
         LIVING_QRT  START 9 CONTROL 1
           (
                OWNED              =      1
                CONDOMINIUM        =      2
                RENTED             =      3
                UNKNOWN            =      ' '
           )
         MORTGAGE  START 10 CONTROL 1
           (
                MORTGAGED          =      'A'
                NOT_MORTGAGED      =      'B'
                NOT_AVAILABLE      =      'C'
           )
         HEADS_NAME  START 11 CHAR 30
/* next 20 positions are skipped; start at 61 */
         HEADS_CLASS_OF_WORK  START 61 CON 1
            (
                WHITE_COLLAR       =      1
                BLUE_COLLAR        =      2
                FARM_WORKERS       =      3
                SERVICE_WORKERS    =      4
                ARMED_SERVICES     =      5
                NOT_REPORTED       =      6
           )
         PERSONS_IN_FAMILY  START 62 OBS 2
         NO_EARNERS  START 64 OBS 2
/* next position is skipped; start at 67 */
         GROSS_INCOME_OF_HEAD  START 67 OBS 7
         GROSS_INCOME_OF_SPOUSE  START 74 OBS 7
END FAMILIES CODEBOOK
```

Note In all cases, the codebook must account for the entire length of the record. If you are not describing a data item that reaches the end of the record, you can describe the last portion of the record with a FILLER entry. See the section on FILLER for more information.

CODEBOOK ENTRIES

The BEGIN Entry

The first entry in a codebook gives the codebook a name. This name is referenced at the beginning of any TPL table request that uses the data being described. The format for the entry is:

Format

```
BEGIN codebookname [CODEBOOK]  [ASCII]
                               BINARY
```

where **codebookname** is a reference name you choose. The word CODEBOOK and the ASCII or BINARY specification are optional.

Example

```
BEGIN SURVEY CODEBOOK
```

Windows Note A codebook name can contain blanks. If it does, the name must be enclosed in ***double*** quotes.

Example

```
BEGIN "FIRE DISPATCHES" CODEBOOK
```

UNIX Note Codebook names cannot contain blanks.

ASCII is the default data format. If you do not include either ASCII or BINARY at the end of the BEGIN clause, TPL TABLES assumes that you are working with a standard ASCII data file. This is the usual format for data files created by PC software.

In an ASCII data file, all data is stored as ASCII characters. In addition, depending on the operating system you are using, each record contains either one or two special characters at the end of each record and a special end-of-file marker at the end of the entire file. On a Windows PC, each record ends with a carriage return/line feed combination; in UNIX, each record ends with a line feed character. TPL TABLES will automatically take these special characters into account so that you do not have to include them in your codebook record descriptions.

If all of the data in your file is stored as ASCII characters, but the records do not have the extra characters at the end, you must include the word BINARY at the end of the BEGIN clause. Files of this type are unusual but are sometimes created by custom programs. For example, a custom program written to transform a particular file from mainframe format to PC

format might not add the end-of-record characters when creating the file in PC format.

If you have any data in your file that is stored in binary or floating point format rather than ASCII characters, you must include the word BINARY at the end of the BEGIN clause. For example,

Example

```
BEGIN SURVEY CODEBOOK  BINARY
```

When a file is described as BINARY, the entire record, including any end-of-record characters, must be included in the codebook description of the data records.

Incomplete Hierarchy Entries

This optional specification follows the BEGIN .... CODEBOOK entry. It is relevant only in codebooks that describe hierarchical data files that may have missing lower level records. For complete details on the rules for processing incomplete hierarchies, please refer to the chapter on Hierarchical Files.

The two possible codebook entries are:

```
TABULATE INCOMPLETE HIERARCHIES = YES;  (NO is the default)
```

and

```
REPORT INCOMPLETE HIERARCHIES = NO;  (YES is the default)
```

These entries can be used to control the treatment of incomplete hierarchies. By choosing

```
TABULATE INCOMPLETE HIERARCHIES = YES;
```

you can tabulate data from higher level records even though records are missing at the lowest levels, as long as information from the missing records is not required to do the tabulation.

By default, incomplete hierarchies will be reported even if they are tabulated. To suppress incomplete hierarchy messages, use the statement

```
REPORT INCOMPLETE HIERARCHIES = NO;
```

The INCOMPLETE HIERARCHIES entries can also be included in a table request following the USE statement. An entry in the table request will override any conflicting entry in the codebook.

The RECORD Entry

One or more record descriptions follows the BEGIN .... CODEBOOK entry to describe each type of record in detail. In the most common type of data file, there is only one type of data record, so only a single record description is needed. A record description begins with a RECORD entry that assigns a name to the record. Descriptions of individual data fields follow the RECORD entry.

For Files with a Single Record Type

If all of the records in your data file are of the same type, use the following format for the RECORD entry.

Format

```
recordname  ['print label']  [USING MASK mask]  RECORD
                              USING
                              MASK
```

The **recordname** names the record whose component data items will be described following this entry. The record name is a default observation variable with a value of 1. In other words, the record name can be used in a TABLE statement to count records.

The optional **print label** following the record name is displayed in the table if the record name is used in the TABLE statement. If no label is provided, the record name is used as the label.

The optional **MASK** clause can be used to specify a special print format for record counts when the record name is used in a TABLE statement. Masks are described briefly later in this chapter and in detail in a separate chapter.

The keyword **RECORD** is always required in the RECORD entry.

Example

An example of a RECORD entry for a file in which each record contains information about a person is:

```
PERSONS  'Number of Persons' RECORD
```

For Files with More than One Record Type

If you are working with a data file that has more than one type of record, you need to describe the format for each type of record. Multiple record types are allowed within any hierarchical level, but the first record type at any level must be identified uniquely. A maximum of 30 record types (including repeating groups) can be described in one codebook.

In files with multiple record types, each type may differ from the other in length, but each record of the same type must be of the same length.

The RECORD entry for each type of record is the same as described above, except that two additional clauses provide information about the level and record identifier.

Format

```
recordname  ['print label']  [USING MASK mask]  RECORD
                              USING
                              MASK

            [LEVEL n  MARKER rec-id = 'value']
```

If a MARKER is present, a LEVEL number is required. The LEVEL number and MARKER can be in any order, but they must be at the end of the RECORD entry.

Following the word MARKER is the name of the data field that contains the record identifier for the type of record being described. The record identifier must be a value that uniquely identifies the type of record. The identifier should be in the same position for each type of record that requires a marker. The data field that contains the identifier must be a control variable that is described somewhere else within the record description. The identifier value follows the equal sign in the MARKER clause and must be within quote marks.

LEVEL n indicates the level of each type of record. For a hierarchical file, choose a low level number (e.g. LEVEL 0) for the master level. Each subordinate record level must increase in level number by one; that is, LEVEL 1, LEVEL 2,...LEVEL n. For a file that has multiple record types but only one level, any level number can be used.

The most common type of file with more that one record format is a **hierarchical** file in which records of different formats represent different levels of the hierarchy. For example, a household record could be followed by a record for each person in the household. The following example shows how LEVEL and MARKER are used in the description of the hierar-

chy. Each FAMILIES record contains the letter 'A' in the data field called CODE. Each MEMBERS record contains a 'B' in the data field called KIND. The MARKER fields are located at the beginning of each type of record.

Example

```
FAMILIES RECORD  LEVEL 0  MARKER CODE = 'A'
        CODE CON 1 (='A')
        (other record description entries)
                .       .       .
                .       .       .
MEMBERS RECORD  LEVEL 1  MARKER KIND = 'B'
        KIND CON 1 (='B')
        (other record description entries)
```

If the first record type at any level of the hierarchy is present in the file, all the remaining record types at that level must be present. In the following example, two record types make up a processing unit in a single-level file. For each family, both the FAMILIES and HOUSING records must be present. The FAMILIES record needs the LEVEL and MARKER clauses, because it is the first record of the pair. Assuming each FAMILIES record can be identified by the letter A in the first position, the record descriptions could begin as follows:

Example

```
FAMILIES RECORD  LEVEL 0  MARKER CODE = 'A'
        CODE CON 1 (='A')
        (record description entries)
                .       .       .
                .       .       .
HOUSING RECORD
        (record description entries)
                .       .       .
                .       .       .
```

Variable Entries

Each data **field** that you want to use in TPL TABLES must be described in a **variable entry**. The essential items in a variable entry are the name that you wish to use to reference the variable, the type of variable (OBSERVATION, CONTROL or CHAR) and the size. Optionally, you may include the START position of the variable. Depending on the variable type, additional information may be required. The variable entries must be entered in the order that they occur in the data record, and the codebook must account for the entire length of the data record.

Note that neither record names nor field names may be duplicated anywhere within one codebook.

If there are sections of a data record that you do not want to use, you do not need to describe them. You can skip over these spaces in the codebook with FILLER entries that tell TPL TABLES the size of the space you want to skip. Alternately, you can give the START position for the next variable you describe.

Another type of codebook entry can be used to describe a GROUP. A GROUP variable can be sudivided by one or more OBSERVATION, CONTROL or CHAR variables and FILLER entries. A repeating GROUP can be used to describe a group of variables that repeats a fixed number of times within a data record. Groups are described only briefly in this chapter. A separate chapter called Repeating Groups provides detailed information on their uses.

The attributes CONTROL, OBSERVATION and CHAR are assigned by you. The choice for a particular variable depends on how you want to use it and the types of values it can have. Details on CONTROL, OBSERVATION, CHAR, FILLER and GROUP entries follow.

CONTROL Variable Entries

Control variables are classifying or qualitative variables such as CITY or JOB_TYPE. When you describe a variable as CONTROL, you are telling TPL TABLES that you want the values for the variable to be treated like codes, where each code can represent a classification for producing summaries in tables.

Control variables can have both numeric and non-numeric values which must be stored in the data file as ASCII characters. Each control variable entry in the codebook must include a list of all possible values for the variable. These values are called **condition values**. When you list the values in the codebook, you can assign labels to the values. These labels will automatically be used in tables, if you use the variable in a TABLE statement.

See also the section on CHAR variables. A CHAR variable can contain any combination of numbers and characters, and you do not need to list the values in the codebook. CHAR variables are similar to CONTROL variables, but they cannot be used directly in TABLES statements. Instead, you must use DEFINE statements to select the values you wish to use in tables.

Format for CONTROL Variable Entries

```
name ['print label']  CONTROL  [fill specification]  n
                       CON

[CONDITION LABEL clause]   [DISPLAY AS  LISTED]
                                        SORTED
                                        NUMERIC
  (
       condition value list
  )
```

where, **n** represents the length of the field in bytes.

The **condition value list** is enclosed in parentheses. Values can be listed using any combination of the following formats:

1. [condition name] ['condition label'] = n
   'string'
2. = 'string'
3. m n ....
4. m:n

The word **IF** can be substituted for the = sign in value lists.

The optional **fill specification** can be any one of the following:

- LEFT BLANK FILL
- RIGHT BLANK FILL
- LEFT ZERO FILL
- RIGHT ZERO FILL

Examples of CONTROL Variable Entries

```
STATE_OF_RESIDENCE CON 2 DISPLAY AS SORTED
  (
       ALASKA        =       20
       NEW YORK      =       10
       UNKNOWN       =       ' '
  )

ROOMS CONTROL  RIGHT BLANK FILL  2
  (
       '1 ROOM'      =       1
       2:10
   )
```

```
YEAR CON 4
CONDITION LABEL IS VALUE
(1980:2000)

AGE CON 3 (65:100)

INDUSTRY CONTROL 2
  (
       'Oil & Gas'              =       'A1'
       'Steel'                  =       'B4'
       'Automobile'             =       'C3'
       'Wood Products'          =       'D1'
  )
```

Condition names for control variable values may not be duplicated within one control variable, but may be duplicated in separate control variables. Condition labels bounded by quote marks may be duplicated anywhere within a codebook without restriction.

Tip You can enter a value list with a format that is similar to the DEFINE statement by using the word IF in place of the = sign and adding **;** after each entry. For example:

```
INDUSTRY CONTROL 2
  (
       'Oil & Gas'              IF      'A1';
       'Steel'                  IF      'B4';
       'Automobile'             IF      'C3';
       'Wood Products'          IF      'D1';
  )
```

If you expect to select subsets of a variable using a DEFINE statement in table requests, this format can help you, because you can use your editor to copy the codebook description for the variable into your table request and then simply delete the entries that you don't want. The format for the entries you retain will match the format required for the DEFINE entries, so you will not need to do any additional editing in your table request.

Default Assumptions about Values

When the data values are listed in the codebook, TPL TABLES makes certain assumptions about listed values that are shorter than the field length. These default assumptions are explained immediately below. If your data does not fit the default assumptions, you may be able to simplify your value lists by using the FILL specifications that are described following the explanation of the default treatment.

Non-numeric. Any value that contains something other than a number is considered to be non-numeric. This includes values that contain blanks. When non-numeric values are listed in the codebook, they must be enclosed in single or double quote marks. If you list a non-numeric value that is shorter than the size of the field, it is assumed to be filled with blanks on the right. For example, if a control variable is a two byte field with values

```
|a |
|b |
|cc|
```

these values can be listed in the codebook as

```
'a'
'b'
'cc'
```

If, on the other hand, the short values are stored in the data file with blanks on the left, the blanks must be included in the listed values as follows.

```
' a'
' b'
'cc'
```

Numeric. Values that contain only numbers can be listed without quotes. If they are listed without quotes, values that are shorter than the size of the field are assumed to be filled with 0's on the left. If they are listed in quotes, short values are treated the same way as non-numeric values. This means that they are assumed to be filled with blanks on the right. For example, if you have a two byte control variable with numeric values 1 to 20, filled with 0's on the left as follows,

```
|01|
|02|
  .
  .
|09|
|10|
|11|
  .
  .
|20|
```

you can simply list the values as

```
1
2
.
.
19
20
```

or as

```
1:20
```

If, instead, the short values are stored with blanks on the right, you need to enclose the short values in quotes and include the blanks:

```
'1 '
'2 '
  .
  .
'9 '
10:20
```

Fill Specifications for Values

The following optional fill specifications can be used to simplify listing of values when control variables do not conform to default assumptions. The fill specification goes between the word CONTROL (or CON) and the field size.

```
COMPANY_TYPE 'Company Type' CONTROL RIGHT BLANK FILL 5
 (
        'Manufacturing'  =  'MF'
        'Aerospace'      =  'ASPAC'
        'Construction'   =  'CONS'
 )
```

LEFT BLANK FILL can be used when short values are always right-adjusted within their space and filled to the left with blanks. If the non-blank characters are numeric, the values do not need to be listed within quotes. Otherwise, the values must be enclosed in quotes, but the blanks on the left do not need to be included in the quoted strings.

RIGHT BLANK FILL can be used when short values are always left-adjusted within their space and filled to the right with blanks. If the non-blank characters are numeric, the values do not need to be listed within quotes. Otherwise, the values must be enclosed in quotes, but the blanks on the right do not need to be included in the quoted strings. RIGHT BLANK FILL is the default for values enclosed in quotes.

LEFT ZERO FILL can be used when short values are always right-adjusted within their space and filled to the left with zeros. If the non-zero characters are numeric, the values do not need to be listed within quotes. Otherwise, the values must be enclosed in quotes, but the zeros on the left do not need to be included in the quoted strings. LEFT ZERO FILL is the default for numeric values not enclosed in quotes.

RIGHT ZERO FILL is the opposite of LEFT ZERO FILL. It is tricky to use and not recommended. We think it unlikely that you will have any variables that require this option.

Display Order for Condition Values

For a control variable, the order in which its values are used is determined either by the order of listing in the codebook or by an ordering clause. When one of these clauses is used, it must immediately precede the parentheses at the top of the condition value list. The format is:

```
DISPLAY  [AS]  order-type
```

where the word **AS** is optional and the **order-type** can be **LISTED** (the default), **SORTED** or **NUMERIC**.

When the variable is referenced in a TABLE statement, the conditions are displayed in the specified order. The order is also used in other statements such as SELECT or DEFINE to determine whether one value is "higher" or "greater than" another.

DISPLAY AS LISTED This clause is the default. If it is entered in the codebook or if no order is explicitly specified for a control variable, the values will be used according to the order of their listing in the codebook.

With the DISPLAY AS LISTED clause, the codebook entry,

```
REGION CON 1 DISPLAY AS LISTED
  (
       SW     =      'D'
       NW     =      'C'
       NE     =      'A'
       SE     =      'B'
  )
```

would cause the REGION conditions to be displayed in the order:

```
SW
NW
NE
SE
```

If DISPLAY AS LISTED is chosen, all references to a range of values expressed in a SELECT, Conditional Compute, or DEFINE statement must be based on the order of the conditions listed, rather than the sort sequence of the condition values.

For example, since the DISPLAY AS LISTED clause is used in the preceding example, the region value of 'A' is considered to be greater than 'D' since 'A' is listed after 'D'. In a SELECT statement, a reference to REGION > 'C' would select region codes of 'B' and 'A'. A reference to REGION > 'B' would result in an error message since 'B' is considered to be the highest (last listed) value of region. Similarly, a DEFINE statement used to combine "SE" and "NE" into one classification would express the condition as 'A':'B' or > 'C'.

DISPLAY AS SORTED This clause causes the conditions to be ordered based on the sort sequence of condition values, regardless of how they are listed in the codebook. In the following example, the control variable has a DISPLAY AS SORTED clause.

```
REGION  CON 1 DISPLAY AS SORTED
  (
        SW      =       'D'
        NW      =       'C'
        NE      =       'A'
        SE      =       'B'
  )
```

If REGION is used in the stub, the rows for REGION will be displayed in the order of the values, 'A' 'B' 'C' and 'D', with the labels:

```
NE
SE
NW
SW
```

DISPLAY NUMERIC If all of the values for a control variable are numeric, you can specify DISPLAY NUMERIC **to order the values numerically**, regardless of leading or trailing blanks. If the variable is used in a table statement, the values will be displayed in numeric order. If you do comparisons to the values, for example in a SELECT or DEFINE statement, the comparisons will be evaluated numerically.

With the DISPLAY AS SORTED clause, described above, values are sorted in **character sort order**. This ordering may not always be appropriate for certain sequences of numeric values, especially if the values have leading or trailing blanks.

Assume that we have a control variable in which all of the values are numeric codes for different types of food. The variable has a width of 2 and short values are filled on the right with blanks. A sampling of values might be:

```
'Meat'          =       '1  '
'Lamb'          =       '11'
'Beef'          =       '12'
'Chicken'       =       '13'
'Vegetables'    =       '2  '
'Carrots'       =       '21'
'Beans'         =       '22'
```

It would be reasonable to used these values in the order shown. This is no problem. However, we might instead wish to use them in strictly numeric order as follows:

```
'Meat'        =     '1  '
'Vegetables'  =     '2  '
'Lamb'        =     '11'
'Beef'        =     '12'
'Chicken'     =     '13'
'Carrots'     =     '21'
'Beans'       =     '22'
```

We could reorder the values in the codebook, but this would be a big job if there were many values. If we were to specify DISPLAY AS SORTED, the values would be ordered in character sort sequence. The values with the blanks on the right would sort to the end rather than the beginning of the list.

The solution is to use DISPLAY NUMERIC. Note also that if we use the clause RIGHT BLANK FILL, described elsewhere in this chapter, we do not need to put the values in quotes. For example:

```
FOOD  CONTROL RIGHT BLANK FILL  2
DISPLAY NUMERIC
  (
        'Meat'        =     1
        'Lamb'        =     11
        'Beef'        =     12
        'Chicken'     =     13
        'Vegetables'  =     2
        'Carrots'     =     21
        'Beans'       =     22
  )
```

Note that the maximum value for a DISPLAY NUMERIC variable is about 2 billion on most computers. On certain 64-bit computers the maximum is much larger.

Listing Condition Values

Each control variable entry must include a list of admissible values for that field. These values can be listed in a variety of ways. The alternate ways may be combined in any order within the set of parentheses that encloses the value list. Each way of describing control variable values follows. (NOTE: Entries in the value list can end with semicolons (;) but not commas.)

- **Entering the value with its name or label**

Format

```
[condition name] ['condition label']  =  n
                                         'string'
```

Example

```
(
     ALABAMA              =     1
     ARK 'Arkansas'       =     2
     'Not Available'      =     'NA'
)
```

where a condition name and/or a condition label is assigned to one value of the control variable. A condition name identifies a value but does not appear within quote marks and therefore must not contain spaces or special characters other than '_' and '#'. In a table request, condition names can be referenced directly in SELECT, Conditional Compute, and DEFINE statements but not in TABLE statements.

When a condition name is assigned to a value and no condition label follows the name, the name is used as the print label for the value. When a condition label is assigned, with or without a condition name, it is used as the print label for the value. A condition label can contain lower case letters, special characters, or other label options such as footnote references. Condition label strings must be enclosed in quotes.

To the right of the equal sign is a specific control variable value. The value, n, represents an integer. If the value of n contains fewer digits than the character length of the data item, leading zeros will be assumed unless a fill specification is used for the variable. A control variable value that contains any character other than a number, including blanks, must be bounded with quote marks. If the 'string' value contains fewer characters than the length of the data item, then the data item will be assumed to have blanks to the right unless a fill specification is used.

- **Entering character values alone**

Format

```
= 'string'
```

Example

```
(
=  'A1'
=  'DF'
)
```

where 'string' is listed as a value without being equated to a condition name or label. Each string value must be preceded by an equal sign. Since no name or label is assigned, TPL TABLES will create one from the combination of the value and the variable name. For example, if we specify,

```
REGION CON 2
  (
       =  'A1'
       =  'DF'
  )
```

using REGION in the stub expression would result in the labels:

```
A1 REGION....
DF REGION....
```

- **Entering numeric values alone**

Format m n ....

where numeric values m n ... etc. are listed separately. Labels are created for the values in the format:

```
m variable-name   n variable-name ... etc.
```

- **Entering ranges of numeric values**

Format m : n

Example (20 : 100)

where m and n represent the lower limit and upper limit, respectively, of a range of numeric values. Each integer value, inclusively, within these limits is considered to be a separate value for the control variable being described. TPL TABLES will create labels for each value in the range in the format:

```
m variable-name....n variable-name
```

For example, if we specify,

```
YEARS_OF_AGE  CON 2
       (20:99)
```

using YEARS_OF_AGE in the stub would result in the labels:

```
20 YEARS OF AGE....
21 YEARS OF AGE....
.       .
.       .
99 YEARS OF AGE....
```

The range provides a convenient way of listing values where many, but not necessarily all, of the values within the range are present in the data file. For example, in a file containing the field described as YEARS_OF_AGE, there need not be a person of every age.

The CONDITION LABEL Clause for Automatic Generation of Formatted Labels

The condition value list can be preceded by a CONDITION LABEL clause to specify the type of label that you want for values that do not have individual labels assigned. This clause must follow the variable size specification and precede the value list that is enclosed in parentheses. The labels can consist of only the condition values themselves, or a combination of label strings and the values. The keyword VALUE in this clause indicates the relative position of each value to the label strings. The value can precede or follow a label string, or be inserted between strings. For example:

```
YEAR CON 2
        CONDITION LABEL IS '19' VALUE
          (50:99)
```

causes condition labels of '1950', '1951', ...'1999' to be generated for the values 50 through 99.

```
AGE CON 2
        CONDITION LABEL IS 'Age = ' VALUE
          (1:99)
```

causes condition labels of 'Age = 1', 'Age = 2', ...'Age = 99' to be generated for the values 1 through 99.

```
AGE CON 2
        CONDITION LABEL IS 'Age ' VALUE ' Total'
          (1:99)
```

causes condition labels of 'Age 1 Total', 'Age 2 Total', ...'Age 99 Total' to be generated for the values 1 through 99.

```
CODE CON 4
        CONDITION LABEL IS VALUE
          (0:999)
```

causes condition labels of '0', '1', ...'999' to be generated.

If a condition name or label is assigned to any specific value in the list, no label will be generated for that value. For example:

```
AGE CON 2
        CONDITION LABEL IS 'Age ' VALUE
          (
                0:99
                'Unknown'       =       'XX'
          )
```

causes condition labels of 'Age 0', 'Age 1', ...'Age 99' to be generated for the values 0 through 99, while the assigned label 'Unknown' is used for the value 'XX'.

The CONDITION LABEL clause can contain any of the options available for labels, such as footnotes and slashes for line spacing. If the values contain leading or trailing blanks, the blanks will not be included in the labels, unless the values are listed in quotes with the blanks included. This is true even when left or right blank fill is specified.

Control Variable Labels

If you enter a print label immediately following the control variable name, this label will be printed in tables where the variable is used. If used in the stub or heading, the variable label will be printed above and spanning over the condition labels. If you do not assign a variable label, only condition labels will be printed in the table. Variable labels are important when the value labels do not provide enough information to stand alone.

Examples

```
PERSONNEL RECORD
EDUCATION 'Education Level' CON 2
  (
        'High School'   =       1
        'Some College'  =       2
  )
```

```
JOB_CODES CONTROL  2
  (
        'Economist'      =       'B1'
        `Statistician'   =       'B2'
  )
MARITAL_STATUS 'Married?' CON 1
  (
        'Yes'            =       'Y'
        'No'             =       'N'
        'No response'    =       ' '
  )
```

The variables EDUCATION and MARITAL_STATUS have labels following their names. These variable labels will print above the labels for the individual condition values. Since the variable JOB_CODES does not have a label following its name, only the condition labels will print. If each of the three variables were concantenated in the stub expression, they would be printed as:

```
Education Level
High School.....
Some College.....
Economist.....
Statistician.....
Married?
Yes.....
No.....
No response.....
```

Control Variable Notes

1. More than one value cannot be combined into a single classification in a codebook. See the chapter on the DEFINE statement for ways to do this in a table request.

2. Blanks are not treated the same as zeros in control variable fields. If numeric fields are blank filled to the left or right, the condition values having blanks must be enclosed in quotes or the LEFT or RIGHT BLANK FILL option must be used.

3. If a control variable has a large number of potential values, but you want to use only a few values or ranges of values in a table request, you may find it much easier to simply describe the variable as OBS or CHAR in the codebook. A DEFINE statement in the table request can then be used to create a control variable with only the values you want to use.

OBSERVATION Variable Entries

Observation variables contain quantitative values that can be used in computations and added into table cells. Typical examples of observation variables are INCOME and WEIGHT (weighting factor).

Observation variables can contain only numeric values. TPL TABLES assumes that the numbers are stored as ASCII characters unless you indicate otherwise. An observation variable description does not include the value list that is required for a control variable description.

If an observation value does not contain a decimal point in the data file, it is assumed to be an integer. You can create decimal places with a SHIFT clause in the codebook or with COMPUTE or POST COMPUTE statements in a table request.

Note Arithmetic operations on two or more observation variables cannot be done with codebook statements. Deriving new observations from old ones must be done by computations in the table request.

Format for OBS Variable Entries

```
name ['print label']  [USING MASK mask] [blank or error treatment]
                      USING
                      MASK

OBSERVATION           n            [SHIFT DECIMAL       LEFT   m]
OBS                   BINARY  1                         RIGHT
                      BINARY  2
                      BINARY  4
                      UNSIGNED BINARY  1
                      UNSIGNED BINARY  2
                      FLOAT  4
                      FLOAT  8
                      FLOAT DOUBLE
```

The type and size (e.g. 2 or BINARY 4) indicate how the data is represented in the data file. The letter 'n' shown in the above format represents the

length of the field in bytes if the data values are stored as ASCII characters.

The variable name is followed by an optional print label that can contain any of the options available for labels. If you do not provide a label, TPL TABLES will use the variable name as the label when the variable is used in a table. The optional mask indicates how the summarized data should be displayed. Masks are described briefly in this section and in more detail in a separate chapter.

The clause shown as "blank or error treatment" can be placed anywhere between the optional label and the word OBSERVATION (or OBS). The possible treatments are:

```
BLANK = 0 (or ZERO)
DATA ERROR = NULL
REPORT ERROR = NO
```

These options are described in the section on Errors in Observation Values.

The optional SHIFT DECIMAL clause, at the end of the entry, can change the location of the decimal point. It is described in the section called The SHIFT DECIMAL Clause.

Examples of OBS Entries

```
INCOME 'GROSS FAMILY INCOME' OBS 6
VARIANCE MASK 999.99 OBSERVATION FLOAT 4
VAR OBS 3
WEIGHT_FACTOR 'Family weighting factor' OBS BINARY 4
GROSS "Gross Income" USING MASK $99,999 OBS FLOAT 8
WEIGHT OBS UNSIGNED BINARY 2
SALARY  DATA ERROR = NULL  OBS 6 SHIFT DECIMAL LEFT 2
```

Types of Observation Values

OBS n

When the data is described with only its length 'n', TPL TABLES assumes that the data is stored as ASCII characters. The data values can have optional leading blanks or tabs, an optional '+' or '-' character, and a number with an optional decimal point. Trailing blanks are permitted. The numbers can contain commas. For example, 43,586.43 is an acceptable value.

Note If a value is entirely blank, it is considered to be a data error. See the section on Errors in Observation Values for ways to specify different treatment for blanks.

Values for ASCII observation variables can also be stored in E notation, e.g. 53.6e10 or .53E13, but processing will be slower for E notation values. For values stored in E notation, the E cannot be preceded by a blank. TPL TABLES will end the value at the blank and will not find the E. Commas are not allowed in E notation values.

OBS BINARY n

Data is stored as a signed binary number 'n' bytes in length. The value of 'n' can be 1, 2, or 4.

OBS UNSIGNED BINARY n

Data is stored as an unsigned binary number 'n' bytes in length. The value of 'n' can be 1 or 2.

OBS FLOAT 4
OBS FLOAT 8
OBS FLOAT DOUBLE

Numeric data is stored in the floating point form that is standard for your computer. Floating point data can have a length of four bytes (FLOAT 4) for single-precision floating point, or eight bytes (FLOAT 8 or FLOAT DOUBLE) for double precision floating point.

The Mask Clause

The codebook mask clause is used to specify how the final values of an observation variable are to be displayed in tables. If no mask is provided, data are right justified within the column and no decimal places or special symbols other than commas are shown. A mask may be added to force centering based on the largest expected cell value, to indicate the number of decimal places to be displayed, to display values with text, or to cause values to be footnoted.

Each mask clause in the codebook begins with the words MASK, USING, or USING MASK. Some examples of mask clauses with codebook observations are shown below. All mask options are described in detail in a separate chapter on masks.

```
USING  $99,999
```

Center data based on mask size and add a dollar sign to the first value of the variable appearing in a column. If the displayed value is smaller than the mask, the dollar sign will "float" to the right.

```
MASK  RIGHT $99,999 ' '
```

Offset the data one space from the right column divider, adding a dollar sign to the first entry in each column.

```
USING MASK  99.99 FOOTNOTE (SOURCE)
```

Center the data based on mask size and precede it with the symbol for the footnote called SOURCE. The data will be displayed to two decimal places. Note that if the data values are integers, only 0's will be printed to the right of the decimal point. For example a final cell value of 1,237 would be displayed as 1,237.00. The footnote symbol and text can be specified in a SET FOOTNOTE statement in the codebook, table request, or format request.

```
USING '**' 9999 '**'
```

Bound all cell values with two asterisks. The cell values and special symbols will be centered based on mask size.

```
USING MASK  999.99 '%' RIGHT
```

Each cell value will be followed by the character % and will be right-justified in the cell.

```
MASK  999,999 FONT HBI 10
```

Each cell value will be centered based on the size of the mask. In PostScript mode, the values will be printed using the Helvetica-BoldOblique font, size 10.

The SHIFT DECIMAL Clause

A SHIFT DECIMAL clause can be included at the end of an OBSERVATION variable entry to adjust the location of the decimal point. The adjustment takes place as the data values are read.

Format The format of the SHIFT DECIMAL clause is:

```
SHIFT  DECIMAL  direction  m
```

where **direction** can be LEFT or RIGHT and **m** is the shift amount. The shift amount must be a positive integer. Negative amounts are not allowed and 0 has no effect.

Although a SHIFT DECIMAL clause can be used with any observation variable, it is used most often to get the effect of a decimal point for variables that do not have explicit decimal points included in the data values. If you do not specify a SHIFT DECIMAL clause for this type of variable, TPL TABLES assumes that its values have no decimal places.

Example

For example, assume we have a variable called **SALARY** with values of **1000**, **2000** and **3000** for hourly salary amounts. No explicit decimal points are included in the data, but there are two decimal places so that the meaning of the values is actually **10.00**, **20.00** and **30.00**. If we describe salary as

```
SALARY  OBS  4
```

and then tabulate the salary values, TPL TABLES will assume that there are no decimal places and give a sum of **6000** for the three values.

If we add a SHIFT DECIMAL as follows,

```
SALARY  OBS  4  SHIFT DECIMAL LEFT  2
```

the data values will be divided by 100 as they are read, giving a sum of **60**.

If we want the two decimal places to be printed in the table, we must add a mask:

```
SALARY  MASK 99.99  OBS  4  SHIFT DECIMAL LEFT  2
```

gives the same sum of **60**, but it will print as **60.00** to show the two decimal places indicated in the mask.

The shift can be either LEFT or RIGHT. For example,

```
SHIFT DECIMAL RIGHT 3
```

causes data values to be multiplied by 1000 as they are read.

```
SHIFT DECIMAL LEFT 3
```

causes data values to be divided by 1000 as they are read.

A SHIFT DECIMAL without direction is interpreted as SHIFT DECIMAL LEFT. For example,

```
SHIFT DECIMAL 3
```

is the same as

```
SHIFT DECIMAL LEFT 3
```

Errors in Character (ASCII) Observation Values

For a character (ASCII) observation variable, TPL TABLES will detect and report as errors values that are completely blank and values that contain other non-numeric characters. Records containing data errors are discarded.

The following settings can be entered in the codebook to change the treatment of observation variables that have blank or other error values:

```
DATA ERROR = NULL [or ERROR]
REPORT ERROR = NO [or YES]
BLANK = ZERO [or ERROR]
```

The default settings are shown in brackets.

- **Global or individual settings**

You can specify these treatments for individual variables, or you can give global settings in the codebook. If you want to treat all of your variables the same way, you will want to use the global approach and enter the settings at the beginning of the codebook.

For *global settings*, place the settings of your choice after the BEGIN ... CODEBOOK statement before any variables, or put the statements between variables. The settings will apply to all following variables. If you put in a different global setting further down in the codebook, it will apply from that point on. Note that settings cannot be placed between the RECORD clause and the first variable.

You can override the global setting by entering an **individual setting** in a variable description. This techniques is described below, along with a detailed decription of how each setting works.

- **Converting error values to NULL**

One or both of the following clauses can be used in converting error values to NULL. They should follow the variable name and optional label and precede the word OBS (or OBSERVATION). If both are used, either one can precede the other.

```
DATA ERROR = NULL
REPORT ERROR = NO
```

If **DATA ERROR = NULL** is specified, blanks or other erroneous values for the variable will be set to NULL and no records will be discarded because of the error. NULL values are not counted in tabulations.

Note that if your data has other values that you do not want to count in tabulations, you can convert these values to NULL using Conditional Compute statements in a table request. For more information on creation and use of NULL values, see the COMPUTE statement chapter.

For data files that have huge numbers of records, the DATA ERROR = NULL clause may significantly increase the time it takes to read the data. If you are concerned about performance, use the clause only with data fields that are known to contain errors.

REPORT ERROR = NO will suppress error messages for the variable. Be cautious in using REPORT ERROR = NO, since it can allow data errors to pass unnoticed.

Example

```
Income 'Family Income'
DATA ERROR = NULL   REPORT ERROR = NO
MASK $999,999  OBS 7
```

- **Converting blanks to zeros**

By default, blank observation values are considered to be data errors. The DATA ERROR and REPORT ERROR clauses can be used with blank values. Sometimes, however, a blank value is used to represent a value of 0. If you have observation fields of this type, you can use the clause

```
BLANK = 0    (or BLANK = ZERO)
```

in the codebook. If you use this clause, blank values will be replaced with the value 0, and no error will be reported. The clause should follow the variable name and optional label and precede the word OBS (or OBSERVATION).

Example

```
Income 'Family Income'   BLANK = 0   MASK $999,999  OBS 7
```

Errors in Binary and Floating Point Observations

TPL TABLES cannot detect errors in binary or floating point observation variables, since any values are possibly valid. In some cases, a "floating point overflow" error may occur, but usually the only evidence of errors is unexpected numbers in the table output.

CHAR Variable Entries

CHAR variables can contain any combination of numbers and characters. In this way, they are similar to CONTROL variables, but you not need to list the values in the codebook. A CHAR variable can be used in any TPL TABLES statements where a CONTROL variable can be used, except that it cannot be used directly in TABLES statements. Instead, you must use DEFINE statements in a table request to select the values you wish to use in tables.

Format for CHAR Variable Entries

```
name ['print label']  CHAR   n
```

where, **n** represents the length of the field in bytes.

Example

```
INDUSTRY  CHAR  8
```

where the variable called INDUSTRY can have any value that is 8 characters long.

Example

```
CITY  CHAR  30
```

where CITY can have any city name up to 30 characters long.

Using START Position in Variable Entries

START position is optional in variable entries. If used, it should follow the variable name and label (if present) and precede the variable type. Following are some examples:

Examples

```
AGE 'Age of student' START 1 CONTROL 2
(1:99)
NUMBER_OF_CHILDREN START 3 OBS 2
INCOME 'Family Income' START 20 MASK 999,999 OBS 6
INCOME 'Family Income' MASK $999,999 START 20 OBS 6
```

FILLER Entries

FILLER is the standard name given to any area of a record that you do not wish to describe or use.

Format

```
FILLER [START position] n
```

The length of the filler area, **n**, is simply skipped over and the next data entry is considered.

A record description in a codebook must account for the entire length of the record. If you are not describing the last portion of the record, you will need a FILLER entry for the undescribed portion.

Assume that you have described the first 40 positions for a record that has a length of 80 and that you do not need to use anything else in the record. You can fill out the record with:

Example

```
FILLER  40
```

If you are not sure how many positions are left to describe but you know the record length, you can end your record description with a filler that covers the entire length of the record by giving a **START position** of 1 and a size equal to the record size. For a record with length of 80, the FILLER would be:

Example

```
FILLER START 1 80
```

GROUP Entries

A GROUP variable in one that is sudivided by one or more OBSERVATION variables, CONTROL variables, CHAR variables and FILLER entries. There are two types of GROUP variables: simple groups and repeating groups. For either type, there is both a **BEGIN GROUP** entry and an **END GROUP** entry.

Simple Groups

A simple group is a optional codebook entry that can be used to organize the description of a collection of items. This type of group has no other function. The group variable cannot be referenced in a table request, although any variables contained within it can be referenced. In the codebook, the group variable can be referenced in a REDEFINE entry.

Format

```
BEGIN GROUP  group-variable-name

        one or more entries for variables or FILLER

END GROUP  group-variable-name
```

Example An example of a simple group is:

```
BEGIN GROUP  HOUSING
  LIVING_QUARTERS  CONTROL 1
    (
      OWNED              =     1
      CONDOMINIUM        =     2
      RENTED             =     3
      NO_RENT_PAID       =     4
      NOT_AVAILABLE      =     5
    )
  MORTGAGE  CONTROL 1
    (
      MORTGAGED          =     'A'
      NOT_MORTGAGED      =     'B'
      NOT_AVAILABLE      =     'C'
    )
END  GROUP  HOUSING
```

In this example, **HOUSING** is the name of the group variable. It can be referenced in a REDEFINE entry in the codebook, but it cannot be used anywhere else. The variables within the group, **LIVING_QUARTERS** and **MORTGAGE**, are CONTROL variables that can be referenced just like any other CONTROL variables.

Repeating Groups

In this chapter, we provide only introductory information on repeating groups. A variety of repeating group structures and their uses are described in a separate chapter called Repeating Groups.

A repeating group can be used to describe a group of variables that repeats a fixed number of times within a data record. Both the repeating group variable and the variables within the group can be referenced in a table request. The repeating group variable can also be referenced in a REDEFINE entry in the codebook.

Format The format for the repeating GROUP is:

```
BEGIN GROUP  group-variable-name  REPEATS  n
[  (  optional names or labels for the repetitions  )  ]

        one or more entries for variables or FILLER

END GROUP  group-variable-name
```

where **n** is the number of times the group repeats in the data record.

Example An example of a repeating group is one in which certain data items repeat for a number of time periods such as monthly. In the following repeating group, monthly data on **HOURS** and **EARNINGS** is repeated for **12** months of the year. Each pair of **HOURS** and **EARNINGS** is separated by one character of **FILLER**. A label is provided for each of the **12** repetitions.

```
BEGIN GROUP  MONTH  REPEATS  12
('Jan', 'Feb', 'Mar', 'Apr', 'May', 'June',
'July', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec')
        HOURS  OBS  2
        FILLER  1
        EARNINGS  OBS  8
END GROUP  MONTH
```

REDEFINES Entries

The REDEFINES entry provides an alternative description of a record area previou!sly described. One common use of REDEFINES is to describe a field as both OBSERVATION and CONTROL. For example, if you have a field that contains the actual age of persons in the data file, you may wish to use it as a control variable for tabulating persons by age but also use it as an observation variable to calculate average age for other categories.

REDEFINES can also be used to describe a code as 1 digit, 2 digit, 3 digit, ... where the first digit of the code gives the major category and additional digits provide additional detail. Examples are commodity codes or industry codes.

Format The format for REDEFINES is:

```
dataname2 ['print label']  REDEFINES dataname1  [variable attributes]
FILLER
```

where **dataname1** is the name of a previous data description entry within the same record description. **Dataname2** is an alternate name for the area starting with **dataname2**. Redefinition starts at **dataname1** and continues from that point forward within the data record.

REDEFINES may not be part of a RECORD entry. In other words, one record may not redefine another at the record level.

Example

```
A1  OBS 1
A2  OBS 3
A3  OBS 2
B  REDEFINES A1  OBS 6

ROOMS  CON 2 (1:10)
#_OF_ROOMS REDEFINES ROOMS  MASK 99  OBS 2
```

In this case, **B** redefines **A1**. When **B** redefines **A1**, the redefinition includes **A1**, **A2**, and **A3**. Thus the six byte field can be tabulated as a whole, or separately for each field **A1**, **A2** and **A3**. **#_OF_ROOMS** redefines **ROOMS** so that the same field can be used as both a control variable and an observation variable.

It is not necessary that a redefinition have the same length, or immediately follow the variable redefined. The REDEFINES entry specifies only the point at which the redefinition begins. For example:

```
A OBS 1
B OBS 4
C CON 2
  (     25
        37
        43
  )
X REDEFINES B CON 2 DISPLAY AS SORTED
  (     16
        12
        10
        0:9
  )
Y OBS 6
Z OBS 3
```

The entry **Y** following the REDEFINES entry will begin describing the record area after the first two bytes of **B** and will extend two bytes beyond **C**. The area from **A** through **Z** is 12 bytes.

Note that **if a record description ends with a redefinition**, the redefinition must cover the entire space already described for the record. Otherwise, when your codebook is processed, you will get a message requesting that you add a FILLER of the appropriate length. In the following codebook, the last redefinition is four characters short of covering the space previously described by COMMODITY_CODE, so FILLER 4 is needed at the end.

```
BEGIN SAMPLE CODEBOOK
.
.
.
COMMODITY_CODE OBS 10
COM3 REDEFINES COMMODITY_CODE OBS 3
COM6 REDEFINES COMMODITY_CODE OBS 6
FILLER 4

END SAMPLE CODEBOOK
```

Redefining Space with START Position

If you know the start positions for the data items in your records, you can use START position in codebook entries as an alternative to REDEFINES. For example, the variable AGE could be used as either an observation or a control variable. You can describe the variable twice using the same START position for each one.

Example

```
BEGIN PERSONS CODEBOOK
AGE START 1 CON 3
(1:120)
AGE_OBS START 1 OBS 3
```

The END Entry

The last codebook entry can be used to end the codebook. Its use is optional. The format is:

Format

```
END  [ codebookname [CODEBOOK] ]
```

A Codebook Example Describing Multiple Record Types

The most useful application of multiple record types is with hierarchical files, where each level is of different length and contains different information. For example, a family expenditures file may consist of a family characteristics records of 159 characters at level 0 of the hierarchy, each followed by one or more family expenditure records of 79 characters at level 1 of the hierarchy. The complete codebook for such a file might resemble the following.

```
BEGIN FAMILIES CODEBOOK

  FAMILY RECORD  LEVEL 0  MARKER ID = 'A'
       FILLER 4
       ID CON 1 ( = 'A')
       REGION CON 1
        (
              'Northeast'       =       1
              'North Central'   =       2
              'South'           =       3
              'West'            =       4
        )
       MORTGAGE CON 1 (1 : 3)
       INCOME_OF_HEAD  OBS 7
              .
              .
              .
       PERSONS_IN_FAMILY  OBS 2

  EXPENDITURE RECORD  LEVEL 1  MARKER CODE = 'B'
       FILLER 4
       CODE CON 1 (='B')
       ITEM_CODE 'Item Code' CON 4
       CONDITION LABEL IS VALUE
          (
              1357
              1367
              1377
              1387
          )
       EXPENDITURE_CLASS  CON 2 (1:20)
              .
              .
              .
       COST  OBS 7
```

END FAMILIES CODEBOOK

The MARKER field that contains the record identifier must be located within the shortest record length.

CODEBOOKS FOR CSV AND OTHER TYPES OF DELIMITED DATA FILES

Codebooks for delimited files are similar to codebooks for other ASCII text data files. The main differences are:

- Delimited data files do not have the data fields in columns of fixed size. Instead, the rows of data have values separated by commas or some other delimiter character. This means that fields are identified by their order in a row of data and that data values for a particular field may have different widths.

- In delimited files, the first row can optionally have names for each field.

Windows If you are using the Windows version of TPL, you have the option of using Codebook Builder to generate the codebook interactively. See Codebook Builder Help for instructions.

The BEGIN Entry

The first entry in a codebook for a delimited file tells TPL that the file is delimited, what the delimiter is, whether the first row contains labels, and whether any items in the file can be surrounded by quotes. The format for the entry is:

Format

```
BEGIN codebookname  [CODEBOOK]  CSV [ ( file-specifications ) ]
```

where **codebookname** is a name you choose, **CSV** indicates that the file is delimited, and the word **CODEBOOK** is optional. If there are no additional **file-specifications**, the following will be assumed:

```
DELIMITER = COMMA
HEAD = YES
QUOTES = ""  (double quote)
```

Example

```
Begin BIRTHS Codebook CSV
```

One or more additional **file-specifications** can follow in parentheses for files that differ from the defaults. They can be entered in any order. The options are:

Format

```
[DELIMITER = string]      [HEAD = YES]    [QUOTES = string]
             COMMA                NO                  NONE
             TAB
             SEMICOLON
             BLANK
```

Delimiter

The default delimiter is the comma. The delimiter is the character that separates values in the file. The most commonly used delimiters can be entered by their names, for example TAB for a tab-delimited file. You can also enter a delimiter character as a string in quotes. The delimiter can be only a single character. If multiple characters are entered in a string, only the first will be taken to be the delimiter.

Examples

```
Begin BIRTHS Codebook CSV (DELIMITER = TAB)

Begin BIRTHS Codebook CSV (DELIMITER = '|')
```

Head

The default is YES. This means that the first row of the file will be assumed to be a header record that contains names for the fields. Items in this record will not be tabulated or used as data in any way. If the first row of your file contains data, specify HEAD = NO.

Example

```
Begin BIRTHS Codebook CSV (DELIMITER = TAB  HEAD = NO)
```

Quotes

Sometimes single or double quotes are included around some or all of the values in a csv file. This is especially true if some fields contain blanks or if blank is used as a separator. If any fields in your file have quotes, enter the appropriate quote symbol. *The default is double quotes*. Note that if you are entering a single quote, enter it inside double quotes """". If you are entering a double quote, it must be inside single quotes '"'.

Example

```
Begin BIRTHS Codebook CSV
        ( DELIMITER = BLANK  HEAD = NO  QUOTES = """ )
```

Variable Entries

Variable entries are generally the same as for other non-database codebooks. The main differences are that you identify fields by field number, you do not need to account for all fields, and you do not need to list the fields in order of occurrence in a record.

Enter the field number specification somewhere between the field name and the field type of Observation, Control or Char.

Example

```
AGE "Age of Father" Field = 2 OBS 3
```

Following are a file that has data about births and a sample codebook for the file. Comma is the delimiter. Note that the first record of the file has names for each of the fields and that some of the values are inside of double quotes. Since the default setting for QUOTES is double quotes, we do not need to specify it in codebook. The HEAD and DELIMITER specifications in the codebook are also the same as the defaults, so we could omit them if we wished.

Data

```
RCOUNTY,HOSPITAL,SEX,RACE,FAGE,MAGE,WEIGHT
Adams,1P001,F,2,23,22,128
"Ben Franklin",1A002,M,3,27,24,98
Adams,4K001,M,1,40,37,142
Washington,1P001,F,1,29,27,123
```

Codebook

```
Begin Births Codebook CSV (Head = Yes Delimiter = COMMA)

Births Record

RCOUNTY "County of Residence" Field = 1 Right Blank Fill Control 12
Condition label is value
  (
   =  "Adams"
   =  "Ben Franklin"
   =  "Washington"
  )
SEX "Sex" Field = 3 Control 1
Condition label is value
  (
   "Male"  =  "M"
   "Female"  =  "F"
  )
```

```
FAGE "Father's Age" Field  = 5 Obs 2

FAGE_CON Field 5 Control 2
Condition Label = Value
(10:99)

MAGE "Mother's age" Field  = 6 Obs 2

WEIGHT "Weight in oz." Field  = 7 Mask 999 Obs 3
```

Key Points to Note about the Codebook

- FILLER is not used in a delimited codebook. If you do not want to use all of the fields in your file, simply omit them from the codebook. In the above example, we omitted Fields 2 and 4, because we did not want to use them.
- When a field is to have more than one type, do not use REDEFINE. Instead, you can create a a variable entry for each type, each specifying the same Field number. FAGE and FAGE_CON are two TPL variables based on Field 5. This gives the same result as what you would get with REDEFINE in codebooks for non-delimited files.
- When entering the size for a field, use a size that is at least as large as the maximum size of any value for that field. If any values are enclosed in quotes, you do not need to include the quotes in the size. If there are values in a field that are larger than the size in the codebook, you will get error messages when using the field.
- Codebooks for delimited files cannot have repeating groups or hierarchies.
- **Important note for Control variables.** For fields in delimited files, TPL always assumes that short values are filled to the right with blanks. If you have fields that can have numeric values and the values can be of different sizes, we recommend that you assign the RIGHT BLANK FILL attribute to these fields. Otherwise, you will need to put quotes around the values, numeric or not, when you refer to them in statements such as Select or Define. You can use RIGHT BLANK FILL in

describing any Control variable, regardless of the widths of its values.

Example The control variable RPLACE can have values 1 to 100 or the value 999 if the place of residence is unknown. This means that the values can have widths of 1, 2 or 3. Without RIGHT BLANK FILL, if you are referencing the shorter values in statements such as Select or Define, you will need to put the values in quotes. With the following, you do not need to be concerned about the width of the values you are referencing or put the numeric values in quotes.

```
RPLACE "Place of residence" Field = 3 RIGHT BLANK FILL Control 3
(1:100
 999
)
```

Delimited Fields that Have Blank or No Value

If a field has any values that have only a blank or nothing at all between the delimiters, these values will be treated like blanks. For example, in a comma-separated file, these values could be:

,, , , ,"", or ," ",

There are different options available for these values depending on the field type.

Observation

If you use fields that have these values, they will be data errors since they are not numeric. Any record that has such values will be discarded from processing. If you do not want this to happen, you can assign the attribute DATA ERROR = NULL to any affected fields. NULL values are not included in tabulations. Alternately, you can assign the BLANK = ZERO attribute to any affected fields. These attributes are described in more detail elsewhere in this chapter under "Errors in Character (ASCII) Observation Variables".

Example

```
FAGE "Father's Age" Field  = 5 DATA ERROR = NULL Obs 2
```

Control

If you use fields that have these values and the values are not in the condition values list in your codebook, they will be data errors. The records with these values will be discarded from processing. If you want to prevent this, you can include a blank value in the list.

Example

```
FAGE_CON "Father's Age" Field 5 Control 2
Condition Label = Value
(
 10:99
 "Unknown" = " "
)
```

CHAPTER 7

Use

ACCESSING THE CODEBOOK

The USE statement must be the first TPL statement in a table request. It takes the form:

Format

```
USE codebookname [ CODEBOOK ];
```

where 'codebookname' is the name assigned to the codebook description of the data file to be accessed by the table request. The keyword CODEBOOK is optional following the codebook name.

The USE statement makes available to all following request statements all names defined in the codebook.

For a codebook named **SURVEY**, the USE statement would be

Example

```
USE SURVEY CODEBOOK;
```

or more simply

```
USE SURVEY;
```

Windows Note A codebook name can contain blanks. If it does, the name must be enclosed in ***double*** quotes.

Example

```
USE "FIRE DISPATCHES" CODEBOOK;
```

UNIX Note Codebook names cannot contain blanks.

Before the codebook is used, it must be processed to create a file named codebookname.K. For the codebook named SURVEY, the processed codebook is SURVEY.K . TPL TABLES assumes that it is in the directory where you are running your table job.

If the processed codebook is not in the directory where you are running your job, you can enter the path information needed to find the processed codebook. The following USE statements provide some examples of acceptable codebook references:

Examples

```
USE  D:\MY_CBS\SURVEY.K ;
USE "C:\MY DATA\EMPLOYEE RECORDS" CODEBOOK;
```

Unix Note Slash characters in paths are forward slashes rather than back slashes.

Unix Example

```
USE  ../my_cbs/SURVEY.K ;
```

In any of the above examples, the **.K** suffix can be omitted.

Note Comments cannot be inserted in the USE statement between the word USE and the codebook name.

CHAPTER 8

Select

SELECTING SUBSETS OF THE DATA

The SELECT statement specifies conditions that must be met by each record of the data file to qualify for further processing by TPL TABLES.

A SELECT statement can apply to all tables within a request or it can be applied to an individual table. If more than one SELECT statement is used in the request, the conditions in all SELECT statements must be met for a record to qualify for further processing.

Note that if your data file is hierarchical, the unit to be selected will be a hierarchical unit rather than a single record. The chapter on processing hierarchical files contains additional information on this subject.

See also the chapter on the DEFINE statement for additional ways to select or filter data for all or part of a table.

Data can be selected based on data values, or certain sections or percentages of the data can be selected.

Selection Based on Data Values

Format 1 This type of selection takes one of the following two forms:

```
SELECT   IF          condition1   [ AND condition2.....];
         UNLESS                    OR

                     or

SELECT   IF          NOT ( condition1    [AND condition2...] );
         UNLESS                          OR
```

To specify selection criteria that apply to ***only a single table***, you can reference that table following the word SELECT. The table can be referenced by number or by name.

Format 2

SELECT **FOR TABLE** table-number .....

or

SELECT **FOR TABLE** table-name .....

Examples

```
SELECT  IF  ACCOUNT  NOT = 24765;

SELECT  IF ACCOUNT IS NOT EQUAL TO 24765;

SELECT  UNLESS  (REGION = 'A1' AND INCOME <= 12000);

SELECT  FOR TABLE 1  IF
        (INCOME >= 12000 AND INCOME < 20000) OR
        (STATE = MONTANA AND
        (OCCUPATION = 1 OR OCCUPATION = 5));

SELECT  FOR TABLE A5  IF
        OCC_CODE IN (2000, 2001, 2002, 2003, 3000)
        AND INCOME >= 30000;

SELECT  IF OCCUPATION = ACCOUNTANT AND
        NOT (REGION = 'D3' OR REGION = 'E5');

SELECT  UNLESS (STATE='CA' AND
        (JOB_BANK=10 OR JOB_BANK=20))
        OR (STATE='IL' AND (JOB_BANK=13 OR
        JOB_BANK=15));
```

When the IF option is used, all records meeting the conditions will be selected. When the UNLESS or IF NOT option is used, all records which do not satisfy the conditions will be selected. If no SELECT statement appears in the request, all records will qualify for further processing by TPL TABLES, regardless of their characteristics.

A condition expresses a relationship or a set of values. It can include codebook variables, computed variables, literal values, and arithmetic expressions. If the select references a computed variable, the compute statement must precede the select statement.

Relations can be expressed by either symbols or words. The words IS and TO are optional. Following is a list of the relation symbols and their English equivalents:

Symbol	English expression
<	[IS] LESS THAN
>	[IS] GREATER THAN
=	[IS] EQUAL [TO] EQUALS
^<	[IS] NOT LESS THAN
^>	[IS] NOT GREATER THAN
^=	[IS] NOT EQUAL [TO]
<=	[IS] LESS THAN OR EQUAL [TO]
>=	[IS] GREATER THAN OR EQUAL [TO]

Types of Conditions

A condition can test for a **relationship**, or it can test a variable to see if it has any of the values specified in **a set of values**. We first describe how relationships can be tested, then follow with a section on sets of values.

Relationships

A condition that tests a relationship can take several forms. In each form which follows, **re** stands for a relation.

- **Comparing a variable to a value**

Format

```
variable re literal-value
```

In this type of condition, the **variable** is compared to a **literal-value**.

Examples

```
SELECT IF AMOUNT < 100.75;
SELECT IF REGION = 'A1';
```

In the first example, the statement would cause selection of all records with an AMOUNT less than 100.75. In the second example, selection would be of all records containing the value 'A1' for the control variable REGION.

The **variable** in the condition can be any codebook or computed variable, and **literal-value** can be a number or a character string.

If the variable is an **observation** variable, the value can be a number, with or without a sign. The number can contain a decimal point but no commas.

If a **control** variable is compared to a literal value that is not all numeric, the value must be surrounded by quote marks.

If the variable is a **char** variable, the value must be surrounded by quote marks and must include any leading blanks or zeros. The only relations that should be used with char variables are equal '=' and not equal '^='. If other relations are used, the results are unpredictable. For example, if LAST_NAME is a char variable,

Example

```
SELECT IF LAST_NAME = 'Smith' ;
```

will successfully select all records with a value of 'Smith' for LAST_NAME if the value 'Smith' is left-aligned, i.e. not preceded by blanks in the data records.

When the same variable is compared with two or more distinct values, the subject must be repeated each time. For example:

Example

```
SELECT IF AGE=14 OR AGE=18 OR AGE=29;
```

For a simplified way of expressing this type of condition, see the section on **Sets of values**.

If the "DISPLAY AS SORTED" clause is not used with a codebook **control** variable, all references to a range of values expressed in a SELECT must be based on the order of the conditions listed, rather than the sorted sequence of the condition values. For example, consider the following control variable entry:

```
REGION CON 1
  (
        SW = 'D'
        NW = 'C'
        SE = 'B'
        NE = 'A'
  )
```

Since the "DISPLAY AS SORTED" clause is not used, the region value of 'A' is considered to be greater than 'D' since 'A' is listed after 'D'. In a SELECT statement, a reference to REGION > 'C' would select region codes of 'B' and 'A'. A SELECT statement such as "SELECT IF REGION > 'A'" would result in an error message since 'A' is considered to be the highest (last listed) value of region.

Observation variables can be tested for null values, but if a null-valued variable is tested for a value other than null, the test will fail. For example, the test: x > 0 will fail if the value of x is NULL.

Arithmetic expressions

Conditions can include arithmetic expressions. Any arithmetic expression allowed in the COMPUTE statement can be used in the SELECT statement. See the chapter called "COMPUTING NEW VARIABLES: The COMPUTE Statement" for complete details on arithmetic expressions.

Arithmetic operations can contain only **observation** variables and numbers. The result of a computation can be compared to an observation variable, a number, or another arithmetic expression.

Example

```
SELECT IF ANNUAL_INCOME / 12 > 2000;
```

- **Comparing one variable to another**

Format

```
name1 re name2
```

where the names refer to codebook or computed variables. **Name1** and **name2** must be the same type of variable: **observation**, **control** or **char**. For example, a control variable cannot be compared to an observation variable.

If **control** or **char** variables are being compared, their values must have the same justification and padding for the comparisons to work correctly.

Example

```
SELECT IF STATE_OF_RESIDENCE = STATE_OF_EMPLOYMENT;
```

- **Comparing a control variable to a condition name**

Format

```
control-name re control-condition-name
```

A **control** variable name may be used to identify one of its conditions by referencing it by condition name rather than condition value.

Example

```
(codebook)
STATE CON 2
  (
       ALABAMA = 1
       .   . .
       .   . .
       MARYLAND = 26
       .   .  .
  )
```

```
(request)
SELECT IF STATE = MARYLAND;
```

Sets of Values

Another type of condition lets you select records that have any of the values specified in **a set of values**. This feature is particularly useful if you need to select for a long list of values.

The format for entering a set of values in a SELECT statement is:

Format

```
SELECT IF  var  IN (val1, val2, val3, .... );
```

where **val1**, **val2**, **val3**, etc. are distinct values. Values must be separated by commas. Non-numeric values must be surrounded by quote marks. For observation variables, negative values such as -25 cannot be used in sets.

Comparisons such as less than or greater than cannot be used, but you can enter ranges of values separated by :, - or the word TO. For example, ranges such as 3:5, 3-5 or 3 TO 5 are allowed.

Example

```
SELECT IF INDUSTRY IN (1000, 2000, 3000:3999);
```

If the variable INDUSTRY has the value 1000, 2000, or any of the values 3000 through 3999, the record will be selected. This example gives the same result as the statement:

```
SELECT  IF INDUSTRY = 1000 OR INDUSTRY = 2000
          OR INDUSTRY >= 3000 AND INDUSTRY <= 3999;
```

The word IN cannot be preceded by the word NOT. If you want to select records that do NOT have any of the values in the set, you can specify:

```
SELECT IF  NOT  ( var IN (val1, val2, val3, .... ) );
```

Selecting records on the basis of a set of values will give more efficient processing than the individual comparisons if there are more than a few (e.g. 3 or 4) values in the set.

The clause **var IN (set of values)** can be used alone in the SELECT statement as shown in the preceding example, or it can be combined with other conditions. An example that combines the set of values with other conditions is:

```
SELECT IF REGION = 3 AND (INDUSTRY IN (410, 420, 425)
        OR INDUSTRY >= 450);
```

Records will be selected if they have REGION code 3 and INDUSTRY code greater than or equal to 450 or equal to 410, 420 or 425.

- **Sets of CHAR values**

If you are listing values for a CHAR variable and the values are all numeric, you do not need to enclose them in quotes. Non-numeric values must be enclosed in quotes. ***If they are shorter than the field width, you must enclose them in quotes and include any leading blanks or zeros.*** Otherwise, you will not get a match and nothing will be selected for these values. Trailing blanks need not be included in the quotes, but a good general rule is to make the value within the quotes have the same length and padding as the value in the data. The length should be the length given for the variable in the codebook. For example, if you have

```
INDUSTRY  CHAR  5
```

in the codebook, where some INDUSTRY values are less then 5 characters and filled on the left with blanks, an example of a correct SELECT statement for a set of INDUSTRY values is:

```
SELECT IF INDUSTRY IN ( '    410', '  4420', '66425' );
```

Compound Conditions

You can use compound conditions consisting of clauses separated by AND and OR. There is no limit to the number of compound conditions per statement. Conditions may be grouped by the use of parentheses to determine the order of evaluation. When parentheses are used, evaluation begins with the conditions contained in the inner-most sets of parentheses.

If the order of evaluation is not specified by parentheses, the expression is evaluated in the following order:

- arithmetic expressions
- relational operators
- AND and its surrounding conditions, starting at the left of the expression and proceeding to the right
- OR and its surrounding conditions, also proceeding from left to right

An expression such as:

```
SELECT IF A = B AND D > E OR F <=  WEIGHT * INCOME;
```

would be evaluated as:

```
SELECT IF ((A = B) AND (D > E)) OR
          (F <=  (WEIGHT * INCOME));
```

Selecting Data for a Specific Table

You can select data for a specific table by referencing that table in a SELECT statement. Multiple SELECT statements can be used to select different data for different tables. These statements can be in any order and can be inserted anywhere in the table request, either before or after the table statements to which they apply.

Example

```
USE CPS CODEBOOK;

SELECT IF YEAR = 2003;

TABLE ONE:
        HEADING TOTAL THEN RESIDENCE;
        STUB TOTAL THEN STATE_CODE;
TABLE TWO:
        HEADING TENURE;
        STUB HOUSEHOLD_SIZE;

SELECT FOR TABLE TWO IF INCOME > 40000;
SELECT FOR TABLE 3  IF EDUCATION IN (1, 4, 6);

TABLE THREE_A:
        HEADING TOTAL THEN SEX;
        STUB EDUCATION;
```

The first SELECT applies to all tables, so only data from the year 2003 will be included. In addition, TABLE TWO will be restricted to records with INCOME > 40000, and TABLE THREE_A (TABLE number 3) will be restricted to records with EDUCATION values of 1, 4, or 6.

Deleting Empty Columns

You may sometimes wish to use a variable in a table after selecting only certain values for that variable. If you use the variable in the table stub or wafer, the values that were not selected will "disappear" from the table. On the other hand, if you use the variable in the heading, you will get

columns for the values that were not selected in addition to those that were selected.

For example, if you had a data file that contained monthly data, you might want to tabulate data for only the first quarter of the year, January through March. Assuming that these months are coded 1 through 3 in the data file, you could select them with the statement,

```
SELECT IF MONTH >= 1 AND MONTH <= 3;
```

If you then use MONTH in a table stub, only the rows for January through March will appear in the table. The other MONTH rows will be "empty" because no data was selected for those rows, and TPL TABLES does not print empty rows unless you specifically request them.

TPL TABLES does the reverse with empty columns. It prints empty columns unless you specifically request that they be deleted. Thus, if MONTH is used in the table heading, you will get twelve columns, one for each month, even though you have selected data only for January through March.

A simple way to delete the empty columns is to use a FORMAT request with the statement

```
DELETE EMPTY COLUMNS;
```

With this technique, you can use the same table request for different groups of months by changing only the SELECT statement to choose the months you want. The empty columns for the other months will always be deleted and the table heading format will be adjusted automatically.

Selection Using the NUMBER and PERCENT Options

The SELECT percent and SELECT number options of the SELECT statement can be used to process a subset of your data without regard to the data values. Instead, they allow you to select a specific section of your data or a randomly selected percentage of your data. These options are especially useful when you have a very large data file, because they enable you to experiment with your table requests without processing all of the data.

Note The SELECT percent and SELECT number options apply to all tables in a request. Individual tables cannot be referenced.

Format

```
SELECT  number % ;
        number PERCENT ;
        number ;
        number START number;
        number record-name ;
        number record-name START number;
```

Examples

```
SELECT  10 %;
SELECT  20;
SELECT  20 START 200;
SELECT  20 MEMBERS;
```

SELECT Percent

The SELECT percent statement gives you a representative sample of your data. If you select 1% of your data then as each record is read, a function is applied which gives it a 1% probability of being selected. *Note that the exact records selected and even the exact number of records selected is not fixed.* In some cases, more than 1% of the records will be selected. In other cases, less than 1% will be selected. If you run the same job multiple times, you will get different numbers in your tables each time.

If your data file is hierarchical, selection is done at the highest level of the hierarchy. For example, if you have a hierarchical file of families and family members, then either a family and all of its members are selected or the family and its members are not selected at all.

SELECT Number

The SELECT number option takes the first records from the file. If you select 10 records, then the first 10 records will be processed.

If your file is a hierarchical file, for example, with family and member records, then the first 10 families and all of their members will be selected. If your SELECT statement is SELECT 10 MEMBERS; then exactly 10 members plus their family records will be selected. In this case, if the 10th member record in the file occurs before the last member record for its family, the members of the family that follow the 10th member record will not be processed.

The SELECT statement SELECT 10 START 5; will result in records 5 through 14 being selected.

Interaction Between Multiple SELECT Statements

The SELECT number and SELECT number START number statements are affected by other SELECT statements in the request. If there is a SELECT statement at a higher hierarchical level or if a SELECT statement at the same level occurs earlier in the request, then the SELECT number statement applies to records which pass the earlier SELECT.

Examples

Suppose that your data file contains information about persons, where there is one record per person, and that your table request contains the statements:

```
SELECT IF SEX = 'm';
SELECT 10 START 5;
```

Further suppose that the first 20 records have a sex of female and the next 20 records have a sex of male. TPL TABLES will exclude the first 20 records because they fail to pass the first SELECT. It will then skip records 21 to 25 because of the START clause. It will include records 26 through 35 and exclude the remainder of the file.

If the SELECT statements are reversed, records 1 through 5 will be excluded by the START clause. Records 6 through 15 will pass the SELECT 10 clause but will fail the SELECT IF SEX = 'm' condition. Records 16 to the end of the file will be excluded by the SELECT 10 clause. Thus no records will pass the two SELECT statements.

CHAPTER 9

Define

RECLASSIFYING DATA BY DELETING, REGROUPING, AND REORDERING VARIABLE VALUES

With the DEFINE statement, you can create a new control variable based on the values of one or more existing variables and assign labels to the new categories. The old variables can be described in the codebook or created by COMPUTE statements. The new variable definition can regroup, reorder or delete old variable values. Variables created with DEFINE can be used in TABLE statements.

Suppose that AGE is a codebook control variable with possible values of one through 99. If we use AGE directly in a TABLE statement, we will get totals for each year of age. We can use the DEFINE statement to get totals for any age groups instead of for single years. For example, we may wish summaries for the following three age groups.

from 1 to 15 years
from 16 to 25 years
over 25 years

The DEFINE statement could be:

```
DEFINE AGE_GROUPS ON AGE;
        '1 to 15 years'         IF  1:15;
        '16 to 25 years'        IF  16:25;
        'Over 25 years'         IF  > 25;
```

where AGE_GROUPS would be substituted for AGE in the TABLE statement to get the desired new classifications.

In another example, using the codebook observation variable Income, we create a control variable which groups income into four categories:

$0 to $12,000
$12,001 to $19,000
$19,001 to $30,000
over $30,000

The DEFINE statement for this grouping is:

```
DEFINE INCOME_GROUPS ON INCOME;
        '$0 to $12,000'          IF  < 12001;
        '$12,001 to $19,000'     IF  12001:19000;
        '$19,001 to $30,000'     IF  19001:30000;
        'Over $30,000'           IF  > 30000;
```

DEFINE ON A SINGLE VARIABLE

The examples above define new categories based on a single variable. We begin by describing this type of DEFINE statement. A later section of the chapter describes DEFINE statements based on multiple variables.

Any DEFINE that can be done based on a single variable can also be done using the multiple-variable type of DEFINE, but the single-variable DEFINE is simpler and more efficient when only one variable needs to be referenced.

Format

```
DEFINE new-variable-name ['var label'] ON old-variable-name;
                                                    :

[condition-name-1]  ['print label'] IF [re] value-entry-1;
                                                :
[condition-name-2]  ['print label'] IF [re] value-entry-2;
                                                :
          .                  .
          .                  .
[condition-name-n]  ['print label'] IF [re] value-entry-n;
                                                :
```

A **value-entry** can be any of the following:

- value
- condition name (if old variable is CONTROL)
- value1 : value2
- OTHER
- ALL
- NULL

If a **value-entry** does not have a name or label assigned to it, it is grouped into the first category above it that does have a name or label.

The optional **re** stands for any relation symbol or the equivalent English as shown below. A relation symbol can precede any value. If no relation is provided, "equal" is assumed.

Symbol	English expression
<	[IS] LESS THAN
>	[IS] GREATER THAN
=	[IS] EQUAL [TO] EQUALS
^<	[IS] NOT LESS THAN
^>	[IS] NOT GREATER THAN
^=	[IS] NOT EQUAL [TO]
<=	[IS] LESS THAN OR EQUAL [TO]
>=	[IS] GREATER THAN OR EQUAL [TO]

Examples

```
DEFINE INCOME_FILTER ON INCOME;
        'Less than $9,000'       IF  < 9000;

DEFINE OVERLAP_RANGES 'Income Categories' ON INCOME;
        'Less than $2,000'       IF  < 2000;
        'Less than $4,000'       IF  < 4000;
        'Less than $6,000'       IF  < 6000;

DEFINE CERTAIN_STATES ON STATE;
        'IDAHO AND INDIANA' IF  3;
                            IF  15;
        MARYLAND            IF  21;

DEFINE INCOME_RANGES ON INCOME;
        'Below $10,000 and above $25,000'     IF  < 10000;
                                              IF  > 25000;
        'From $10,000 to less than $25,000'   IF  10000 TO < 25000;
```

```
DEFINE REORDER_REGION_CODES ON REGION;
        'Southwest'      IF  SOUTHWEST;
        'Southeast'      IF  SOUTHEAST;
        'Northwest'      IF  NORTHWEST;
        'Northeast'      IF  NORTHEAST;

DEFINE ALPHA_GROUPS ON ALPHA_CODE;
        'G and T and E '         IF  'G';
                                 IF  'T';
                                 IF  'E';
        'X through Z'            IF  'X' : 'Z';
        'A through D'            IF  < 'E';
        'Other codes'            IF  OTHER;

DEFINE SOME_STATES ON STATES;
        ''          IF  NOT 27:42;

DEFINE DUMMY_SPANNER ON REGION;
        'All Industries'   IF  ALL;
```

Description of the DEFINE Statement

The DEFINE statement can be thought of as a two-column table. The new variable and its entries are on the left; the old variable and its entries are on the right.

The first row begins with the word DEFINE, the new variable name and its optional label. The new variable is always a control variable. Following the new variable information is the word ON, then the name of the the old variable. The old variable can be a codebook variable of type control, observation or char, or it can be a computed variable. It cannot be a variable created by a DEFINE statement or POST COMPUTE statement.

If you provide a label for the new variable, it will be displayed in tables where the variable is used and will span above the categories for the new variable. There are many options associated with print labels such as upper and lower case letters, special characters and footnotes. A separate chapter describes print labels in more detail.

There can be multiple DEFINE statements within a table request but each new variable must be created before it is used in a TABLE statement. If a computed variable is used as the old variable in a DEFINE statement, the COMPUTE statement must precede the DEFINE statement.

The same old variable can be used in any number of DEFINE statements provided unique new variable names are used.

After the new and old variables are named, there are rows of entries that assign old variable values into categories for the new variable. The order of the entries in the statement determines the order of display in a table. The left and right entries are separated by the word IF or a colon(:). Each entry on the right is followed by a semicolon.

The following lists the valid entries for the DEFINE statement:

Old Variable Entries

A semicolon is required after each entry in the old variable column. An entry can be any of the following:

1. An old variable value, for example **1000** or **'F'**.

   If the old variable is a **control** variable, condition values that are not all numeric must be surrounded by quote marks according to the rules for listing values in the codebook.

   If the old variable is a **char** variable, values must be surrounded by quote marks and must include any leading blanks or zeros.

2. A codebook condition name, for example **MALE**.

3. A relation followed by a value or name, where relation is any of those listed earlier in this chapter. An example is **< 25**.

   Note on char variables: The only relations that should be used with **char** variables are equal '=' and not equal '^='. If other relations are used with char variables, the results are unpredictable.

4. A range of old variable values specified as **value1 : value2** where value1 is less than or equal to value2. For example, if **2:5** is specified, all values not less than 2.000... and no greater than 5.000... will be accepted. A range of values can be non-numeric, in which case each lower and each upper value must be surrounded by quote marks; an example is **'A':'D'**. The keyword **TO** can be used in place of **:** to separate lower and upper range values. An example is **2 TO 5**.

   The NOT operator can precede a range of values, e.g. NOT 2:5, means any value that is either less than 2.000... or greater than 5.000....

The relations > and < can also be used in ranges in certain circumstances. The valid formats are listed below. The value **m** must always be less than the value **n**. If the old variable is an observation, the values can contain decimal points.

```
[NOT]       m : n;
          > m : n;
            m : < n ;
          > m : < n ;
```

5. ALL, meaning all values of the old variable.

6. The word OTHER, meaning all values for the old variable which are not specified by any other entry. ALL and OTHER may not be used together in the same DEFINE statement. Since ALL specifies all values of the old variable, OTHER used with ALL would specify nothing.

7. If the old variable is an observation variable, the word NULL can be used to create a category that counts null values. Null values will not be counted in any other type of defined category, including ALL or OTHER.

New Variable Entries

1. An optional **condition name** which you assign to one or more values of the old variable. Each value or range of values which apply to a condition name must appear in succeeding entries preceded by either IF's or colons, and followed by semicolons. Each condition name in the new variable column is considered unique, even if two or more are identical. The condition name will be used as a print label if no other label is provided.

2. An optional **print label**. Labels can contain upper and lower case letters, special characters and footnotes. A separate chapter describes print labels in more detail. Each label in the new variable column is considered unique, even if two or more are identical.

3. A **condition name followed by a print label**. In this case the print label is used in the table in place of the condition name.

4. Some entries as above and some entries blank. A blank entry will be grouped into the category of the nearest entry that precedes it and has a name and/or label.

5. All entries blank. In this case, each entry will define a new category and a label will be generated for each category. The format of the label will be "n variable name", where the name is taken from the newly defined variable. The number n starts with "1" for the first entry, and increments by one for each successive entry. To use this option, **all** entries must be blank. You cannot have some blank entries with generated labels and some entries with assigned condition names or labels.

If condition names and/or labels are assigned, the first must appear to the left of the first old variable entry and will apply to all following old variable entries until another condition name or label appears.

Note on Value Order in Relations and Ranges

If the old variable is a control variable and has been described in the codebook using the default order DISPLAY AS LISTED, then all relations with values or ranges of values must be based on the order of the conditions listed, rather than the sort sequence of the condition values. For example, consider the following control variable entry:

```
REGION CON 1
  (
        SW      =  'D'
        NW      =  'C'
        SE      =  'B'
        NE      =  'A'
  )
```

Unless the DISPLAY AS SORTED clause is used, the region value of 'A' is considered to be greater than 'D' since 'A' is listed after 'D'. A DEFINE statement used to combine "SE" and "NE" into one classification would have to express the condition as 'B':'A' or as > 'C'.

Referencing Values Not Listed in the Codebook

If the old variable is control and has display order "as listed", DEFINE entries for that variable must refer to valid values for that variable. Otherwise, the DEFINE statement will not be processed. For example, if AGE with values 16 to 99 is to be recoded with a DEFINE statement, an entry under AGE which specifies >100 would be invalid. Since AGE never has a value greater than 100, that entry specifies an empty set of values. An entry of >90 would, however, be acceptable since some valid values would be included in that specification.

If the old variable is control and has display order "as sorted", a reference to an invalid value in a DEFINE statement will be noted with a message when the DEFINE statement is processed.

Grouping Values with DEFINE

One purpose of the DEFINE statement is to group old variable values so that they fall into one or more new categories. For example, in the case of an observation variable INCOME, we may wish to group INCOME values in the following way:

```
DEFINE NEW_INCOME ON INCOME;
'Less than $15,000'     IF  < 15000;
'Exactly $15,000'       IF  15000;
'More than $15,000'     IF  > 15000;
```

The new control variable NEW_INCOME can be used in TABLE statements to tabulate into the three new income groups.

Similarly, codebook control variables or char variables can be grouped. For example, we may wish to group states according to geographical location.

Reordering Values with DEFINE

The default display order for control variables described in the codebook is the order in which the conditions are listed in the codebook. If the DISPLAY AS SORTED clause is used, the values will be displayed according to the collating sequence of the variable values; that is, a control variable with values from 1 through 10 will be displayed in numerical order, regardless of the order of listing in the codebook. Likewise, if DISPLAY NUMERIC is used, the values are displayed in numeric sort order.

The DEFINE statement can be used to change the order specified in the codebook by listing the values under the old variable name in the order they are to be displayed. Condition names or labels for each of the old variable values would be assigned under the new variable column.

Excluding Values with DEFINE

All possible values of the old variable need not be included in a DEFINE statement. Thus, in addition to regrouping and reordering, the DEFINE statement can be used to delete old variable values by omitting them from the old variable column. This type of DEFINE statement can be used to get the same effect as a SELECT FOR TABLE statement, but it can also be applied to only a part of a table rather than the whole table.

A simple example would be the case where only male heads of households are to be tabulated. The codebook sex code could be defined on as follows, where MALE is a condition name.

```
DEFINE MALES ON SEX;
'MALE HEAD OF HOUSEHOLD'  IF  MALE;

TABLE MALES_ONLY:
        WAFER  MALES,
        stub  expression,
        heading  expression;
```

The new control variable, MALES, has the effect of filtering out female heads of households from the table. Note that MALES could have been nested into the stub expression or the heading expression with the same effect, except for the location of the print label in the table.

Many DEFINE statements combine the functions of regrouping, reordering, and deleting in one statement.

The COPY Option for Using Labels from the Codebook

The COPY option allows control variable conditions to be copied exactly as they appear in the codebook to form DEFINE statement conditions. The keyword COPY appears to the left of the keyword "IF" or "EACH". The keyword "EACH" is preferred with COPY because it suggests that even with a range of values a condition will be formed for every item in the range. The conditions to be copied are expressed the same as for other DEFINE entries including ranges, single values, relational operators, and the optional use of condition names. The keywords "ALL" and "OTHER" may not be used with COPY.

Unless "DISPLAY AS SORTED" is used with a codebook control variable, the ranges used with "EACH" or "IF" must match the order listed in the codebook, and each condition will be copied in that order.

The COPY option can appear before, after, or interspersed among other DEFINE entries, and more than one COPY can be used within a DEFINE.

Example

It is desired to reclassify the codebook control variable STATE so that summaries are produced for all states, then for each of four Midwestern states, followed by a combined summary for those four Midwestern states. Condition labels for each of the Midwestern states are to be copied from the codebook.

```
DEFINE SOME_STATES ON STATE;
        'All States'          IF      ALL;
        COPY                  IF      13;
        COPY                  EACH    29:31;
        'Midwestern Total'    IF      13;
                              IF      29:31;
```

Using SOME_STATES in the stub expression of a TABLE statement would result in:

```
All States.....
Indiana.....
Michigan.....
Missouri.....
Ohio.....
Midwestern Total...
```

where Indiana=13, Michigan=29, Missouri=30, and Ohio=31.

Tip on Using Value Lists from the Codebook

In a DEFINE statement, there is another way to take advantage of the codebook information for a CONTROL variable if you list the values in the codebook in a form that is compatible with the DEFINE statement.

In the codebook, use the word **IF** in the value list instead of **=** and add **;** after each entry. For example:

```
INDUSTRY CONTROL 2
  (
        'Oil & Gas'           IF  'A1';
        'Steel'               IF  'B4';
        'Automobile'          IF  'C3';
        'Wood Products'       IF  'D1';
  )
```

If you expect to select subsets of a variable using a DEFINE statement, this format can help you, because you can use your editor to copy the codebook description for the variable into your table request and then simply delete the entries that you don't want. The format for the entries you retain will match the format required for the DEFINE entries, so you will not need to do any additional editing in your table request.

Applications

For these applications, suppose that we are using an input data file which has one record per person containing observations about weekly income and hours worked per week, along with control variables such as age, sex, state, and region.

Example 1 We are interested in tabulating hours worked observations for certain categories of income. In the following DEFINE, each person in the data file will fall into a category based on the person's income.

```
DEFINE INCOME_GROUP 'Income brackets' ON INCOME;
        '0 to $4,999'          IF  0:4999;
        '$5,000 to $10,000'    IF  5000:10000;
        'Over $10,000'         IF  > 10000;
```

The colon within old variable entries indicates an inclusive range from the value on the left through the value on the right. The range represents a single classification.

The above DEFINE statement is equivalent to the following.

a. If INCOME is within the range 0 through 4999, then INCOME_GROUP gets the name '0 to $4,999' which is the category defined by that range.

b. If the INCOME category is from 5000 through 10000, then INCOME_GROUP gets the name '$5,000 to $10,000'.

c. If INCOME is greater than 10000, then that classification of INCOME_GROUP gets the name 'Over $10,000'.

A TABLE statement using the defined variable might be:

```
TABLE WORKTIME:
        STUB  INCOME_GROUP,
        HEADING  HOURS BY SEX;
```

where the stub would begin with the variable label, 'Income brackets', followed by the stub labels of the three condition names. The first line of the table would contain the total hours worked by sex for all those whose income is between 0 and 4999. The second line would contain hours worked for income between 5000 and 10000, etc.

Example 2 Suppose that it is desired to group together as one category the states California, Texas, Illinois, and New York. In effect we want these four states to be treated as a single classification. Assume that each state has been set up as a condition name in the codebook.

```
STATE CON 2
 (
        ALABAMA      = 1
                . . .
                . . .
        WYOMING      = 50
 )
```

We will define a new variable called POPULATION_RANK.

```
DEFINE POPULATION_RANK ON STATE;
        LARGE_STATES        IF  CALIFORNIA;
                            IF  TEXAS;
                            IF  ILLINOIS;
                            IF  NEW_YORK;
        OTHER_STATES        IF  OTHER;
```

Each entry to the right of an 'IF' must be either a single value or a range of values. In this example, since we have four discrete values, each of the four must appear to the right of the 'IF'. Since no condition names appear to the left of the "IF", these states will be associated with LARGE_ STATES, along with California.

OTHER is a reserved word which collects into one category all old variable values not defined elsewhere in the DEFINE statement. All State codes other than the four will be assigned collectively the name OTHER_ STATES when the variable POPULATION_RANK is used in a TABLE statement.

The TABLE statement

```
TABLE RANK:
        STUB  POPULATION_RANK,
        HEADING  SEX BY INCOME;
```

would have a stub label of LARGE STATES (with, to its right, the income total for the four large States broken down by sex), followed by the stub label OTHER STATES (with, to its right, the income total for all other States, broken down by sex). No spanner label will print above the first stub label.

Example 3 A geographical region code appears in the codebook with the possible values of 'A', 'B', 'C', and 'D'. We wish to produce a table in which one entry reflects a total for all regions (U.S.), followed by a total combining region codes 'A' and 'B', followed by a total for 'A' and 'D', and finally a total for 'A', 'C', and 'D'.

We now define a new control variable as follows:

```
DEFINE REGION_GROUP 'REGION GROUPINGS' ON REGION;
        'U.S.'                  IF  ALL;
        EASTERN                 IF  'A':'B';
        NORTHERN                IF  NOT 'B':'C';
        'ALL BUT SOUTHEAST'     IF  'A';
                                IF  'C':'D';
```

When old variable values are not numeric they must be bounded by quote marks; however, the range symbol (:) can still be used between non-numeric values.

NORTHERN is assigned all values for REGION except 'B' and 'C', that is 'A' and 'D'. Note that the old variable ranges for different entries can overlap.

ALL is a reserved word which means collectively all values listed in the codebook for REGION, i.e. 'A', 'B', 'C' and 'D'. The reserved words ALL or OTHER may appear as entries, but they may not be used together in the same DEFINE statement.

The order of the condition names displayed in a table will be the order in which they are listed in the DEFINE statement, starting with 'U.S.' and ending with 'ALL BUT SOUTHEAST'. If we would like totals for all regions to appear last in the table, then the 'U.S.' entry would have to be specified last in the DEFINE statement.

A Technique for Working with Alphanumeric Codes

Assume that a variable occupies five character positions of which the first and fifth are alphabetic and the rest numeric.

```
AnnnA
BnnnB
CnnnN
DnnnB
```

There are many possible codes, but we want to use only a few. It would be inconvenient to describe this variable as a control variable in the codebook, because we would need to list all possible values. It cannot be described as an observation variable since it is not all numeric. We can, however, assign it a type of **char** in the codebook:

```
PRODUCT_CODE  CHAR  5
```

Then, if we want to use selected values in a table, we can pick them out with a DEFINE statement. For example:

```
DEFINE  SELECTED_CODES ON PRODUCT_CODE;
        'Gloves'          IF      'A114B';
        'Hats, straw'     IF      'B325A';
        'Shirts, nylon'   IF      'D327N';
        'Shirts, cotton'  IF      'D425B';
```

Tip on Using NOT in DEFINE

One use of the DEFINE statement can lead to unexpected results as shown in the following example where the DEFINE statement is intended to reclassify all values other than 'A' and 'C' into a single category.

```
DEFINE NEW ON OLD;
'Not A or C'     IF NOT 'A';
                 IF NOT 'C';
```

The entries in this statement are really equivalent to a specification of 'ALL'. Each old variable entry is assumed to be joined by 'OR', so an incoming 'A' fails the first test but passes the second and thus would be included. An incoming 'C' would pass the first condition and thus be included. However, NOT 'A' : 'C' is acceptable if there is no 'B' value. Also, two separate DEFINE statements, each containing one NOT condition, could be nested together to get the same results.

The combination of a Conditional Compute and DEFINE could be used as follows:

```
COMPUTE NEW ON OLD;
        1  IF 'A';
        1  IF 'C';
        0  IF OTHER;

DEFINE FILTER ON NEW;
'Not A or C'  IF  0;
```

Another option would be to use the type of DEFINE statement described in the next section of this chapter. The statement would be:

```
DEFINE NEW;
'Not A or C'      IF OLD NOT = 'A' AND OLD NOT = 'C';
```

DEFINE ON MULTIPLE VARIABLES

DEFINE statements that reference multiple variables provide much more flexibility than DEFINEs on a single variable. They are similar in syntax to Select Style Conditional Compute statements but differ in two important ways. First, they create new control variables with categories instead of computed observation values. Second, in a conditional compute, the testing ends with the first test that succeeds, whereas in a DEFINE on multiple variables, all tests are evaluated so that values can go into multiple categories .

Although this type of DEFINE is called "DEFINE on multiple variables", any DEFINE that can be done based on a single variable can also be done using the multiple-variable type of DEFINE.

Format

```
DEFINE new-variable-name ['var label'];
[condition-name-1]  ['print label'] IF test-1;
                                    :
```

```
[condition-name-2]  ['print label'] IF test-2;
                                        :
          .                  .
          .                  .
[condition-name-n]  ['print label'] IF test-n;
```

A **test** can be any of the following:

- condition-test
- ALL
- OTHER

where a **condition-test** is one or more comparisons between variables and values connected by **AND** or **OR** with parentheses as needed. Sets of values and computations can be included. Except for defined or post computed variables, any type of variable can be referenced in a test. Each test can reference entirely different variables. Any test which is valid for the SELECT statement is valid for this type of DEFINE statement.

If OTHER is used, it must be the last entry. If there is a test with no label or condition to its left, then values passing the test will be grouped with the category above it.

Note that if there is a computation error such as a divide by zero in a test, the test results cannot be predicted.

Example

```
Define Insurance_Group;
"Safe"              if  AGE > 30;
"Moderately Safe"   if  AGE > 25 and SEX = 'M';
                    if  AGE >= 20 and SEX = 'F';
"Unsafe"            if  AGE < 20 or (AGE < 25 and SEX = 'M');
"All Drivers"       if  ALL;
```

Example

In this example, people have been asked to check one or more of several reasons why they like their neighborhood. We would like to know how many people checked each reason and also have a total of the number of respondents in the survey. The following DEFINE statement will provide a single new variable with all of the categories we want. All people will be counted in the first (Total) category and also in one or more of the other categories.

```
DEFINE REASONS;
'Total'          IF ALL;
'Arts/Culture'   IF ARTS = 1;
'Parks'          IF PARKS = 1;
'Shopping'       IF SHOPS = 1;
'Housing'        IF HOUSES = 1;

TABLE D2 'What do you like about your neighborhood?':
        HEADING REASONS;
        STUB TOTAL THEN GENDER;
```

What do you like about your neighborhood?

	Total	Arts/Culture	Parks	Shopping	Housing
Total	592	220	378	193	200
Female	362	127	231	104	119
Male	230	93	147	89	81

Example In the next example, we select data for the last quarter of one year and the first quarter of the next, creating a category for each month and a total for each quarter

```
DEFINE QUARTERLY ;
/"Oct." IF YEAR = 2008 and MONTH = "Oct";
"Nov." IF YEAR = 2008 and MONTH = "Nov";
"Dec." IF YEAR = 2008 and MONTH = "Dec";
/"4th Qtr. 2008" IF YEAR = 2008 AND
          MONTH IN ("Oct", "Nov", "Dec");
/"Jan." IF YEAR = 2009 and MONTH = "Jan";
"Feb." IF YEAR = 2009 and MONTH = "Feb";
"Mar." IF YEAR = 2009 and MONTH = "Mar";
/"1st Qtr. 2000" IF YEAR = 2009 AND
          MONTH IN ("Jan", "Feb", "Mar");

Table ONE "Bushels of grain exported for the last two quarters:
STUB  QUARTERLY;
HEADING CORN then SORGHUM then BARLEY then OATS;
WAFER BUSHELS;
```

Bushels of grain exported for the last two quarters

	Corn	Sorghum	Barley	Oats
Oct.	173,544,585	19,139,313	6,926,280	8,493,134
Nov.	172,416,779	23,241,936	4,189,716	11,489,270
Dec.	167,570,732	26,546,006	7,700,744	6,998,313
4th Qtr. 2008	513,532,096	68,927,256	18,816,741	26,980,717
Jan.	158,402,211	25,348,172	5,426,203	6,812,804
Feb.	145,507,999	24,945,411	3,668,906	9,290,378
Mar.	156,266,042	25,926,515	3,832,875	5,993,486
1st Qtr. 2009	460,176,252	76,220,098	12,927,985	22,096,668

CHAPTER 10

Compute

COMPUTING NEW VARIABLES

Format COMPUTE new-variable ['print label'] [USING MASK mask] = computation;
USING :
MASK

where **new-variable** is an **observation** variable.

The optional **print label** following the variable name replaces the variable name on the printed table. There are many options associated with print labels such as upper and lower case letters, special characters and footnotes. A separate chapter describes print labels in more detail.

The optional **mask** is used to specify the print format for the computed data values if the computed variable is used in a TABLE statement.

Examples

```
COMPUTE WEIGHTED_INCOME USING MASK $999,999 =
        INCOME * WEIGHT;

COMPUTE NO_FAMS 'Number of Families'
        USING MASK 9999 = FAMILIES;

COMPUTE AMOUNT = INCOME**2 / (4.5 + WEIGHT);

COMPUTE FACTOR = .571;
```

Introduction

The COMPUTE statement provides a way of creating a new observation variable which has not been defined in the codebook. The new variable is calculated from other observation variables (including a record name) and

numeric literals using addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (**).

The calculations are performed using the normal rules for evaluation order and parentheses. Unless parentheses are used to change the evaluation order, exponentiation is performed before division and multiplication, which, in turn, are performed before addition and subtraction. Strings of operations at the same level are performed left to right. For example, 10 - 5 + 6 is evaluated as (10 - 5) + 6 = 11, not as 10 - (5 + 6) = -1.

A special division operator called **DIV** is available for performing integer division, and the **SQRT** and **ABS** functions can be used to get square roots and absolute values.

Computed variables are always considered to be observation variables and can be used in following TPL statements in any place that codebook-defined observation variables can be used. A computed variable is aggregated over the entire file when used in a TABLE statement.

COMPUTE statements are executed in ANSI standard double precision floating point.

Compute Entries

A computation can reference numeric literals and observation variables from either the codebook or previous COMPUTE or Conditional Compute statements. The numeric literals can contain an actual decimal point. Parentheses can be used in the computation to any level. Computed values aggregated over the entire file are rounded, if necessary, just before being displayed. An optional mask to the left of the equal sign indicates the number of decimal places and special symbols to be displayed.

The statement:

```
COMPUTE AMT 'Expenditure Amount' =  ((INTEREST + 3)  /
        (GROSS - INTEREST)) * .5;
```

is a valid COMPUTE statement if INTEREST and GROSS are observation variables from either the codebook or a previous COMPUTE or Conditional Compute statement. A computed variable can be set equal to a constant value or to another observation variable.

For example:

```
COMPUTE A = 5;
COMPUTE B = + 3;
COMPUTE C =  -95;
COMPUTE FAMILY_INCOME = INCOME;
```

The "new-variable" being computed cannot have the same name as any other variable created or used within the table request. If it has the same name as a variable in the codebook named by the USE statement, references to that name will be assumed to refer to the computed variable rather than the codebook variable.

For example:

```
COMPUTE WAGES = INCOME  / 50;
```

If a variable called WAGES has already been created in an earlier statement, an error will be reported.

If **division by zero** is attempted, a warning message will be issued and the quotient will be assigned a value of zero.

If any variable used in a computation has a **NULL** value, then the computed variable will be assigned a NULL value. For example, if you specify DATA ERROR = NULL when describing the variable INCOME in the codebook, and you then use INCOME in a COMPUTE statement, the computed result will be NULL for any record that has a data error for the variable INCOME. NULL values are not included in tabulations.

Absolute Value

You can obtain the absolute value of any expression within a computation by enclosing the expression in parentheses and preceding the left parenthesis with the keyword **ABS**, as in:

```
COMPUTE AMT 'Weighted Income' = ABS (WEIGHT  * INCOME);
```

Square Root

Square root can be obtained by following the keyword **SQRT** with a computation within parentheses. If square root is attempted on a negative value, a warning message will be issued and a value of zero will be returned.

OBS

The OBS function changes a character variable into an observation variable which can be used in a computation. The conversion discards leading blank space. It then supports an optional plus ("**+**") or minus ("**-**") sign followed by numbers including an optional decimal point ("."). Note that comma cannot be used as a decimal point. When a non-number or the end of the field is encountered, the conversion is terminated.

"123.456" is converted to 123.456.
"123,456" is converted to 123 since the comma terminates the conversion.
"+ 12345" is converted to 0 since there is a blank following the "+".
"-123x45" is converted to -123 since the 'x' terminates the conversion.

Integer Division

If you want division to be performed in integer mode so that all decimal places are truncated for each computed result, then use the **DIV** operator in place of the divide symbol (/). For example, 3 DIV 4 is zero, whereas 3/4 is 0.75. A DIV by 1 will simply truncate the decimal places. For example, 2.688 DIV 1 gives a result of 2.

Note

The DIV function can only guarantee 11 to 12 digits of accuracy. The reason for this is that TPL has code which prevents possible larger errors when data values are converted to floating point in the computer. Suppose the data value is really 7.000000000000000 but because of data conversion errors the number is represented in the computer as 6.999999999999999. If we do a straight truncation for DIV we will get an incorrect number, 6, even at the integer level. TPL corrects for this by adding a small number to the value before truncation. Hence the value becomes something like 7.000000000000754. Now if we truncate to integers or even to 7.0000000, we get correct numbers but if we truncate to 7.0000000000007 we get an incorrect value. TPL opts to give up a few digits of accuracy to prevent errors from showing up in higher digits.

Masks for Output Formatting

The COMPUTE statement can also contain a concise expression of how the computed variable is to be formatted when printed. This expression, known as a mask, consists of a succession of 9's, one for each digit position of the expected maximum aggregated value. Embedded within the 9's can be commas, a decimal point, dollar sign, or percent symbol in positions where they would appear when the computed variable is displayed.

For example, a mask of 9,999.99 would cause a cell value of 2467.34 to be displayed as 2,467.34 and be centered within the column width. Without

the mask it would be displayed as 2,467 right-justified within the column width. A dollar sign can appear before the leftmost 9 or a percent symbol(%) can appear after the rightmost 9 to cause these symbols to be printed.

A footnote reference can optionally be included in the mask. If a text is provided for the footnote in a SET FOOTNOTE statement, the associated footnote symbol will precede all cell values containing the computed value. Please refer to the chapter on Footnotes for complete details on the use of footnotes in masks.

```
COMPUTE FAMILY_INCOME USING MASK $999,999.99 =
        (HEAD_INC + OTHER_INC) / 100;
```

The total of HEAD_INC and OTHER_INC, which are recorded in cents, are to be aggregated for all families and displayed in dollars and cents. A dollar sign, comma, and decimal point are also to be printed. Since the incomes are to be expressed in dollars and cents, the totals must be divided by 100 to move the decimal point two positions to the left. Note that if HEAD_INC and OTHER_INC were desribed in the codebook with the clause SHIFT DECIMAL LEFT 2, we would not need to divide by 100 in order to display the incomes in dollars and cents.

If FAMILY_INCOME is used in the heading expression of a TABLE statement, the mask applies to each entry under FAMILY_INCOME; however, dollar signs, and percent symbols only appear with the first non-empty cell in the column. Since a mask is used, the data is automatically centered within the column width based on the number of symbols in the mask.

If FAMILY_INCOME is used in the stub expression, the mask applies to each FAMILY_INCOME row.

Data can be rounded and displayed with trailing zeros by inserting zeros in the mask. For example, with a mask of 999,000 the value 876859 will be displayed as 877,000.

For additional details on the use of masks, see the chapter on Masks.

Weighting

A common need in statistical processing is to weight various observed values in a data record which represent a sampling. Typically, each processing unit contains a weighting factor to be applied to the entries before they are tabulated, so that the final table values represent a larger universe.

The simplest example of weighting is creating a table with weighted frequency counts. This does not require a COMPUTE statement. The weighting factor observation variable is nested in the TABLE statement so that weighting factors replace the default observation variable of record name which has a value of one. In the following example, the weight variable WGT is nested with all cells of that table and tabulated for each cell.

```
TABLE W1 'Weighted tabulation of population' :
        HEADING  WGT BY REGION;
        STUB  MARITAL_STATUS BY EDUCATION;
```

To obtain a weighted tabulation other than a frequency count, a COMPUTE statement is needed. A new weighted variable can be created by multiplying the variable to be weighted by the weight factor. For example,

```
COMPUTE WEIGHTED_INCOME = INCOME * WEIGHT_FACTOR;

COMPUTE WEIGHTED_COST = COST * WEIGHT_FACTOR;
```

where, INCOME, COST, and WEIGHT_FACTOR are codebook observation variables. WEIGHTED_INCOME and WEIGHTED_COST are computed for each record. They can then be used in a TABLE statement such as:

```
TABLE T1:  STUB  AUTO BY (REGION THEN TOTAL),
        HEADING  WEIGHTED_INCOME THEN WEIGHTED_COST;
```

The computed weighted values will be aggregated from each record.

THE CONDITIONAL COMPUTE STATEMENT

Introduction

An extension of the COMPUTE statement can be used to create an observation variable for which the computation varies depending on the values of one or more other variables. The new variable will be assigned the computation associated with the first condition satisfied. This type of COMPUTE statement is called Conditional Compute.

There are two types of Conditional Compute. We call the first type "Select Style" Conditional Compute, because any condition that can be expressed in the SELECT statement can be used in this type of Conditional Compute. We call the second type "Define Style" Conditional Compute, because its form is similar to that of a DEFINE statement. The Define Style statement provides a short-cut approach that can be used when the choice of computations depends on the values of only one variable.

Select Style Conditional Compute

Format

```
COMPUTE new-obs-var ['print label'] [USING MASK mask] =
                                          USING
                                          MASK

computation-1  IF condition-1A [AND condition-1B...];
NULL           :               OR

[computation-2 IF condition-2A [AND condition-2B...]; ]
NULL           :               OR
       .              .  .
       .              .  .
[computation-n  IF  OTHER; ]
NULL           :
```

The keyword 'IF' and the colon can be used interchangeably.

Examples

```
COMPUTE WEIGHTED_INCOME  =
        WEIGHT * INCOME    IF  INCOME > 20000;
        INCOME             IF  OTHER;

COMPUTE TEST_ZERO_DENOMINATOR  =
        EARNINGS / HOURS   IF  HOURS > 0;
        NULL               IF  OTHER;
```

```
COMPUTE AMT3 USING MASK 999 =
        A       IF  L > 25.55;
        L       IF  OTHER;

COMPUTE CHILDREN USING MASK 9999 FOOTNOTE(C) =
        PERSONS_IN_FAMILY - 2  IF
                MARITAL_STATUS = 1 AND SEX = MALE;
        PERSONS_IN_FAMILY - 1  IF
                MARITAL_STATUS  IN  (1, 2, 3, 5);
                0  IF  OTHER;           /* Never Married and */
                                        /* Not available */
```

Condition Term

The variables used to the right of the 'IF' in the Conditional Compute can be either **control** variables (but not a defined variable), **char** variables or **observation** variables (but not a post computed variable).

The conditions are expressed identically to the IF form of the SELECT statement. Conditions can test for **relationships** or **sets of values**. All conditions which are valid for the SELECT statement are valid for the Conditional Compute. Each condition can reference entirely different variables.

If there is a computation error such as a divide by zero in a condition, the test will fail and the evaluation will go on to the next condition.

Compute Term

The entries to the left of the 'IF' can be:

1. numeric literals and observation variables from the codebook or computed variables. A record name cannot be used.

or

2. the keyword NULL as explained later in this chapter.

As a first example, suppose that the control variable PAY_TYPE contains either an 'H' or a 'W' to indicate whether the observation variable EARNINGS is stored as an hourly wage rate in cents or a weekly salary in dollars. To create a new observation variable WEEKLY_SALARY to be displayed in dollars and cents, we could write:

```
COMPUTE  WEEKLY_SALARY USING MASK  $999.99 =
    (EARNINGS * USUAL_WEEKLY_HRS)/100  IF  PAY_TYPE = 'H';
    EARNINGS/100                       IF  PAY_TYPE = 'W';
```

Note that if EARNINGS had been described in the codebook with the clause SHIFT DECIMAL LEFT 2, we would not need to divide by 100 in order to display the results in dollars and cents.

The ordering of the entries is important in the Conditional Compute. The conditions are evaluated in the order in which they are specified. ***The new variable will be assigned the computation associated with the first condition satisfied.***

Consider an expansion of the last example:

```
COMPUTE WEEKLY_SALARY USING MASK $999.99 =
    (EARNINGS * USUAL_WEEKLY_HRS)/100  IF  PAY_TYPE = 'H';
    EARNINGS/100                       IF  PAY_TYPE = 'W';
    EARNINGS                           IF  PAY_TYPE = 'H';
```

If PAY_TYPE equals 'H', WEEKLY_SALARY will be set equal to the first computation and no more testing will be done. Although 'H' occurs more than once, any computation other than the one associated with the first occurrence of 'H' will be ignored.

If none of the conditions are satisfied, then the new variable will be assigned the value of zero. Thus in the above example, if PAY_TYPE='H', then the new variable WEEKLY_SALARY will contain the value of the computation expressed by (EARNINGS * USUAL_WEEKLY_HRS)/100. If PAY_TYPE='W', then WEEKLY_SALARY will contain the value EARNINGS/100. If neither condition is satisfied, WEEKLY_SALARY will be assigned the value zero.

If PAY_TYPE cannot take values other than 'H' and 'W', the statement could have been written using 'OTHER' as in:

```
COMPUTE  WEEKLY_SALARY USING MASK $999.99 =
    (EARNINGS * USUAL_WEEKLY_HRS)/100  IF  PAY_TYPE = 'H';
    EARNINGS/100                       IF  OTHER;
```

In the Conditional Compute statement at least one computation or numeric literal must appear in the first entry in the column under the new variable. If subsequent entries do not contain a computation on the left, they will be associated with the previous computation. For example, if we need a new variable **WEIGHT** which varies depending on the value of the variable

STATE, we could use the following statement. Codebook condition names associated with STATE are used.

```
(codebook)
STATE CON 2
  (
        ALABAMA = 1
                . . .
                . . .
        WYOMING = 50
  )

(request)
COMPUTE WEIGHT =
        1       IF  STATE = Arizona;
                IF  STATE = Utah;
                IF  STATE = Nevada or STATE = New_Mexico;
        3       IF  STATE = California;
                IF  STATE = New_York;
        2       IF  OTHER;
```

WEIGHT will have a value of 1 if STATE = Arizona, Utah, Nevada or New Mexico, and a value of 3 if STATE = California or New York. All other state values will cause weight to have a value of 2.

The same computation can be repeated in different entries. For example, to create WEIGHT we might wish to list the states in alphabetic order on the right as follows:

```
COMPUTE WEIGHT =
        2       IF  STATE = Alabama;
        2       IF  STATE = Alaska;
        1       IF  STATE = Arizona;
        2       IF  STATE = Arkansas;
        3       IF  STATE = California;
        2       IF  STATE = Colorado;
        2       IF  OTHER;
```

An observation variable can be tested and also used in the computation. For example:

```
COMPUTE NEW_WEIGHT =
        WEIGHT          IF  WEIGHT < 25;
                1       IF  OTHER;
```

A conditionally computed variable can be used anywhere that an observation variable can appear, including in another Conditional Compute statement.

If the computation associated with the first condition satisfied contains a null value, the new variable value for that record will be null-valued.

Define Style Conditional Compute

If your calculations depend on the values of a single variable, you can use an abbreviated form of Conditional Compute that looks similar to a DEFINE statement. This type of Conditional Compute works the same as the SELECT style but has a more simple format.

Note that Define style Conditional Compute cannot be used if the **old-variable-name** references a variable of type CHAR.

Format

```
COMPUTE new-obs-var ['print label']  [ USING MASK mask] ON old-variable;
                                       USING
                                       MASK
                    computation-1  IF  entry -1;
                    NULL           :

                    [computation-2 IF  entry-2; ]
                    NULL           :
                     .             .
                      .            .
                    [computation-n IF  OTHER; ]
                    NULL           :
```

The keyword IF and the colon can be used interchangeably.

The **old-variable** value is compared to the values in the entries below it. The first match will determine the value to be assigned to the **new-obs-var**.

Examples

```
COMPUTE WEIGHTED_INCOME  ON  INCOME;
        WEIGHT * INCOME          IF  > 20000;
        INCOME                   IF  OTHER;

COMPUTE TEST_ZERO_DENOMINATOR  ON  HOURS;
        EARNINGS / HOURS         IF  > 0;
        NULL                     IF  OTHER;
```

```
COMPUTE WEEKLY_SALARY USING MASK $999  ON  PAY_TYPE;
        EARNINGS * WEEKLY_HRS    IF  'H';
        EARNINGS                 IF  'W';
        EARNINGS / 2             IF  'B';
```

Entries on the Right

The entry to the right of an IF can be any value or range of values, with or without relation symbols, that is valid for a DEFINE statement. The word ALL cannot be used. If the word OTHER is used as an entry, it must be the last entry.

Computations on the Left

An entry to the left of the IF can be a computation or the value NULL as allowed in the Select style Conditional Compute.

Assigning NULL Values

A null value can be assigned to a conditionally computed variable by using the keyword NULL on the left side of a condition. NULL values are not included in tabulations.

Null assignments can be used to eliminate invalid values from computations such as averages, medians and other quantiles. These invalid values may come directly from the data file or may be the result of computation errors, such as "divide by zero".

Suppose that we want a table with average family income values and median income values. Assume that INCOME is five bytes long and that values of zero or 99999 are to be treated as null values so that erroneous average or median calculations are avoided. We can use the following statements to exclude null INCOME values:

```
COMPUTE VALID_INCOME ON INCOME;
        INCOME          IF  > 0 TO < 99999;
        NULL            IF  OTHER;

COMPUTE VALID_FAMILY ON INCOME;
        1         IF  > 0 TO < 99999;
        NULL    IF  OTHER;

POST COMPUTE AVERAGE = VALID_INCOME / VALID_FAMILY;

MEDIAN MEDIAN_INCOME ON VALID_INCOME (4);
```

```
TABLE SAMPLE:  STUB   REGION,
       HEADING   AVERAGE THEN MEDIAN_INCOME;
```

If a computation specified in a COMPUTE statement results in a division by 0 or the square root of a negative number, TPL TABLES produces an error message and assigns a value of 0 to the computation. If the computed variable is used directly in a TABLE statement this action is probably acceptable. On the other hand, if the computed variable is used in another computation or define, the result is probably not what is desired.

In such a case you should replace the COMPUTE with a Conditional Compute which produces the result you want. For example, if HOURS can sometimes have the value 0, the COMPUTE statement

```
COMPUTE RATE = COST / HOURS;
```

should be replaced by the Conditional Compute

```
COMPUTE RATE       ON     HOURS;
       NULL        IF     0;
       COST/HOURS  IF     OTHER;
```

Then you will get correct results for calculations such as:

```
MEDIAN MEDIAN_RATE ON RATE(10);
```

If a null-valued observation variable is referenced in a DEFINE statement, the null values will be counted only if there is a specific entry that specifies NULL as a category. Null values will not be counted in any other of the defined categories, including ALL or OTHER.

In SELECT and Conditional Compute statements, no test involving a null-valued variable will succeed unless it specifically references NULL. For example, if variable "A" has a null value, then "A = 5" will not be satisfied. Similarly, "A NOT = 5" will not be satisfied. If all of the variables referenced in a Conditional Compute are null-valued, all of the tests will fail and the newly computed variable will take on the value associated with the "OTHER" category if one is provided; it will take on the default value of zero if no "OTHER" category is specified.

The keyword NULL can be explicitly used in the SELECT statement and on the right side of a Conditional Compute statement. For example, suppose that there are two income variables called MONTH_EARNINGS and

WEEK_EARNINGS. Suppose also that one of the two incomes can be null-valued. We want to compute a new variable EARNINGS which is expressed in monthly earnings. We could use a statement like the following:

```
COMPUTE EARNINGS =
        MONTH_EARNINGS  IF  MONTH_EARNINGS NOT = NULL
                AND  WEEK_EARNINGS = NULL;
        (52 / 12) * WEEK_EARNINGS IF
                WEEK_EARNINGS NOT =  NULL
                AND  MONTH_EARNINGS = NULL;
        NULL IF  OTHER;
```

NULL or Zero for OTHER

If there is no OTHER category in a Conditional Compute, then the computed variable is assigned a value of zero for records that do not meet any of the conditions. In most cases, this will be acceptable treatment. However, in the case where the computed variable is to be used in computations such as averages or in QUANTILE or MEDIAN statements, you will probably want to specify "NULL IF OTHER" to eliminate the possibility of including unwanted zero values in the calculation.

In general, processing will be more efficient if you add the condition "NULL IF OTHER" at the end of the Conditional Compute. When the computed variable is used in a table, cells that have only NULL contributions will be treated the same as empty cells. A dash will be displayed in these cells as it would be for any other situation where there is no value for the cell. If you prefer that a zero value be displayed in the cells that have only NULL contributions, you should use the default assignment of zero or explicitly specify "0 IF OTHER" at the end of the Conditional Compute.

A Technique for Computing Ratios

A useful technique with the Conditional Compute is to compute ratios of classifications within a control variable, also using the POST COMPUTE statement (see POST COMPUTE chapter). Suppose a codebook control variable entry is expressed as:

```
EMPLOYMENT_STATUS CON 1
  (
        EMPLOYED     =  1
        UNEMPLOYED =  2
  )
```

If we wish to compute a ratio of UNEMPLOYED to EMPLOYED workers, we can write the following statements.

```
COMPUTE EMPLOYED ON EMPLOYMENT_STATUS;
        1       IF   1;
        NULL    IF   OTHER;

COMPUTE UNEMPLOYED ON EMPLOYMENT_STATUS;
        1       IF  2;
        NULL    IF  OTHER;

POSTCOMPUTE RATIO USING MASK 99.9 =
                                UNEMPLOYED / EMPLOYED;
```

Then, within a TABLE statement, we might have:

```
TABLE SAMPLE_RATIOS:
HEADING  INDUSTRY BY
        (TOTAL THEN EMPLOYMENT_STATUS THEN RATIO);
STUB  CITY;
```

The table would show the total number of workers, the employed, the unemployed, and the ratio of unemployed to employed for each city.

CHAPTER 11

Post Compute

COMPUTING NEW VARIABLES ON FINAL TABULATED VALUES

Format

```
POST COMPUTE new-var ['print label'] [USING MASK mask] =
POSTCOMPUTE                          USING
                                     MASK
        computation;
```

where **new-var** is an observation variable

The optional **print label** following the variable name replaces the variable name on the printed table. There are many options associated with print labels such as upper and lower case letters, special characters and footnotes. The Labels chapter describes print labels in more detail.

Examples

```
POST COMPUTE AVG_INC 'Average Income' MASK $99,999 =
        INCOME / PERSONS;

POST COMPUTE PERCENT_INCREASE =
        ((AMT_1983 - AMT_1982)/AMT_1982) * 100;

POST COMPUTE AMOUNT USING $99,999.99 =
         EXPENDITURES/100;

POST COMPUTE FACTOR  =  (LEN **2  - SQRT(ALEN *
        (BLEN + (CLEN-EXPEN)))) / .456;
```

The purpose of the POST COMPUTE statement is to compute cell values based on variables aggregated over the entire file. While the COMPUTE statement results in a computation on each processing unit, the POST COMPUTE computation is not done until each observation variable value within the computation is accumulated over the entire file. For example, to

produce averages within a table, the arithmetic operation of division would not be done until the amounts and counts from each record were accumulated over the entire file.

Post Compute Entries

Each variable used in the computation must be an observation variable, and each will be aggregated as a final total before the computation takes place. The observation variable can be a record name. The computation can also include the MAX and MIN built-in functions which are described below.

The observation variables used in the calculation of a new variable do not need to be displayed in the table. All terms valid in the COMPUTE statement are valid in the POST COMPUTE.

If the POST COMPUTE involves taking the square root of a negative value, or if division by zero is attempted, the cell value will be displayed as '**' with a footnote at the end of the table '** Computation error'. This built-in footnote can be changed or suppressed. Please refer to the section on footnotes for complete details.

Numeric literals appearing in a Post Compute expression are not aggregated but are used in the final computation. All computations are done in ANSI standard double precision floating point.

It is permissible in a POST COMPUTE to include a variable created in a preceding COMPUTE or POST COMPUTE statement. Post Computed variables must be created prior to their use in a TABLE statement.

For any table cell containing a Post Computed value, if any variable used in the Post Compute computation has only null values for that cell, the result of the Post Compute will be a null value.

MAX

MAX is an operator which is used in POST COMPUTE statements only. Its form is 'MAX(var)' where the argument, 'var', is any single observation variable. The observation variable can come from the codebook or from a computed variable. The contribution of MAX(var) to the Post Compute is the largest value of 'var' from any record which contributes to a cell.

Let us assume that the first condition value of REGION is 'Northeast', the first of SEX is 'Male', and that INCOME is an observation variable. Now consider the following example:

```
POST COMPUTE DOUBLE_HIGH_INCOME = 2 * MAX (INCOME);

TABLE T1:  REGION BY SEX,  DOUBLE_HIGH_INCOME;
```

The value occurring in the first row of the table will be twice the highest income of any member of the data set who is male and lives in the Northeast.

MIN

MIN is an operator which follows the same rules as MAX except that the contribution of MIN(var) to the Post Compute for a cell is the smallest value of 'var' from any record which contributes to the cell.

Masks for Output Formatting

Masks can be used in the POST COMPUTE statement in the same way as in the COMPUTE statement to insert dollar signs, decimal points, footnotes, etc. Use of masks will cause the post computed values to be centered within the column widths based on the size of the mask. Masks are explained in more detail in the chapter called "Masks".

Sample Applications

Example

A table is to be produced consisting of a column of total income followed by a column of number of persons in each of three regions. It is desired to POST COMPUTE a third column of average income derived from the count of number of persons and total income for each region. Each person is described by a record named PERSONS which contains a region code, income amount, and years of schooling.

The required statements are:

```
POST COMPUTE AVERAGE_INCOME = INCOME / PERSONS;

TABLE A:
        REGION,
        INCOME THEN PERSONS THEN AVERAGE_INCOME;
```

	INCOME	PERSONS	AVERAGE INCOME
REGION=1	603,280	35	17,237
REGION=2	543,080	31	17,519
REGION=3	298,000	23	12,957

Example

It is desired to produce a table showing the count of persons, their total income, and total years of schooling for each of three regions. Separate wafers are to be produced for each of two age groups. We wish to POST COMPUTE average income and average years of schooling for each wafer.

The required statements are:

```
POST COMPUTE AVG_INCOME = INCOME / PERSONS;
POST COMPUTE AVG_SCHOOLING = SCHOOL / PERSONS;

TABLE B:
        WAFER  AGE,
        STUB  PERSONS THEN INCOME THEN AVERAGE_INCOME
                THEN SCHOOL THEN AVG_SCHOOLING,
        HEADING  REGION;
```

AGE=2

	REGION=1	REGION=2	REGION=3
PERSO			
INCOM			
AVERA			
SCHOO			
AVG S			

AGE=1

	REGION=1	REGION=2	REGION=3
PERSONS	35	31	23
INCOME	603,280	543,080	298,000
AVERAGE INCOME	17,237	17,519	12,957
SCHOOL	350	341	322
AVG SCHOOLING	10	11	14

Example

A table consisting of two wafers is to be produced. The first wafer is to contain a count of persons for each city and income class. The second wafer will contain the aggregated income for persons in each city and income class. We wish to POST COMPUTE a third wafer whose corresponding cells show their average income.

The required statements are:

```
POST COMPUTE AVG_INCOME = INCOME / PERSONS;

TABLE C:
    WAFER  PERSONS THEN INCOME THEN AVERAGE_INCOME,
    STUB  CITY,
    HEADING INCOME_CLASS;
```

PERSONS

	0 - 6 THOUSAND	7 - 10 THOUSAND	over 10 THOUSAND
BOSTON	10	24	38
CHICAGO	44	30	43

INCOME

	0 - 6 THOUSAND	7 - 10 THOUSAND	over 10 THOUSAND
BOSTON	4,000	219,300	1,264,260
CHICAGO	4,000	271,700	1,498,760

AVERAGE INCOME

	0 - 6 THOUSAND	7 - 10 THOUSAND	over 10 THOUSAND
BOSTON	400	9,138	33,270
CHICAGO	91	9,057	34,855

Standard Deviation

Example Standard Deviation is to be computed for all persons' income by using square root.

```
COMPUTE SQ_INCOME = INCOME ** 2;

POST COMPUTE ST_DEV_INCOME =
   SQRT (SQ_INCOME / PERSONS  - (INCOME / PERSONS) ** 2);

TABLE DEVIATION:   INCOME_CLASS,
         ST_DEV_INCOME BY (TOTAL THEN REGION);
```

Using Post Computed Variables in Post Computes

Post computed variables can be referenced in subsequent POST COMPUTE statements. In the following example, the first statement calculates the mean number of vehicles per household in the year 2000. The second calculates the mean for the year 1990. The third calculates the difference between the two means, and the fourth references both the 1990 mean and the difference between the means to calculate percent change.

Example

```
POST COMPUTE MEAN_VEHICLES_00
"Mean vehicles per household"  USING MASK 999,999.99 right
         = CARS_00  / HHLDS_00;

POST COMPUTE MEAN_VEHICLES_90
"Mean vehicles per household"  USING MASK 999,999.99 right
         = CARS_90  / HHLDS_90;

POST COMPUTE VEHICLES_DIFF
font h 8 "Change from 1990 - 2000"  USING MASK 999,999.99 right
         = (MEAN_VEHICLES_00) - (MEAN_VEHICLES_90);

POST COMPUTE VEHICLES_PCT_DIFF
font h 8 "Percent change from 1990 - 2000"
USING MASK 999,999.99 right
         = 100 * VEHICLES_DIFF / MEAN_VEHICLES_90;
```

The DISPLAY Function

A POST COMPUTE calculation is done using unrounded values. No rounding is done until just before the result is displayed in a table. If the inputs to the POST COMPUTE are displayed as rounded values in a table, you may occasionally see a small difference between the calculation based on unrounded values and the calculation based on displayed values. If you need to do the calculations based on rounded displayed data values, you can use DISPLAY.

The DISPLAY function can be used in a POST COMPUTE to convert an unrounded value into the same value as that which would be displayed in the table after rounding. Thus the result of the POST COMPUTE will be the same as a calculation using the values displayed in the table.

The function has two arguments: first, the variable or arithmetic expression for the value that was displayed with rounding; second, the number of digits following the decimal point in the mask that was used to display the rounded value.

Format

```
DISPLAY(arith-expr, n)
```

Example

```
POST COMPUTE CHGWG_PAST =
        DISPLAY(TOT_WAGES_PRES, 0) -
        DISPLAY(TOT_WAGES_PAST, 0);
```

Example

In the following table, columns 3 and 4 are obtained by post computing the difference between the values displayed in columns 1 and 2. The first difference is calculated from the unrounded values. The second difference is calculated using DISPLAY.

```
POST COMPUTE diff_regular 'Difference' mask 999,999.99 =
        pop_thousands_wgt - pop_thousands;

POST COMPUTE diff_display 'Difference using DISPLAY'
        mask 999,999.99 =
        DISPLAY(pop_thousands_wgt,2) - DISPLAY(pop_thousands,2);
```

Table D-1. Households by State (in thousands)

	Weighted Count of House-holds	Number Surveyed	Difference	Difference using DISPLAY
Total	46,333.47	30.00	46,303.47	46,303.47
New England				
Connecticut	682.18	0.33	681.84	681.85
Maine	241.43	0.29	241.14	241.14
Massachusetts	1,203.60	1.20	1,202.40	1,202.40
New Hampshire	211.67	0.27	211.40	211.40
Rhode Island	183.52	0.26	183.26	183.26
Vermont	105.19	0.26	104.93	104.93

In this example, the results of the two Post Computes are the same except in one row where the cells are shaded. There is a small difference of .01 between these cells.

Subtracting the unrounded values gives the following:

682.175080 - 0.334000 = 681.84108 which rounds to 681.84

Subtracting the displayed values give:

682.18 - 0.33 = 681.85

THE CONDITIONAL POST COMPUTE STATEMENT

Introduction

The Conditional Post Compute lets you adjust table cell values by testing tabulated values to see if certain conditions have been met. The first successful test for a table cell will determine the final cell value. Cell values can be replaced with post computed values and footnotes.

The general format is similar to the format of the Conditional Compute statement:

Format

```
POST COMPUTE new-obs-var ['print label'] [USING MASK mask] =

replacement1   IF  condition1   [AND condition2.....] ;
                                 OR
       .         .       .     .       .
       .         .       .     .       .
replacement-n  IF  condition-m [AND condition-p.....]  ;
                                 OR
```

Example

```
POST COMPUTE NEW_INCOME =
        20000           IF  INCOME > 20000;
        INCOME          IF  OTHER;
```

A replacement expression can be a numeric literal, a single observation variable, a computation or the keyword NULL. Any computation that is valid for a POST COMPUTE statement can be used in the Conditional Post Compute. Another example shows some valid entries, where each of the letters represents an observation variable.

```
POST COMPUTE NEW  MASK 99,999.99 =
        3.65 + A + B + MAX(C)       IF  E * F < (D - G);
        SQRT ((L * M)/N)            IF  H > 5.1;
```

A footnote reference or complete conditional print mask can be included at the end of the replacement expression. If a replacement expression ends with a print mask, that mask will be used in place of the mask associated with the post computed variable. If a replacement expression contains ONLY a print mask and does not provide a replacement value, the contents of the mask will be used in the cell. In this case, if there are 9's in the mask, a zero replacement value will be assumed.

Conditions to the right of "IF" can contain numeric literals, observation variables, including computed and post computed variables, computations, the keyword OTHER or the keyword NULL. All of the relational operators permitted in the SELECT statement can be used.

Sometimes the treatment of table cells cannot be determined simply by looking at tabulated values but instead depends on other aggregate properties. Additional operators are available for testing other aggregate properties of variables. These operators are discussed later under the heading "Testing Aggregate Properties with Status Variables".

Conditional Masks and Footnotes

The Conditional Post Compute can be used to assign conditional masks to cells. As an example, assume that we are aggregating expenditure amounts for different types of purchases. For large items such as automobiles or boats, the tabulated values may be quite large and could be displayed in hundreds of dollars; for small items such as candy or stockings, it might be more appropriate to display the tabulated values in dollars and cents. We can determine the cell format for each cell by testing the final tabulated values as follows:

```
POST COMPUTE DISPLAY_VALUE =
        AMOUNT / 10000 MASK 999 FOOTNOTE HUNDREDS
                                        IF  AMOUNT > 100000;
        AMOUNT MASK 999.99              IF  OTHER;
```

The Conditional Post Compute can be used for conditional footnoting, where a unique footnote is associated with each condition. For example:

```
POST COMPUTE FOOTNOTE_TEST MASK 999,999 =
        INCOME FOOTNOTE(A)      IF  INCOME < 100000;
        INCOME FOOTNOTE(B)      IF  INCOME >= 100000;
```

Note A REPLACE MASK statement in a format request will override a conditional mask applied to the same cells. This means that the conditional footnotes will be lost. To retain the conditional footnotes when replacing a mask, see **KEEP DATA FOOTNOTE** in the **FORMAT** section of the manual.

In TPL TABLES, footnotes for data cells are normally attached to masks. In the Conditional Post Compute, there are two special cases where a footnote reference can be used alone:

1. If the cell is to contain only a footnote symbol, the replacement expression can consist of just a footnote reference.

   ```
   POST COMPUTE NEW_INCOME 99,999 =
           INCOME                  IF  INCOME >= 20000;
           FOOTNOTE (A)            IF  OTHER;
   ```

   In this example, cells with tabulated INCOME values less than 20000 will contain only the symbol for FOOTNOTE A.

2. If the cell is to be footnoted, but will also contain a data value, a footnote reference can be used at the end of the replacement expression. In this case, the rest of the cell format will be determined by the mask associated with the post computed variable. In other words, the footnote reference will act as a supplement to the mask of the post computed variable.

```
POST COMPUTE NEW_INCOME 99,999 =
        INCOME                      IF  INCOME >= 20000;
        INCOME FOOTNOTE (A)         IF  OTHER;
```

   In this example, cells with tabulated INCOME values less than 20000 will contain values formatted with the MASK 99,999 and footnoted with FOOTNOTE A.

Tabulated values may need to be tested in situations where incomplete data causes division by zero in some cells, or where too few contributions to a cell can result in confidential information being revealed. Conditional Post Compute statements can be used to make the required adjustments by deleting, altering or footnoting cell values.

Suppose that we want to display average family incomes in all cells for which at least 10 families have made contributions to the average. If fewer than 10 families contributed to a cell, we want to replace the value with a footnote to avoid disclosure of confidential information. This can be done by specifying:

```
POST COMPUTE AVERAGE MASK $99,999 =
        INCOME / FAMILIES    IF  FAMILIES >= 10;
        FOOTNOTE(FEW)        IF  OTHER;
```

with the footnote statement:

```
SET FOOTNOTE(FEW) TEXT   'Fewer Than 10 Families Contributed';
```

Suppose that we use AVERAGE in a TABLE statement that has age of family head as the stub expression and geographical region in the heading.

```
TABLE SAMPLE:  AGE, AVERAGE BY REGION;
```

We can interpret the logic of the Conditional Post Compute as follows: The observation variables referenced in the Conditional Post Compute will be summarized for each table cell. After all data have been summarized, final aggregated amounts for INCOME and FAMILIES will be available in

each cell. If FAMILIES >= 10, an average will be computed for the cell. Otherwise, the cell will contain only the footnote symbol for the footnote FEW.

If we wish to compute the average for the "OTHER" condition but footnote it, we can do so by specifying the calculation followed by the footnote reference:

```
POST COMPUTE AVERAGE  MASK $99,999 =
        INCOME/FAMILIES                      IF  FAMILIES >= 10;
        INCOME/FAMILIES FOOTNOTE(FEW)   IF  OTHER;
```

Next, suppose that we want to avoid division by zero by replacing those cells having a zero divisor with a null value. The null value is equivalent to "data not available" and is displayed as a dash symbol. We can specify:

```
POST COMPUTE AVERAGE =
        INCOME / FAMILIES    IF  FAMILIES > 0;
        NULL                 IF  OTHER;
```

If an entire row of data contains null values, partially or fully because of null-valued Post Computes, the entire row will be treated as empty and by default will be deleted.

Suppose now that we want to display final income values that are at least $30,000. For cells less than $30,000 we want to display the word 'SMALL'. We can specify:

```
POST COMPUTE NEW_INCOME MASK $99,999 =
        INCOME          IF  INCOME >= 30000;
        MASK 'SMALL'            IF  OTHER;
```

In some cases, conditional footnoting or suppression of data values depends on the summarized values that contribute to a cell. For example, in calculating average family incomes, we might wish to replace average income with a footnote in cells where the family with the highest income has an income greater than 1/2 of the total for the cell. This we could accomplish simply with the conditional post compute:

```
POST COMPUTE AVERAGE_INCOME MASK $99,999.99 =
FOOTNOTE(LARGE_CONTRIBUTOR)
                        IF  MAX(INCOME) > 1/2 * INCOME;
INCOME / FAMILIES    IF  OTHER;
```

Status Variables

Sometimes, the treatment of cell values cannot be determined based on tabulated values. Instead, we may wish to footnote a cell if any one contributor to the cell has a special property or if all contributors have the property. To do this, we need to use a separate variable that we call a **status variable** in order to keep track of the occurrence of the special property. We can then test the status of cells after tabulation to decide whether footnotes are required.

Status tests can be applied using the two operators, **U for Union** and **I for Intersection**. The Union and Intersection operators treat numbers as strings of binary 1's and 0's. Thus, for example, the number 3 will be thought of as BIT '11' and the number 6 will be thought of as BIT '110'. For the special cases of 1 and 0, their representations are just BIT '1' and BIT '0' respectively.

Now suppose we wish to label a cell as revised if any data value contributing to the cell has been revised. Assume there is a field, REVISED, in our data file which contains 1 if the INCOME field is revised and 0 if it is not revised. We could now write our post compute as:

```
POST COMPUTE AVERAGE_INCOME =
INCOME/FAMILIES FOOTNOTE (NEW_RESULT)
                          IF  U(REVISED) = BIT '1';
INCOME/FAMILIES           IF  OTHER;
```

U(REVISED) will be equal to 1 if one or more records contributing to a table cell has a 1 in it.

Alternately we might want to footnote a cell as final if all of the cells contributing to it are final. If we have a field, FINAL, in our data file which contains 1 if income is final and 0 otherwise we can accomplish this with:

```
POST COMPUTE AVERAGE_INCOME =
INCOME/FAMILIES FOOTNOTE (FINAL_RESULT)
                          IF  I(FINAL) = BIT '1';
INCOME/FAMILIES           IF  OTHER;
```

In this case I(FINAL) will be 1 only if all of the records contributing to a cell have a FINAL value of 1.

Testing Aggregate Properties with Status Variables

More complicated requirements can be accommodated with more complex testing clauses. However, things become unmanageable as the number of conditions grows. To simplify the testing, you can include all status information in a single status variable. Up to 31 switches can be grouped into a single status variable that can be tested for a combination of results.

If your data file already contains fields with status switches in them, you can describe them in your codebook as observation variables and use them directly in conditional post computes. The fields can be 1 to 4 bytes long.

If your data file contains status information that is recorded in some other form, you can create a status variable in a conditional compute statement by assigning bit string values of 1 to 31 bits.

For example, suppose the data file has a control variable INCOME_STATUS which is coded with "r" for revised, "p" for preliminary and "b" for both. We can turn this into a usable status field with the following compute:

```
COMPUTE NEW_STATUS =
        BIT '01'        IF  INCOME_STATUS = 'r';
        BIT '10'        IF  INCOME_STATUS = 'p';
        BIT '11'        IF  INCOME_STATUS = 'b';
        BIT '00'        IF  OTHER;
```

The right-most bit is "on" (= 1) if INCOME is revised. The left bit is "on" if INCOME is preliminary. Both bits are "on" if INCOME is both revised and preliminary.

If this same status information were stored in two fields, REVISED and PRELIMINARY, we could get the same results with:

```
COMPUTE NEW_STATUS =
        BIT '01'        IF  REVISED = 1 AND PRELIMINARY = 0;
        BIT '10'        IF  REVISED = 0 AND PRELIMINARY = 1;
        BIT '11'        IF  REVISED = 1 AND PRELIMINARY = 1;
        BIT '00'        IF  OTHER;
```

We can now use NEW_STATUS in a post compute.

```
POST COMPUTE AVERAGE_INCOME =
        INCOME / FAMILIES FOOTNOTE(REVISE)
                                IF  U(NEW_STATUS) = BIT '01';
        INCOME / FAMILIES FOOTNOTE(PRELIM)
                                IF  U(NEW_STATUS) = BIT '10';
        INCOME / FAMILIES FOOTNOTE(P_AND_R)
                                IF  U(NEW_STATUS) = BIT '11';
        INCOME / FAMILIES    IF  OTHER;
```

In table cells where any contributing INCOME is revised, we will get the REVISE footnote; in cells where any contributing INCOME is preliminary, we will get the PRELIM footnote; and in cells where there is at least one occurrence of a revised INCOME AND at least one occurrence of a preliminary INCOME_R footnote.

Now suppose that instead of just 2 status conditions (revised and preliminary in the preceding example), we have 10 that need to be tested for certain combinations. In such a case, we would have 10 ** 2 = 100 possible status combinations. We probably do not want 100 different tests and footnotes. Instead we would like to test for only a few of the possible combinations, adding footnotes to only those cells with those attributes.

TPL TABLES lets you simplify the testing by using a **"don't care"** bit denoted by the letter **x**. The bit string BIT '0xx1' will compare as equal to BIT '0111' or BIT '0101' or BIT '0011' or BIT '0001'. In other words, it will compare as equal with any string of four bits that has 0 in the left-most bit and 1 in the right-most bit. It will not compare as equal if the left-most bit is 1 or the right-most bit is 0. The bits you "don't care" about will be ignored in the comparison.

If you compare a value to a bit string that is shorter than the value, TPL TABLES will assume that you don't care about the bits at the left end. For example, if you compare a 4-bit value to the string BIT '11', the result will be the same as if you had specified BIT 'xx11'.

Restrictions

The following is a summary of the restrictions and other rules which govern the behavior of the Conditional Post Compute.

1. Only observation variables can be referenced.

2. No further testing is done after the first condition is satisfied.

3. An IF condition test must always have a replacement expression to the left of the "IF". The following is invalid, because there is no replacement expression specified to the left of "IF R < 1.5;".

```
POST COMPUTE INVALID =
        1       IF  Y > 3;
                IF  R < 1.5;
        2       IF  OTHER;
```

4. If OTHER is used, it must be the last condition.

5. If a computation error (divide by zero, etc.) occurs in evaluation of a condition, no value will be post computed. All affected tables cells will be footnoted with an error message. If there is a computation error to the left of the "IF", all cells that fit the corresponding condition will be footnoted with an error message.

6. If conditionally computed values are used in percent calculations, footnotes or other aspects of masks associated with those values will be ignored. The mask for the percent variable will determine the format for the cell value.

   One exception to this rule occurs if a conditionally computed value has a built-in footnote for a missing value or a computation error. In that case, the missing (EMPTY) or error footnote will be displayed in the percent cell.

   Another exception occurs if the numerator for a percent cell is conditionally computed to have a mask with only a footnote. Then no percent will be calculated for that cell and the footnote symbol will be displayed instead.

7. Variables can be tested for null values, but if a null-valued variable is tested for a value other than null, the test will fail. For example, the test: x > 0 will fail if the value of x is NULL.

8. If no condition is satisfied, the new variable is set to zero.

CHAPTER 12

Percent

CALCULATING PERCENTS FROM TABULATED VALUES

Introduction

The percent features in TPL TABLES allow nearly complete flexibility in calculating and displaying percents in tables. You may specify whether only percents or both original cell values and percents are to be displayed.

In specifying percent calculations to be performed on a table, it is necessary to specify both where the percents should appear and what the base (i.e. denominator) of each percent calculation should be.

For any percent calculation, a PERCENT statement is required to create a **percent variable** to be referenced in a TABLE statement. We can specify a variety of percent calculations for a table through a combination of a percent variable and the use of that variable surrounded by the symbols "<" and ">" to indicate where the base cells are located. When a percent variable is surrounded by these symbols, it is called a **base marker**. Alternately, instead of using a base marker, we can specify the location of the bases in the PERCENT statement itself or at the beginning of the TABLE statement.

Percent Variables

A percent variable is created by the PERCENT statement. The general form of the PERCENT statement is:

Format

```
PERCENT variable-name  ['print label']  [USING [MASK] mask]
        [percent-location]  [base-location]  ;

[percent-conditions]
```

The optional percent-location is:

```
WHERE  [=]    STUB   (Default)
       [IS]   WAFER
              HEAD   (or HEADING)
```

The optional base-location is:

```
BASE  [=]   [FIRST]  ROW
      [IS]           COLUMN   (Default)
                     CELL
```

The format for the optional percent-conditions is:

```
[['print label'] [mask] : VALUE ;]
[['print label'] [mask] : PERCENT;]
```

The two percent-conditions can appear in either order. The percent condition can appear alone. If a mask is attached to a percent condition, it will override the mask associated with the percent variable.

Percent **conditions** are similar to conditions in other types of statements such as DEFINES, although there can be at most two of them. When both conditions are present for a percent variable referenced in a TABLE statement, there will be both original values and percents in the table.

Examples

An example of the most simple kind of PERCENT statement is:

```
PERCENT PCT;
```

An example of a PERCENT statement using all of the options is:

```
PERCENT PCT "Percent" MASK 999%
WHERE = STUB
BASE = FIRST ROW;
"Percents" MASK 999.99% : PERCENT;
"Counts" MASK 99,999 : VALUE;
```

The following simple table will be used as the basis for many examples throughout this chapter to illustrate how percent variables can be used in tables.

```
TABLE P_1:    AGE,
              REGION;
```

P 1

	U.S	North	South
All Ages	2,000	1,200	800
Under 21	600	400	200
21 and Over	1,400	800	600

Tables without Percent Markers

Percent base markers are used to specify which cells are to be used as bases. For simple cases, base markers are not necessary since the base locations can be specified within the percent statement itself. We will first look at some of these simple cases before moving on to tables requiring base markers.

Consider the following simple PERCENT and TABLE statements:

```
PERCENT P2 ""
        WHERE = HEAD
        BASE = FIRST COLUMN;

TABLE P_2 PERCENT P2:
        AGE,
        REGION;
```

P 2

	U.S	North	South
All Ages	100	60	40
Under 21	100	**67**	33
21 and Over	100	57	43

The percent variable name is included in the title line of the TABLE statement which turns the table into a table of percents. When a percent is used in the title line of a TABLE statement, the PERCENT statement usually include both a WHERE clause and BASE clause though the defaults may be relied on.

The PERCENT statement specifies that the BASE should be the first column so the values in this column are used as the denominators for the percent calculations. Since the percent variable does not have a label or conditions, WHERE=STUB would produce an identical table.

Note Format statements were used to shade the base cells to make the discussion easier. They have also been used to underline other cells. They are not required.

A percent cell is calculated by dividing its numerator by its base cell and then multiplying by 100. The numerator is the value the cell would have if the table did not have percent variables. For most tables the base for a cell is the nearest base cell which comes before or matches the percent cell.

Consider the underlined cell. From the previous table P_1 we know that the numerator for this cell should be 400. The base for this cell is the value that is in the first column of this row since this is the nearest base cell. Again from our previous table we see this is 600. So our percent cell is (400 / 600) * 100 = 67. Note that base cells which are also percent cells are always 100 percent since they are their own base cells.

If we change our PERCENT statement slightly so the bases are the first row we get:

```
PERCENT P3 ""
  WHERE = HEAD
  BASE = FIRST ROW;

TABLE P_3 PERCENT P3:
AGE,
REGION;
```

P 3

	U.S	North	South
All Ages	100	100	100
Under 21	30	33	25
21 and Over	70	67	75

Now suppose we want a table with both values and percents. To do this we use percent conditions:

```
PERCENT P4 ""
  WHERE = HEAD
   BASE = FIRST COLUMN;
"Value": VALUE;
"Percent": PERCENT;

TABLE P_4  PERCENT P4:
AGE,
REGION;
```

P 4

	U.S		North		South	
	Value	Percent	Value	Percent	Value	Percent
All Ages	2,000	100	1,200	60	800	40
Under 21	600	100	400	**67**	200	33
21 and Over	1,400	100	800	57	600	43

If we reversed the order of the conditions in the PERCENT statement, the percents would come first in the table.

The base for the underlined cell is 600, the first cell in its row. The numerator for the cell is 400, the same as the number from its associated **value** cell.

If we modify our PERCENT statement so the percent variable is in the stub we get:

```
PERCENT P5 ""
  WHERE = STUB
  BASE = FIRST COLUMN;
"Value": VALUE;
"Percent": PERCENT;
```

```
TABLE P_5  PERCENT P5:
AGE,
REGION;
```

P 5

	U.S	North	South
All Ages			
Value	**2000**	1,200	800
Percent	100	60	40
Under 21			
Value	600	400	200
Percent	**100**	67	33
21 and Over			
Value	1,400	800	600
Percent	100	57	43

Since the percent variable is in the stub, the **values** and **percents** are in alternating rows. The underlined **2000** is a cell which is a base but does not apply to any percent since there is no percent cell in its row. There is nothing wrong with this. In many cases we will not shade unused bases. The base for the underlined **100** is itself.

The next table shows what we get when the percent base is a single cell:

```
PERCENT P6 ""
        WHERE = HEAD
        BASE = FIRST CELL;
"Value": VALUE;
"Percent": PERCENT;

TABLE P_6  PERCENT P6:
AGE,
REGION;
```

P 6

	U.S		North		South	
	Value	Percent	Value	Percent	Value	Percent
All Ages	2,000	100	1,200	60	800	40
Under 21	600	30	400	20	200	10
21 and Over	1,400	70	800	40	600	30

Percents in Parts of Tables

Placing a percent variable in the title line of a table statement will result in an entire table of percents or alternating values and percents. In some cases we wish to have percents in only part of a table. We can do this by nesting a percent variable in the wafer, stub, or header of a table and using a base specification in the title line of the table.

```
PERCENT P7 "" mask right 99 %
  WHERE = HEAD
  BASE = FIRST ROW;

TABLE P_7  PERCENT BASE FIRST COLUMN:
AGE THEN P7 BY TOTAL,
REGION;
```

P 7

	U.S	North	South
All Ages	2,000	1,200	800
Under 21	600	400	200
21 and Over	1,400	800	600
Total	100%	60%	40%

When a percent is used in a table, the WHERE and BASE clauses are discarded. Percent conditions, if they exist, are retained. All cells nested with the percent variable are percent cells. The base cells are determined by the base specification in the table title line.

A mask with % has been added to the PERCENT statement for clarity.

In the next table we place the percents in the last column. Note that the bases are now in the first row. If we left them as the first column, then all of the percents would have 100%.

```
PERCENT P8 "" mask right 99 % ;

TABLE P_8  PERCENT BASE FIRST ROW;
        AGE,
        REGION THEN P8 BY TOTAL;
```

P 8

	U.S	North	South	Total
All Ages	2,000	1,200	800	100%
Under 21	600	400	200	30
21 and Over	1,400	800	600	70

It is possible in TPL to combine these tables to produce a table with both a row and column of percents as shown below. But this table requires use of both percent markers and multiple percents so we will delay discussing it until later.

P 9

	U.S	North	South	Total
All Ages	2,000	1,200	800	100%
Under 21	600	400	200	30
21 and Over	1,400	800	600	70
Total	100%	60%	40%	–

Base Markers

Using percents and base markers in the wafer, stub and header of a table provides more flexibility than just placing specifications in the title line. Unfortunately, it is also more complicated. We will begin by reproducing some of the tables we produced earlier and then move on to more complicated tables.

Consider first the following simple PERCENT statement which defines a percent variable called P:

```
PERCENT P;
```

This percent variable can be used in a TABLE statement along with a corresponding percent marker **<P>** to specify percent calculations.

Suppose we wish to specify a table of all percents using the first column (in this case, U.S.) as the base for the percents. In other words, we want to see regional percentages for each age group. We could specify:

```
TABLE  P_10:  AGE,
               P BY <P>REGION;
```

P 10

	P		
	U.S	North	South
All Ages	100	60	40
Under 21	100	67	33
21 and Over	100	57	43

This is the same table as P_2 except that the percent variable **P** does not have a blank label so **P** is shown. In this table specification, the percent variable P is nested with all cells in the table; thus all cells in the table contain percents. The variable P could have been nested with the stub expression or used in the wafer expression to indicate that the entire table contains percents. The percent marker <P> associated with the percent variable P appears directly in front of the variable REGION in the heading expression. This means that the first element of REGION is the base for the percent calculations. The first element of REGION describes the column of values under U.S. in the table. Each cell value for U.S. is the base for the percents in its row.

Now, suppose that we wish to specify the same table with all percents using the first row of the table as the base. In other words, we want age group percentages for each region category. We could specify:

```
TABLE  P_11:
       P BY <P>AGE,
       REGION;
```

P 11

	U.S	North	South
P			
All Ages	100	100	100
Under 21	30	33	25
21 and Over	70	67	75

This is the same table as P_3 except that the percent variable P does not have a blank label so **P** is shown. As in TABLE P_10, the percent variable P is nested with all cells of the table so that all cells contain percents. In TABLE P_11, however, the percent marker **<P>** precedes the variable AGE in the stub to indicate that the first element of AGE defines the bases. The

first element of AGE describes the row of values to the right of "All Ages" in the table. Each cell value for "All Ages" is the base for the percents in its column.

Use of Base Markers

The general rule regarding percent bases is that they are always applied from left to right, from top to bottom, or from front to back (i.e. through wafers) within a table. If you want to show the base in the rightmost column or bottom row, the base may be duplicated and the first base deleted using a DELETE statement in a FORMAT request.

Before proceeding with further examples, we consider more fully how the base marker works. A base marker associated with a particular percent variable is written as the name of the percent variable enclosed in "pointed brackets". For example, the marker associated with the percent variable P was written as <P>. For a percent variable named PER, the marker would be <PER>; for a percent variable named PCT, the marker would be <PCT>.

A base marker applies to the expression directly following it in a TABLE statement. If the base marker applies to an expression containing more than one variable, then parentheses must surround the expression to show the scope of the base marker's application. An example is:

```
TABLE P_12:  <P>(TOTAL THEN INDUSTRIES),
             P BY REGION;
```

P 12

	P		
	U.S	North	South
Total	100	100	100
White colar	45	46	44
Blue colar	55	54	56

Parentheses are required in the above TABLE statement to show that the base marker applies to the entire stub expression. All cells of the table will be percents since P is nested into the heading expression.

A base marker indicates that the first element of the "marked" expression will determine the bases for all percents in all cells nested with its associated percent variable. For example, in

```
TABLE P_13:  P BY <P>AGE,
             REGION;
```

P 13

	U.S	North	South
P			
All Ages	100	100	100
Under 21	30	33	25
21 and Over	70	67	75

the cells associated with the first element of AGE, "All Ages", are the bases for the table of percents. In the next TABLE statement the percent variable is nested only with SEX. The result is that the first element of AGE will serve as the bases for SEX but not REGION since REGION is not within the scope of the percent variable. In this way we can get tables of numbers and percents.

```
TABLE P_14:  <P> AGE,
             REGION THEN P BY SEX;
```

P 14

	U.S	North	South	P		
				Both sexes	Male	Female
All Ages	2,000	1,200	800	100	100	100
Under 21	600	400	200	30	33	27
21 and Over	1,400	800	600	70	67	73

This table could also be specified as:

```
Table P_14  PERCENT BASE FIRST ROW:
        AGE,
        REGION THEN P BY SEX;
```

We will now examine more tables which are similar to the first examples but show additional possibilities. In the following table of three variables we will first consider how the placement of the base marker in a table stub affects the types of percents we will get.

TABLE P_15: AGE BY SEX,
REGION;

P 15

	U.S	North	South
All Ages			
Both sexes	2,000	1,200	800
Male	900	550	350
Female	1,100	650	450
Under 21			
Both sexes	600	400	200
Male	300	150	150
Female	300	250	50
21 and Over			
Both sexes	1,400	800	600
Male	600	400	200
Female	800	400	400

With TABLE P_16, we give another illustration of percents calculated with the first row of the table used as the base:

TABLE P_16: P BY <P>(AGE BY SEX),
REGION;

P 16

	U.S	North	South
P			
All Ages			
Both sexes	100	100	100
Male	45	46	44
Female	55	54	56
Under 21			
Both sexes	30	33	25
Male	15	12	19
Female	15	21	6
21 and Over			
Both sexes	70	67	75
Male	30	33	25
Female	40	33	50

In this example, the placement of the percent marker in front of the expression (AGE BY SEX) specifies that the first element of (AGE BY SEX) will determine the base cells. The first element is "All Ages"-"Both Sexes", or, in other words, the first element of Age and the first element of Sex. *All cells nested with this element are base cells.* This includes each category of REGION.

The effect of removing the parentheses from the stub expression is to get the percent of each sex category within each age group as shown in the next table.

```
TABLE P_17:   P BY <P>AGE BY SEX,
              REGION;
```

P 17

	U.S	North	South
P			
All Ages			
Both sexes	100	100	100
Male	100	**100**	100
Female	100	100	100
Under 21			
Both sexes	30	33	25
Male	33	**27**	43
Female	27	38	11
21 and Over			
Both sexes	70	67	75
Male	67	**73**	57
Female	73	62	89

In TABLE P_17 the percent marker <P> precedes the variable AGE. Thus the first element of AGE determines the location of the bases for the percent calculations. Since all cells nested with the first category of AGE are also base cells, the first element of AGE, "All Ages", applies to all cells in the first three rows of the table. The values in the "All Ages" category are the set of bases for the corresponding values of "Under 21", and "21 and Over". For example, females 21 and over represent 62 percent of all females in the North region, and males under 21 represent 43 percent of all males in the South.

This is the first table which requires percent base markers. It is also rather confusing. Earlier we stated that "For most tables the base for a cell is the nearest base cell which comes before or matches the percent cell." This is a table for which this is not true. The underlined 100 is the base

for the other underlined cells even though it is not the nearest base to them. It would be better to rearrange the table:

```
TABLE P_18:  P BY SEX BY <P>AGE,
             REGION;
```

P 18

	U.S	North	South
P			
Both sexes			
All Ages	100	100	100
Under 21	30	33	25
21 and Over	70	67	75
Male			
All Ages	100	**100**	100
Under 21	33	**27**	43
21 and Over	67	**73**	57
Female			
All Ages	100	100	100
Under 21	27	38	11
21 and Over	73	62	89

The first condition of AGE is "All Ages". Each "All Ages" cell is a base. This includes the cells for each category of REGION and each category of SEX. The percent variable is nested into the stub expression, so the entire table will be percents.

As with the previous table, the underlined **100** is the base for the other underlined cells. But now the base and its related percents are near each other.

Nesting Percent Markers

It is possible to nest percent markers for a given percent variable to further restrict the number of base cells. For example, if we wished to specify a table of percents where all percents were calculated using the first cell of the table, we would specify:

```
TABLE P_19:  P BY <P>AGE,
             <P>REGION;
```

P 19

	Region	
	North	South
P		
All Ages	100	67
Under 21	33	17
21 and Over	67	50

In this case, we say that the bases are determined by the first element of Age, "All Ages", and the first element of Region, "U.S.". Since only one cell of the table is "All Ages"-"U.S.", i.e., the upper left-hand cell of the table, all cells of the table will be divided by this cell's value in calculating percents.

Tables of Original Values and Percents

So far we have emphasized tables in which only percents were shown or tables in which there are alternating rows or columns of numbers and percents. However, it is often the case that we would like to see some or all of the original values as well as the percents in more varied arrangements. For example we might like to see the original values of only the base cells along with the percents. All such possibilities can be accommodated through the use of the usual table operations. For a simple example of a table with both original values and percents, we can specify TABLE P_20 to display alternating columns of original values and percents as follows:

```
TABLE  P_20:  AGE,
       <P>REGION BY (TOTAL THEN P);
```

This table could also be written using percent conditions as:

```
PERCENT  PC;
       "Total" : VALUE;
       "P" : PERCENT;

TABLE P_20: AGE:
       AGE,
       <PC>REGION BY PC;
```

As before, the first element of Region will determine the bases, but only the cells nested with the percent variable P will contain percents. These include the cells formed by P, REGION, and AGE.

P 20

	U.S		North		South	
	Total	P	Total	P	Total	P
All Ages	2,000	100	1,200	60	800	40
Under 21	600	100	400	67	200	33
21 and Over	1,400	100	800	57	600	43

We could specify a block of original values followed by a block of percents as follows:

```
TABLE P_21:  AGE,
             (TOTAL THEN P) BY <P>REGION;
```

Alternately we could write this as:

```
TABLE P_21:  AGE,
             PC BY <PC>REGION;
```

P 21

	Total			P		
	U.S	North	South	U.S	North	South
All Ages	2,000	1,200	800	100	60	40
Under 21	600	400	200	100	67	33
21 and Over	1,400	800	600	100	57	43

Or, suppose we wish to show original values for only the bases.

```
TABLE P_22:      AGE,
                 TOTAL THEN P BY <P>REGION;
```

P 22

	Total	P		
		U.S	North	South
All Ages	2,000	100	60	40
Under 21	600	100	67	33
21 and Over	1,400	100	57	43

Perhaps Region has only the values "North" and "South", but we wish to duplicate the results of the preceding table. We could then specify:

```
TABLE P_23:  AGE,
             TOTAL THEN P BY <P>(TOTAL THEN REGION);
```

According to this specification, the first element of (TOTAL THEN REGION) will determine the bases for the percents. Since the first element is TOTAL, we will have exactly the same result as in the preceding table (only with slightly different labels in the heading):

P 23

	Total	P		
		Total	Region	
			North	South
All Ages	2,000	100	60	40
Under 21	600	100	67	33
21 and Over	1,400	100	57	43

Using Percents with Different Observation Variables

It is important to examine more closely how the percent variable works. The percent variable initially represents values that form the numerator for calculating the percentages. These numerator values are then replaced with the actual percentage. **If the percent variable P is not nested with an observation variable, the numerator for calculating percents will be frequency counts.** If the percent variable has conditions, the VALUE will just be a count and the numerator of the PERCENT will also be a count.

To illustrate these points, we go back to the table P1 from the beginning of the chapter. Assume for this example that each record in the data represents one family. There are no percents in table P1. It simply counts the number of families according to region and the age of the head of the household.

P 1

	U.S	North	South
All Ages	2,000	1,200	800
Under 21	600	400	200
21 and Over	1,400	800	600

Now assume also that each family record has a variable called PERSONS that contains the number of persons in the family. If we wish to see the number and percent of persons in each region, then we might accidentally specify it as follows:

```
PERCENT P 'Percent' USING MASK 999.9%;

TABLE P_24  'P_24 - Wrong Usage of Percent':
        AGE BY <P>(PERSONS THEN P),
        <P>REGION;
```

P 24 - Wrong usage of percent

	U.S.	North	South
All Ages			
Persons	5,250	3,300	1,950
Percent	**38.1%**	22.9%	15.2%
Under 21			
Persons	2,450	1,700	750
Percent	24.5%	16.3%	8.2%
21 and Over			
Persons	2,800	1,600	1,200
Percent	50.0%	28.6%	**21.4%**

The use of the two base markers in the TABLE statement identifies the percent base as PERSONS within each category of AGE for the U.S. category of REGION. Since P is nested (crossed) with only the control variable AGE, the numerators for the percents will be family frequency counts which appear in table P1 above. These numerators will be replaced by percents calculated using PERSONS as the base. Thus, the percents are based on the ratio of family frequency counts to PERSONS aggregations. The results are not what we intended.

For example, the top left underlined "percent" is 2000 families/5250 persons*100 = 38.1 when we actually intended it to be 5250 persons/5250 persons = 100%. The bottom right underlined "percent" is 600 families/2800 persons = 21.4 when we actually intended it to be 1200 persons/2800 persons = 42.9%.

We could try:

```
PERCENT P25 "";
        "Value" MASK 99: VALUE;
        "Percent" MASK 999.9% : PERCENT;

TABLE P_25 :
         AGE by P25,
          <P25>REGION;
```

P 25

	U.S.	North	South
All Ages			
Value	23	12	11
Percent	100.0%	52.2%	47.8%
Under 21			
Value	15	8	7
Percent	100.0%	53.3%	46.7%
21 and Over			
Value	8	4	4
Percent	100.0%	50.0%	50.0%

This is a valid table but it shows FAMILY percents rather than PERSONS percents. Changing the label "Value" to "Persons" would not change the numbers.

If we wish to see percents of persons in each region, then PERSONS must be nested with P to make PERSONS the numerator for the percents as well as the denominator. This may be done by either adding PERSONS to the wafer expression, nesting PERSONS into the heading expression, or by writing the stub expression as follows:

```
TABLE P_26:   AGE BY PERSONS BY <P>(TOTAL THEN P),
              <P> REGION;
```

P 26

	U.S.	North	South
All Ages			
Persons			
Total	5,250	3,300	1,950
Percent	**100.0%**	62.9%	37.1%
Under 21			
Persons			
Total	2,450	1,700	750
Percent	100.0%	69.4%	30.6%
21 and Over			
Persons			
Total	2,800	1,600	1,200
Percent	100.0%	57.1%	**42.9%**

We can actually produce a table with both FAMILY and PERSONS precents with the following:

```
PERCENT P27 "":
        "" mask 9,999 : VALUE;
        "Percent" mask 999.9% : PERCENT;

TABLE P_27:
        AGE_GP BY PERSONS BY P27 THEN
        AGE_GP BY FAMILY BY P27,
        <P27>REGION;
```

P 27

	U.S.	North	South
All Ages			
Persons	5250	3300	1950
Percent	100.0%	62.9%	37.1%
Under 21			
Persons	2450	1700	750
Percent	100.0%	69.4%	30.6%
21 and Over			
Persons	2800	1600	1200
Percent	100.0%	57.1%	42.9%
All Ages			
Families	23	12	11
Percent	100.0%	52.2%	47.8%
Under 21			
Families	15	8	7
Percent	100.0%	53.3%	46.7%
21 and Over			
Families	8	4	4
Percent	100.0%	50.0%	50.0%

No label was assigned to VALUE in the PERCENT statement so it would get its label from the variable nested above. This enabled the same percent to be used for both PERSONS and FAMILY. To make the table look more natural, format statements were added to line up PERSONS, PERCENT, and FAMILY. These involved making the stub indent smaller and adding indents to the start of PERSONS and FAMILY.

Multiple Percent Variables within a Table

It is also possible to specify more than one type of percent distribution within the same table through the use of more than one percent variable. To combine the percent calculations of tables P2 and P3 (first column as base; first row as base), we could create two percent variables:

```
PERCENT P_ACROSS;
PERCENT P_DOWN;
```

The following TABLE statement would produce the desired results:

```
TABLE P_28:
        <P_DOWN>AGE,
        (P_ACROSS THEN P_DOWN) BY <P_ACROSS>REGION;
```

P 28

	P ACROSS			P DOWN		
	U.S	North	South	U.S	North	South
All Ages	100	60	40	100	100	100
Under 21	100	67	33	30	33	25
21 and Over	100	57	43	70	67	75

Here, the first element of AGE determines the base for all cells nested with P_DOWN; the first element of REGION determines the base for all cells nested with P_ACROSS. It is important to note that a base marker for one percent variable will have no effect on the cells within the scope of a different percent. We have emphasized this by using different shading for the different bases.

The next table is the one we showed earlier with the last row and columns being percents. The table is produced from the following:

```
PERCENT P1 "" MASK RIGHT 999%;
PERCENT P2 "" MASK RIGHT 999%;

TABLE P_29:
        <P1>(AGE THEN P2 BY TOTAL),
        <P2>(REGION THEN P1 BY TOTAL);
```

P 29

	U.S	North	South	Total
All Ages	2,000	1,200	800	100%
Under 21	600	400	200	30
21 and Over	1,400	800	600	70
Total	100%	60%	40%	–

This table has percents in two different dimensions. This results in the informational warning:

*** WARNING: Percent variables occur in more than one dimension. All cells in their intersection will be set to null.

Treatment of Masks in Percents

In general, the mask associated with the percent variable will determine the format for the cell value. Footnotes and other aspects of masks associated with the numerator and denominator will be ignored.

If the numerator value for a percent cell has a mask with no 9's, in other words, a mask with only a footnote reference and/or a character string, then no percent will be calculated for that cell and the footnote reference and/or character string will be displayed instead. The only exception to this rule is when either the numerator or the denominator carries the built-in footnote called ERROR. In this case, the ERROR footnote symbol will be displayed.

Summary of Rules for Producing Percents

- Bases for percentages are derived from top to bottom, from left to right, and from front to back through wafers within a table.
- All cells that are nested with a percent variable or percent condition are percent cells. If the percent variable is not nested with an observation variable, the numerator for calculating percents will be frequency counts.
- The set of bases for these percent cells includes every cell that is nested with the first element that the markers apply to.
- The bases for percents remain in the table unless removed by FORMAT statements.

Checking for Percent Errors in Post Translator

Some error messages related to percents are displayed in the Post Translator (POSTRANS) or Sisyphus portions of the output listing and not by the TPL Translator (TRANSLATOR). This means that although the TPL Translator may indicate that no errors were detected and a reasonable table layout is produced, the messages produced by Post Translator and Sisyphus will still have to be examined. For example, messages related to missing base markers will be produced by Post Translator. Sisyphus will produce messages warning of missing or zero percent bases.

CHAPTER 13

Percent Change

Creating Table Requests with Percent Change or Numeric Change

The **Percent Change** and **Numeric Change** statements create new observation variables which can be used in tables statements. Specification of the statements requires 2 *On variables*, an observation variable and a control variable. The changes that appear in the table are the change in the *On Observation* value that occur when you move from one condition of the *On Control* variable to the next.

Format

PERCENT CHANGE new-variable-name ['variable-label'][mask] ON control-var AND obs-var ;

NUMERIC CHANGE new-variable-name ['variable-label'][mask] ON control-var AND obs-var ;

How Percent Change is Calculated

If the change variable appears in a table statement but is not in the same cross tabulation (*See Cross Tabulation in the Tables chapter*) as its *on control* variable, cells in the cross tabulation will just get the values they would get if the change variable were replaced by the *on observation* variable.

If a cross tabulation contains both a percent change variable and its *on control variable*, cells nested below the first condition of the control variable will get a **-**. Other cells will get a value calculated by the formula

$$\text{cell-value}_n = 100 * ((v_n - v_{n-1}) / v_{n-1})$$

where

v_n is the cell-value which would occur as the n^{th} condition of the control variable if the *on observation* variable had been used instead of the change variable.

If **Numeric Change** is used instead, the cells will get a value calculated by the formula:

$$\text{cell-value}_n = v_n - v_{n-1}$$

Examples

We begin with a simple table request based on hospital data.

```
Define FILTERED_YEAR on YEAR;
"1985" if "85";

Define WHY on PROBLEM;
"Not yet Classified"            if "000";
"Brain & Nervous System"        if "001": "035";
"Respiratory System"            if "076": "102";
"Heart & Circulatory System"    if "103": "145";

Table TABLE_1:
Stub FILTERED_YEAR by MONTH;
Head WHY;
```

TABLE 1

	Not yet Classified	Brain & Nervous System	Respiratory System	Heart & Circulatory System
1985				
JANURARY	1	39	66	90
FEBRUARY	3	28	89	87
MARCH	4	31	65	79
APRIL	5	32	49	69
MAY	8	29	30	64
JUNE	2	21	33	68
JULY	–	25	22	59
AUGUST	2	18	16	63
SEPTEMBER	2	21	29	63
OCTOBER	4	17	27	58
NOVEMBER	4	18	34	64
DECEMBER	21	23	39	60

– Data not available.

We now add a Percent Change statement to our table request. Note that **Count** is used as the observation since there is no observation in the request.

```
Percent Change PCH "Monthly Change" Mask 99.99% center
on COUNT and MONTH;

Table Table_2:
Stub FILTERED_YEAR by MONTH;
Head PCH by WHY;
```

TABLE 2

	Monthly Change			
	Not yet Classified	Brain & Nervous System	Respiratory System	Heart & Circulatory System
1985				
JANURARY	–	–	–	–
FEBRUARY	200.00%	-28.21%	34.85%	-3.33%
MARCH	33.33	10.71	-26.97	-9.20
APRIL	25.00	3.23	-24.62	-12.66
MAY	60.00	-9.38	-38.78	-7.25
JUNE	-75.00	-27.59	10.00	6.25
JULY	–	19.05	-33.33	-13.24
AUGUST	–	-28.00	-27.27	6.78
SEPTEMBER	0.00	16.67	81.25	0.00
OCTOBER	100.00	-19.05	-6.90	-7.94
NOVEMBER	0.00	5.88	25.93	10.34
DECEMBER	425.00	27.78	14.71	-6.25

– Data not available.

Notice that the first row is all dashed because there is no previous month. Also note that July and August of the first column are dashed. July is because July is also dashed in the based-on table. August is dashed because the calculation of the August value involves dividing by the July value which is zero. If Numeric Change were used instead of Percent Change, July would still be dashed since it is zero but August would have a value.

Combining the two tables we get.

```
Table TABLE_3:
Stub FILTERED_YEAR by (MONTH);
Head WHY by (COUNT then PCH);
```

TABLE 3

	Not yet Classified		Brain & Nervous System		Respiratory System		Heart & Circulatory System	
	Count	Monthly Change	Count	Monthly Change	Count	Monthly Change	Count	Monthly Change
1985								
JANURARY	1	–	39	–	66	–	90	–
FEBRUARY	3	200.00%	28	-28.21%	89	34.85%	87	-3.33%
MARCH	4	33.33	31	10.71	65	-26.97	79	-9.20
APRIL	5	25.00	32	3.23	49	-24.62	69	-12.66
MAY	8	60.00	29	-9.38	30	-38.78	64	-7.25
JUNE	2	-75.00	21	-27.59	33	10.00	68	6.25
JULY	–	–	25	19.05	22	-33.33	59	-13.24
AUGUST	2	–	18	-28.00	16	-27.27	63	6.78
SEPTEMBER	2	0.00	21	16.67	29	81.25	63	0.00
OCTOBER	4	100.00	17	-19.05	27	-6.90	58	-7.94
NOVEMBER	4	0.00	18	5.88	34	25.93	64	10.34
DECEMBER	21	425.00	23	27.78	39	14.71	60	-6.25

– Data not available.

If we replace our Percent Change statement with a Numeric Change statement we get:

```
Numeric Change NCH "Numeric Monthly Change" mask 99 center;

Table TABLE_4:
Stub FILTERED_YEAR by (MONTH);
Head WHY by (COUNT then NCH);
```

TABLE 4

	Not yet Classified		Brain & Nervous System		Respiratory System		Heart & Circulatory System	
	Count	Numeric Monthly Change	Count	Numeric Monthly Change	Count	Numeric Monthly Change	Count	Numeric Monthly Change
1985								
JANURARY	1	–	39	–	66	–	90	–
FEBRUARY	3	2	28	-11	89	23	87	-3
MARCH	4	1	31	3	65	-24	79	-8
APRIL	5	1	32	1	49	-16	69	-10
MAY	8	3	29	-3	30	-19	64	-5
JUNE	2	-6	21	-8	33	3	68	4
JULY	–	–	25	4	22	-11	59	-9
AUGUST	2	2	18	-7	16	-6	63	4
SEPTEMBER	2	0	21	3	29	13	63	0
OCTOBER	4	2	17	-4	27	-2	58	-5
NOVEMBER	4	0	18	1	34	7	64	6
DECEMBER	21	17	23	5	39	5	60	-4

– Data not available.

Suppose we want to create a table with percent changes that spans years. The Percent Change statement can only take one control variable so we must combine the years and months into a single control variable. If Month and Year are separate fields, the easy way to do this is with a Define on multiple variables.

```
Define YEAR_MONTH;
"1984" /
"  September" if YEAR = 84 and MONTH = '9 ';
"  October" if YEAR = 84 and MONTH = 10;
"  November" if YEAR = 84 and MONTH = 11;
"  December" if YEAR = 84 and MONTH = 12;
"1985" /
"  January" if YEAR = 85 and MONTH = '1 ';
"  February" if YEAR = 85 and MONTH = '2 ';
"  March" if YEAR = 85 and MONTH = '3 ';
"  April" if YEAR = 85 and MONTH = '4 ';
"  May" if YEAR = 85 and MONTH = '5 ';
"  June" if YEAR = 85 and MONTH = '6 ';
"  July" if YEAR = 85 and MONTH = '7 ';
"  August" if YEAR = 85 and MONTH = '8 ';
"  September" if YEAR = 85 and MONTH = '9 ';
"  October" if YEAR = 85 and MONTH =  10;
"  November" if YEAR = 85 and MONTH = 11;
"  December" if YEAR = 85 and MONTH = 12;
"1986" /
"  January" if YEAR = 86 and MONTH = '1 ';
"  February" if YEAR = 86 and MONTH = '2 ';
"  March" if YEAR = 86 and MONTH = '3 ';
"  April" if YEAR = 86 and MONTH = '4 ';
"  May" if YEAR = 86 and MONTH = '5 ';
"  June" if YEAR = 86 and MONTH = '6 ';
"  July" if YEAR = 86 and MONTH = '7 ';
"  August" if YEAR = 86 and MONTH = '8 ';

Percent Change PCH "Monthly Change" Mask 99.99 % ON
YEAR_MONTH AND COUNT;

Table TABLE_1:
        Stub YEAR_MONTH;
        Head WHY by (COUNT then PCH);
```

TABLE 5

	Not yet Classified		Brain & Nervous System		Respiratory System		Heart & Circulatory System	
	Count	Monthly Change	Count	Monthly Change	Count	Monthly Change	Count	Monthly Change
1984								
September	–	–	–	–	–	–	–	–
October	4	–	19	–	50	–	78	–
November	2	-50.00%	24	26.32%	41	-18.00%	79	1.28%
December	7	250.00	29	20.83	61	48.78	91	15.19
1985								
January	1	-85.71	39	34.48	66	8.20	90	-1.10
February	3	200.00	28	-28.21	89	34.85	87	-3.33
March	4	33.33	31	10.71	65	-26.97	79	-9.20
April	5	25.00	32	3.23	49	-24.62	69	-12.66
May	8	60.00	29	-9.38	30	-38.78	64	-7.25
June	2	-75.00	21	-27.59	33	10.00	68	6.25
July	–	–	25	19.05	22	-33.33	59	-13.24
August	2	–	18	-28.00	16	-27.27	63	6.78
September	2	0.00	21	16.67	29	81.25	63	0.00
October	4	100.00	17	-19.05	27	-6.90	58	-7.94
November	4	0.00	18	5.88	34	25.93	64	10.34
December	21	425.00	23	27.78	39	14.71	60	-6.25
1986								
January	14	-33.33	31	34.78	68	74.36	62	3.33
February	13	-7.14	22	-29.03	43	-36.76	65	4.84
March	15	15.38	25	13.64	60	39.53	81	24.62
April	10	-33.33	26	4.00	39	-35.00	88	8.64
May	7	-30.00	21	-19.23	29	-25.64	69	-21.59
June	8	14.29	34	61.90	24	-17.24	52	-24.64
July	410	5025.00	–	–	–	–	–	–
August	297	-27.56	–	–	–	–	–	–

– Data not available.

CHAPTER 14

Statistics

STATISTICAL FUNCTIONS AND STATEMENTS

The built-in functions **MAX** and **MIN** can be entered in POST COMPUTE statements.

The following built-in statistical functions can be used to create new variables. These variables can be referenced in POST COMPUTE and TABLE statements.

MEDIAN
FMEDIAN - Alternate median calculation
QUANTILE
FQUANTILE - Alternate quantile calculation
MEAN
VAR - variance of sample
VARP - variance of whole population
STDEV - standard deviation of sample
STDEVP - standard deviation of whole population
STDERR - standard error of means

Note that in some cases you may wish to exclude certain values from these calculations in order to get the desired results. For example, if you are applying these functions to variables that have been computed by division, you may need to screen out "divide by zero" error values using Conditional Compute statements. If you are applying these functions to variables in your data file and have records with data errors, you may need to eliminate those records. For more information on this subject, see the discussion of "Null Values" in the section on Conditional Compute in the "Compute" chapter, the section on "Data Errors" in the "Data" chapter and the discussion on "Errors in Observation Values" in the "Codebook" chapter.

If you are applying these functions to values that are taken directly from the data file and there are no data errors, you do not need to be concerned with the above paragraph.

MAX

MAX is an operator which is used in POST COMPUTE statements only. Its form is **MAX(var)** where the argument, **var**, is any single observation variable. The observation variable may come from the codebook or from a computed variable. The contribution of MAX(var) to the Post Compute is the largest value of 'var' from any record which contributes to a cell.

Example Assume that we want to show the maximum family income for each region according to sex of the head of household. We can Post Compute the maximum value, then nest it into the table statement as follows:

```
POST COMPUTE MAX_INCOME 'Maximum income' =
        MAX(INCOME);

TABLE Q1 'Maximum family income by region and sex':
        HEADING  MAX_INCOME  BY  SEX;
        STUB  REGION;
```

Each cell of the table will contain a maximum income value.

MIN

MIN is an operator which follows the same rules as MAX except that the contribution of **MIN(var)** to the Post Compute for a cell is the smallest value of **var** from any record which contributes to the cell.

Example Assume that we want to show the minimum family income for each region according to sex of the head of household. We can Post Compute the minimum value, then nest it into the table statement as follows:

```
POST COMPUTE MIN_INCOME 'Minimum income' = MIN(INCOME);

TABLE Q2 'Minimum family income by region and sex':
        HEADING  MIN_INCOME  BY  SEX;
        STUB  REGION;
```

Each cell of the table will contain a minimum income value.

Now assume that in some cases, the family income is not reported so that for some families the income is zero. This would result in the minimum value being zero for all or most of the cells. If we wish, we can eliminate the zero values from the MIN calculation by using a conditional Compute to create a new income variable that has only the non-zero income values.

```
COMPUTE NONZERO_INCOME =
        INCOME          IF INCOME > 0;
        NULL            IF OTHER;
```

Then we can change our Post Compute of the minimum values to calculate the minimum of the non-zero values.

```
POST COMPUTE MIN_INCOME 'Minimum income' =
        MIN(NONZERO_INCOME);
```

MEDIAN and FMEDIAN

The MEDIAN statement is used to find the median of an observation variable. The variable generated by the statement may be used in a POST COMPUTE statement or a TABLE statement. The FMEDIAN statement is a variation of the Median Statement which uses a slightly different algorithm for its calculation. FMEDIAN should only be used when you are ranking integer values. MEDIAN may be used for integers or other numbers. See *Quantile Algorithm* for a discussion of the differences in the calculation.

The format of the statement is:

Format
```
MEDIAN  new-var  ['print label']  USING MASK mask] ON
                                  USING            :
                                  MASK
        rank-var(isd) [WEIGHTED BY weight-var];

FMEDIAN  new-var  ['print label'] USING MASK mask] ON
                                  USING            :
                                  MASK

        rank-var  [WEIGHTED BY weight-var];
```

where *new-var* is the new observation variable generated by the statement, *rank-var* is the observation variable to be ranked which may come from

either the codebook or a COMPUTE statement, and the *isd* value is an integer between 1 and 23 which must be specified to regulate the accuracy of the results for median statements but must not be specified for fmedians. A discussion of how to select an **ISD** value appears in the section *Choosing the ISD*. A print label and mask are optional parameters for medians.

Weighted Medians

In some cases, records contain weighting factors. These records may be, for example, records from sampling. The contribution of each such record to the median of a rank variable should be the value of the **weight-var** occurrences of the rank variable value. The clause, **WEIGHTED BY *weight-var*** is an optional specification to allow for this type of processing. The values of *weight-var* need not be integers.

For example, we might have an Industry Wage Survey file where each establishment record contains an industry code (retail, manufacturing, etc.), region code, an hourly rate, number of workers earning that hourly rate, and the hours worked at that hourly rate. A request specifying two types of medians might be as follows:

```
MEDIAN MED_HR_PAY_HOUR
Median hourly rate for each hour worked'
        ON HOUR_RATE(5) WEIGHTED BY HOURS_WORKED;

MEDIAN MED_HR_PAY_WORKER
Median hourly pay rate for each worker'
        ON HOUR_RATE(5) WEIGHTED BY WORKERS;

TABLE INDUSTRY_MEDIANS:  INDUSTRY,  REGION BY
        (MED_HR_PAY_HOUR THEN MED_HR_PAY_WORKER);
```

In a hierarchical data file with LEVEL 0 family records and level 1 expenditure records, to find the median total family expenditures, the individual expenditures for each family must be added and be available as a total expenditure value at the family record level. Since TPL TABLES cannot summarize lower level expenditure values to the family level, the median family expenditure cannot be found. If each total family expenditure were available at the family level, the next statement would be meaningful.

```
MEDIAN MEDIAN_FAMILY_EXPENDITURE ON
        TOTAL_FAMILY_EXPENDITURE(4);
```

Special care must be taken when weighting is used with hierarchical files. In most cases, you will want the rank variable and weighting variable to be at the same level so that weighting will apply once to a variable for each occurrence of its hierarchical level. Therefore, to avoid confusion about the effect of weighting, the rank variable must be at the same record level as the weighting variable. A weighting variable or a rank variable must be brought down to the lower level containing the other type of variable by a COMPUTE statement before weighting is used in the MEDIAN statement. See an explanation of the COMPUTE statement in the Hierarchical Files chapter.

QUANTILE and FQUANTILE Statements

An FQUANTILE or QUANTILE statement is a generalization of the MEDIAN statement. Fquantiles should only be used when the rank variable has integers values while the quantile statement has no such restriction. The statement format is:

Format

```
QUANTILE[S](quantity#) new-var ['print label'] [USING MASK mask]
                                                 USING
                                                 MASK
ON rank-var (isd)  [WEIGHTED BY weight-var]

;

or

FOR EACH ;

or

;
[label 1] IF quantile#1 ;
.         .         .
.         .         .
[label n] IF quantile#n ;
```

The format of an FQUANTILE is identical to that of a QUANTILE except that no ISD is specifield

Examples

```
QUANTILE (10) DECILE 'Income Decile' ON INCOME (4);
        '1st Decile'      IF 1;
        '2nd Decile'      IF 2;
        '3rd Decile'      IF 3;
        '4th Decile'      IF 4;
        'Median'          IF 5;
        '6th Decile'      IF 6;
        '7th Decile'      IF 7;
        '8th Decile'      IF 8;
        '9th Decile'      IF 9;

FQUANTILE (100) TEST_SCORES ON SCORE;
        '1st Percentile'  IF  1;
        '1st Decile'      IF 10;
        '1st Quartile'    IF 25;
        'Median'          IF 50;
```

The *quantity#* is a positive integer which gives the number of divisions of the quantile; e.g., a quantity# of 4 yields quartiles, a quantity# of 100 yields percentiles. The observation variable, ***rank-var,*** is the rank variable on which the quantile is created. It may be either a codebook variable or a computed variable. For a QUANTILE statement, the *isd* number is a required value between 1 and 23 which controls the accuracy of the quantile. For an FQUANTILE statement, no *isd* is specified. See the section *Choosing the ISD* for a discussion of this parameter. The weighting variable, *weight-var*, is optional as in the MEDIAN statement. The print label and mask are optional.

The **quantile numbers** (*quantile#*$_1$...) are integers between 0 and quantity#. The quantiles need not be in any special order and only those of interest need be specified. If the quantity# is 4, a quantile# of 2 would result in a value of the 2nd quartile (median). If the quantity# is 100, the quantile# of 2 would yield the 2nd percentile. A quantile# of 0 yields the approximate minimum. A quantile# of quantity# yields the approximate maximum.

The *new variable* is the name for the one or more observation variables created by the QUANTILE statement.

Referencing the Quantile Variable

There are two ways that the new variable may be used in a table request.

The new variable may be used in a POST COMPUTE if it has a single quantile number, or if it is qualified by a quantile number, using the form: *new-var (quantile#)*, where *quantile#* is one of the quantile numbers ap-

pearing in the QUANTILE statement which generates the new variable. Suppose we have:

```
QUANTILE (4) SCORE_QUARTILE 'Score Distribution' ON SCORE(7);
             '1st Quartile'    IF 1;
             '3rd Quartile'    IF 3;
             'Median'          IF 2;

    POSTCOMPUTE SECOND_TO_THIRD_SPREAD =
             SCORE_QUARTILE(3) - SCORE_QUARTILE(2);
```

The contribution of SCORE_QUARTILE(3) to a Post Compute for a cell would be the third quartile score of those records which contribute to the cell. The contribution of SCORE_QUARTILE(2) would be medians.

Note Referencing the new variable qualified by a quantile number is restricted to quantile variables that have quantity numbers less than 256.

The new variable created by a QUANTILE statement may also be used directly in a TABLE statement. For such use it must not have a quantile number as a qualifier. The effect of using a variable created by a QUANTILE statement in a TABLE statement is that of replacing the single variable by the concatenation of the quantiles created in the QUANTILE statement.

If the specification of the heading of a table is SCORE_QUARTILE BY SEX, and SEX is a control variable, the heading will look like:

Score Distribution					
1st Quantile		3rd Quantile		Median	
Male	Female	Male	Female	Male	Female

The FOR EACH Option

As a convenience the keywords, "FOR EACH", can be used in QUANTILE statements. The clause, "FOR EACH" is used when all quantities for a given quantity# are to appear in a QUANTILE statement and they are to appear in ascending order, as in:

```
QUANTILE (100) SCORE_PERCENTILE 'Test Scores' ON SCORE (4)
        FOR EACH;
```

This statement is equivalent to:

```
QUANTILE (100) SCORE_PERCENTILE 'Test Scores' ON SCORE (4);
        '1'     IF 1;
        '2'     IF 2;
                .
                .
                .
        '99'    IF 99;
```

Choosing the ISD

Calculating the quantiles exactly can take a long time, especially for large data files. The approach in TPL TABLES is to use a grouping technique to calculate the quantiles. That is, the range of values are divided into a set of intervals, the interval which contains the desired quantile is determined, and the quantile is interpolated within that interval. The **interval size designator(ISD)** is used to control the grouping intervals.

There is an upper bound on the relative error of the computation. For each ISD number,

```
Maximum Relative Error =
Interval Size / Lower bound of Interval containing the quantile
```

The maximum possible relative error for each ISD number regardless of the value of the quantile is given in the following table:

ISD Number	Maximum Relative Error	ISD Number	Maximum Relative Error
1	100.00%	13	.024
2	50.00	14	.012
3	25.00	15	.0061
4	12.50	16	.0031
5	6.25	17	.0015
6	3.13	18	.00076
7	1.56	19	.00038
8	0.78	20	.00019
9	0.39	21	.000095
10	0.19	22	.000044
11	0.09	23	.000022
12	0.048		

The smoother the distribution of rank variables and the more data, the smaller will be the error in quantile calculations. This means that although an ISD of 4 is chosen, most cells will probably have a percentage error of considerably less than 12.5%. However, if a guaranteed accuracy of 12.5% is required for all cells, then ISD=4 should be used.

The ISD establishes the interval size for grouping of the rank variable values used in calculating the quantiles. The interval size for a given ISD number is not fixed but varies relative to the rank variable values. The following table shows the actual interval sizes used for selected rank variable values and selected ISD numbers.

Rank Variable Values		ISD Number				
		1	3	4	8	21
From	Up To					
1	2	1	1/4	1/8	1/128	1/1048576
2	4	2	1/2	1/4	1/64	1/524288
4	8	4	1	1/2	1/32	1/262144
8	16	8	2	1	1/16	1/131072
16	32	16	4	2	1/8	1/64436
32	64	32	8	4	1/4	1/32768
64	128	64	16	8	1/2	1/16384
128	256	128	32	16	1	1/8192
256	512	256	64	32	2	1/4096
512	1024	512	128	64	4	1/2048
1024	2048	1024	256	128	8	1/1024
2048	4096	2048	512	256	16	1/512
32768	65536	23768	8192	4096	256	1/32
1048576	2097152	1048576	262144	131072	8192	1

It can be seen from the table that any rank variable value which is twice as large as another rank variable value will be grouped within an interval which is twice as large. Also, increasing the ISD number by 1 will halve the interval size.

For all practical purposes, specifying an ISD number of 23 will produce quantiles without any grouping of the rank variable values. However, for a variety of reasons, some related to the nature of the rank variable data,

but often related to the processing time required to produce the quantiles, it may be desirable to group the data before calculating the quantiles.

Specifying the ISD number provides you with a flexible and convenient way of establishing the interval size. For example, in producing quantiles for such diverse values as the price of candy, television sets, and automobiles, a single quantile statement may be nested with the control variable ITEM. For any ISD number the interval size for the price of candy will be small and for automobiles will be large. The appropriate intervals should be determined by someone who is familiar with the nature of the data being processed.

Processing Time and the ISD

Quantile calculations on moderate to large datasets may take a long time. The conditions which contribute to a long processing time are:

- a large number of table cells in a run which will have medians or quantiles;
- a high ISD number;
- the field values on which quantile computations are to be done are not clustered and are distributed over a large range;
- limited memory.

If you would like to reduce the processing time for calculating quantiles, try one or more of the following:

1. Use as few distinct QUANTILE statements as possible in a run. For example, it will usually be faster to get three quantiles from a single quantile statement than to get two quantiles from separate statements.

2. Use the lowest ISD number you can. A value of 4 should be reasonable for moderate to large datasets.

3. Use Conditional Computes to clump rank variable values which are not likely to be quantile results. Suppose your run consists of a median based on a rank variable, A, crossed with a control variable. Further, suppose you know all the resulting medians will be between 300 and 350. A Conditional Compute like the following may result in internal processing efficiency, especially if there are many values outside the median range.

```
COMPUTE B =
        -99999          IF A < 300;
        A               IF A >= 300 AND A <= 350;
        99999           IF A > 350;
```

B should then be used as the rank variable in the QUANTILE statement instead of A. If the actual median fell below 300 or above 350, an obviously wrong answer would be displayed.

Quantile Algorithm

The method used to find Medians and Quantiles is as follows: For each median or quantile, an ordered distribution of the occurrences of grouped rank variable values is prepared. For Quantiles and Medians, the size of the intervals into which the rank variable values are grouped is determined by the ISD you supply. For Fquantiles and Fmedians. the interval size is always 1. For cases in which weighting has been specified, weighted occurrences are used. After determining the total occurrences for a quantile, the quantile point, QP, is computed to be:

QP = Total Occurrences of ranking variable values * quantile# / quantity#

The occurrences in each interval, starting with the lowest, are then added together until the i-th sum, S(i), equals or exceeds the quantile point. Thus the quantile is either some value in the i-th interval or between the i-th and the (i+1)-st interval.

If S(i) exceeds the quantile point the quantile value is within the i-th interval and is computed as follows:

Quantile Value = V(i) + R(i) * (QP - S(i-1)) / N(i)

If S(i) equals the quantile point the quantile value is between the i-th and the (i+1)-st interval.

```
Quantile Value =
        V(i) + R(i) + (V(i+1) - (V(i) + R(i))) * Quantile# / quantity#
```

Where:

V(i) = the lowest rank-var value of the i-th interval.
R(i) = the size of the i-th interval.
N(i) = the number of occurrences in the i-th interval.
V(i+1) = the lowest rank-var value of the next higher interval having any occurrences.

Sample Quantile Tables

A sample request, illustrating various capabilities of order statistics, is shown on the following pages.

```
USE HIERARCH CODEBOOK;

DEFINE INCOME_CLASS /'Family count for income ranges:'
                                        ON INCOME;
        'Exactly Equal to $0.00'        IF  0 ;
        'From $1 to $5,000'             IF  1 : 5000;
        'From $5,001 to $10,000'        IF  5001 : 10000;
        'From $10,001 to $20,000'       IF  10001 : 20000;
        'From $20,001 to $30,000'       IF  20001 : 30000;
        'Greater than $30,000'          IF  > 30000;

COMPUTE RECORD_NAME 'Number of Families Surveyed'
        USING 999 = FAMILIES;

COMPUTE NEW_INCOME /'Income of Families Surveyed'
        USING $9,999,999 = INCOME;

MEDIAN MED_INCOME /'Median Income of Surveyed Families'
        USING $99,999 ON INCOME (15);

MEDIAN OTHER_MED_INCOME
        /'Median Income of Surveyed Families'
        USING MASK $99,999 ON INCOME (15);

POST COMPUTE MAX_INCOME
        /'Largest Income of Surveyed Families'
        USING  MASK $99,999 = MAX (INCOME);

POST COMPUTE MIN_INCOME
        /'Smallest Income of Surveyed Families'
        USING MASK $99,999 = MIN (INCOME);

QUANTILE(4) QUARTILE_INCOME
        'Quartile Ranking of Family Income'
        USING MASK $99,999  ON  INCOME (15);
                '1st Quartile'     IF  1 ;
                'Median'           IF  2 ;
                'Third Quartile'   IF  3 ;

QUANTILE(10) RANK_INCOME 'Decile Ranking of Family Income'
        USING MASK $99,999 ON INCOME  (15)  FOR EACH ;
```

```
DEFINE ALL_FAM  ON RECORD_NAME;
        'Families Surveyed'      IF  ALL;

DEFINE NEW_HEADS_CLASS_OF_WORK
/ 'Class of Work for Family Head' ON  HEADS_CLASS_OF_WORK;
        'White collar'           IF  1;
        /  'Blue collar'         IF  2;
        /  'Farm workers'        IF  3;
        /  'Service workers'     IF  4;
        /  'Other workers'       IF  OTHER;

DEFINE NEW_LIVING_QRT
'Type of Family Living Quarters' ON LIVING_QRT;
        'Owned'          IF  1;
        'Condominium'    IF  2;
        'Rented'         IF  3;
        'Other types'    IF  OTHER;

DEFINE NEW_EARNER_COMP
/ 'Earner Composition of Families' ON  EARNER_COMP;
        'Head only Employed'                              IF  0;
        'Head & Spouse Employed'                          IF  1;
        'Head & Family members >= 18 yrs.'                IF  2;
        'Head & Family members < 18 yrs.'                 IF  3;
        'Head & other types of members Employed'          IF  4:5;
        'Head Unemployed'                                 IF  6;
        'Head Unemployed & Spouse Employed'               IF  7;
        'Other types'                                     IF  OTHER;
```

```
TABLE SAMPLE_01  'First Special Table for Order Statistics '
    'Using Median, Maximum and Minimum' :
            STUB RECORD_NAME THEN NEW_INCOME THEN
                 INCOME_CLASS THEN MED_INCOME THEN
                 MAX_INCOME THEN MIN_INCOME,
            HEADING  TOTAL THEN REGION;
```

First Special Table for Order Statistics Using Median, Maximum and Minimum

	Total	Northeast	North Central	South	West
Number of Families Surveyed	117	35	31	23	28
Income of Families Surveyed	$1,058,371	$327,531	$179,990	$252,995	$297,855
Family count for income ranges:					
Exactly Equal to $0.00	9	3	2	–	4
From $5,001 to $10,000	34	12	12	–	10
From $10,001 to $20,000	32	6	4	11	11
From $20,001 to $30,000	12	6	–	4	2
Median Income of Surveyed Families	$7,205	$6,005	$5,400	$12,100	$9,610
Largest Income of Surveyed Families	27,200	27,200	13,500	23,060	23,500
Smallest Income of Surveyed Families	0	0	0	3,000	0

– Data not available.

```
TABLE SAMPLE_02  'Second Table for Order Statistics Using the'
      /'Variable Created in the Quantile Statement in the Stub' :
      STUB   (TOTAL THEN NEW_HEADS_CLASS_OF_WORK)
             BY QUANTILE_INCOME,
      HEADING  SEX THEN NEW_LIVING_QRT;
```

Second Table for Order Statistics Using the Variable Created in the Quantile Statement in the Stub

	Head's sex		Type of Family Living Quarters			
	Male	Female	Owned	Condomin-ium	Rented	Other types
Total						
Quartile Ranking of Family Income						
1st Quartile	$5,890	$1,200	$4,750	–	$3,100	–
Median	10,015	3,605	11,063	–	6,005	–
Third Quartile	15,515	5,400	14,000	–	9,700	–
Class of Work for Family Head						
White collar						
Quartile Ranking of Family Income						
1st Quartile	5,695	1,501	5,695	–	6,005	–
Median	12,100	6,303	12,100	–	9,856	–
Third Quartile	20,200	9,700	19,851	–	16,700	–
Blue collar						
Quartile Ranking of Family Income						
1st Quartile	7,361	3,605	8,351	–	6,000	–
Median	11,565	4,503	13,163	–	8,108	–
Third Quartile	14,150	5,400	13,875	–	14,200	–
Farm workers						
Quartile Ranking of Family Income						
1st Quartile	1,800	–	–	–	1,800	–
Median	1,800	–	–	–	1,800	–
Third Quartile	1,800	–	–	–	1,800	–
Service workers						
Quartile Ranking of Family Income						
1st Quartile	7,205	–	12,820	–	5,776	–
Median	9,000	–	12,820	–	8,103	–
Third Quartile	15,050	–	15,050	–	22,650	–
Other workers						
Quartile Ranking of Family Income						
1st Quartile	2,550	600	0	–	2,194	–
Median	2,925	2,400	600	–	3,013	–
Third Quartile	3,100	4,600	2,550	–	4,690	–

– Data not available.

```
TABLE SAMPLE_03
        'Third Table for Order Statistics Using the Variable'/
        'Created in the Quantile Statement in the Heading' :
                STUB  ALL_FAM BY (REGION THEN
                        NEW_EARNER_COMP),
                HEADING  RANK_INCOME;
```

Third Table for Order Statistics Using the Variable Created in the Quantile Statement in the Heading

	Decile Ranking of Family Income								
	1	2	3	4	5	6	7	8	9
Families Surveyed									
Northeast	$1,735	$2,320	$4,600	$5,331	$6,005	$6,560	$11,650	$17,102	$24,001
North Central	600	1,800	2,925	3,100	5,400	7,215	9,000	9,000	13,200
South	3,600	3,740	4,750	10,025	12,100	12,820	14,000	14,200	20,201
West	0	5,695	5,890	7,205	9,610	13,125	15,515	18,800	19,851
Earner Composition of Families									
Head only Employed	3,000	4,750	5,400	6,005	7,205	9,000	10,005	12,000	20,201
Head & Spouse Employed	7,215	9,610	12,100	13,125	13,500	14,200	15,515	16,701	17,103
Head & Family members >= 18 yrs.	5,890	5,890	9,000	9,000	14,425	19,850	19,851	23,060	23,061
Head & other types of members Employed	5,695	7,120	12,820	13,292	14,000	16,880	18,800	22,560	23,500
Head Unemployed	0	600	1,800	2,158	2,400	2,550	2,925	4,600	5,331
Head Unemployed & Spouse Employed	3,100	3,100	3,100	3,100	4,800	6,500	6,500	6,500	6,500
Other types	3,600	3,600	3,600	3,600	3,600	3,600	3,600	3,600	3,600

MEAN

Format

```
MEAN mean-var ['print label']  [USING MASK mask] ON obs-var;
                               USING              :
                               MASK
[Weighted by weight-var]
```

where *mean-var* is the new mean variable and *obs-var* is an observation variable from the codebook or a COMPUTE statement.

The mean is calculated using the formulas:

Formula

$$\frac{\sum_i v_i}{n_T} \quad \text{or, with weighting} \quad \frac{\sum_i w_i v_i}{w_T}$$

where the v_i's are the values of the *obs-var* mapped into a table cell and the w_i's, if used, are the corresponding weights. n_T is the number of values that contribute to the cell. w_T is the sum of the weights.

Note Be sure the ON variable uses NULL for bad or missing values rather than 0. Otherwise incorrect answers may result.

Example

```
Mean MEAN_INC 'Mean Income' on INCOME;
```

This statement creates the mean variable MEAN_INC that can be used in POST COMPUTE and TABLE statements.

VAR - Variance of a Sample

Format

```
VAR var-samp ['print label']    [USING MASK mask] ON obs-var;
                                USING          :
                                MASK
    [Weighted by weight-var]
```

where *var_samp* is the new variance variable and *obs-var* is an observation variable from the codebook or a COMPUTE statement. The variance is calculated using the formulas:

Formula

$$\frac{n_T \sum_i v_i^2 - \left(\sum_i v_i\right)^2}{n_T\left(n_T - 1\right)} \quad \text{or, with weighting} \quad \frac{w_T \sum_i w_i v_i^2 - \left(\sum_i w_i v_i\right)^2}{w_T\left(w_T - 1\right)}$$

where the v_i's are the values of the *obs-var* mapped into a table cell and the w_i's, if used, are the corresponding weights. n_T is the number of values that contribute to the cell. w_T is the sum of the weights.

Note Be sure the *ON* variable uses NULL for bad or missing values rather than 0. Otherwise incorrect answers may result.

Example

```
VAR VAR_INC 'Income Variance' Mask 99,999 on INCOME;
```

This statement creates the variance variable VAR_INC that can be used in POST COMPUTE and TABLE statements.

VARP - Variance of Whole Population

Format

VARP var-pop ['print label'] [USING MASK mask] ON obs-var;
USING : MASK

[Weighted by weight-var]

where *var_pop* is the new variance variable and *obs-var* is an observation variable from the codebook or a COMPUTE statement. The variance is calculated using the formulas:

Formula

$$\frac{n_T \sum_i v_i^2 - \left(\sum_i v_i\right)^2}{n_T^2} \quad \text{or, with weighting} \quad \frac{w_T \sum_i w_i v_i^2 - \left(\sum_i w_i v_i\right)^2}{w_T^2}$$

where the v_i's are the values of the *obs-var* mapped into a table cell and the w_i's, if used, are the corresponding weights. n_T is the number of values that contribute to the cell. w_T is the sum of the weights.

Note

Be sure the *ON* variable uses NULL for bad or missing values rather than 0. Otherwise incorrect answers may result.

Example

```
VARP VARP_INC 'Income Variance' Mask 99,999 on INCOME;
```

This statement creates the variance variable VARP_INC that can be used in POST COMPUTE and TABLE statements.

STDEV - Standard Deviation of a Sample

Format

STDEV std-samp ['print label'] [USING MASK mask] ON obs-var;
USING : MASK

[Weighted by weight-var]

where *std-samp* is the new standard deviation variable and *obs-var* is an observation variable from the codebook or a COMPUTE statement. The standard deviation is calculated using the formulas:

Formula

$$\sqrt{\frac{n_T \sum_i v_i^2 - \left(\sum_i v_i\right)^2}{n_T\left(n_T - 1\right)}} \quad \text{or, with weighting} \quad \sqrt{\frac{w_T \sum_i w_i v_i^2 - \left(\sum_i w_i v_i\right)^2}{w_T\left(w_T - 1\right)}}$$

where the v_i's are the values of the *obs-var* mapped into a table cell and the w_i's, if used, are the corresponding weights. n_T is the number of values that contribute to the cell. w_T is the sum of the weights.

Note Be sure the *ON* variable uses NULL for bad or missing values rather than 0. Otherwise incorrect answers may result.

Example

```
STDEV DEV_INC 'Standard Deviation of Sample' Mask 99,999 on
      INCOME;
```

This statement creates the standard deviation variable DEV_INC that can be used in POST COMPUTE and TABLE statements.

STDEVP - Standard Deviation of Whole Population

Format

```
STDEVP std-pop ['print label'] [USING MASK mask] ON obs-var;
                               USING              :
                               MASK
[Weighted by weight-var]
```

where *std-pop* is the new standard deviation variable and *obs-var* is an observation variable from the codebook or a COMPUTE statement. The standard deviation is calculated using the formulas:

Formula $\sqrt{\frac{n_T \sum_i v_i^2 - \left(\sum_i v_i\right)^2}{n_T^2}}$ or, with weighting $\sqrt{\frac{w_T \sum_i w_i v_i^2 - \left(\sum_i w_i v_i\right)^2}{w_T^2}}$

where the v_i's are the values of the *obs-var* mapped into a table cell and the w_i's, if used, are the corresponding weights. n_T is the number of values that contribute to the cell. w_T is the sum of the weights.

Note Be sure the *ON* variable uses NULL for bad or missing values rather than 0. Otherwise incorrect answers may result.

Example

```
STDEVP DEVP_INC 'Income Deviation' Mask 99,999 on INCOME;
```

This statement creates the standard deviation variable DEVP_INC that can be used in POST COMPUTE and TABLE statements.

STDERR - Standard Error of the Mean

Format

```
STDERR std-err ['print label']  [USING MASK mask] ON obs-var;
                                USING              :
                                 MASK
[Weighted by weight-var]
```

where *std-err* is the new standard error variable and *obs-var* is an observation variable from the codebook or a COMPUTE statement. The standard error is calculated using the formulas:

Formula

$$\sqrt{\frac{n_T \sum_i v_i^2 - \left(\sum_i v_i\right)^2}{n_T\left(n_T - 1\right)}} \Big/ \sqrt{n_T}$$

or, with weighting

$$\sqrt{\frac{w_T \sum_i w_i v_i^2 - \left(\sum_i w_i v_i\right)^2}{w_T\left(w_T - 1\right)}} \Big/ \sqrt{w_T}$$

where the v_i's are the values of the *obs-var* mapped into a table cell and the w_i's, if used, are the corresponding weights. n_T is the number of values that contribute to the cell. w_T is the sum of the weights.

Note Be sure the *ON* variable uses NULL for bad or missing values rather than 0. Otherwise incorrect answers may result.

Example

```
STDERR ERR_INC 'Income Standard Error' Mask 9,999.99 on
       INCOME;
```

This statement creates the standard error variable ERR_INC that can be used in POST COMPUTE and TABLE statements.

Example Showing Multiple Statistics

Multiple statistics can be calculated for the same table. The following shows several statistics calculated for AGE.

```
Mean MEAN_AGE "Mean age" Mask 99.999 on AGE;
Mean MEAN_AGE_WGTD "Mean age weighted" Mask 99.999
        on AGE Weighted by WEIGHT;
Var VAR_AGE "Variance of sample" Mask 99.999 on AGE;
Varp VAR_AGE_WHOLE "Variance of whole population" Mask 99.999
        on AGE;
Stdev STD_DEV_AGE "Standard deviation of sample" Mask 99.999
        on AGE;
Stdevp ST_DEV_AGE_WHOLE
        "Standard deviation of whole population"
         Mask 99.999 on AGE;
Stderr STD_ERROR_AGE "Standard error of means" Mask 99.999
        on AGE;

Table STATS "Table showing statistics for age":
Stub TOTAL then SEX;
Heading TOTAL then MEAN_AGE then MEAN_AGE_WGTD then
        VAR_AGE then VAR_AGE_WHOLE then STD_DEV_AGE then
        ST_DEV_AGE_WHOLE then STD_ERROR_AGE;
```

Table showing statistics for age

	Total	Mean age	Mean age weighted	Variance of sample	Variance of whole population	Standard deviation of sample	Standard deviation of whole population	Standard error of means
Total	30,000	47.988	47.799	308.408	308.398	17.562	17.561	0.101
Sex of Householder								
Male	20,821	46.553	46.466	265.816	265.803	16.304	16.303	0.113
Female	9,179	51.243	50.787	389.792	389.750	19.743	19.742	0.206

CHAPTER 15

Ranking

ORDERING ROWS BASED ON THE VALUES IN A TABLE COLUMN

Introduction

With the RANK statement, you can create a variable for ordering table rows based on the values in a selected table column. The ranking can be descending or ascending. An option lets you keep only the top or bottom **n** rows for a particular ranking. With this option, you can request a row to display the residual. The residual category includes all values not in the top or bottom n rows, respectively.

You can also add a RANK DISPLAY column to the table. This column contains the rank number for each row. For rows that are not included in the ranking, blanks can be displayed in this column or you can use the built-in NORANK footnote to choose something other than blank.

A second method of displaying ranking is specified by the Format statement RANK ON VALUES. One or more columns of the table are picked, and the values in those columns are replaced by a rank number for the values. See RANK ON VALUES in the Format chapter for more information.

The RANK Statement

The RANK statement is similar to a DEFINE statement. It creates a new control variable based on the values of an existing variable and assigns labels to the new categories. The old variable can be one already described in the codebook or created by a Compute statement. The new variable definition can regroup or delete old variable values.

RANK variables can be used in TABLE statements. Since they are used to order the rows of a table, they **can only be used in the table stub**. They can be nested with other variables but **must be at the bottom level of the nest**.

More than one RANK variable can be used in a table, so different groups of rows can be ranked in different ways.

In a DEFINE, items are displayed in the table in the order they are listed in the DEFINE. In a RANK statement, the order of display in the table is determined by the table values in the specified *ranked-on* column.

Format

```
RANK new-variable-name ['var label']  ON ranked-on-var-name
                                                :
[COLUMN c] [KEEP n] [direction] ;

[condition-name-1]  ['print label'] IF [re] value-entry-1;
                                              :
[condition-name-2]  ['print label'] IF [re] value-entry-2;
                                              :
          .                  .
          .                  .
[condition-name-n]  ['print label'] IF [re] value-entry-n;
                                                                    :
```

The optional **COLUMN c** specifies the number of the column on which ranking occurs. If no column is specified, column 1 is assumed. There are no restrictions on what can be in the column. For example, it can contain counts, post computed values or percents.

The optional **KEEP n** specifies the number of rows that will be kept in each group. If no value is provided, all of the rows are kept.

The optional direction can be ascending or descending. Ascending order can be specified with **UP** or **ASCENDING**. Descending order can be specified with **DOWN** or **DESCENDING**. The direction is used to specify whether the first value in a group is the largest and the values from there go DOWN or whether the first value in a group is the smallest and the values go UP. The default is DOWN.

A **value-entry** can be any of the following:

value
condition name (if old variable is CONTROL)
value1 : value2
OTHER
ALL
NULL

If a **value-entry** does not have a name or label assigned to it, it is grouped into the first category above it that does have a name or label.

The optional **re** stands for any relation symbol or the equivalent English as shown below. A relation symbol can precede any value. If no relation is provided, "equal" is assumed.

Symbol	English expression
<	[IS] LESS THAN
>	[IS] GREATER THAN
=	[IS] EQUAL [TO] EQUALS
^<	[IS] NOT LESS THAN
^>	[IS] NOT GREATER THAN
^=	[IS] NOT EQUAL [TO]
<=	[IS] LESS THAN OR EQUAL [TO]
>=	[IS] GREATER THAN OR EQUAL [TO]

NULL value-entries

If your *ranked-on* variable contains NULL values, you need to reference NULL specifically to include these values in a RANK category. ALL and OTHER will not include NULL's.

OTHER value-entries

OTHER includes any values, except NULL, that do not fall into any other rank category. If you have limited the number of values to be displayed in a grouping, then the excluded values (residuals) will also fall into this category.

ALL value-entries

ALL includes all values except NULL.

Examples Following is a simple table request that selects only states in the Northeast region and creates a table with rows ranked by mean income.

```
USE CPS Codebook;

SELECT IF STATE_CODE IN (CONNECTICUT,  MAINE,
        MASSACHUSETTS, NEW_HAMPSHIRE, RHODE_ISLAND,
        VERMONT, NEW_JERSEY, NEW_YORK, PENNSYLVANIA);

MEAN MEAN_INCOME "Mean Income" ON INCOME;

MEAN WT_MEAN_INCOME "Weighted Mean Income" ON INCOME
        WEIGHTED BY WEIGHT;

RANK NE_RANK ON STATE_CODE COLUMN 1;
"Connecticut"    IF CONNECTICUT;
"Maine"          IF MAINE;
"Massachusetts" IF MASSACHUSETTS;
"New Hampshire" IF NEW_HAMPSHIRE;
"Rhode Island"  IF RHODE_ISLAND;
"Vermont"        IF VERMONT;
"New Jersey"     IF NEW_JERSEY;
"New York"       IF NEW_YORK;
"Pennsylvania"   IF PENNSYLVANIA;

TABLE RANK_1 "Northeast States Ranked on Mean Income":
STUB NE_RANK;
HEADING MEAN_INCOME THEN WT_MEAN_INCOME
        THEN COUNT;
```

Northeast States Ranked on Mean Income

	Mean Income	Weighted Mean Income	Count
Connecticut	38,872	38,721	599
New Jersey	38,858	39,195	2,243
Massachusetts	38,631	38,842	2,262
New Hampshire	36,676	36,454	515
New York	33,513	34,010	4,095
Rhode Island	32,827	32,769	517
Vermont	31,306	31,162	531
Pennsylvania	30,310	30,340	2,453
Maine	27,810	27,648	612

Next, we change the *ranked-on* column from 1 to 2 so the ranking is done on weighted mean income. Note that for the first 3 rows, the rank order is different from the rank order for (unweighted) mean income in the preceding example.

```
RANK NE_RANK_ON_2 on STATE_CODE COLUMN 2;
"Connecticut"    IF CONNECTICUT;
"Maine"          IF MAINE;
"Massachusetts" IF MASSACHUSETTS;
"New Hampshire" IF NEW_HAMPSHIRE;
"Rhode Island"  IF RHODE_ISLAND;
"Vermont"        IF VERMONT;
"New Jersey"     IF NEW_JERSEY;
"New York"       IF NEW_YORK;
"Pennsylvania"   IF PENNSYLVANIA;

TABLE RANK_2 "Northeast States Ranked on Weighted "
        "Mean Income":
STUB NE_RANK_ON_2;
HEADING MEAN_INCOME THEN WT_MEAN_INCOME
        THEN COUNT;
```

Northeast States Ranked on Weighted Mean Income

	Mean Income	Weighted Mean Income	Count
New Jersey	38,858	39,195	2,243
Massachusetts	38,631	38,842	2,262
Connecticut	38,872	38,721	599
New Hampshire	36,676	36,454	515
New York	33,513	34,010	4,095
Rhode Island	32,827	32,769	517
Vermont	31,306	31,162	531
Pennsylvania	30,310	30,340	2,453
Maine	27,810	27,648	612

Nested RANK Variables

RANK variables can can be nested with other variables in the stub but they **must be at the bottom level of the nest**.

Example In this example, we nest our rank variable below the control variable HOUSEHOLD_TYPE. This gives us multiple groups of rows to be ranked. For each HOUSEHOLD_TYPE, the rows are ranked on column 2. Notice that the ranking varies from group to group.

```
TABLE RANK_3 "Household Type by Northeast States Ranked on "
      "Mean Income" :
STUB HOUSEHOLD_TYPE BY NE_RANK_ON_2;
HEADING MEAN_INCOME THEN WT_MEAN_INCOME THEN
      COUNT;
```

Household Type by Northeast States Ranked on Mean Income

	Mean Income	Weighted Mean Income	Count
Type of Household			
Married couple			
New Jersey	49,863	50,484	1,284
Connecticut	49,995	50,387	313
Massachusetts	49,672	49,826	1,218
New York	44,574	45,223	2,075
New Hampshire	42,695	42,359	305
Rhode Island	41,042	40,957	297
Vermont	38,642	38,543	287
Pennsylvania	38,062	38,133	1,432
Maine	34,024	34,235	360
Other family			
New Hampshire	36,549	35,955	61
Connecticut	30,992	31,414	96
New Jersey	28,890	29,682	321
Massachusetts	26,906	27,291	367
Vermont	24,691	25,122	73
Rhode Island	24,566	24,322	61
Pennsylvania	24,211	24,289	332
New York	23,391	24,024	767
Maine	22,026	22,315	77
Nonfamily household			
Massachusetts	25,121	25,377	677
New Hampshire	24,407	24,981	149
Connecticut	24,531	23,913	190
New Jersey	21,725	21,835	638
Vermont	21,816	21,738	171
New York	21,390	21,388	1,253
Rhode Island	20,652	20,421	159
Pennsylvania	17,135	17,176	689
Maine	17,573	17,000	175

COPY as a Shortcut for Ranking on Codebook Variables

In some cases, you may be ranking on a control variable that has many values listed in the codebook and labels that you would like to use for the entries in a RANK statement. In this case, you can use COPY as a shortcut to simplify your RANK statement and take advantage of the labels you already have in the codebook.

Examples

Assume that the following variable is in the codebook for data about fire reports.

```
SITUATION CONTROL 2
(       'Fire -- No Other Info'   = 10
        'Structure Fires'         = 11
        'Outside Str Fires'       = 12
        'Vehicle Fires'           = 13
        'Trees, Brush Fires '     = 14
        'Refuse Fires'            = 15
        'Explosions, No Fire'     = 16
        'Other Fires'             = 19
)
```

We could rank all fire situations, taking advantage of all the codebook labels, with this simple RANK statement:

```
RANK FIRES ON SITUATION COLUMN 1;
COPY IF 10:19;
```

Note

As in the DEFINE statement, COPY cannot be used with ALL, OTHER or NULL. To copy all values, enter a range with the first and last condition values as shown above.

In the next RANK example, COPY is combined with some specific grouping of values.

```
RANK FIRES ON SITUATION COLUMN 1;
COPY                  IF 11:16;
'Type Not Known'      IF 10;
                      IF 19;
```

```
TABLE F1 'Fires Ranked byType':
HEADING COUNT;
STUB TOTAL THEN FIRES,
```

Fires Ranked by Type

	Count
Total	2,665
Vehicle Fires	886
Structure Fires	818
Refuse Fires	456
Trees, Brush Fires	354
Outside Str Fires	107
Type Not Known	33
Explosions, No Fire	11

Keeping the Top or Bottom Ranked Rows

The optional **KEEP** can be used in the RANK statement to specify the number of rows that will be kept in each group of ranked rows. If there is no KEEP in the statement, all of the rows are kept.

Example

In this example, we rank on column 1 and limit the number of included categories to the top 4 rows in each ranking. As in the previous example, ranking is done for each of 3 groups, so the top 4 rows are kept for each group.

```
RANK NE_RANK_LIMIT ON STATE_CODE COLUMN 1 KEEP 4;
"Connecticut" IF CONNECTICUT;
"Maine" IF MAINE;
"Massachusetts" IF MASSACHUSETTS;
"New Hampshire" IF NEW_HAMPSHIRE;
"Rhode Island" IF RHODE_ISLAND;
"Vermont" IF VERMONT;
"New Jersey" IF NEW_JERSEY;
"New York" IF NEW_YORK;
"Pennsylvania" IF PENNSYLVANIA;
```

```
Table RANK_4  "Household Type by Top 4 Northeast States "
        "Ranked on Mean Income":
STUB HOUSEHOLD_TYPE BY NE_RANK_LIMIT;
HEADING MEAN_INCOME THEN WT_MEAN_INCOME THEN
        COUNT;
```

Household Type by Top 4 Northeast States Ranked on Mean Income

	Mean Income	Weighted Mean Income	Count
Type of Household			
Married couple			
Connecticut	49,995	50,387	313
New Jersey	49,863	50,484	1,284
Massachusetts	49,672	49,826	1,218
New York	44,574	45,223	2,075
Other family			
New Hampshire	36,549	35,955	61
Connecticut	30,992	31,414	96
New Jersey	28,890	29,682	321
Massachusetts	26,906	27,291	367
Nonfamily household			
Massachusetts	25,121	25,377	677
Connecticut	24,531	23,913	190
New Hampshire	24,407	24,981	149
Vermont	21,816	21,738	171

Treatment of Ties

Sometimes you may have two or more rows that are tied in the ranking, having the same value in the *ranked-on* column. If there is a tie between the last row and one or more rows above it, all of these rows will be retained.

Example

In the following table, tied values are shown in bold type. The rank statement had **KEEP 30,** but since the last two rows are tied, the actual number of rows kept is **31**.

Top 30 counties ranked by number of plural births

	Total	Single Births	Plural Births
Lincoln	12,296	11,860	436
Taylor	10,924	10,501	423
Eisenhower	5,831	5,638	193
Comanche	4,517	4,341	176
Cheyenne	5,433	5,270	163
Clinton	3,900	3,762	138
Coral	2,586	2,494	92
Bannock	2,564	2,474	90
Massachusett	3,114	3,025	89
Maricopa	2,039	1,956	83
Harrison	1,809	1,732	77
Cameron	2,079	2,004	75
Blackfoot	2,190	2,117	73
Mohawk	1,987	1,917	***70***
Chippewa	1,903	1,833	***70***
Norfolk	2,032	1,970	62
Vermillion	1,659	1,602	57
Adams	1,814	1,759	55
Holland	2,116	2,062	54
McKinley	1,356	1,303	53
Manhattan	1,200	1,149	51
Sky	2,312	2,262	50
Gramblin	1,075	1,027	48
Cherokee	1,519	1,474	45
Cayuse	1,202	1,158	***44***
Wilson	1,079	1,035	***44***
Paris	1,690	1,649	41
Oxford	1,140	1,104	***36***
Ford	1,423	1,387	***36***
Niagra	1,757	1,722	***35***
Benton	1,047	1,012	***35***

Note

If you are ranking on decimal values rather than integers, you may have values that look like ties in the table but are actually different values because of additional decimal places not shown in the table.

Using OTHER to Get Residuals

An OTHER category can be entered on any row of the RANK statement. OTHER includes all values, except NULL, that do not fall into any other rank category. When you use KEEP to limit the number of values to be displayed in a grouping, then the excluded values (residuals) will also fall into this category. The row for the OTHER category will always follow the ranked rows, regardless of where it is entered in the RANK statement.

Example

In this table, we add an **OTHER** category to our RANK statement. Note that each grouping has a new "Lower Income States" category which is the sum of all of the excluded categories.

```
RANK NE_RANK_LIMIT_OTHER ON STATE_CODE COLUMN 1
        KEEP 4;
"Connecticut" IF CONNECTICUT;
"Maine" IF MAINE;
"Massachusetts" IF MASSACHUSETTS;
"New Hampshire" IF NEW_HAMPSHIRE;
"Rhode Island" IF RHODE_ISLAND;
"Vermont" IF VERMONT;
"New Jersey" IF NEW_JERSEY;
"New York" IF NEW_YORK;
"Lower Income States" IF OTHER;
"Pennsylvania" IF PENNSYLVANIA;

TABLE RANK_5 "Household Type by Top 4 Plus OTHER Northeast "
        "States Ranked on Mean Income":
STUB HOUSEHOLD_TYPE BY NE_RANK_LIMIT_OTHER;
HEADING MEAN_INCOME THEN WT_MEAN_INCOME THEN
        COUNT;
```

Household Type by Top 4 Plus OTHER
Northeast States Ranked on Mean Income

	Mean Income	Weighted Mean Income	Count
Type of Household			
Married couple			
Connecticut	49,995	50,387	313
New Jersey	49,863	50,484	1,284
Massachusetts	49,672	49,826	1,218
New York	44,574	45,223	2,075
Lower Income States	38,439	38,311	2,681
Other family			
New Hampshire	36,549	35,955	61
Connecticut	30,992	31,414	96
New Jersey	28,890	29,682	321
Massachusetts	26,906	27,291	367
Lower Income States	23,646	24,078	1,310
Nonfamily household			
Massachusetts	25,121	25,377	677
Connecticut	24,531	23,913	190
New Hampshire	24,407	24,981	149
Vermont	21,816	21,738	171
Lower Income States	20,188	20,101	2,914

Using ALL and OTHER

The **OTHER** category includes both residuals and any values which do not appear in other categories. If **ALL** is added to the list of categories, then **OTHER** will include only residuals. If the RANK statement does not have a **KEEP** clause, then **OTHER** will be empty.

Note that if you have ALL in a RANK statement, the ALL category will be included in the ranking. Thus, if you have requested the top 5 rows, for example, and the ALL row has a value in the top 5, it will be one of the 5 rows kept. If you do not want the ALL row to be ranked, create a separate variable for ALL and add it to the TABLE statement.

Example

```
DEFINE ALL_STATES ON STATE_CODE;
"All New England States" IF ALL;

RANK NE_RANK_LIMIT ON STATE_CODE COLUMN 1 KEEP 4;
"Connecticut" IF CONNECTICUT;
"Maine" IF MAINE;
"Massachusetts" IF MASSACHUSETTS;
"New Hampshire" IF NEW_HAMPSHIRE;
"Rhode Island" IF RHODE_ISLAND;
"Vermont" IF VERMONT;
"New Jersey" IF NEW_JERSEY;
"New York" IF NEW_YORK;
"Pennsylvania" IF PENNSYLVANIA;
```

```
Table RANK_6  "Household Type by Top 4 Northeast States "
        "Ranked on Mean Income":
STUB HOUSEHOLD_TYPE BY
        (ALL_STATES THEN NE_RANK_LIMIT);
HEADING MEAN_INCOME THEN WT_MEAN_INCOME THEN
        COUNT;
```

Displaying the Rank Number with RANK DISPLAY

RANK DISPLAY is a simple statement that lets you add a column to your table to display rank numbers. This column contains the rank number for each row. For rows that are not included in the ranking, blanks can be displayed or you can use the built-in NORANK footnote to choose something other than blank.

Format

```
RANK DISPLAY display-variable-name ['var label'] [mask] ;
```

If you do not provide a label, the **display-variable-name** will be used as the label. If you do not provide a mask, the rank numbers will be right-aligned in the column.

Place the RANK DISPLAY variable in the heading of your table.

Example

In the following example, the RANK DISPLAY variable named COUNTY_RANK is at the end of the heading, so the rank number will be displayed in the last column.

```
DEFINE SINGLE_PLURAL ON PLURAL;
"Single Births" if "1";
"Plural Births" if "2": "5";

RANK COUNTY_RANK ON RCOUNTY COLUMN 3 KEEP 30;
COPY IF 1:100;

RANK DISPLAY RANK_NUM "Rank";
```

```
TABLE TABLE_2 "Top 30 counties ranked by number of plural "
        "births with rank numbers in last column":
HEADING     TOTAL THEN SINGLE_PLURAL THEN RANK_NUM;
STUB        COUNTY_RANK;
```

Top 30 counties ranked by number of plural births with rank numbers in last column

	Total	Single Births	Plural Births	Rank
Lincoln	12,296	11,860	436	1
Taylor	10,924	10,501	423	2
Eisenhower	5,831	5,638	193	3
Comanche	4,517	4,341	176	4
Cheyenne	5,433	5,270	163	5
Clinton	3,900	3,762	138	6
Coral	2,586	2,494	92	7
Bannock	2,564	2,474	90	8
Massachusett	3,114	3,025	89	9
Maricopa	2,039	1,956	83	10
Harrison	1,809	1,732	77	11
Cameron	2,079	2,004	75	12
Blackfoot	2,190	2,117	73	13
Mohawk	1,987	1,917	***70***	***14***
Chippewa	1,903	1,833	***70***	***14***
Norfolk	2,032	1,970	62	16
Vermillion	1,659	1,602	57	17
Adams	1,814	1,759	55	18
Holland	2,116	2,062	54	19
McKinley	1,356	1,303	53	20
Manhattan	1,200	1,149	51	21
Sky	2,312	2,262	50	22
Gramblin	1,075	1,027	48	23
Cherokee	1,519	1,474	45	24
Cayuse	1,202	1,158	***44***	***25***
Wilson	1,079	1,035	***44***	***25***
Paris	1,690	1,649	41	27
Oxford	1,140	1,104	***36***	***28***
Ford	1,423	1,387	***36***	***28***
Niagra	1,757	1,722	***35***	***30***
Benton	1,047	1,012	***35***	***30***

Treatment of Ties in the RANK DISPLAY Column

Sometimes you may have two or more rows that are tied in the ranking, having the same value in the *ranked-on* column. When two or more values are tied, they will have the same rank number and the rank number following will be adjusted upward to account for the ties. In the table above, the ties are shown in bold type. Note, for example, that the two rows with the value 70 have the same rank number of 14 and the next rank number is 16.

Troubleshooting the RANK DISPLAY Column

If you get a table with 1's in the RANK DISPLAY column where you would expect to see meaningful rank numbers, the likely cause is that you accidentally used the column number of your RANK DISPLAY column in the RANK statement. This is easy to do, for example, if you are ranking on a Total column at the beginning of the table and then add a RANK DISPLAY column to the left of it without remembering to change the RANK column to COLUMN 2.

If you put a RANK DISPLAY column in a table that does not have a RANK variable, you will get a column of 0's.

The NORANK Footnote

Sometimes you may have rows in your table for which there is no rank number. This will happen if not all rows of the table are ranked. The symbol for the built-in footnote named NORANK will appear in the RANK DISPLAY column for these rows. **Blank is the default** for the symbol and text. To change the symbol or text, use a SET FOOTNOTE statement.

In the following table, there is ranking by age but also a total row above the ranked rows and unranked month rows below. Column 1 is the RANK DISPLAY column. There is no rank number for the rows not included in the ranking, so the NORANK symbol appears for these rows. In this case, a SET FOOTNOTE statement has been used to replace the default blank with ... and add a footnote text to be displayed at the bottom of the table.

```
RANK M_AGE_GROUPS "Mother's Age" on MAGE COLUMN 2;
"<15" if < "15";
"15-19" if "15": "19";
"20-24" if "20": "24";
"25-29" if "25": "29";
"30-34" if "30": "34";
"35-39" if "35": "39";
"40-44" if "40": "44";
"45+" if "45": < "99";
"N.S." if "99";

RANK DISPLAY RNUM "Rank" MASK 9;

SET FOOTNOTE NORANK SYMBOL "..."
        TEXT "Rank number not applicable.";
```

```
TABLE TABLE_NR "Total Births, Births Ranked by Mothers Age, "
      "and Births by Birth Month":
HEADING      RNUM then TOTAL then PLURAL;
STUB         TOTAL then M_AGE_GROUPS then BMONTH;
```

Total Births, Births Ranked by Mothers Age, and Births by Birth Month

	Rank	Total	Plurality				
			Single	Twins	Triplets	Quadruplets	Quintuplets or more
Total	...	119,349	115,365	3,761	206	12	5
Mother's Age							
20-24	1	32,605	31,764	826	15	–	–
25-29	2	31,666	30,611	1,006	40	4	5
30-34	3	27,287	26,116	1,078	85	8	–
15-19	4	13,895	13,645	241	9	–	–
35-39	5	11,376	10,845	489	42	–	–
40-44	6	2,119	2,007	97	15	–	–
<15	7	303	299	4	–	–	–
45+	8	96	76	20	–	–	–
N.S.	9	2	2	–	–	–	–
January	...	10,123	9,811	304	8	–	–
February	...	9,281	8,975	304	–	2	–
March	...	9,914	9,569	315	30	–	–
April	...	9,508	9,184	309	15	–	–
May	...	9,866	9,502	349	15	–	–
June	...	9,723	9,385	311	27	–	–
July	...	10,575	10,198	354	21	2	–
August	...	10,704	10,364	328	12	–	–
September	...	10,155	9,866	260	20	4	5
October	...	10,205	9,890	297	14	4	–
November	...	9,321	9,011	296	14	–	–
December	...	9,974	9,610	334	30	–	–

... Rank number not applicable.
– Data not available.

Referencing Ranked Rows in Format Statements

If you are using Format statements to reference ranked rows, you cannot determine row numbers by counting data rows in the printed table. The row numbers are assigned before the ranking, so they will not match the order in the table after the ranking. You can find the row numbers for **PRINTED ROWS** in the OUTPUT file.

If you have ranked rows and reference an empty row, the statements EJECT AFTER ROW and BANK AFTER ROW will have no effect. For ranked rows, you need to reference a row that has data.

CHAPTER 16

Weighting

CREATING MULTIPLIERS WITH THE WEIGHTING STATEMENT

The WEIGHTING statement lets you create a variable that contains multipliers for use in a TABLE statement. It is especially useful in tables where there are many observation variables that need to be multiplied by one or more weights or other variables within the same table.

Format

```
WEIGHTING new-variable ['print label'] :
                                          ON

'label-1'       =       obs-var-1;
['label-2'      =       obs-var-2;
.
.
.
['label-n'      =       obs-var-n;
```

where **new-variable** is the name of the WEIGHTING variable that can be used in a TABLE statement. If you assign a print label to the WEIGHTING variable, it will be used in the table; otherwise it will have no label in the table.

Listed under the WEIGHTING variable are one or more observation variables, **obs-var-1**, **obs-var-2**, ...**obs-var-n**, that will be used as multipliers for the values in the table cells. You must assign a print label to each of these variables.

There are many options associated with print labels such as upper and lower case letters, special characters and footnotes. Any of these options, as described in the "Labels" chapter, can be used in this statement.

When the WEIGHTING variable is used in a TABLE statement, a set of table cells is created for each of the multiplier variables listed beneath it. Each set of cells will be multiplied by its corresponding multiplier variable.

The multiplier variables can be codebook observation variables or observation variables created with a COMPUTE statement. You can use the built-in variable COUNT or a record name as a multiplier if you want to multiply by 1 to get the equivalent of no multiplication.

A constant cannot be used directly as a multiplier. If you want to use a contstant as a multiplier, create one with a COMPUTE statement using COUNT. For example:

```
COMPUTE three = COUNT * 3;
```

Only one WEIGHTING variable can be used per table request, but the WEIGHTING variable does not need to be used in all tables. Within a table, it can be nested with only part of the table.

Example

```
Weighting MULTIPLIERS:
'Weighted' = WEIGHT;
'Unweighted' = COUNT;
```

If we nest the variable called MULTIPLIERS in a TABLE statement, then we will get two sets of cells. One set will be labeled 'Weighted'. For this set, the observation values will be multiplied by WEIGHT as they are aggregated. The other set will be labeled 'Unweighted'. For this set, all observation values will be multiplied by COUNT as they are aggregated.

Note that since COUNT is a built-in variable with the value 1, multiplication by COUNT is the same as multiplication by 1; it will have no effect on its set of cells. Thus, the resulting table will have a set of weighted cells and a set of unweighted cells.

In the following TABLE statement, the columns contain values for the observation variables, ASSETS, CASH, BAD_DEBTS, INVENTORY and DEPRECIATION. By nesting MULTIPLIERS in the stub of the table with the BY operator, we get alternating rows of weighted and unweighted values for each of these observation variables.

```
table W1  font hb 11 'Table W1—Weighted and unweighted financial '
          'information for corporations in the mining industry.';
     stub MINING BY MULTIPLIERS;
     heading ASSETS
          then CASH_
          then BAD_DEBTS
          then INVENTORY
          then DEPRECIATION;
```

Table W1--Weighted and unweighted financial information for corporations in the mining industry.

	Assets	Cash	Bad debt	Inventory	Depreciation
Total					
Weighted	$39,712,367	$3,863,969	$4,999	$348,220	$33,804,634
Unweighted	2,056,827	117,527	2,287	121,858	1,021,388
Metal mining					
Weighted	5,083,702	390,324	–	57,496	946,775
Unweighted	207,935	11,438	–	7,379	40,277
Coal mining					
Weighted	6,489,075	538,630	1,156	169,496	11,579,667
Unweighted	579,445	27,832	1,156	65,241	435,290
Oil and gas extraction					
Weighted	10,500,000	672,444	3,843	111,560	12,326,483
Unweighted	709,059	24,865	1,132	47,651	335,446
Nonmetalic minerals, except fuels					
Weighted	17,639,591	2,262,571	–	9,668	8,951,710
Unweighted	560,389	53,391	–	1,587	210,375

– Data not available.

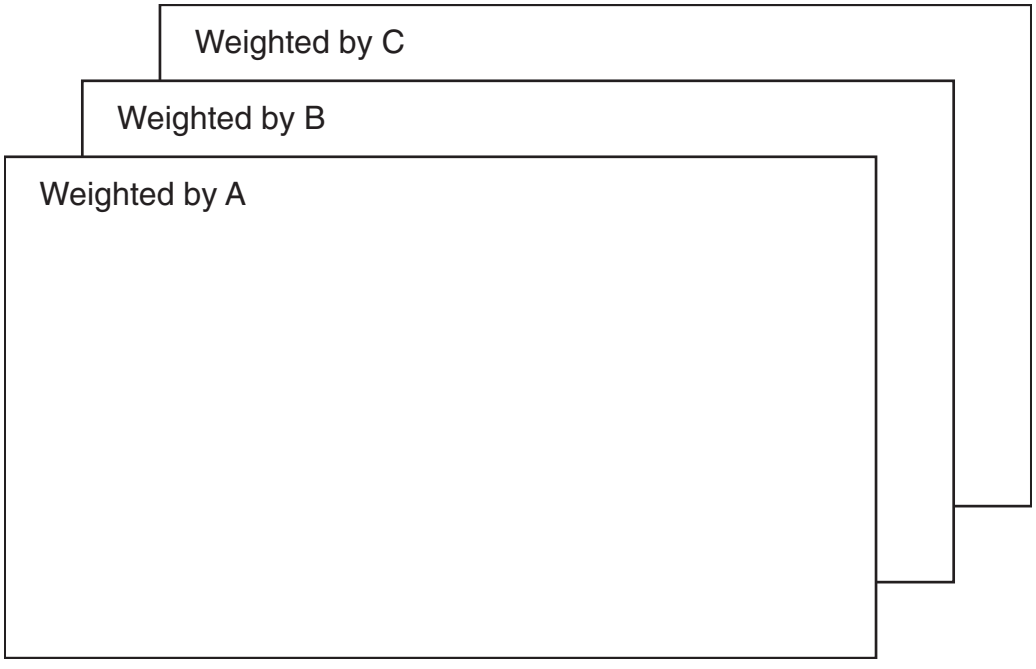
If we reversed the stub and heading in the table statement, we would get alternating columns of weighted and unweighted values.

If we used the WEIGHTING variable in a wafer expression, we would get a wafer of weighted values and a wafer of unweighted values.

Example Suppose our data has three different weights, WT_A, WT_B and WT_C. We would like to apply each weight in a different wafer of a table and then compare the results. To do this, we list the three weights in a WEIGHTING statement and then use the WEIGHTING variable as the wafer variable in a TABLE statement.

```
weighting WGT:
'Weighted by A' = WT_A;
'Weighted by B' = WT_B;
'Weighted by C' = WT_C;

table WAFER_EXAMPLE:    WAFER WGT,
                        STUB stub expression;
                        HEADING heading expression;
```



Effect of WEIGHTING on Variables Created with other Statements

Percents

If the base and numerator of a percent have the same WEIGHTING variable, the percent value will not change.

Medians and other Quantiles

Only the **weight** variable is weighted, if present. The **rank** variable is not.

Max, Min, Union (U) and Intersection (I)

These functions are not affected by nesting with a WEIGHTING variable.

COMPUTE

Constants are not weighted, but each variable within a COMPUTE statement is weighted individually.

POST COMPUTE

If a POST COMPUTE variable is nested with a WEIGHTING variable, then each variable within the POST COMPUTE will be weighted separately. Thus, for example, an average will become a weighted average.

Constants are not weighted. For example, suppose that we wish to create a table similar to the one in our earlier example, but we want to show the data in thousands of dollars. We can do this by dividing each value by 1000 in a POST COMPUTE statement. Each variable used in the POST COMPUTE is multiplied by the WEIGHTING multipliers, but the constant 1000 is not affected.

Example

```
post compute ASSETS_T 'Assets' mask $99,999,999  =
        ASSETS/1000;
post compute CASH_T 'Cash' mask $99,999,999 = CASH/1000;
post compute BAD_DEBTS_T 'Bad debt' mask $99,999,999 =
        BAD_DEBTS/1000;
post compute INVENTORY_T 'Inventory' mask $99,999,999 =
        INVENTORY/1000;
post compute DEPRECIATION_T 'Depreciation' mask $99,999,999 =
        DEPRECIATION/1000;

Weighting MULTIPLIERS:
'Weighted' = WEIGHT;
'Unweighted' = COUNT;

table W2  font hb 11 'Table W2—Weighted and unweighted financial '
        'information for corporations in the mining industry.'/
        font h 9 '(Money amounts are in thousands of dollars.)':
    stub MINING BY MULTIPLIERS;
    heading ASSETS_T
        then CASH_T
        then BAD_DEBTS_T
        then INVENTORY_T
        then DEPRECIATION_T;
```

Table W2--Weighted and unweighted financial information for corporations in the mining industry.

(Money amounts are in thousands of dollars.)

	Assets	Cash	Bad debt	Inventory	Depreciation
Total					
Weighted	$39,712	$3,864	$5	$348	$33,805
Unweighted	2,057	118	2	122	1,021
Metal mining					
Weighted	5,084	390	–	57	947
Unweighted	208	11	–	7	40
Coal mining					
Weighted	6,489	539	1	169	11,580
Unweighted	579	28	1	65	435
Oil and gas extraction					
Weighted	10,500	672	4	112	12,326
Unweighted	709	25	1	48	335
Nonmetalic minerals, except fuels					
Weighted	17,640	2,263	–	10	8,952
Unweighted	560	53	–	2	210

– Data not available.

A Note of Caution on Multiplication in Post Compute

Multiplication is rarely used in Post Compute statements. However, if you do have a Post Compute in which one variable is multiplied with another, please note that nesting with a WEIGHTING variable is likely to produce a different result from what is desired or expected.

For example:

```
post compute X  = A * B;
```

Since each variable in the Post Compute is weighted separately, the result will be a tabulation of W * A multiplied by a tabulation of W * B. In other words, the weight will be factored in twice.

Masks for Output Formatting

WEIGHTING statements cannot contain masks. When a WEIGHTING variable is nested in a table, the masks, if present, are taken from the observations variables that are being weighted.

CHAPTER 17

Char

CREATING A NEW CHARACTER VARIABLE

The CHAR statement creates a new character variable by combining all or part of other character variables and text. Its primary use is in creating variables for use in TPL reports. It also may be used in TPL TABLES to create variables for use in SELECT statements.

Format

```
CHAR new-variable ['print-label'] = construction ;
```

where **new-variable** is a character variable and **construction** is made up of one or more of 'character-string', substr(character-variable, start) and substr(character-variable, start, length) concatenated together using '+' or '||'.

Example

In our first example we wish to select all **items** which begin with 'A';

```
CHAR SELECT_VAR = SUBSTR(ITEM,1,1);
SELECT IF SELECT_VAR = 'A';
```

In our next example we wish to create a TPL report which includes the full names of people in a data file. The data is stored with first name, FNAME, in one field and last name, LNAME, in another. If we just use FNAME and LNAME in the report, then the fields will be in separate columns with varying numbers of blanks between them. Instead we use:

```
CHAR FULL_NAME 'Name' = FNAME + ' ' + LNAME;
```

Note that we have included a blank between FNAME and LNAME since otherwise the fields would be run together.

CHAR SPLIT: DIVIDE A CHARACTER VARIABLE

The SUBSTR function can be used to split a fixed-format data field into parts. If the data is not fixed-format, then CHAR SPLIT must be used.

Format

CHAR SPLIT old-variable: variable$_1$ "divider$_1$" variable$_2$ "divider$_2$" ...
CHAR SPLIT old-variable: "divider$_1$" variable$_1$ "divider$_2$" variable$_2$ ...

where old-variable is the variable which is to be divided into two or more new variables. The dividers, which are a list of characters, are entered in quotes. They separate the subfields in the data.

Example

Suppose your CSV data file has a DATE field. The field on different records are:

```
1/23/45    and
12/22/46
```

You wish to split the DATE field into three new observation variables, MONTH, DAY, and YEAR. You can do this with:

```
CHAR SPLIT DATE: CMONTH "/" CDAY "/" CYEAR;
COMPUTE MONTH = OBS(CMONTH);
COMPUTE DAY = OBS(CDAY);
COMPUTE YEAR = OBS(CYEAR) + 1900;
```

Note

Dividers need not be the same from subfield to subfield. All characters in the divider list between two subfields are discarded. regardless of order.

Example

Suppose you have a data field **VALUES** which is: [+ 34 +A 23] and you wish to extract just the two numbers. You can use

```
CHAR SPLIT VALUES: " +" VALUE1 "A+ " VALUE2;
```

VALUE1 will get the value 34 while VALUE2 will get 23.

CHAPTER 18

Hierarchies

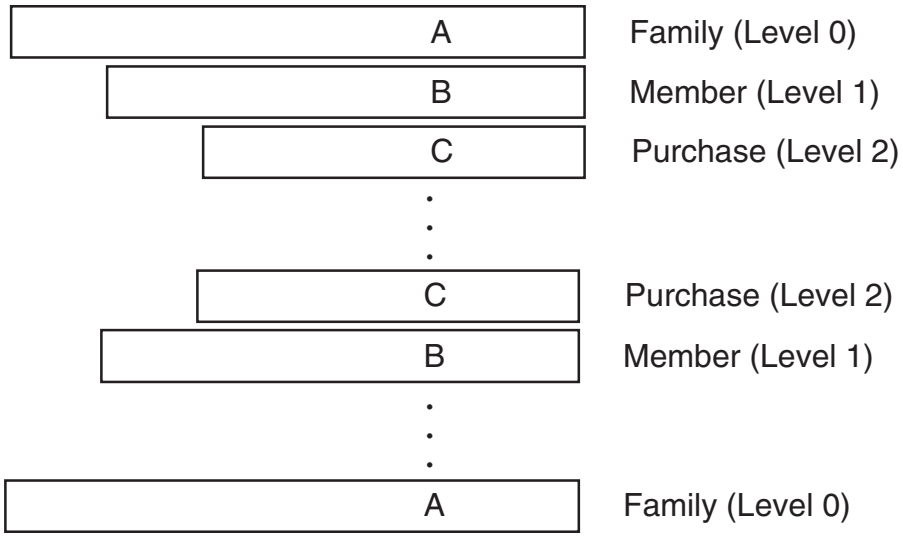
PROCESSING HIERARCHICAL FILES

Introduction

A hierarchical file consists of multiple record types, each related to the other but describing a different level of detail. The records are sequenced so that for records at any level of detail, a variable number of more detailed records may follow.

TPL TABLES can tabulate at different levels of a hierarchical file as will be described in this chapter. A repeating group is another type of data structure that has many of the attributes of a hierarchical file. Repeating groups are described in a separate chapter. Note that the level at which tabulation occurs can change for a hierarchical file if repeating groups are added to its codebook. The interaction is described in the TABLE statement section of this chapter and in the repeating group chapter.

The following diagram illustrates the concept of hierarchically related records. Lower level records are shown indented to suggest subordination within the hierarchy. Records containing the greatest detail are dependent on the next higher level of the hierarchy and so on back up to the Master level. The master record has a level number of zero indicated by LEVEL 0. The record type immediately subordinate to the Master record is LEVEL 1. Level numbers increase in increments of one to the level number of the lowest order of subordination in the hierarchy. The structure of the three-level FAMILY hierarchical file is shown next, followed by the codebook description. Note that some unique value (e.g., A,B,C) within each record must uniquely identify each level of the hierarchy.



Each level of the hierarchy is identified in the codebook by a record marker and a level number. For example, each family record may be identified by a marker of **1** in record position 1 and each member record may be identified by a marker of **2** in record position 1.

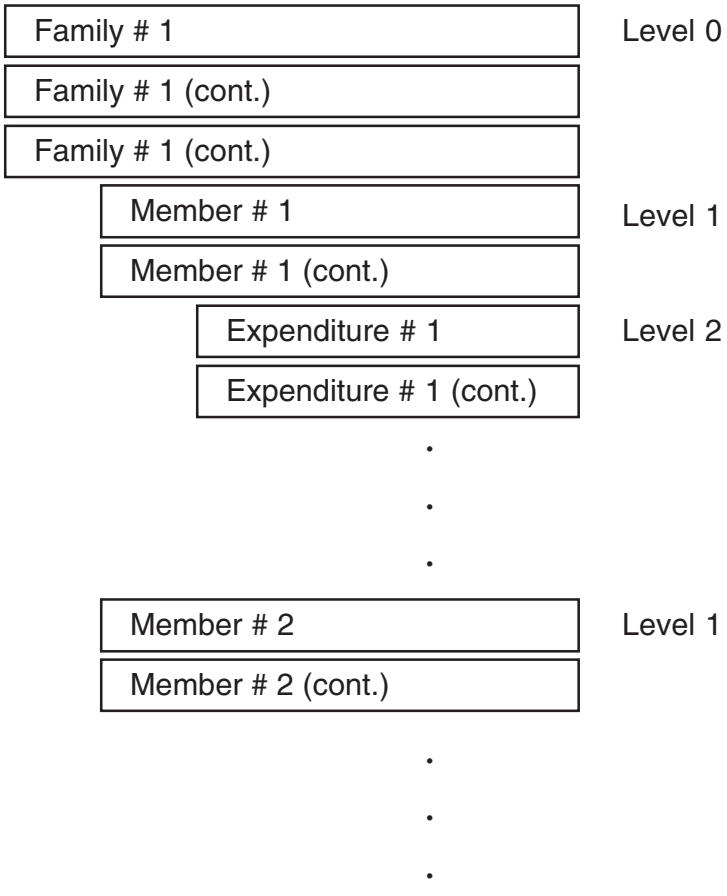
Let us assume that each family record in a data file has one or more family members, and each member record has at least one purchase record associated with it. When processing this file, TPL TABLES will read the first family record, the first member record, and the first purchase record. These three levels will form a hierarchical unit for TPL TABLES to act on. Next, the second purchase record (if any) will replace the first to form another hierarchical unit.

After all purchase records have been combined individually with the first family and member record, the second member record is read, if any. The first family record is paired with the second member record and each purchase record of that member in turn. After all member records for the first family and their purchase records have been processed, the next family record is read and the cycle is repeated.

Each record or collection of records is assigned a level number whose value depends on its hierarchical relationship to other records. The record type which identifies a major new processing unit in the file, and which is not subordinate to other records, is known as a master record or level 0 type record. In our example, the family record is a level zero record and is identified as such in the codebook. The first record subordinate to the family record is the member record which is identified as a level 1 type record in the codebook.

Since the purchase record occurs one or more times for each occurrence of the member record, it has a level number of 2. Each member record must be followed by at least one purchase record. Two successive member records without at least one purchase record in between means that the hierarchy is incomplete.

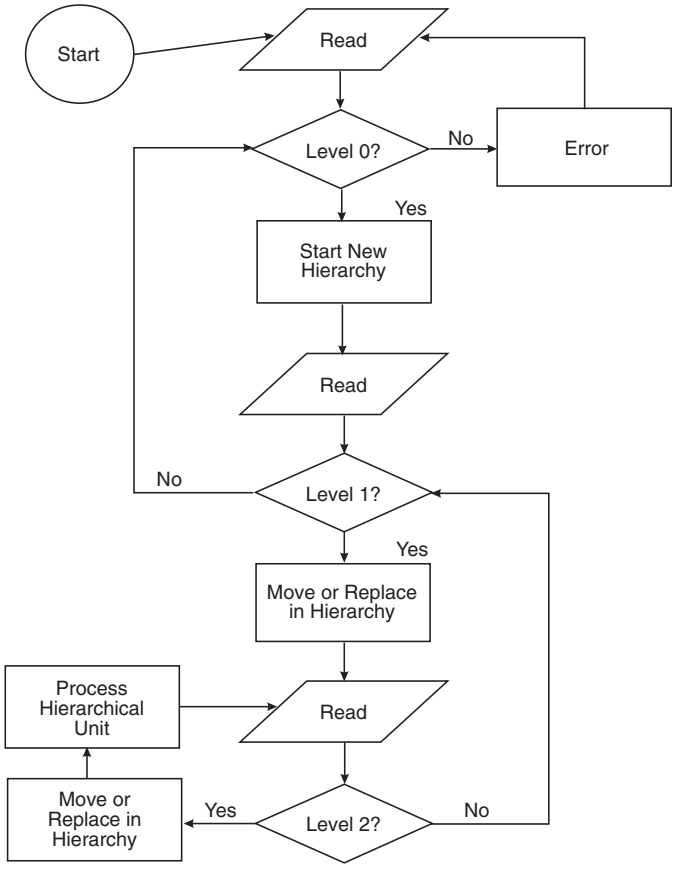
Each level of the hierarchy can consist of more than one record type, but there must be only one of each type in succession at that level. In the following illustration member information spans over two records and family information spans over three records.



The first record type of each hierarchical level must be uniquely identified to TPL TABLES by specifying a unique record value, of any length, in the codebook. In this way missing levels can be detected when the data file is read. If a level is skipped (e.g., LEVEL 0 to LEVEL 2), an error message will indicate that there are records that are not in the expected sequence. When the next record of the highest level (e.g. LEVEL 0) is found, processing will resume.

Below is a flowchart showing the tests made for a three-level hierarchical file.

General Processing for a Three-Level Hierarchical File



Codebook Entries

The first record of each level of a hierarchical file must contain a record identifier by which the record can be uniquely identified. This record identifier follows the key word MARKER in the codebook RECORD clause and is a data name which must be described somewhere within the record description. The data name must be the name of a control variable, and the value following the equal sign must be within quote marks. The record MARKER must be accompanied by a record LEVEL number. (For additional details, see the Record Name Clause section of the chapter on "Describing a TPL TABLES Input Data File".)

```
BEGIN HIERARCHY CODEBOOK

FAMILIES RECORD MARKER FMID ='A' LEVEL 0
    FMID CON 1    ( = 'A' )
    REG CON 1
      (
         'Northeast'  =  1
         'Midwest'    =  2
       )
    JOB CON 1 (1:9)
    AGE CON 2
    CONDITION LABEL IS VALUE
      (     16:99     )
    PERSONS OBS 2
    INCOME OBS 5

MEMBER RECORD MARKER MBID='B' LEVEL 1
    MBID CON 1 (='B')
    AGE_M OBS 2
    SEX CON 1
      (
         'Male'       =  1
         'Female'     =  2
      )
    OCCUPATION OBS 3
    FILLER 5

PURCHASES RECORD MARKER PRID ='C' LEVEL 2
    PRID CON 1 (='C')
    ITEM CON 1
      (
         'Bread'      =  1
         'Fish'       =  2
         'Milk'       =  3
         'Eggs'       =  4
      )
    COST OBS 3
    DAY  CON 1
      (
         'Day 1'      =  1
         'Day 2'      =  2
         'Day 3'      =  3
      )
    PKG_T CON 1 (1:2)
    FILLER 5

END HIERARCHY CODEBOOK
```

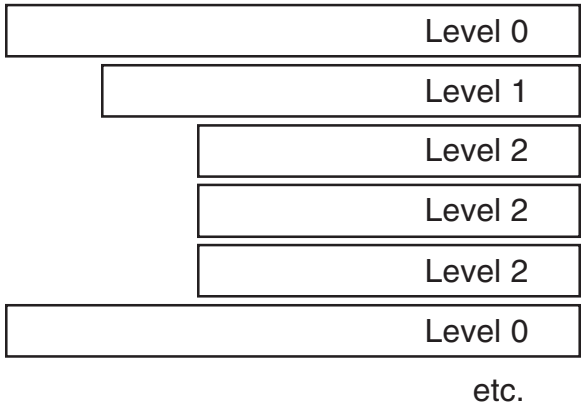
Using Incomplete Hierarchies

Default Treatment

Normally, a hierarchical unit processed by TPL TABLES must be complete. For example, a three level hierarchy cannot consist of only Level 0 records and Level 2 records. Before any processing is done on a hierarchical unit, at least one record must be present at each level. Incomplete hierarchies are reported as errors. If any level is missing, all subsequent records are discarded until a new record is found at the highest level (i.e. the lowest level number).

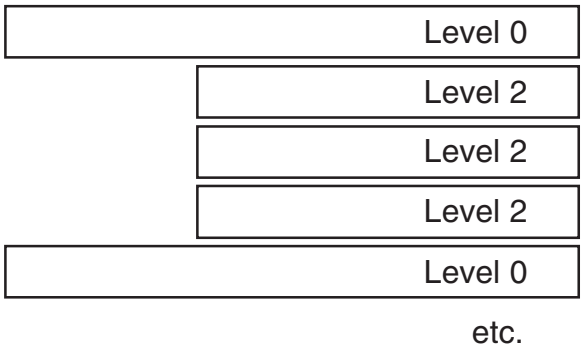
The following sequence of records showing a complete hierarchy will be used as the basis for two examples of incomplete hierarchies.

Complete Hierarchy

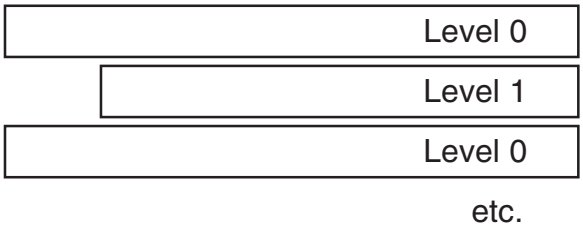


Examples of Incomplete Hierarcies

1. Missing Level 1



2. Missing Level 2



Incomplete hierarchy messages show where the problem occurred in the data file and where processing was resumed. For example:

*** ERROR: Records out of sequence. Level 2 expected for record 6

*** ERROR: Recovered from level error on record 8

Forcing Tabulation of Incomplete Hierarchies

The statements

```
TABULATE INCOMPLETE HIERARCHIES = YES; (NO is the default)
```

and

```
REPORT INCOMPLETE HIERARCHIES = NO; (YES is the default)
```

can be used to control the treatment of incomplete hierarchies. By choosing TABULATE INCOMPLETE HIERARCHIES = YES; you can tabulate

data from higher level records even though records are missing at the lowest levels, as long as information from the missing records is not required to do the tabulation.

Note that records cannot be missing from middle levels. For example, a Level 1 record cannot be tabulated if it is followed by a Level 3 record.

The INCOMPLETE HIERARCHIES statements can be entered either in the codebook after the BEGIN CODEBOOK statement or in the table request after the USE statement. A statement in the table request will override any conflicting statement entered in the codebook.

Example of Statements in Codebook

```
BEGIN hierarchy CODEBOOK

TABULATE INCOMPLETE HIERARCHIES = YES;

families RECORD MARKER achar = 'A' LEVEL 0

  FILLER 4

  achar CON 1          /* The marker field.  'A' is the only      */
        ( = 'A' )      /* valid value for this type of record.   */

  month OBS 2          /* The month of the survey.               */
        .
        .
        .
```

Example of Statements in Table Request

```
USE hierarchy CODEBOOK;

TABULATE INCOMPLETE HIERARCHIES = YES;
REPORT INCOMPLETE HIERARCHIES = NO;

DEFINE quarter ON month;
        '1st Quarter' if 1:3;
        '2nd Quarter' if 4:6;

TABLE toplevel:  HEADING TOTAL THEN quarter;
                 STUB  .
                       .
                       .
```

If variables referenced in a SELECT statement or used in a TABLE statement are located in a missing level, no tabulation will take place for that hierarchical unit.

If a request combines tabulations requiring the missing level with tabulations that do not depend on the missing level, each will be evaluated independently.

If a tabulation depends on a lower level record, such as when only control variables are used, then all records which make up the hierarchical unit down to that record must be present for processing to take place.

For example, consider a three level hierarchical file consisting of family (Level 0), member (Level 1), and expenditure records (level 2). If a crosstabulation consists of control variables and an observation from the family record, including record name, then member and expenditure records do not contribute to the tabulation, so they can be missing if you have specified TABULATE INCOMPLETE HIERARCHIES = YES; in your codebook or table request. However, if a tabulation involves only control variables from the family record, then the lowest level expenditure records need to be counted and both member and expenditure records must be present.

Message Suppression

By default, incomplete hierarchies will be reported even if they are tabulated. To suppress incomplete hierarchy messages, use the statement

```
REPORT INCOMPLETE HIERARCHIES = NO;
```

How Hierarchies Interact with TPL TABLES Statements

The following sections explain how each TPL statement reacts with a hierarchical file. First, you should think of all records which make up a hierarchical unit (level 0 through a single occurrence of the lowest level) as being read into one contiguous area. The result may be thought of as one long record representing a processing unit. After this unit is processed, another record will be read. If this record belongs to the lowest level of the hierarchy it will replace the preceding record having the same level number, and the resulting new hierarchical unit will be processed again.

If a record is read which belongs to a higher level of the hierarchy (lower level number), that level will be replaced, and successive reads will replace

all lower level records until another hierarchical unit is formed. Incomplete hierarchical units will be recognized and an appropriate diagnostic message issued. Normally, only complete hierarchical units will be processed. (See the section on Using Incomplete Hierarchies if you need to tabulate data from incomplete units.)

TABLE Statement

To aid in understanding what tabulations result when accessing variables from different hierarchical levels, we will use a sample data file of six records, with the variable name and value appearing in each field for convenient reference, followed by a codebook description for this file.

```
FAMILIES   REG=1   JOB=2   AGE=18   PERSONS=3   INCOME=11000   A

 PURCHASES                    BREAD    COST=70   DAY=1   PKG_T=1      C

 PURCHASES                    FISH     COST=80   DAY=2   PKG_T=1      C

 PURCHASES                    BREAD    COST=25   DAY=1   PKG_T=2      C

FAMILIES   REG=2   JOB=1   AGE=20   PERSONS=4   INCOME=13000   A

 PURHASES                     FISH     COST=100  DAY=3   PKG_T=2      C
```

```
BEGIN TWO LEVEL CODEBOOK

FAMILIES RECORD MARKER FMID='A' LEVEL 0
  REG CON 1
        (
          'REG=1'      = 1
          'REG=2'      = 2
        )
  JOB CON 1 (1:2)
  AGE CON 2
        (
          'AGE=18'     = 18
          'AGE=20'     = 20
        )
  PERSONS OBS 2
  INCOME OBS 5
  FMID  CON 1 (='A')
```

```
PURCHASES RECORD MARKER PRID='C' LEVEL 1
  ITEM CON 1
       (
         BREAD       = 1
         FISH                = 2
         MILK = 3
         EGGS        = 4
       )
  COST OBS 3
  DAY  CON 1
       (
         'DAY=1'     = 1
         'DAY=3'     = 3
       )
  PKG_T CON 1 (1:2)
  PRID CON 1 (='C')
END TWOLEVEL CODEBOOK
```

This section will show the actual tabulations that will result when various TABLE statements are processed against the above data file and codebook. The results of a tabulation depend on the hierarchical levels at which the variables are located, according to three rules. Each TABLE statement is followed by a brief explanation of the table content.

Rule 1

When only control variables are crossed or nested in a TABLE statement, the default observation variable is the lowest level (highest level number) record name, having an assumed value of 1.

Note that there is one exception to this rule. If one or more repeating groups are included in the codebook for a hierarchical file, the default observation variable is the top (level 0) record name. Thus, the addition or deletion of a repeating group in a codebook can affect the results of tabulations that do not have explicit observation variables specified. See the chapter on repeating groups for additional information. This chapter assumes that no repeating group variables are in the codebook.

Examples TABLE H1: ITEM, DAY;

Counts purchases made of each item on each day of the week.

	DAY=1	DAY=2	DAY=3
BREAD	2	–	–
FISH	–	1	1

TABLE H2: REG BY ITEM, DAY;

Same as above except purchases summarized by each region.

	DAY=1	DAY=2	DAY=3
REG=1			
BREAD	2	–	–
FISH	–	1	–
REG=2			
FISH	–	–	1

TABLE H3: REG, AGE;

Counts purchases according to region and age of household head.

	AGE=18	AGE=20
REG=1	3	–
REG=2	–	1

Rule 2

When an observation variable is crossed or nested with a control variable at the same or higher level (lower level number), that observation variable is aggregated from each occurrence of the record in which it is located.

Examples

```
TABLE H4:   WAFER  FAMILIES,
            STUB   REG,
            HEADING  AGE;
```

Counts families according to region and age categories. Lower level purchase records are not tabulated.

```
TABLE H5: ITEM, COST;
```

Cost for each item is aggregated from the purchase record.

	COST
BREAD	70+25
FISH	80+100

```
TABLE H6: REG, ITEM THEN COST;
```

Counts purchases of each item and overall cost of each item for each region.

	BREAD	FISH	COST
REG=1	1+1	1	70+80+25
REG=2	–	1	100

```
TABLE H7: TOTAL THEN REG, ITEM THEN COST;
```

Same as above, except also summarized over all regions.

	BREAD	FISH	COST
Total	1+1	1+1	70+80+25+100
REG=1	1+1	1	70+80+25
REG=2	–	1	100

TABLE H8: AGE THEN PKG_T, COST;

Total cost of all purchases summarized by age of household head and package type.

	COST
AGE=18	70+80+25
AGE=20	100
1 PKG T	70+80
2 PKG T	25+100

TABLE H9: REG, PERSONS THEN INCOME;

Total number of persons and total family income summarized by region.

	PERSONS	INCOME
REG=1	3	11,000
REG=2	4	13,000

Rule 3

When an observation variable is crossed or nested with a control variable at a lower level, that observation variable (including the record name, having a value of 1) is aggregated once for each unique value of that control variable. The control variable TOTAL can be thought of as belonging to any level of the hierarchy since the results are the same.

Examples

This third rule is extremely useful in the situations where you want, for example, to count families that made purchases of a particular item or collection of items, no matter how many of those items appeared at the lower level. For example, you can count families who bought bread, no matter how many purchases of bread were made.

In another application involving Establishment records at level 0 and Employee records at level 1, it may be desired, for example, to count Establishments having at least one bookkeeper.

```
TABLE H10: ITEM, FAMILIES;
```

Counts the families who bought each item at least once.

	FAMILIES
BREAD	1
FISH	2

```
TABLE H11: ITEM, PERSONS;
```

Counts persons in families who bought each item at least once.

	PERSONS
BREAD	3
FISH	3+4

```
TABLE H12:  WAFER  FAMILIES,
            STUB  REG,
            HEADING  DAY;
```

Counts families in each region who bought something on each day.

FAMILIES

	DAY=1	DAY=2	DAY=3
REG=1	1	1	–
REG=2	–	–	1

TABLE H13: REG, FAMILIES BY ITEM THEN COST;

For each region, the number of families who purchased each item and the total cost of the purchases (combines Rule 2 and Rule 3).

	FAMILIES		COST
	BREAD	FISH	
REG=1	1	1	70+80+25
REG=2	–	1	100

TABLE H14: ITEM, FAMILIES BY (TOTAL THEN REG);

Total families who purchased each item and the total purchases for each region.

	FAMILIES		
	Total	REG=1	REG=2
BREAD	1	1	–
FISH	1+1	1	1

TABLE H15: ITEM, TOTAL THEN FAMILIES BY REG;

Total number of purchases of each item, and the number of families in each region making at least one purchase. Note that there were two purchases of bread, all made by one family in REG=1.

	Total	FAMILIES	
		REG=1	REG=2
BREAD	1+1	1	–
FISH	1+1	1	1

TABLE H16: ITEM, FAMILIES BY AGE THEN PERSONS;

Counts families who bought each item according to the age of the head of the household, plus the total number of persons in families who bought each item.

	FAMILIES		PERSONS
	AGE=18	AGE=20	
BREAD	1	–	3
FISH	1	1	3+4

TABLE H17: WAFER INCOME,
STUB REG THEN DAY,
HEADING ITEM THEN PKG_T;

Aggregates income by region and day of week, according to items bought and package type. Income is aggregated once for each unique item code and package type.

INCOME

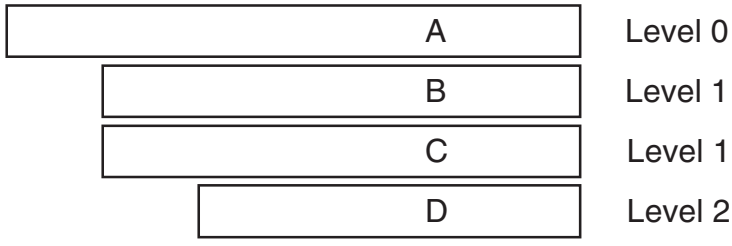
	BREAD	FISH	MILK	EGGS	1 PKG T	2 PKG T
REG=1	11,000	11,000	–	–	11,000	11,000
REG=2	–	13,000	–	–	–	13,000
DAY=1	11,000	–	–	–	11,000	11,000
DAY=2	–	11,000	–	–	11,000	–
DAY=3	–	13,000	–	–	–	13,000

The observation variable COUNT is equivalent to the record name at the lowest hierarchical level and when used in a TABLE statement will count the units at the lowest hierarchical level. Thus a TABLE statement containing only control variables will count the very lowest hierarchical units no matter at which level the control variables are located.

SELECT Statement

The result of a SELECT statement is that either the entire hierarchical unit is selected for processing by following TPL statements, or the next hierarchical unit is formed and tested again. Individual records of a hierarchical unit are never selected for processing alone, even though all variables tested may belong to an individual record.

Assume that a hierarchical file has the following structure.



If a variable is tested in record **D** and does not meet the SELECT condition, a new hierarchical unit is formed by replacing **D** with the next **D**, if one exists. This new hierarchical unit is tested again.

If a variable is tested in record **B** or **C** (both at level 1) and does not meet the SELECT condition, a new hierarchical unit is formed by reading past all following **D** records until a new pair of **B** and **C** records plus a new **D** record is found. This new hierarchical unit is tested again.

If a variable is tested in record **A** and does not meet the SELECT condition, a new hierarchical unit is formed from the next **A**, **B**, **C**, and **D** records. This new hierarchical unit is tested again.

COMPUTE Statement

A computed variable is assumed to belong to the lowest level record of the hierarchy which contains a referenced variable in the COMPUTE statement. For example, if a COMPUTE statement references only variables in

level 0 of a hierarchical file, then the computed variable will be assumed to belong to level 0. If a COMPUTE statement references variables in levels 0 and 1 of a three level hierarchy, then the computed variable will be assumed to belong to level 1. If variables in levels 0 and 2 are referenced, the computed value will become part of level 2. If the computation consists only of a literal value, the computed variable will be associated with level 0.

Example Assume a hierarchical file consists of a family characteristics record at level 0 followed by one or more member characteristics records at level 1. If the family record contains number of rooms in the household (ROOMS) and family size (PERSONS), the statement:

```
COMPUTE  ROOMS_PER_PERSON = ROOMS / PERSONS;
```

will associate ROOMS_PER_PERSON with the family record. A table can then be produced showing the sum of the average rooms per person for all families.

Conditional Compute Statement

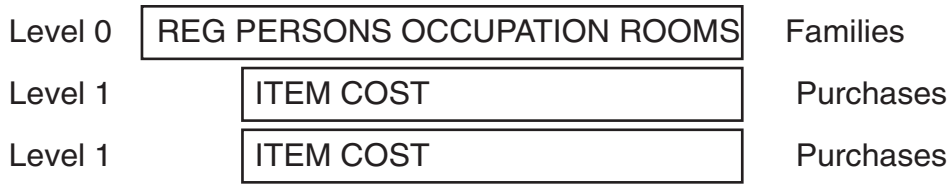
Like the COMPUTE Statement, the conditionally computed variable is assumed to belong to the lowest level record which contains a referenced variable (control or observation) in the Conditional Compute statement. For example, if we have the statement:

```
COMPUTE NEW_WEIGHT =
        1                   IF WEIGHT=4;
        PERSON_WEIGHT       IF OTHER;
```

If WEIGHT were at level 0 and PERSON_WEIGHT were at level 1, NEW_WEIGHT would always be assigned to level 1.

POST COMPUTE Statement

Assume that a hierarchical file of two levels consists of a family characteristics record at level 0 containing control variables region (REG) and occupation, and observation variables, number of persons in the household (PERSONS), and number of rooms in the household (ROOMS). The level 1 record contains a code for an item purchased and its cost.



Suppose it is desired to calculate the average rooms per person for each region and occupation. We can use the POST COMPUTE statement:

```
POST COMPUTE ROOMS_PER_PERSON = ROOMS / PERSONS;
```

Since only variables from the level 0 records are referenced in the POST COMPUTE statement, ROOMS and PERSONS will each be aggregated once for each family characteristics record. For each category of region and occupation, the final aggregated value of ROOMS will be divided by the final aggregated value of PERSONS, to get the average by using the Post Computed variable in the TABLE statement:

```
TABLE AVG:  REG, ROOMS_PER_PERSON BY OCCUPATION;
```

If we wish to know the average cost of each item classified by region, we need separate aggregations of COST and PURCHASE records containing the cost. The POST COMPUTE statement,

```
POST COMPUTE COST_PER_ITEM = COST/PURCHASE;
```

where PURCHASE is the name of the LEVEL 1 record, calculates the average. By using the Post Computed variable in the following TABLE statement, separate totals of COST and PURCHASE records will be obtained for each item code and each region over the entire file. The final count of PURCHASE records will be divided into the final aggregation of COST amounts for each item and region.

```
TABLE AVG_COST:  ITEM, COST_PER_ITEM BY REGION;
```

In general, the variables within a POST COMPUTE statement may be thought of as being aggregated separately as sub-cells and then replaced by a single amount according to the arithmetic expression of the POST COMPUTE statement.

As a final example we wish to produce a table showing the average cost per family member for each type of item purchased and occupation of head of household. For each family the number of persons in the family is to be

aggregated once for each unique item code. Cost is to be aggregated for each item code. We first use the POST COMPUTE statement:

```
POST COMPUTE COST_PER_PERSON = COST/PERSONS;
```

The Post Computed variable is then put into the TABLE statement:

```
TABLE AVG_PERSON_COST:
        ITEM, COST_PER_PERSON BY OCCUPATION;
```

Since COST_PER_PERSON is derived from COST (level 1) and PERSONS (level 0), each is aggregated according to its hierarchical relationships with control variables appearing in the TABLE statement. PERSONS will be aggregated once from each family for each unique value of item code, since PERSONS is at a higher level within the hierarchy than ITEM. COST will be aggregated for each ITEM code since they are both at level 1. Each final cell value may then be thought of as containing sub-cell values of total cost and total persons values which are then replaced with the average cost for display in the table.

DEFINE Statement

The defined variable may be assumed to apply to the record containing the old variable value, whether the old variable is in the codebook or computed. Using the TWOLEVEL codebook shown earlier, suppose that we want to count families in each region who made at least one purchase of either bread or eggs and the total cost of both.

```
DEFINE EGGS_BREAD ON ITEM;
        'Bread & Eggs' IF 1;
                       IF 4;

TABLE DEF_EXAMPLE:
        REGION, EGGS_BREAD BY (FAMILIES THEN COST);
```

MEDIAN and QUANTILE Statement

The observation variables created by these statements are associated with the hierarchical level containing the rank and weight variables. The rank variable must be at the same level as the weight variable. Note that if no weight variable is specified, the record name variable at the level of the rank variable is assumed to be the weight variable. This record variable has a value of 1 so the quantile is unweighted. See the discussion of weighting in the chapter on "Statistics" for more details.

CHAPTER 19

Repeating Groups

TABULATING VARIABLES THAT REPEAT WITHIN RECORDS

Introduction

When one variable or a collection of variables repeat within a record, they can be described as a repeating group. Repeating groups can greatly simplify tabulation with this type of data. They also allow table structures to be produced that would be awkward or impossible otherwise.

One example of a repeating group is a time series in which each data record contains a sequence of 12 values, one for each month of the year. Another example would be a survey questionnaire that contains a series of questions where each question has the same set of possible responses.

The repeating group feature lets you describe the repeating unit only once in the codebook and assign a name to it. You can also assign a name and/or label to each repetition so that the repeating group variable looks like a control variable with the same number of values as the number of repetitions in the group.

A repeating group can be viewed as a lower hierarchical level within a record and behaves in much the same way as a hierarchical level represented on separate records. The one exception occurs in a cross tabulation that does not have an explicit observation variable. In this case, the default observation variable is the record name at the highest level of the data file. Group repetitions are not counted unless a group-level observation variable is added to the tabulation. When a repeating group is described in a codebook, TPL TABLES automatically provides an observation variable that can be used for this purpose.

If you want to use a section of a data record both as individual items and as a repeating group, you can use the REDEFINE feature in the codebook to describe that section of data both ways. Then you can use either or both ways of looking at the data in your table requests.

Effect in Hierarchical Files

Note that the inclusion of one or more repeating groups in a codebook that describes a hierarchical data file *causes the default observation variable COUNT to be transferred from the lowest level of the hierarchy to the top level* (level 0) with the result that COUNT is the same as the record name at the top level.

A Time Series Example

In a single EMPLOYEE data record there could be a CITY field and an INCOME field followed by 12 successive months of employment and hours data arranged as follows:

```
CITY  | INCOME |January data        | .... |December data
      |        |(EMPLOYMENT)(HOURS) |      |(EMPLOYMENT)(HOURS)
```

Rather than assigning a unique name to each of 12 employment and hours fields within the record, a repeating group name can be assigned for the monthly data and each field within the group can be described only once. A REPEATS clause on the group variable says that the monthly data repeats 12 times. During tabulations of the repeating data, each monthly occurrence is processed in turn. This is similar to the treatment of a lower level record in a hierarchical data file.

Following are codebook entries that describe the record containing the 12 months of employment and hours data:

```
EMPLOYEE RECORD
  CITY CON 1
    (
       'Concordia'      = 1
       'Frostburg'      = 2
       'Silver Spring'  = 3
    )
  INCOME OBS 5
```

```
BEGIN GROUP MONTHLY_E_AND_H REPEATS 12
  (
     'January', 'February', 'March', 'April',
     'May', 'June', 'July', 'August',
     'September', 'October' ,'November', 'December'
  )
     EMPLOYMENT CON 1
       (
             'Employed'       = 1
             'Unemployed'     = 2
       )
     HOURS OBS 5
END GROUP MONTHLY_E_AND_H
```

The name following the phrase "BEGIN GROUP" is called the GROUP variable. It is an implied control variable with 12 values as specified in the REPEATS clause. A list of labels for each of the 12 values is shown in parentheses following the REPEATS clause. If the GROUP variable name is used in a TABLE statement, each of the repetitions will be displayed as a separate control variable condition in the table.

Each of the 12 repetitions of EMPLOYMENT and HOURS is associated with one of the group values and labels. The subordinate control and observation variables, EMPLOYMENT and HOURS, occur within each repetition of month. Any number of subordinate control or observation entries can be included within the repeating group.

The effect of repeating group usage will be illustrated using the variables from the above time series codebook in various TABLE statements.

```
TABLE G1:  STUB CITY,  HEADING HOURS;
```

Within each record, the HOURS value for each month will be aggregated into one of the CITY categories.

```
TABLE G2:  STUB EMPLOYMENT,  HEADING HOURS;
```

Within each record, the HOURS value for each month will be aggregated according to its employment code.

```
TABLE G3:  STUB EMPLOYMENT,
       HEADING HOURS BY MONTHLY_E_AND_H;
```

Since the group variable MONTHLY_E_AND_H has been used, hours will be aggregated by month for each employment code.

```
TABLE G4:  STUB EMPLOYMENT,  HEADING INCOME;
```

Income will be aggregated just once for each unique employment code in the record. This is because the repeating group is treated as a sub-record hierarchical level which is subordinate to the record entries outside the group. This follows the rule for hierarchical processing.

```
TABLE G5:  STUB EMPLOYMENT,
        HEADING INCOME BY MONTHLY_E_AND_H;
```

INCOME will be aggregated for each month according to the employment code. An aggregation will take place for each month regardless of employment code because each repetition is a unique condition. This table displays the total employee income for each month.

A Survey Questionnaire Example

For another application of repeating groups, suppose that an evaluation of county services is taken in which three successive character positions rate the quality of police protection, library services, and street maintenance, respectively. Each repetition of the quality variable rates a separate service and has a value of poor = 1, fair = 2 or good = 3.

Without using the repeating group feature, we would need to describe each question as a separate field with the same set of answers repeated for each one as follows:

```
POLICE 'Police' CON 1
  (
        'Poor'   = 1
        'Fair'   = 2
        'Good'   = 3
  )
LIBRARY ‘Library’ CON 1
  (
        'Poor'   = 1
        'Fair'   = 2
        'Good'   = 3
  )
STREETS ‘Streets’ CON 1
  (
        'Poor'   = 1
        'Fair'   = 2
        'Good'   = 3
  )
```

This data description could become quite lengthy with additional questions all having the same set of answers. In addition, it is impossible to create a table structure that shows the questions in one dimension and the answers in another, because there is no code to identify the data for the different questions. Instead, we only know which question we are looking at by its location in the record. We can solve these problems by describing the questions as a repeating group in the codebook:

```
BEGIN GROUP  SERVICES  REPEATS  3
  ('Police', 'Library', 'Streets')
        QUALITY CON 1
          (
                'Poor'   = 1
                'Fair'   = 2
                'Good'   = 3
          )
END GROUP  SERVICES
```

The repeating group SERVICES repeats 3 times. The control variable QUALITY within the group describes the possible answers for all of the 3 questions about services. Additional control or observation entries could have been included within the repeating group if required.

To show the table formatting flexibility, we will use the variables in three TABLE statements:

```
TABLE G6:  HEADING SERVICES BY QUALITY,  STUB TOTAL;
```

	Police			Library			Streets		
	Poor	Fair	Good	Poor	Fair	Good	Poor	Fair	Good
Total	X	X	X	X	X	X	X	X	X

```
TABLE G7:  HEADING QUALITY BY SERVICES,  STUB TOTAL;
```

	Poor			Fair			Good		
	Police	Library	Streets	Police	Library	Streets	Police	Library	Streets
Total	X	X	X	X	X	X	X	X	X

TABLE G8: HEADING QUALITY, STUB SERVICES;

	Poor	Fair	Good
Police	X	X	X
Library	X	X	X
Streets	X	X	X

Nesting or crossing the repeating group variable SERVICES with the variable QUALITY within the group has caused each type of service to be evaluated and cross tabulated.

The CONTINUE Option

Repeating groups can also be organized so that instead of groups of two or more adjacent data items being repeated, they may be repeated in parallel; that is, some items of the group are exhausted before the other items of the group are continued. In the case of twelve months of Hours and Employment data, the twelve months of Hours data may appear in succession, followed by the twelve months of Employment data.

January ..... December January ..... December
(EMPLOYMENT) (EMPLOYMENT) (HOURS) (HOURS)

The processing of the repeating group is identical regardless of whether the two data items repeat in pairs or the repetitions of one are followed by the repetitions of the other. The codebook language for indicating a continued group is:

```
BEGIN GROUP MONTHLY_E_AND_H REPEATS 12
  (
        'January','February','March','April','May','June','July',
        'August','September','October','November','December'
  )
  EMPLOYMENT OBS 5
END GROUP MONTHLY_E_AND_H
  .
  .
  .
CONTINUE GROUP MONTHLY_E_AND_H
  HOURS OBS 5
END GROUP MONTHLY_E_AND_H
```

All of the items within the first appearance of the group name will be repeated, then the items within the group continuation will be repeated. The group continuation need not immediately follow the first part of the group.

Continued repeating groups cannot bridge record types. If you have repeating groups that continue across multiple records at the same hierarchical level, you can change the codebook to describe the multiple records as a single record. Unless your codebook has the attribute BINARY at the beginning, you will need to account for characters between records. If you are running under Windows, use FILLER 2 in the codebook to account for the end-of-record characters between records at the same level. If you are running under UNIX, use FILLER 1.

Describing Repeating Groups in the Codebook

Format

```
BEGIN GROUP  group-name ['print label']
        [REDEFINES var-name]  REPEATS n

[(Name1 and/or Label1, Name2 and/or Label2, ...
        Name-n and/or Label-n)]

    elementary-item description(s) follow

END GROUP  group-name
```

The commas are required between group repetition names, labels, or name/label pairs. The repetition value, n, must have a value of 1 or more.

To describe a repeating group continuation, use:

Format

```
CONTINUE GROUP  group-name

    elementary-item description(s) follow

END GROUP  group-name
```

The following rules and comments apply to the description of repeating group variables in the codebook.

1. The REPEATS clause must have a value of 1 or more.

2. Within a repeating group there must be at least one elementary item. The elementary items can be control, observation, char or filler. In addition, groups can be contained within groups. We refer to this situation as "nested" repeating groups.

3. **The repeating group name is a control variable** which takes the values of 1 through n, where n is the repetition value. Each repetition can have an optional name and/or print label. If a name is provided for a repetition, but no label is provided, the name will be used as a print label. If no name or label is provided for a repetition, the label "n group-name" will be assigned to that occurrence.

4. Repeating groups can appear anywhere in codebooks, except that they cannot span across data records of different types.

5. Group variables can be redefined and group variables can redefine other variables.

6. The CONTINUE GROUP clause describes the situation where all fields of the group are not stored side by side. Instead, the repetitions of one or more fields follow after all of the repetitions of the field(s) in the location where the group is first defined.

The Special Repeating Group Observation Variable

For each repeating group, TPL TABLES automatically creates a corresponding observation variable and adds it to the codebook. If you look at the codebook abstract created during codebook processing (the codebookname.L file), you will see the variable included in the variable list. This observation variable has the same name as the repeating group variable but with **_OBS** appended to the name. It can be used to tabulate at the repeating group level and has a value of 1 for each repetition of the repeating group. Its label is (''), so if you use the variable in a table, no label will print for it.

If you want to add a label for the special **_OBS** variable, you can assign it to a new computed variable and use the new variable instead. For example:

```
COMPUTE MONTH_OBS_LABEL 'Months' = MONTH_OBS;
```

Or, you can replace the label with a FORMAT statement. For example:

```
FOR VARIABLE MONTH_OBS:  REPLACE LABEL WITH 'Months';
```

How Repeating Groups Affect Tabulations

In describing how the use of repeating group variables affects tables, we will reference the following data record and the codebook which describes it. Only the first and last months are shown. The sample values shown under each data field will be used in the tabulation examples.

WORKER Record

January

				Hour_Type	
		Taxes	Leave	Regular	Overtime
Occupation	Earnings	Withheld	Category	Hours_Worked	Hours_Worked
1	800	200	1	160	4

December

				Hour_Type		
		Taxes	Leave	Regular	Overtime	
.....	Earnings	Withheld	Category	Hours_Worked	Hours_Worked	Sex
	900	225	2	160	8	1

```
BEGIN EMPLOYEE CODEBOOK
```

```
WORKER RECORD

OCCUPATION CON 1
  (
        'White Collar'    = 1
        'Blue Collar'     = 2
        'Farm Worker'     = 3
  )

BEGIN GROUP MONTH REPEATS 12
  (
        'January',
        'February',
            .
            .
            .
        'December'
  )
     EARNINGS 'Earnings'  OBS 8
     TAXES_WITHHELD 'Taxes Withheld' OBS 8
     LEAVE_CATEGORY CON 1
       (
          '4 Hours/Pay Period'  = 1
          '6 Hours/Pay Period'  = 2
          '8 Hours/Pay Period'  = 3
       )

     BEGIN GROUP HOUR_TYPE REPEATS 2
       (
          'Regular',
          'Overtime'
       )
          HOURS_WORKED 'Hours Worked' OBS 3
     END GROUP HOUR_TYPE

END GROUP MONTH

SEX CON 1
  (
     'Male'      = 1
     'Female'    = 2
  )

END EMPLOYEE CODEBOOK
```

MONTH is a group item containing several elementary items. Following the group name MONTH is the number of repetitions of monthly data within the record, in this case 12. Next, within parentheses, is a

list of labels to be used for each of the 12 repetitions of MONTH. The variables EARNINGS, TAXES_WITHHELD, LEAVE_CATEGORY and HOUR_TYPE all repeat 12 times within each data record.

HOUR_TYPE is itself a group that repeats 2 times for each occurrence of MONTH.

The notion of hierarchical level can be applied to repeating groups if we view the repeating groups as sub-records or records within a record. For example, in the sample codebook, WORKER and OCCUPATION can be thought of as level 0; MONTH, EARNINGS, TAXES_WITHHELD and LEAVE_CATEGORY as level 1; and HOUR_TYPE and HOURS_WORKED as level 2.

The sub-record structure may be viewed as:

```
level  0  WORKER|OCCUPATION

level  1  JANUARY  |EARNINGS|TAXES_WITHHELD|LEAVE_CATEGORY

level  2  REGULAR  |HOURS_WORKED
          OVERTIME|HOURS_WORKED

level  1  FEBRUARY|EARNINGS|TAXES_WITHHELD|LEAVE_CATEGORY

level  2  REGULAR  |HOURS_WORKED
          OVERTIME|HOURS_WORKED
                        .
                        .
                        .
```

In general, you will get the results you expect when using repeating groups. If you do not, you may wish to consult the following rules regarding certain specific uses in a TABLE statement.

RULE 1 The use of an observation variable within a repeating group without nesting it with the group variable(s) above it causes it to be aggregated over all occurrences of the group. For example, consider the next TABLE statement.

```
TABLE G9: HEADING EARNINGS THEN HOURS_WORKED,
          STUB TOTAL;
```

Since neither of the repeating group control variables, MONTH or HOUR_TYPE, is used in the TABLE statement, EARNINGS will be aggregated over 12 months, and HOURS_WORKED will be aggregated over both regular and overtime for the 12 months.

Using the sample WORKER record values shown with the codebook description, the resulting table would be,

	Earnings	Hours Worked
Total	1700	332

RULE 2 Nesting observation variables with control variables within a group, including the group name itself, will result in one or more cross tabulations from each repetition of the group. For example, the TABLE statement:

```
TABLE G10:  HEADING  EARNINGS THEN TAXES_WITHHELD,
            STUB  LEAVE_CATEGORY;
```

will cause EARNINGS and TAXES_WITHHELD to be aggregated from each of the 12 months according to the LEAVE_CATEGORY value for each month, producing the table:

	Earnings	Taxes Withheld
4 Hours/Pay Period	800	200
6 Hours/Pay Period	900	225

Adding the control variables MONTH and HOUR_TYPE in the next TABLE statement results in HOURS_WORKED being aggregated from REGULAR and OVERTIME for each occurrence of MONTH.

```
TABLE G11:  HEADING EARNINGS THEN HOURS_WORKED BY
            (HOUR_TYPE THEN TOTAL),  STUB MONTH THEN TOTAL;
```

	Earnings	Hours Worked		
		Regular	Overtime	Total
January	800	160	4	164
February ...	.	.	.	.
March	.	.	.	.
. .....	.	.	.	.
. .....	.	.	.	.
December	900	160	8	168
Total	1700	320	12	332

The next table illustrates both rules 1 and 2. The variable HOURS_ WORKED will be summed over both Regular and Overtime since it is not nested with its group variable, HOUR_TYPE (rule 1). Since variables within the group are nested with the group name MONTH, aggregations will occur for each month according to leave category (rule 2).

TABLE G12: HEADING LEAVE_CATEGORY BY (EARNINGS THEN HOURS_WORKED), STUB MONTH THEN TOTAL;

	4 Hours/Pay Period		6 Hours/Pay Period		8 Hours/Pay Period	
	Earnings	Hours Worked	Earnings	Hours Worked	Earnings	Hours Worked
January	800	164				
February ...	.	.				
March	.	.				
. .....	.	.				
. .....	.	.				
December	.	.	900	168		
Total	800	164	900	168		

RULE 3 A repeating group variable should not be nested only with variables that are outside of the repeating group and, therefore, at a higher

level in the record. The results are always meaningless and, in some cases, unpredictable.

RULE 4 If no observation variable is used in a cross tabulation involving variables within a repeating group, the default observation variable is the record name at the highest level in the data file. In the case of a flat (non-hierarchical) file, this is the record containing the repeating group. In other words, only the records will be counted.

For example, suppose we have the TABLE statement,

```
TABLE G13: HEADING LEAVE_CATEGORY, STUB OCCUPATION;
```

Since no observation variable has been used, the default observation variable is the record name WORKER (the same as COUNT). Each worker will be counted once for each unique leave category the worker was in during the 12 months. If a white collar worker was earning 4 hours/pay period from January through September, and then earned 6 hours/pay period in October through December, the table values would appear as:

	4 Hours/Pay Period	6 Hours/Pay Period	8 Hours/Pay Period
White Collar	1	1	
Blue Collar			
Farm Worker			

If we wanted a count of all months for which a worker was in each leave category, we would need an observation variable that could count at the MONTH level in the record. For this purpose, we could use the special observation variable that is created by TPL TABLES for each repeating group. This observation variable has the same name as the repeating group variable but with "_OBS" appended to the name. For our sample codebook, TPL TABLES would have created observation variables called MONTH_OBS and HOUR_TYPE_OBS. These observation variables have a value of 1 for each repetition in their respective repeating groups. They have null labels (''), so if you use them in TABLE statements, no labels will print.

Nesting the special observation variable MONTH_OBS into the above table gives a count of all months for which a worker was in each leave category.

Note that since MONTH_OBS has a null label, no label is printed for it, although it does affect the contents of the data cells.

TABLE G14: HEADING MONTH_OBS BY LEAVE_CATEGORY,
STUB OCCUPATION;

	4 Hours/Pay Period	6 Hours/Pay Period	8 Hours/Pay Period
White Collar	9	3	
Blue Collar			
Farm Worker			

RULE 5 The inclusion of one or more repeating groups in a codebook that describes a hierarchical data file will cause the default observation variable COUNT to be transferred from the lowest level of the hierarchy to the top level. ***Thus, in cross tabulations that contain only control variables, records will be counted at the top level of the hierarchy rather than at the lowest level.***

If you are using repeating groups in hierarchical data files, we recommend that you add explicit record names to cross tabulations that contain only control variables so that you can be sure of getting counts at the correct level. This can be done by nesting (with BY) the record name for the appropriate level into any cross tabulations that contain only control variables. If you do not want the record name or label to show in the table, assign a null " label to the record.

Limits on the Use of Repeating Groups in Tables

1. More than one independent repeating group cannot be used within a single table request, even if they are used in separate TABLE statements. For example, if each record contains a repeating group of MONTHS and a repeating group of INDUSTRIES, only one of the two can be referenced in the same table request. Nested repeating groups (groups within groups) are not considered independent and can be used together.

2. In a hierarchical data file, a repeating group can appear at any hierarchical level; however, the group cannot be referenced together with any variable which is contained in a record below that of the record containing the group. This is because there is no clear interpretation of an intra-record hierarchy (the repeating group) working together with lower level hierarchical records.

Repeating Group Variables in Computations

The repeating group name (control) and any control variables within the group can be used in Conditional Compute statements. Observation variables within the group can be used everywhere that other observation variables can be used. If a repeating group variable or variables within a group are used in COMPUTE statements, the result is a new variable in the group with a sub-record hierarchical level number equal to the lowest (highest numerical) of any variable used. For example,

```
COMPUTE NET_INCOME = EARNINGS - TAXES_WITHHELD;
```

Since both EARNINGS and TAXES are in the repeating group called MONTH, the computation will be done for each repetition of MONTH. The computed variable NET_INCOME will be treated like the other variables in the MONTH group and will be associated with the same level as EARNINGS and TAXES_WITHHELD so that the use of NET_INCOME in a TABLE statement will follow the same rules as using EARNINGS or TAXES_WITHHELD.

Limiting Tabulations to Certain Occurrences with DEFINE Statements

Tabulations can be limited to one or any combination of occurrences of a repeating group variable.

For example, if tabulations are to be limited to the January and December occurrences of a repeating group called MONTH, then a DEFINE such as the following can be used:

```
DEFINE JAN_AND_DEC ON MONTH;
        'January and December'          IF   1;
                                       IF  12;
```

Nesting JAN_AND_DEC into the table will cause all totals of the same classification within MONTH to be combined for January and December. The second through the eleventh months in each record will be ignored.

To get separate totals for January and December, the DEFINE would then appear as:

```
DEFINE JAN_AND_DEC ON MONTH;
        'January'        IF  1;
        'December'       IF  12;
```

Using Dummy Repeating Groups to Associate Repetitions

The fields within a repeating group do not need to be contiguous, because they can be joined with the CONTINUED option. However, the repetitions for repeating groups must be contiguous within a record. If you have fields that you would like to use together as a repeating group, but they are separated by other fields, even on different levels of a hierarchy, dummy repeating groups can sometimes be used to effectively pull the repetitions together as if they were side by side. A specific example will be used to illustrate this.

Note that the technique is easiest to use with observation variables. For control variables with numeric values, redefines can be used to create equivalent observation variables. For control variables with non-numeric values, corresponding observation variables with numeric values must be created using conditional computes.

Let us assume that our data file is hierarchical, with records for families at level 0, and records for persons at level 1. We will suppose that each family record contains a variable indicating the state within which the family resides and another variable with the family income. Each person record contains a variable indicating the state within which that person was born. We will assume that both state variables are observation and have the same coding structure.

We want to produce a table in which the stub has one row for each state. The heading has two columns to count persons, one for persons residing in each state, the other for persons born in each state. A third column contains the total family income for families either residing in each state or containing members born in that state.

Assuming that the names of the two records are "FAMILIES" and "PERSONS", we add the following information within the codebook description of the PERSONS record:

```
BEGIN GROUP DUMMY_GROUP 'State of' REPEATS 2
        ('Birth', 'Residence')
        FILLER 1
END GROUP DUMMY_GROUP
```

This group can be added to the codebook even though there is really no group structure within the actual data record. The "dummy" repeating group must replace one byte of FILLER, or it can overlay the space of another variable if an appropriate REDEFINE is used. For example, the "dummy" group could be inserted in front of the variable AGE, and the AGE entry could then begin with:

```
AGE REDEFINES DUMMY_GROUP .....
```

In the table request, a conditional compute statement will be used to associate each state variable with a repeating group occurrence. The technique is as follows:

```
COMPUTE COMBINED_STATES =
        STATE_OF_BIRTH          IF  DUMMY_GROUP = 1;
        STATE_OF_RESIDENCE      IF  DUMMY_GROUP = 2;
```

A DEFINE statement will then create the desired state labels common to both occurrences. The proper values of state code for each state must be used on the right-hand side of the ‘IF’ as shown in the next DEFINE statement.

```
DEFINE STATE_LABELS ON COMBINED_STATES;
        ILLINOIS        IF 1;
        OHIO            IF 2;
        FLORIDA         IF 3;
           .
           .
           .
```

The TABLE statement will then be:

```
TABLE G15:  HEADING PERSONS BY DUMMY_GROUP
        THEN INCOME,
        STUB TOTAL THEN STATE_LABELS;
```

The table would appear as:

	Persons		Income
	State of		
	Birth	Residence	
Total	7	7	115,054
ILLINOIS	2	3	72,130
OHIO	3	1	42,924
FLORIDA	2	3	84,424

Note that if other variables (such as INCOME in the above example) are concatenated with the "dummy" repeating group, the tabulations for these other variables will be correct, but the meaning of the numbers may be obscure.

Additional Sample Tables Using Repeating Groups

To provide more examples of how repeating groups work, we show a small codebook followed by sample data records and two table requests which reference the codebook and data. It is important to note that two separate table requests are required to produce all of the tables because only one repeating group (not counting nested groups) can be accessed in one table request. The table contents are explained in the table titles.

```
BEGIN HH CODEBOOK

HOUSEHOLDS MASK 999 RECORD

CITY CON 2
  (
     'Concordia'  =  1
     'Frostburg'  =  2
     'Silver Spring'  =  3
  )
BEGIN GROUP SERVICES REPEATS 3
  (
     'Police Protection',
     'Library Services',
     'Street Maintenance'
  )
```

```
    EVALUATION CON 1
      (
          'Good'  =    'G'
          'Fair'  =    'F'
          'Poor'  =    'P'
          'No Response'  =  ' '
       )
END GROUP SERVICES

BEGIN GROUP MEMBERS 'Family Members' REPEATS 5
  YEARS_OLD CON 2
  CONDITION LABEL IS VALUE ' Years Old' (10:45)
  SEX 'Sex of Respondent' CON 1
    (
      'Male'  =        'M'
      'Female'  =      'F'
      'Not Reported'  =  ' '
    )
  VIEWING_HOURS 'Weekly Hours of TV Viewing' CON 1
    (
      'Less than 5'    =  'A'
      '6 to 10'        =  'B'
      'More than 10'   =  'C'
      'No Response'    =  ' '
    )
  INCOME 'Income'  MASK $99,999 DATA ERROR = NULL  OBS 5
END GROUP MEMBERS

END HH CODEBOOK
```

In the above MEMBERS repeating group, no names or labels have been assigned to the repetitions because it is assumed that the family members may be stored in the record in any order. If tabulations are to be done based on age, sex, income, and viewing hours and not on whether the members are listed in order by husband, wife, etc., then names or labels for each repetition are not necessary.

The following four data records were used in producing the tables shown on following pages.

```
CITY                                                          End of record
|                                                                        |
|     SERVICES REPEATING GROUP (3)                                       |
|     |                                                                  |
|     |     MEMBERS REPEATING GROUP (5)                                  |
|     |     |                                                            |
|     |     |                                                            |
01    GFP 37MC25000  32FC15000  11M                                      |
01    GFP 37MC25000  32FC15000  11M                                      |
02    PFG 25MA15000  24FB10000                                           |
03    FF   45M  40000  45FB         20FA15000 18MC08000 16FA             |
```

First Table Request

```
USE HH CODEBOOK;

TABLE T1 'TOTAL THEN SERVICES, SERVICES_OBS BY EVALUATION;'//
    'Summary of ratings for each type of municipal service.':
            STUB TOTAL THEN SERVICES,
            HEADING  SERVICES_OBS BY EVALUATION;

TABLE T2 'CITY, EVALUATION;'//  'Count of households in each city '
    'which rated at least one evaluation category, without reference '
    'to type of municipal service.  The default observation variable is '
    'the record name HOUSEHOLDS.  To count total occurrences of '
    'all evaluations, the group observation variable SERVICES_OBS, '
    'generated by TPL TABLES, would need to be nested '
    'into the table.':
            STUB CITY,  HEADING EVALUATION;

TABLE T3 'SERVICES BY EVALUATION, TOTAL THEN CITY;'//
    'Ratings of each municipal service by city.':
            STUB SERVICES BY EVALUATION,
            HEADING TOTAL THEN CITY;

TABLE T4 'EVALUATION BY SERVICES, TOTAL THEN CITY;'//
    'Ratings of each municipal service within the major '
    'category of evaluation for each city.':
            STUB EVALUATION BY SERVICES,
            HEADING TOTAL THEN CITY;
```

```
TABLE T5 'TOTAL THEN EVALUATION, TOTAL THEN HOUSEHOLDS '
    'BY CITY;'//  'Counts households which made at least one of each '
    'type of evaluation in each city.  If a particular household has more '
    'than one of the same type of  evaluation, the household will be '
    'counted only once for that type.':

        STUB TOTAL THEN EVALUATION,
        HEADING TOTAL THEN HOUSEHOLDS BY CITY;
```

TOTAL THEN SERVICES, SERVICES_OBS BY EVALUATION;

Summary of ratings for each type of municipal service.

	Good	Fair	Poor	No Response
Total	3	5	3	1
Police Protection	2	1	1	–
Library Services	–	4	–	–
Street Maintenance	1	–	2	1

– Data not available.

CITY, EVALUATION;

Count of households in each city which rated at least one evaluation category, without reference to type of municipal service. The default observation variable is the record name HOUSEHOLDS. To count total occurrences of all evaluations, the group observation variable SERVICES_OBS, generated by TPL TABLES, would need to be nested into the table.

	Good	Fair	Poor	No Response
Concordia	2	2	2	–
Frostburg	1	1	1	–
Silver Spring	–	1	–	1

– Data not available.

SERVICES BY EVALUATION, TOTAL THEN CITY;

Ratings of each municipal service by city.

	Total	Concordia	Frostburg	Silver Spring
Police Protection				
Good	2	2	–	–
Fair	1	–	–	1
Poor	1	–	1	–
Library Services				
Fair	4	2	1	1
Street Maintenance				
Good	1	–	1	–
Poor	2	2	–	–
No Response	1	–	–	1

– Data not available.

EVALUATION BY SERVICES, TOTAL THEN CITY;

Ratings of each municipal service within the major category of evaluation for each city.

	Total	Concordia	Frostburg	Silver Spring
Good				
Police Protection	2	2	–	–
Street Maintenance	1	–	1	–
Fair				
Police Protection	1	–	–	1
Library Services	4	2	1	1
Poor				
Police Protection	1	–	1	–
Street Maintenance	2	2	–	–
No Response				
Street Maintenance	1	–	–	1

– Data not available.

TOTAL THEN EVALUATION, TOTAL THEN HOUSEHOLDS BY CITY;

Counts households which made at least one of each type of evaluation in each city. If a particular household has more than one of the same type of evaluation, the household will be counted only once for that type.

	Total	HOUSEHOLDS		
		Concordia	Frostburg	Silver Spring
Total	4	2	1	1
Good	3	2	1	–
Fair	4	2	1	1
Poor	3	2	1	–
No Response	1	–	–	1

– Data not available.

Second Table Request

```
USE HH CODEBOOK;

SELECT IF SEX = 'M' OR SEX = 'F';

POST COMPUTE AVG_HH_INCOME 'Average Household Income'
    MASK $99,999 = INCOME / MEMBERS_OBS;

TABLE RG1 'CITY BY SEX, MEMBERS_OBS BY VIEWING_HOURS;'//
    'Count of family members in each city by sex and category '
    'of TV viewing.':
            STUB CITY BY SEX,
            HEADING MEMBERS_OBS BY VIEWING_HOURS;

TABLE RG2 'VIEWING HOURS BY (MEMBERS_OBS THEN INCOME), '
    'CITY;'//  'Counts members and aggregates income according to TV '
    'viewing pattern.  Note that since MEMBERS_OBS has a null label, '
    'its data rows "collapse up" and are labelled by VIEWING_HOURS '
    'categories.':
            STUB VIEWING HOURS BY (MEMBERS_OBS THEN
                    INCOME),
            HEADING CITY;
```

```
TABLE RG3 'CITY, INCOME;'// 'Income aggregation for all members by city.':
        STUB CITY,  HEADING INCOME;

TABLE RG4 'TOTAL THEN CITY, MEMBERS_OBS BY (TOTAL THEN '
    'SEX);'//  'Counts members according to city and sex.':
        STUB TOTAL THEN CITY,
        HEADING MEMBERS_OBS BY (TOTAL THEN SEX);

TABLE RG5 '(INCOME THEN MEMBERS_OBS THEN AVG_HH_INCOME)'
    ' BY (TOTAL THEN SEX),  TOTAL THEN VIEWING_HOURS;'//
    'Member income, member count and average household'
    ' income classified by sex and hours of TV viewing.':
        STUB (INCOME THEN MEMBERS_OBS THEN
                AVG_HH_INCOME) BY (TOTAL THEN SEX),
        HEADING TOTAL THEN VIEWING_HOURS;

TABLE RG6 '(TOTAL THEN SEX) BY YEARS_OLD, MEMBERS_OBS '
    'THEN VIEWING_HOURS BY (MEMBERS_OBS THEN INCOME);'//
    'Counts members and aggregates member income according to '
    'hours of TV viewing, age and sex.  Note that a FORMAT '
    'statement has been used to replace the null label originally '
    'generated for the special repeating group observation variable '
    'called MEMBERS_OBS.':
        STUB (TOTAL THEN SEX) BY YEARS_OLD,
        HEADING MEMBERS_OBS THEN VIEWING_HOURS BY
                (MEMBERS_OBS THEN INCOME);
```

CITY BY SEX, MEMBERS_OBS BY VIEWING_HOURS;

Count of family members in each city by sex and category of TV viewing.

	Weekly Hours of TV Viewing			
	Less than 5	6 to 10	More than 10	No Response
Concordia				
Sex of Respondent				
Male	–	–	2	2
Female	–	–	2	–
Frostburg				
Sex of Respondent				
Male	1	–	–	–
Female	–	1	–	–
Silver Spring				
Sex of Respondent				
Male	–	–	1	1
Female	2	1	–	–

– Data not available.

VIEWING_HOURS BY (MEMBERS_OBS THEN INCOME), CITY;

Counts members and aggregates income according to TV viewing pattern. Note that since MEMBERS_OBS has a null label, its data rows "collapse up" and are labelled by VIEWING_HOURS categories.

	Concordia	Frostburg	Silver Spring
Weekly Hours of TV Viewing			
Less than 5	–	1	2
Income	–	$15,000	$15,000
6 to 10	–	1	1
Income	–	$10,000	–
More than 10	4	–	1
Income	$80,000	–	$8,000
No Response	2	–	1
Income	–	–	$40,000

– Data not available.

CITY, INCOME;

Income aggregation for all members by city.

	Income
Concordia	$80,000
Frostburg	25,000
Silver Spring	63,000

TOTAL THEN CITY, MEMBERS_OBS BY (TOTAL THEN SEX);

Counts members according to city and sex.

	Total	Sex of Respondent		
		Male	Female	Not Reported
Total	13	7	6	–
Concordia	6	4	2	–
Frostburg	2	1	1	–
Silver Spring	5	2	3	–

– Data not available.

(INCOME THEN MEMBERS_OBS THEN AVG_HH_INCOME) BY (TOTAL THEN SEX), TOTAL THEN VIEWING_HOURS;

Member income, member count and average household income classified by sex and hours of TV viewing.

	Total	Weekly Hours of TV Viewing			
		Less than 5	6 to 10	More than 10	No Response
Income					
Total	$168,000	$30,000	$10,000	$88,000	$40,000
Sex of Respondent					
Male	113,000	15,000	–	58,000	40,000
Female	55,000	15,000	10,000	30,000	–
Total	13	3	2	5	3
Sex of Respondent					
Male	7	1	–	3	3
Female	6	2	2	2	–
Average Household Income					
Total	$12,923	$10,000	$5,000	$17,600	$13,333
Sex of Respondent					
Male	16,143	15,000	–	19,333	13,333
Female	9,167	7,500	5,000	15,000	–

– Data not available.

(TOTAL THEN SEX) BY YEARS_OLD, MEMBERS_OBS THEN VIEWING_HOURS BY (MEMBERS_OBS THEN INCOME);

Counts members and aggregates member income according to hours of TV viewing, age and sex. Note that a FORMAT statement has been used to replace the null label originally generated for the special repeating group observation variable called MEMBERS_OBS.

	Member Count	Weekly Hours of TV Viewing							
		Less than 5		6 to 10		More than 10		No Response	
		Member Count	Income	Member Count	Income	Member Count	Income	Member Count	Income
Total									
11 Years Old	2	–	–	–	–	–	–	2	–
16 Years Old	1	1	–	–	–	–	–	–	–
18 Years Old	1	–	–	–	–	1	$8,000	–	–
20 Years Old	1	1	$15,000	–	–	–	–	–	–
24 Years Old	1	–	–	1	$10,000	–	–	–	–
25 Years Old	1	1	15,000	–	–	–	–	–	–
32 Years Old	2	–	–	–	–	2	30,000	–	–
37 Years Old	2	–	–	–	–	2	50,000	–	–
45 Years Old	2	–	–	1	–	–	–	1	$40,000
Sex of Respondent									
Male									
11 Years Old	2	–	–	–	–	–	–	2	–
18 Years Old	1	–	–	–	–	1	8,000	–	–
25 Years Old	1	1	15,000	–	–	–	–	–	–
37 Years Old	2	–	–	–	–	2	50,000	–	–
45 Years Old	1	–	–	–	–	–	–	1	40,000
Female									
16 Years Old	1	1	–	–	–	–	–	–	–
20 Years Old	1	1	15,000	–	–	–	–	–	–
24 Years Old	1	–	–	1	10,000	–	–	–	–
32 Years Old	2	–	–	–	–	2	30,000	–	–
45 Years Old	1	–	–	1	–	–	–	–	–

– Data not available.

CHAPTER 20

Labels

CREATING AND FORMATTING PRINT LABELS

Any variable can have a print label associated with it. The print label follows the variable name when the variable is described in the codebook or created in a table request. When the variable is used in a table, this label will print in place of the original variable name. Other table elements that can have print labels are listed below. Three important table elements, **table titles**, **footnote texts** and **TEXT masks**, are print labels and can contain any of the formatting elements described in this chapter.

If you do not specify print labels, default labels will be created for tables. Default labels are satisfactory for identifying the contents of a table, but you may wish to specify your own labels to make them more informative or to take advantage of some of the label formatting options.

This chapter describes all of the formatting options you can use in individual labels. For a description of default treatments, such as default alignment of stub labels or table titles, see the chapter on "Automatic Formatting".

A typical label consists of text that is bounded by single or double quote marks. The text can include spaces, upper and lower case letters, and special characters. Many formatting options are available for print labels. Break points can be chosen for multiline labels, and alignment can be specified. Labels can also contain references to footnotes. You can vary the type styles within labels by inserting font specifications.

Print label options apply to all of the following table elements:

1. Records described in the Codebook
2. Control variables
3. Control variable values
4. Observation variables
5. Character variables
6. Condition values
7. Table titles
8. New variables created with statements such as DEFINE and COMPUTE
9. Footnote texts
10. TEXT masks

Automatic Print Labels

When print labels are not specified, they are automatically created according to the following rules:

Observation Variables

If no label is assigned to an observation variable, the variable name is used as the print label. This rule applies to variables that are described as RECORD or OBS in the codebook and to variables that are computed in a table request.

Control Variables and Their Values

If no label is assigned to a control variable, no label is printed for the variable. If no labels are assigned to the values, labels are generated from the condition names, if present, or from the values themselves. These rules apply to variables described as CONTROL in the codebook and to variables created by DEFINE statements in a table request.

Table Titles

If no table title is assigned in the TABLE statement, the table name is used as the title.

Whenever a name is used as a label, any letters used in the name are printed in upper case. For example, if the observation variable called Income is not followed by a print label, the name INCOME will be used as the default print label. If the name contains underscore (_) characters,

they will be replaced with blanks when the name is printed. For example, the name Average_Income will print as **AVERAGE INCOME**.

Creating Your Own Print Labels

Labels can be created in the codebook or table request. They can also be created or replaced using REPLACE statements in a FORMAT request.

A simple label consists of a text string surrounded by single or double quotes. An example of a simple label assigned in a codebook is:

```
INCOME 'Annual Income in Thousands' OBS 5
```

When the variable INCOME is used in a TABLE statement the label **Annual Income in Thousands** will be used to identify INCOME values in the table.

In the codebook, print labels can optionally be included following variable names and can be assigned to control variable values. Examples of print labels assigned in a codebook are:

```
FAMILIES 'Family Count'  RECORD
AMT_WK 'Dollars spent per week' OBS 7
AUTO 'Automobile owned?'  CON 1
  (
        YES 'Yes'  = 1
        NO 'No'  = 2
  )
HEADS_WORK  'Class of work of family head' CON 1
  (
        ' White Collar'  = 1
        ' Blue Collar'  = 2
        ' Other'  = 3
  )
```

Within a table request, any of the TPL statements that create new variables can optionally include print labels. A print label can also follow the table name in a TABLE statement, in which case that print label will be used as the table title. The following examples show uses of print labels within a table request.

```
COMPUTE INCOME 'Total Family Income' =
        HEAD_INCOME + OTHER_INCOME;

DEFINE INC_CL 'Income Classifications' ON INCOME;
        'Less than $5,000'      IF < 5000;
        'Less than $10,000'     IF <10000;
        'At least $15,000'      IF >=15000;

POSTCOMPUTE AVG_INC 'Average Income' =
        INCOME / FAMILIES;

TABLE FAM_DAT 'Family Income Classifications' :
        STUB   INC_CL, HEADING INCOME THEN AVG_INC;
```

If an observation variable does not have a print label, the variable name will be used as the label.

If a control variable has a print label, the label spans over the condition labels for that variable. If a control variable does not have a label, then only the condition labels will print.

Characters Allowed in Label Strings

With only a few exceptions, label strings can contain any character that is available on your keyboard. The quote and backslash (\) characters must be entered in a special way as described in the next section.

We recommend that you not enter tabs or carriage returns (typed with the **<Enter>** key) in label strings. Tabs will be printed as blanks, and carriage returns will be removed before printing. You can get the effect of a tab at the beginning of a label by using the INDENT option described later in this chapter. If you are entering a label string that is longer than one line, you can break it into multiple segments by ending each line with a quote, typing **<Enter>**, and continuing on the next line beginning with another quote. The segments will be joined when the label is printed.

If there are characters available on your printer that are not on your keyboard, you can enter them in label strings either by using a character name or code. A character name is preceded by **&** and followed by **;** . For examples **É** represents an E with an acute accent above it. Character names are case sensitive. **é** represents **e** with an accute accent above it.

Character codes are entered by typing **\nnn** where **nnn** is the 3 digit decimal code for the character. Three digits are always required. If the character can be represented by fewer than 3 digits, add leading zeros. For example, for a character represented by the code 65, enter **\065**.

The value nnn must be the DECIMAL code for the character. The character code tables in some software manuals show the octal or hexidecimal codes for the characters. If you are referring to such a table, you must convert the code to its decimal equivalent. Character set tables showing decimal codes and character names are included at the end of this manual in an appendix.

Quotes and Back slashes in Labels

Since quotes are used to show the beginning and end of a label string, they must be entered in a special way if they are to be used inside a label string. If single quotes are used at the beginning and end of the label string, two successive single quotes are required to print one single quote within the label. If double quotes are used at the beginning and end of the label string, two successive double quotes are required to print a double quote within the label.

String expression	Will print as
'"FIRST EXAMPLE"'	'FIRST EXAMPLE'
'USER''S CHOICE'	USER'S CHOICE
"User's Choice"	User's Choice
'BUT "LESS THAN" 100'	BUT 'LESS THAN' 100
"40 BUT LESS THAN 60"""	40 BUT LESS THAN 60"
"""	' (a single quote inside double quotes)
'"'	" (a double quote inside single quotes)
' '	(a blank inside single quotes)

The backslash (\) character has a special use for entering characters that are not on the keyboard. If you want to include a backslash character in a label, enter a double backslash. For example, the label string **'\\In Thousands\\'** will print as:

\In Thousands\

Label Length

Label length is virtually unlimited. The practical limit on the length of a label is imposed by the requirement that there must be room for at least one line of data on each page of a table. In other words, if a label is so long that it takes up a whole page, there will be no space left for the data.

The Null Label

The null label consists of two consecutive quote marks with nothing between them. When a variable with a null label is nested with another variable, no label is printed for the variable with the null label. The difference between a null label and a blank label is that the null label will effectively disappear from the table, while the blank label will be printed just like any other label.

Null labels can be especially useful in the case where you want to eliminate certain values from a tabulation with a DEFINE statement, but you do not want any extra labels printed for the defined variable. For example,

```
DEFINE SELECTED_REGIONS ON REGION;
        ''      IF 1;
                IF 3;
```

If SELECTED_REGIONS is nested into one of the table expressions, only regions 1 and 3 will be included in the tabulation, but no identifying label will be printed for the variable SELECTED_REGIONS.

The null label is also very useful where a label for an observation variable would be redundant. For example, if the variable INCOME is to be used in a table, but the table title includes all necessary information about what is being tabulated, you could prevent an INCOME label from being printed in the stub or heading with a POST COMPUTE that assigns a null label:

```
COMPUTE NO_LABEL '' = INCOME;
```

Then use the variable NO_LABEL in the TABLE statement instead of INCOME.

Note that if only a null label is provided for a table wafer, column or row, it will be completely unidentified. If a wafer has only a null label, there will be no label for the wafer. If a column has only a null label, the space where a label would be will be completely blank. If a row, the stub entry for the row will be blank all the way across. In other words, there will be no label and no stub filler characters.

Labels with Multiple Segments

The print label can be expressed as a single label string or as two or more segments separated by at least one space, as in:

```
'ALL NONMANUFACTURING'  ' INDUSTRIES FOR 1985-90'
```

When multiple segments are used, they will be interpreted as one label combining the individual components. No space will be inserted to separate the merged segments, so for each pair of segments, a space must be included to separate words. One advantage of this format is that each segment can be entered on a separate line, although the segments will be merged as one continuous label. Another advantage is that label formatting options can be inserted between label segments.

Creating Extra Labels

The LABEL Statement

The LABEL statement lets you create variables that can be used to add labels to the wafer, stub or heading of a table. It can be especially useful if you need to add a label that spans over two or more variables in the stub or heading.

Format

```
LABEL  label-variable  'print label' ;
```

The **label-variable** behaves exactly the same as the built-in variable TOTAL but you can assign the label of your choice. The **print label** in a LABEL statement can contain any of the elements that are allowed in other types of TPL TABLES print labels, for example alignments, footnote references or SPANNER attributes. Any number of label variables can be used in a table and they can be nested or concatenated in any dimension.

An example of a label statement is:

```
LABEL   ALL_POP  'All population age 16 and above';
```

If the label called **ALL_POP** is used in the following table heading:

```
HEADING  ALL_POP  BY  (TOTAL THEN AVG_INCOME);
```

then the ALL_POP label will span across the top of the heading.

If you want to change the label in a format request, you can reference the label variable the same way as you can any other variable. For example:

```
FOR VARIABLE ALL_POP:
        REPLACE LABEL WITH 'New label';
```

The following examples illustrate a few of the uses for label variables. We begin with a simple table statement.

```
TABLE SALARY 'Pay information by sex':
HEADING  MIN_RATE THEN MED_RATE THEN MAX_RATE;
STUB  SEX;
```

Pay information by sex

	Min Rate	Median Rate	Max Rate
Female	$5.33	$7.35	$24.19
Male	5.00	11.83	23.19

Assume that we want to add a label to the table heading that spans across the observation variables for minimum, median and maximum pay and says 'Hourly Wages'. We can do this easily by creating a label variable and nesting it into the table heading.

```
LABEL   HOURLY   'Hourly Wages';

TABLE SALARY 'Pay information by sex':
HEADING  HOURLY BY (MIN_RATE THEN MED_RATE
        THEN MAX_RATE);
STUB  SEX;
```

Pay information by sex

	Hourly Wages		
	Min Rate	Median Rate	Max Rate
Female	$5.33	$7.35	$24.19
Male	5.00	11.83	23.19

Similarly, we can nest the label variable in the stub.

```
TABLE SALARY 'Pay information by sex':
HEADING  MIN_RATE THEN MED_RATE THEN MAX_RATE;
STUB  HOURLY BY SEX;
```

Pay information by sex

	Min Rate	Median Rate	Max Rate
Hourly Wages			
Female	$5.33	$7.35	$24.19
Male	5.00	11.83	23.19

In both of the above uses of the label variable, the label variable adds nothing to the table except the label. If we concatenate the label variable in the table using THEN, it will add one or more rows, columns or wafers to the table with the same values we would get by using the TOTAL variable. The next example shows that if we concatenate the label variable at the beginning of the table stub, we add a total row with the label 'Hourly Wages'.

```
TABLE SALARY 'Pay information by sex':
HEADING  MIN_RATE THEN MED_RATE THEN MAX_RATE;
STUB  HOURLY THEN SEX;
```

Pay information by sex

	Min Rate	Median Rate	Max Rate
Hourly Wages	$5.00	$7.44	$24.19
Female	5.33	7.35	24.19
Male	5.00	11.83	23.19

Dummy Variables for Extra Labels

Dummy variables can be created with DEFINE statements and nested or contatenated into tables to add extra labels. The LABEL statement, also described in this chapter, provides a more simple, straight-forward way of doing the exactly the same thing. In case you are modifying a table request that was written before the LABEL statement was introduced, you may need to know how "dummy" variables work.

A dummy variable with a label is created by a DEFINE statement that has a single entry and **ALL** as the old variable value. The general format is:

```
DEFINE dummy-var  ON  COUNT;
        'Extra Label' IF  ALL;
```

If the variable **dummy-var** is nested into a table expression, as in:

```
dummy-var BY (variable1 THEN variable2...)
```

the extra label will be printed in the table but will not affect the data values in the tabulation.

Control of Label Breaks

If a print label is too long for its allotted space, it will be automatically divided over two or more lines. If you want more precise control over label break points, you can use two special formatting options.

Slashes

The first formatting option is provided by the use of the slash (/) symbol. A slash inserted between two label segments will cause the second segment to start on a new line. Each additional slash will cause the insertion of one blank line. Each slash at the beginning or end of a label will cause one blank line to be inserted.

Although slashes cause unconditional breaks, alignment of each segment is according to whether the label is for a heading, stub, or table title. If the labels are heading labels, each segment will be centered within the column width. As a stub segment, the segment following the first slash will be indented to the right. If there is a second slash followed by a third segment, the third segment will be aligned under the second segment. Additional slash/segment pairs will cause identical alignment.

Single slashes cause single spacing between segments. Multiple slashes cause additional line spacing between segments. For example, three slashes separating two print labels would cause triple spacing between them. The expression,

```
'Row One'/'Continue'/'Continue'//'Row Two'
```

would print in a heading label as:

```
Row One
Continue
Continue
(space)
Row Two
```

and in a stub label as:

```
Row One
  Continue
  Continue
  (space)
  Row Two
```

If these segments were used in a table title, they would be treated similarly to the heading label, except that each segment of the title would be left justified within the table width unless an alignment keyword, such as **CENTER**, is included in the title. In that case each segment would be centered within the table width.

A codebook control variable entry might appear as follows:

```
INDUSTRY  /'Industry Types' CON 1
  (
        /'Manufacturing'  =  'A'
        /'Non-Manufacturing'  =  'B'
        /'Farming'  =  'C'
  )
```

The variable label, **Industry Types**, will begin one line below its normal starting line. Each of the three condition names will be preceded by one blank line.

If one or more slashes follow the last segment of stub text that is associated with a data line, the line of text will not be aligned with its data; that is, the line spacing will be forced before the data line is printed. Place the slashes before the text associated with the following stub entry to get the spacing after the data line.

Conditional Hyphens

The second formatting option allows you to specify where the label should break if it is too long for the available space. This conditional hyphenation is best illustrated by an example.

```
'MANU'-'FAC'-'TUR'-'ING'
```

If there is enough space to print all of the components as one consecutive string, they will appear as:

```
MANUFACTURING
```

If there is enough room for only the first seven characters plus a hyphen they will appear as:

```
MANUFAC-
```

with **TURING** appearing on the next line. If only five spaces are available, **MANU-** will appear on one line. **FACTURING** will next be considered for the following line and segmented in the same way if necessary.

When a hyphen at the end of a label segment is followed by a conditional hyphen and the label breaks at that point, only one hyphen will be displayed in the label.

Example For a column of width 10, the label 'Never-'-'Married' will be printed in the heading as:

```
Never-
Married
```

Hierarchy of Label Break Points

Labels are divided into multiple lines according to the following priorities.

1. Unconditional Break ('segment' / 'segment')
2. Blank within label string ('segment segment')
3. Hyphen within label string ('segment-segment')
4. Conditional hyphen ('segment' - 'segment')

If none of the above break points are found, the label will be broken at points that allow the segments to be printed with hyphens at the break points.

Label Alignment

LEFT, RIGHT and CENTER

The words LEFT, RIGHT and CENTER can be used with a label to override the default alignment. Default alignment for different types of labels can be changed with ALIGN statements as described in the FORMAT chapter, but a specification of LEFT, RIGHT or CENTER in an individual label will always override the default alignment.

Note that the word CENTER can also be spelled CENTRE.

LEFT, RIGHT and CENTER are called **alignment markers**. They can be inserted at the beginning of a label before the first quote, at the end after the last quote, or between label segments if there is more than one segment. For example:

```
TABLE ONE
        LEFT 'ESTABLISHMENT DATA'
        RIGHT 'ESTABLISHMENT DATA' / / /
        LEFT 'Table B-1. Employees on nonagricultural '
                'payrolls by industry' / /
                '[in thousands]':
         stub, heading;
```

In this example, the title will be formatted as:

```
ESTABLISHMENT DATA                                ESTABLISHMENT DATA

Table B-1. Employees on nonagricultural payrolls by industry

[in thousands]
```

As illustrated above, a label can have one or more alignment markers. They affect the label according to the following rules.

1. If you put only one alignment marker in a label, regardless of its location in the label, all segments of the label will take on the specified alignment. For example, the following label with a single marker of RIGHT will be formatted as two lines with both aligned to the right, even though the word RIGHT is placed in the middle of the label.

   ```
   'All Establishments' / RIGHT 'Reporting this Year'
   ```

will print as:

All Establishments
Reporting this Year

2. If there are multiple alignment markers in a label, any label **section** that does not have an explicit alignment marker is assumed to be left-aligned.

   For alignment purposes, the first **section** of a label begins at the beginning of the label. A **section** ends with any of the following label elements: /, RIGHT, LEFT, CENTER, RIGHT IN SPACE, SPACE, and SPACE TO.

   For example, the label

   ```
   CENTER 'Workers Compensation' / 'Mining' / RIGHT 'January'
   ```

   has the sections:

   ```
   'Workers Compensation'
   'Mining'
   'January'
   ```

   The section 'Workers Compensation' is centered, because it is preceded by CENTER. It ends with /. The section 'Mining' is left-aligned, because it has no explicit alignment marker. It is ended by both a / and the word RIGHT. The section 'January' is right-aligned, because it is preceded by RIGHT. If this is the table title, it will be displayed as:

   ```
                    Workers Compensation
   Mining
                                                  January
   ```

3. If there are two alignment markers between slashes, or between the beginning and end of the label if there are no slashes, then the sections will be placed on the same line with the specified alignments if there is enough space on the line to do so. If an aligned section doesn't fit, it will be placed on the next line. Consider the following table title:

   ```
   TABLE TITLE_SAMPLE
   LEFT 'Workers Compensation Report' RIGHT 'January'
   ```

If the table is wide enough for all of the title characters to fit on one line without overlapping, the complete title will be placed on one line with a left-aligned section and a right-aligned section:

```
Workers Compensation Report                    January
```

If RIGHT and LEFT were reversed in this title as follows:

```
RIGHT 'Workers Compensation Report' LEFT 'January'
```

then 'Workers Compensation' would be right-aligned on one line and 'January' would be left-aligned on the next, since there would never be space for the 'January' following the right-aligned 'Workers Compensation' section.

For another example, suppose a table is 50 characters wide. This means that the title space is 50 characters. The first section of label is left-aligned and takes up 15 characters. The second section is to be centered and takes up 26 characters. The centered section should start at position 25 (the center) - 13 (half the length of the centered segment) = 12. But the first section extends beyond 12, so there is no room for the centered section. Consequently, the centered section appears on a new line.

4. If you use multiple alignments within the same label, we recommend that you explicitly divide your label into sections that will fit for each line of the label and precede each section with the alignment of your choice. That way, you will always get the expected result.

Alignment in Page Markers

The FORMAT statement called PAGE MARKER can only have an alignment specified at the beginning (before any label segments, if present). This alignment applies to the entire page marker.

If you want a page marker with part on the left and part on the right, try aligning the page marker LEFT and inserting SPACE TO in front of parts of the marker to "push" them over to the desired location. Some experimenting may be needed to get things in the position you want. An example is:

```
PAGE MARKER = LEFT  SPACE TO 3 cm 'Page ' NUMBER
                    SPACE TO 12.5 cm  'HOUSEHOLD DATA';
```

Note that SPACE TO only applies to left-aligned labels, so this technique can only be used with a left-aligned page marker. Note also that a left-aligned page markers begins at the left margin of the page rather than the left edge of the table below it.

See also RIGHT IN SPACE, described elsewhere in this chapter. This is another option that can help you get a left and right section for a page marker. For example:

```
PAGE MARKER LEFT 'Left marker'
        RIGHT IN SPACE 7.5 IN  NUMBER;
```

In this example, the page width is 8.5 inches. Aligning the page NUMBER right to a location of 7.5 inches puts it at the right margin of the page if the default left and right margin widths of .5 inches are being used.

RIGHT with Spanning Stub Labels in Banked Tables

When a table has banks of unequal width, stub labels with the SPANNER attribute are formatted for the width of the narrowest bank in the table. This means that if a table has banks of different widths, a RIGHT label segment will be all the way to the right only in the narrowest bank.

Effect of CENTER when Stub is on the Right

If you have used the FORMAT statement **STUB RIGHT** to display the stub on the right side of the table **and** there is exactly one alignment marker in a stub label **and** that alignment is CENTER, the label will be centered within the entire stub width. If there are multiple alignment markers, centering is within the part of the stub following the standard dot leader.

The following example tables show the difference between STUB LEFT (the default) and STUB RIGHT on a set of stub labels with alignment markers. The special case of CENTER is shaded.

Example of alignments with stub on the left.

Stub on the left	*The stub label is:*
Left	*LEFT 'Left'*
Right	*RIGHT 'Right'*
Center	*CENTER 'Center'*
Left Center Right	*LEFT 'Left' CENTER 'Center' RIGHT 'Right'*

Example of alignments with stub on the right.

The stub label is:	Stub on the right
LEFT 'Left'	..... Left
RIGHT 'Right'	..... Right
CENTER 'Center'	..... Center
LEFT 'Left' CENTER 'Center' RIGHT 'Right'	..... Left Center Right

RIGHT IN SPACE for Right-Alignment to a Selected Point in a Label

A specification of RIGHT in a label causes the following label section to be aligned at the right edge of the label space. If you wish to right-align to some other point within the label space, you can use RIGHT IN SPACE.

Format RIGHT IN SPACE location [unit]

The label **section** following RIGHT IN SPACE will be right-aligned to the **location.** The optional unit of measure can be expressed as inches, cm, or points. If no unit is specified, the unit is assumed to be characters.

The first **section** of a label begins at the beginning of the label. A label **section** ends with any of the following label elements: /, RIGHT, LEFT, CENTER, RIGHT IN SPACE, SPACE, and SPACE TO.

The location is measured from the beginning of the label space. For example, in a table title, the label space begins at the left edge of the table. For a stub, the label space begins at the left edge of the stub. Note, however, that nested stub labels and continuation lines for multi-line stub labels are automatically indented. In these cases, the location is measured from the indented point. If you want a different result, see the section on "Interaction of indent with automatic indentation" for ways of controlling the indentation.

Note that RIGHT IN SPACE applies only to left-aligned label sections. Certain types of labels, such as heading and stub head labels, are centered by default. With these, you must use LEFT to left-align before specifying RIGHT IN SPACE, as shown in the example below.

The location must be within the available space. For example, if you specify the following for a heading label:

```
LEFT  RIGHT IN SPACE 2 INCHES 'Health Insurance'
```

and the column width is only 1.5 inches, the label section can't be aligned to a point 2 inches to the right.

If RIGHT IN SPACE is applied to a label section that cannot fit in the space preceding the location, RIGHT IN SPACE is ignored. For example, if the label section is 5 inches long, and you specify RIGHT IN SPACE 3 INCHES, the label section cannot fit in the 3 inch space.

Example

In the following example, RIGHT IN SPACE is used to right-align two sections of the table title 3 inches into the title space. A series of stub labels is right-aligned to a location 1.5 inches into the stub space.

```
define plan_stub indent 2 'All participants'/ on plan_type;
  right in space 1.5 inches 'Total' if 1;
  right in space 1.5 in 'Single employer' if 2;
  right in space 1.5 in 'Multiemployer' footnote multi if 3;
  right in space 1.5 in 'Mandated benefits' footnote mandate if 4;
  right in space 1.5 in 'Employer association' footnote assoc if 5;

table one 'Table 86.'
        right in space 3 in 'Plan' footnote planf ' administration:'/
        right in space 3 in 'Percent of full-ime participants.' :
heading  total then health then life then other_ins,
stub  plan_stub;
```

Table 86. Plan[1] administration: Percent of full-time participants.

Plan sponsor	Total	Health insurance	Life insurance	Sickness and accident insurance
All participants				
Total	1	100	100	100
Single employer	1	96	97	87
Multiemployer[2]	1	4	3	2
Mandated benefits[3]	1	–	–	11
Employer association[4]	1	([5])	([5])	–

[1] Does not include supplemental plans.
[2] Individual employers in the same or in a related industry.
[3] The majority of the participants with mandated sickness and accident insurance benefits were covered by State temporary disability plans.
[4] Band of small employers in a common trade or business, for example, savings and loan associations. The plan sponsored by the association is not negotiated with the employees.
[5] Less than 0.5 percent.
NOTE: Because of rounding, sums of individual items may not equal totals. Dash indicates no employees in this category.

Using RIGHT IN SPACE to Align Footnote Symbols

For footnotes, the alignment at the bottom of a table is determined according to built-in defaults. If you have footnote symbols of different widths, they will be aligned independently, relative to their footnote text. If you wish, you can right-align the symbols within a space of a specific width using a combination of RIGHT IN SPACE, SPACE TO and SYM. The technique is described with examples in the "Footnotes" chapter.

Footnote References in Labels

Any print label, created in the codebook, table request or FORMAT request, can contain a footnote reference. The footnote reference can be at the beginning of the label, between label segments or at the end. The footnote symbol and text are specified in the SET FOOTNOTE statement. When the label is displayed, the footnote symbol will replace the footnote reference, and the footnote text will be displayed at the end of the table. Complete details on footnotes can be found in the footnote chapter.

In the examples that follow, the footnote reference is shown following various print labels in the codebook and table request.

(*codebook)*

```
REGION 'Regions of U.S.' FOOTNOTE R  CON 1
  (
        'Northeast'  =  1
        'South' FOOTNOTES  =  2
        'East'  =  3
        'West'  =  4
  )
```

(*table request)*

```
COMPUTE INCOME 'Non-farm Income' FOOTNOTE NFI =
        AMOUNT/100;

TABLE SAMPLE
        'Expenditures' FOOTNOTE EXP 'For Plant Equipment':
        STUB  Industry,
        HEADING  Expenditures;
```

Continuation Labels for Table Titles

Table titles are the only print labels that have automatic continuation labels. For example, suppose that we have a table title set up as:

```
TABLE REG_TAB 'Region Summaries for 1981 - 91':
        stub,
        heading;
```

If the table continues beyond a single page, a continuation label will follow the title for pages after the first:

Region Summaries for 1981 - 91 - Continued

If the title contains the keyword **CONTINUATION** at some point before the end of the title, the continuation indicator will be inserted at that point.

The continuation label ' **- Continued**' can be changed using the FORMAT statement **REPLACE TITLE CONTINUATION WITH 'new continuation'**.

SPANNER Labels

Spanning the Table with Wafer Labels

Normally wafer labels are displayed at the top left of each wafer between the table title and the heading. You can move the wafer label down into the body of the table if you convert it to a SPANNER label. This can only be done in a format request. For complete details, see WAFER LABEL SPANNER in the FORMAT chapter of the manual. A SPANNER specification entered directly into a label will only produce a spanner if the label is used in the stub as described in this chapter.

Spanning the Table with Stub Labels

A stub label will span across a table if either of the words SPAN or SPANNER are included in the label. The spanner will have a horizontal rule (line) above and below it and the label will be centered.

The spanner can be limited to the data area or it can span the entire width of the table including the stub. You can choose the spanner style you want using a FORMAT statement:

```
DATA SPAN;
```

will cause all spanner labels to extend only across the data columns. The label will be centered within the data columns. DATA SPAN is the system default. If this is the spanner style you want, you will get it automatically.

```
ROW SPAN;
```

will cause all spanner labels to extend across the entire table, including the stub. The label will be centered relative to the full width of the table. To choose this spanner style, add the statement ROW SPAN to your PROFILE.TPL file or FORMAT request.

You can override alignment defaults with the FORMAT statement ALIGN STUB LABELS. If you put an alignment specification into a stub label, it will take precedence over any other alignment specifications.

If the **$** or **%** characters are used in masks that apply to the data columns, then, for any column with these masks, the **$** or **%** character will be repeated in the first non-empty cell following each spanner.

If a label with the SPANNER attribute is used anywhere in a table other than in the stub, the SPANNER attribute will be ignored. See the FORMAT statement WAFER LABEL SPANNER to span the table with wafer labels.

Note If the stub is deleted or the stub width is set to 0 (zero) with FORMAT statements, all stub entries, including SPANNER labels, will be deleted.

Table spanner example

Assume that we want to collect certain state codes into region categories with a define statement, then tabulate persons by sex and occupation for each region. We want region by occupation in the table stub with region labels spanning across the data columns. We can do this by including the spanner attribute in the region labels, then nesting region with occupation in the table statement.

```
DEFINE REGION  ON STATE;
       SPANNER 'Northeast'       IF 1:7;
       SPANNER 'North Central'   IF 8:11;
       SPANNER 'Southeast'       IF 12:16;
```

```
Table  one  'Table showing how spanner labels works':

    HEADING  TOTAL THEN SEX;
    STUB  REGION BY OCCUPATION;
```

Table showing how spanner labels work.

	Total	Sex of Householder	
		Male	Female
	Northeast		
White collar	294	172	122
Blue collar	308	197	111
Farm workers	949	625	324
Service workers	1,072	771	301
	North Central		
White collar	655	338	317
Blue collar	597	345	252
Farm workers	1,723	1,148	575
Service workers	1,654	1,196	458
	Southeast		
White collar	3,037	2,007	1,030
Blue collar	2,794	1,810	984
Farm workers	8,203	5,739	2,464
Service workers	8,714	6,473	2,241

Alignment of Spanning Stub Labels in Banked Tables

When a table has banks of unequal width, stub labels with the SPANNER attribute are formatted and aligned for the width of the narrowest bank in the table. This means that a centered label will be centered only in the narrowest bank and a RIGHT label segment will be all the way to the right only in the narrowest bank.

Inserting Spanners at the Lowest Level of Nest

Spanner labels cannot be used at the bottom level of label nesting, because the same row of a table cannot contain both data and a spanner. If you have spanner labels in the stub with nothing nested below, you will get a message in the table stub that says "***** BAD SPANNER *****".

Following are two approaches to grouping sets of rows for a variable and inserting spanner labels for the groups. Assume we have a variable called **state_code** in the codebook with condition names and labels included for each state. In each of the two examples, we are grouping states into re-

gions with a spanner for each region followed by individual rows for each state. Condition labels for the states are copied from the codebook. The resulting tables are identical.

1. Define region groups with a spanner label for each group. Then nest the states within the region variable in the table stub.

```
define region on  state_code;
        spanner 'New England' if Connecticut;
                              if  Maine;
                              if  Massachusetts;
        spanner 'Mid Atlantic'  if  New_Jersey;
                              if  New_York;
                              if  Pennsylvania;

table one:
        heading  total then sex;
        stub region by state_code;
```

2. Do separate DEFINE statements for each region, use the variable labels as the spanners, and concatenate the variables for each region in the stub.

```
define new_england spanner 'New England' on state_code;
        copy    if  Connecticut;
        copy    if  Maine;
        copy    if  Massachusetts;

define mid_atlantic spanner 'Mid Atlantic' on state_code;
        copy    if  New_Jersey;
        copy    if  New_York;
        copy    if  Pennsylvania;

table two:
        heading  total then sex;
        stub  new_england then mid_atlantic;
```

Spanners for Nested Variables

Since a particular spanner can only be associated with a single variable or a single condition value, nesting of variables cannot produce a spanner for the combination of the two variables. The following example shows a technique for creating multi-variable spanners.

For each nesting of the upper level variable, prepare a DEFINE statement with spanner labels that include the identifying information for the lower

level variable. Then create null labels for the lower level variable, so that there will not be redundant labels outside of the spanners.

```
DEFINE REGION_HOUSEHOLDS ON STATE_CODE;
  SPANNER 'Northeast - Number of Households'        IF 1:7;
  SPANNER 'North Central - Number of Households'    IF 8:11;

DEFINE REGION_AVG_INCOME ON STATE_CODE;
  SPANNER 'Average Family Income in the Northeast'  IF 1:7;
  SPANNER 'Average Family Income in North Central'  IF 8:11;

POST COMPUTE HH_LABEL '' = HOUSEHOLDS;
POST COMPUTE AVERAGE '' MASK RIGHT $99,999 =
      INCOME / HOUSEHOLDS;

TABLE ONE  'Table showing spanners for nested variables':
      HEADING  TOTAL THEN SEX;
      STUB (REGION_HOUSEHOLDS BY HH_LABEL THEN
      REGION_AVG_INCOME BY AVERAGE) BY STATE_CODE;
```

Table showing spanners for nested variables

	Total	Sex of Householder	
		Male	Female
	Northeast - Number of Households		
New York	2,112	1,283	829
Pennsylvania	1,375	980	395
New Jersey	1,142	764	378
	North Central - Number of Households		
Michigan	1,138	812	326
Illinois	1,226	838	388
Indiana	439	307	132
	Average Family Income in the Northeast		
New York	$33,139	$39,971	$22,567
Pennsylvania	31,190	36,259	18,613
New Jersey	39,777	46,594	26,000
	Average Family Income in North Central		
Michigan	$33,623	$38,547	$21,356
Illinois	33,395	39,702	19,774
Indiana	26,280	30,480	16,511

Indentation and Spacing in Labels

Changing Label Alignment with INDENT

INDENT specifications are used in labels to assist in label alignment.

Format

The format for the indent specification is:

```
INDENT [ + or - ]  amount  [ unit ]
```

where **amount** is the size of the indent (decimal numbers are allowed). The amount can be up to about 25 inches.

The optional **unit** specification can be expressed as inches, cm or points.

Example

```
INDENT .5 INCHES
```

A positive indent amount will shift the label right; a negative amount will shift the label left.

If no unit is specified (as shown in the following example), the unit is assumed to be characters.

Example

```
define selected_regions on state_code;
                  'Northeast'              if 1:9;
        indent 3  'New England'            if 1:5;
        indent 3  'Mid Atlantic'           if 6:9;
                  'Midwest'                if 10:21;
        indent 3  'East North Central'     if 10:15;
        indent 3  'West North Central'     if 16:21;
```

If selected_regions is used in the table stub and a text table is produced the alignment of the labels will be:

```
Northeast
   New England
   Mid Atlantic
   Midwest
   East North Central
   West North Central
```

With a regular proprtional type table, the table stub will be:

```
Northeast
   New England
   Mid Atlantic
Midwest
   East North Central
   West North Central
```

Note that this example assumes that STUB INDENT and STUB CONTINUATION have been set at 0. For other uses of INDENT in the table stub, see the section below on **"Interaction of INDENT with Automatic Stub Indentation"**.

INDENT works properly only with left-justified labels.

INDENT applies to all lines of a label that follow it. If you begin a label with INDENT, then add another INDENT specification in the middle of the label, the second INDENT will take effect at the beginning of the next line. For example:

```
indent 1 cm 'label line 1' indent .5 cm / 'label line 2'
```

will give the result:

```
        label line 1
    label line 2
```

If slashes are included in the label to show where the label should break to go to a new line, an INDENT specification for the new line can be inserted either before or after the slash. The label

```
indent 1 cm 'label line 1' / indent .5 cm 'label line 2'
```

will give the same result as the label shown above. It is identical except that the INDENT for the second line follows the slash rather than preceding it.

If you have not inserted slashes to show the break point for a long label but wish to control the indentation following the break, you must insert an INDENT somewhere in the label before the break point. For example,

```
INDENT 3 'This is' INDENT 6 ' a long multi-line label.'
```

The first line of the label will be indented 3 characters. Continuation lines will be indented 6 characters.

Interaction of Indent with Automatic Indentation

When an INDENT specification is used in the table stub, the specified indentation is added to (or subtracted from) any other indentation in effect.

Example

A nested stub label will, by default, be indented 2 more spaces than the label above it. To "cancel" this indentation, you can use a negative INDENT. The default indentation for the following table

```
COMPUTE INCOME 'Income in Thousands' = FULL_INCOME / 1000;

TABLE ONE:  HEADING TOTAL, STUB TOTAL BY INCOME;
```

would result in a stub of

```
Total
   Income  in  Thousands
```

If we insert a negative INDENT at the beginning of the INCOME label, it will be shifted left and aligned with the Total label.

```
COMPUTE INCOME INDENT -2 'Income in Thousands' =
        FULL_INCOME / 1000;
```

will result in a stub of

```
Total
Income  in  Thousands
```

If you want to replace all automatic stub indentation with your own specifications, you can use the FORMAT statements

```
STUB INCREMENT = 0;  and
STUB CONTINUATION = 0;
```

to turn off the automatic indentations.

Indent Restrictions

There must to be space on the current line for at least two characters of label in addition to the indentation.

Indent with Proportional Fonts

In a table without proportional font, all characters, including blanks, are the same width. If you are working with a proportional font, the character width depends on the character. Numbers will all have the same width, but for other characters the width will vary. For example, the letter **o** will

be wider than the letter **i**. In particular, a blank will take up about half the space of the average character width or the width of a number. If you have specified INDENT in characters, the width used for each unit of indentation will be the same as the width of a number in the font you are using.

If you are producing a text table or a table without proportional fonts, you can often easily align labels by simply adding blanks to move parts of the label left or right. If you are working with proportional fonts, use of INDENT rather than blanks will produce better results.

Spacing within Labels Using SPACE and SPACE TO

You can use the words **SPACE** and **SPACE TO** to add a specific amount of space within a label or to space over to a particular location.

Format The formats for the 2 space options are:

```
SPACE amount  [ unit ]
SPACE TO amount  [ unit ]
```

where **amount** is the size of the space or the location to "space to". The amounts can contain decimal points for fractional amounts such as 3.5 . The amount can be up to about 25 inches.

The optional **unit** can be expressed as inches, cm or points. If no unit is specified, the unit is assumed to be characters.

The spacing options should only be used with left-aligned label segments. If used in centered or right-aligned segments, they will either be ignored or give results other than what you expect. When SPACE TO is used, the location is always calculated from the start position of the label without regard to indents or blanks that may be included at the beginning of the label.

If a label segment is too long for the current line after space is added, it will be continued to another line with no space at the beginning of the next line.

Examples

```
'Total' SPACE 10 CM 'All Universities'
'Total' SPACE TO 10 CM 'All Universities'
```

In the first example, there will be a space of 10 centimeters between 'Total' and 'All Universities'. In the second example, space will be added between

'Total' and 'All Universities' so that the distance from the start of the label to 'All Universities' is 10 centimeters.

Using SPACE TO and INDENT Together

SPACE TO and INDENT can be combined as shown in the following example where SPACE TO is used to move a portion of the first line of a table title to 1 inch from the beginning and INDENT is used to indent additional lines to the same location. For an example of SPACE TO and INDENT in footnote text, see the **FORMAT** statement **FOOTNOTE COLUMNS**.

Example

```
TABLE S1 LEFT 'Table 3.3e'
SPACE TO 1 INCH INDENT 1 INCH
'Petrolum Imports: Angola, Australia, Bahama Islands, Brazil, '
'Canada, and China.' /
FONT H 10 '(Thousand Barrels per Day)':  ..... ;
```

Table 3.3e Petroleum Imports: Angola, Australia, Bahama Islands, Brazil, Canada, and China
(Thousand Barrels per Day)

Both INDENT and SPACE options are designed to work with left-aligned label segments. All segments of the table title are left-aligned by default, but it is possible to get different alignments for independent segments. The next title is the same as above but the last line is centered. We can make the centering work correctly by setting INDENT back to 0 for the last line so that no indentation is in effect for that line.

Example

```
TABLE S2 LEFT 'Table 3.3e'
SPACE TO 1 INCH INDENT 1 INCH
'Petroleum Imports: Angola Australia, Bahama Islands, Brazil, '
'Canada, and China.' /
INDENT 0  CENTER  FONT H 10  '(Thousand Barrels per Day)': ..... ;
```

Table 3.3e Petroleum Imports: Angola, Australia, Bahama Islands, Brazil, Canada, and China
(Thousand Barrels per Day)

Links and Anchors in HTML Export

HTML provides a way for page viewers to jump between web pages or different locations within a web page. This is accomplished by inserting **Links** and **Anchors** within the web pages. An Anchor is a destination. A Link is an instruction to jump to an anchor or web address when the link is clicked by the viewer. Links are displayed to web page viewers by underlining. Anchors are not visible in the displayed web page. Neither Links nor Anchors affect the appearance or behavior of tables which have not been exported to HTML.

Links can be used in tables for such things as providing a way to move from a footnote symbol in the body of a table to its footnote text or even to link to explanatory text not produced in TPL Tables.

Anchor

An Anchor is inserted into a label between text segments using the code

```
HTML  ANCHOR anchor-name
```

where *anchor-name* is any string of characters including internal blanks but excluding special characters such as **#**, **''**, **'**, etc. *The names are case sensitive*. If an anchor name includes a blank, it should be enclosed in quotes.

Example

```
Set Footnote Revised symbol R text HTML ANCHOR "REVISED FN"
"Data has been revised.";
```

Anchors should be unique for any given web page. Special care should be used if an anchor is attached to a variable or condition label which may be repeated on an output page.

Link

A Link is inserted into a label between text segments using the code

```
HTML LINK link-name
```

where *link-name* is a path to a file or anchor.

The path should be relative. If the target file will be in the same directory as the file you are linking from, you should just enter the target file. If you must follow a path it should begin with something like **..** rather than something like **C:**. This is because web pages will typically be moved from a local disk to a web site. Note that you may use forward or backward slashes.

The link should include the full name of the target file; e.g., **table2.htm** not just **table2**. You must know what the target file name will be even if you haven't exported it yet. You may find it easier to do an export, figure out file names, add links, and then export again.

If you wish to jump to a specific location on the target page, you must put an anchor at that point and add the anchor name to your link path. You specify this by writing your link path followed by **#** followed by the anchor name; e.g **"table2.htm#start of footnotes"**. If the target is on the same page as the link, you can just include the anchor; e.g. **"#start of footnotes"**.

Links, unlike anchors, are displayed on the web page. So the location of the link matters. The link causes the label segment following the link to be underlined. If the link is placed after the last segment of the label, nothing is displayed with underline and the link does not work. If you want the underline to span more than one table segment, you must repeat the link for each segment.

Example Assume we have a two page table with the state of Illinois on the first page and footnotes on the second page. We export the HTML with a base name of **PAGE.**

```
Set footnote r symbol 'R' text html anchor "revised footnote" "revised";

For Table 1 Condition State(Illinois): replace label with "Illinois" HTML
LINK "./page2.htm#revised footnote" footnote(r);
```

If the first page of the resulting HTML is looked at in a browser, the footnote symbol **R** will be underlined. If a user clicks on it, the browser will jump to the footnote text for footnote **r** on page 2.

Font Control in Labels

Fonts can be set for different types of labels, including titles and footnote texts, using FONT statements in the FORMAT language. This method of font selection works well if you want all labels of a certain type to have the same font. Sometimes, however, you may need to use a different type style or size for particular labels or for different sections within the same label. You can do this by including fonts in individual labels. These font specifications have no effect on text tables. For a complete list of available

fonts, including **bold**, *italic* and underline fonts, see the **FONT** statement in the **FORMAT** section of the manual.

A font specification within a label takes the same form as in the FORMAT language FONT statement. To change the font for an entire label, simply insert the FONT specification at the beginning. For example, the following label will be printed in **T**imes **B**old **I**talic:

```
FONT TBI '***Revised***'.
```

Fonts can change more than once within a label. For instance, a label could begin with a section of **bold-underlined** type, change to *italic* and end with ***bold-italic***. To change fonts within a label, insert the font specifications anywhere between strings.

The expression **FONT RESET** can be used at any point to restore the default label font for a later section of the label. The following example shows how FONT and RESET can be used in a footnote text:

```
SET FOOTNOTE A TEXT = 'As published in '
        FONT HI 'Three Little Pigs'
        FONT RESET ' by Anon.';
```

The font size is optional. If the font specification is in the middle of a label and does not include a size, the size is the same as for the previous part of the label. If the font specification is at the beginning of the label and does not include a size, the size is the same as the default size for that type of label.

A font remains in effect until another new font is specified or the end of the label is encountered. The special font RESET is the same as the default font for that type of label. Thus, for instance, if we have set the default of **FOOTNOTE TEXT FONT = H 8**, the example shown above would give the same result as:

```
SET FOOTNOTE A TEXT = 'As published in '
        FONT HI 8 'Three Little Pigs'
        FONT H 8 ' by Anon.';
```

In either case, the footnote would print as:

As published in *Three Little Pigs* by Anon.

The advantage of using the RESET font is that if you change the default font for a particular type of label, you will not need to adjust individual labels to match the new default.

For another example, assume that the default title font has been set with:

```
TITLE FONT H 10;
```

If we want all parts of the title to have the default size, but different styles for some sections, we can add FONT specifications to the title without including sizes. For example,

```
CENTER 'Table B-4.  '
        FONT HBU 'Median and Average Sales Prices'
        FONT HB ' of New Houses Sold in the United States, '
        'by Region.' /
        FONT RESET '[Rounded to hundreds of dollars]'
```

This title would print as:

Table B-4. **Median and Average Sales Prices of New Houses Sold in the United States, by Region.**
[Rounded to hundreds of dollars]

If we later find that we need to increase or decrease the size of the title font for all tables, we can do so by changing only the TITLE FONT statement. Size adjustments in the individual titles will be automatic. Assuming that we change the default title font to **TITLE FONT H 8**, the title shown above will print as:

Table B-4. **Median and average Sales Prices of New Houses Sold in the United States, by Region.**
[Rounded to hundreds of dollars]

Font Defaults

When TPL TABLES is installed, default fonts are set in the profile. You can change the defaults in the profile, or you can change them in a format request for an individual job.

If a default font is not specified for some type of label, the font is determined by other font defaults. For example, if CONDITION LABELS IN

HEADING FONT is not specified, the condition labels in the heading get the CONDITION LABELS FONT. Ultimately everything defaults to DEFAULT FONT. See the FONT statement in the Format chapter for more details.

Vertical Spacing

TPL TABLES will determine appropriate vertical spacing of labels and data according to the fonts being used. If you are satisfied with the spacing, you do not need to be concerned with the following details.

If fonts of different sizes are used in a label, the vertical spacing for the label is determined by the largest font specified in the label or the default font size for the type of label, whichever is larger. For data rows, if the DEFAULT FONT is larger than the stub label fonts, the vertical spacing for data rows will be determined by the DEFAULT FONT.

Example

```
TITLE FONT = HB 10;
```

The title is specified as:

```
FONT HB 8 'Table 4.  Civilian employment in occupations with '
'25,000 workers or more, under low, medium, and high scenarios '
'for economic growth' / FONT H 7 '[Numbers in thousands]'
```

This multiline title will have the vertical spacing specified by the default TITLE FONT size of 10, because the default title font size is larger than any font size explicitly included in the title. If you want to reduce the space between the label lines, set a smaller default TITLE FONT size.

Superscripts and Subscripts

Superscripts and subscripts can be used in labels, including footnote texts. For text tables, the superscript and subscript notations are ignored.

Enter the superscript or subscript notation in the label in front of the appropriate label segments. For superscript, use **SUP** or **SUPER**; for subscript, use **SUB**. The superscript or subscript specification will apply from that point in the label, either to the end of the label or to the next occurrence of the notation **NORMAL**. These notations can be mixed with other label features such as font, spacing and line break specifications.

Superscript characters are raised by the same amount as superscripted footnote symbols; subscripts are lowered to the base line of the label.

Example 'Regular label part ' SUP 'Superscript part ' NORMAL 'End'

The label text 'Regular label part ' will be printed at the normal level, the label text 'Superscript part ' will be raised, and the label text 'End' will be at the normal level.

CHAPTER 21

Masks

FORMATTING THE DATA CELLS WITH MASKS

Table cell values that do not have masks are rounded to the nearest whole integer and displayed with no special symbols except commas. The values are right-adjusted in the table columns. If you want a different format for values, you can specify the format using a print mask.

With a mask, you can format data with decimal points, commas, and special characters such as dollar signs and percent symbols. A mask can also reference footnotes. When a mask is used, data is centered in the table columns based on the size of the mask, or right-alignment can be specified. You can choose the type style for table cells by inserting font specifications in masks.

A mask can be assigned to any observation variable described in the codebook or computed in a table request. Whenever the variable is used in a table, the mask determines the format for the variable's tabulated values. The **REPLACE MASK** statement can also be used in a FORMAT request to assign or replace a mask either by variable or by table location. To replace an entire cell value with a different value, see the FORMAT statement REPLACE VALUE.

The mask functions as a pattern for formatting the cell values. In its simplest form, it consists of a succession of 9's, one for each digit position of the largest expected cell value. For example, a mask of:

```
MASK  9999
```

would indicate that the largest expected final cell value has four digits. The values would be centered based on the size of a four digit number and would be printed without commas or other special characters.

Adding Decimal Points, Commas, $ and %

When decimal points, commas and other special symbols are to be displayed with the cell values, the symbols are indicated in positions relative to the 9's. The following mask will format the values with a comma and two decimal places; a dollar sign will precede the values:

```
MASK  $9,999.99
```

Only one of **$** and **%** can be used in the same mask. If a mask with a $ or % symbol is used with a heading or wafer variable, the symbol normally will be displayed only in the first non-empty cell of each column for that variable. An exception to this rule occurs if you have SPANNER labels or if you use FORMAT statements to insert horizontal rules (lines) in the data section of the tables. In these cases, the $ or % characters will be repeated for each column in the first non-empty cell following each spanner or horizontal rule. Another exception occurs if a cell in the column has a mask assigned to it which does not contain the $ or % character.

Tip

Built-in footnotes, such as the EMPTY footnote that appears in cells for which there is no data, override the mask rather than change it. Thus, they do not cause a new $ to be printed on the next line. Footnote-only cells coming from conditional post computes or replace mask statements do result in new masks and so may cause a new $ to be printed. If you wish to suppress the new $, put a $ on the footnote-only mask. For example:

```
REPLACE MASK WITH $ FOOTNOTE UNPUBLISHABLE;
```

No $ will appear on the line following this replacement mask. Also, $'s are always suppressed for footnote-only cells so there is no danger of getting a cell such as $(1).

If a value is to be printed with a dollar sign, the dollar sign will be displayed immediately to the left of the cell value, regardless of the number of digits in the value. If a cell value is larger than the mask and the mask contains one or more commas, additional commas will be inserted as required.

Decimal cell values are rounded to the number of decimal places shown by the mask. If a decimal cell value is formatted without a mask, it is rounded to the nearest integer value.

Rounding Rule

By default, rounding is done according to the "round even" rule.

Note You can override the "round even" rule and choose to round up instead. See the ROUND statement in the "Format" chapter for details.

With "round even", 5 is rounded up or down depending on the digit to the left of the 5. If the digit to the left of the 5 is even, it rounds down. If the digit to the left is odd, it rounds up. (A blank to the left is considered to be a zero and thus even.)

For example, with a mask of 99.9:

```
5.8500 -> 5.8      (8 is even -> round down)
5.7500 -> 5.8      (7 is odd  -> round up)
```

This rounding rule is part of the IEEE and ANSI standards for binary and floating point arithmetic.

Note that detail cells in a tabulation may not add to totals because of rounding. This is true regardless of the rounding rule being used. The following illustrates results with the “round even” rule. In this case, the sum of the rounded detail cells is greater than the rounded sum.

```
2.5 -> 2      (2 is even -> round down)
4.5 -> 4      (4 is even -> round down)
------------
7.0  ≠  6
```

Creating Decimal Places

A cell value is assumed to be a whole number with no decimal places unless:

- it contains values described with a SHIFT LEFT clause in the codebook;
- it contains values described as floating point in the codebook; or
- it contains values resulting from computations that add decimal places (for example, division in a Compute or Post Compute statement).

If the cell value is assumed to be a whole number and is formatted with a mask that contains a decimal point, the whole number will be printed to the left of the decimal point with 0's to the right of the decimal point. For example, if the mask **$99,999.99** is used to display a cents aggregation of 47378, the displayed result will be **$47,378.00**, since the decimal point is assumed after the 8.

To show values of this type with the correct number of decimal places, the decimal places must be created by division in a COMPUTE or POST COMPUTE statement. For the dollars and cents example, we can create two decimal places by dividing the tabulated value by 100 in a POST COMPUTE statement as in:

```
POST COMPUTE DOLLARS USING $99,999.99 = CENTS / 100;
```

The cell value of **473.78** used with the mask **$99,999.99** will then be displayed correctly as **$473.78**.

Leading Zeros

When a decimal value less than zero is printed, it is always displayed with a zero to the left of the decimal point. An example is **0.48**. This is true regardless of what mask is used for the value, even a mask such as **MASK .99**. If you do not want to display these zeros, you can remove them by using the FORMAT statement DELETE LEADING ZEROS; If this statement is used, our example value will print as **.48** instead of **0.48**.

Character Strings in Masks

A mask can be preceded or followed by a character string bounded by quote marks. In this case, the character string will be displayed with all cell values to which the mask applies. For example, if an entire column is to be printed with a trailing percent symbol, a mask such as **99.9'%'** could be used.

A mask can consist of only a character string bounded by quote marks. In this case, the character string will be displayed alone without the cell value. You can even make a cell blank by using **MASK ' '**.

See also the section called **TEXT Masks** for additional ways of putting text in table cells.

Moving the Decimal Point before Display

You can add a DISPLAY DECIMAL clause to move the decimal point to the left or right before values are formatted for output.

Example

```
POST COMPUTE AVG_INCOME
        MASK 999 DISPLAY DECIMAL LEFT 3 =
        INCOME / PERSONS;
```

Assume that AVG_INCOME values are in dollars. For each value of AVG_INCOME, the decimal point will be shifted left three positions and the value will be displayed as a whole number. The effect is to show the average income values in thousands of dollars. For the value 75724.36, the decimal point will be moved left three positions. The resulting value of 75.72436 will then be rounded to a whole number according to the mask of 999 and will be displayed as 76.

DISPLAY DECIMAL can be added to any mask, in the codebook, table request or format request. The mask can be a regular mask or a TEXT mask. Regardless of where it is entered, it is used only for display purposes and does not affect tabulation or other computations.

Restriction

The DISPLAY DECIMAL clause will not be applied in any cell where you have replaced the value using the FORMAT statement REPLACE VALUE.

Replacing Rounded Digits with Zeros

Data can be rounded and displayed with trailing zeros by inserting zeros in the mask. For example, a mask of **999,000** causes data to be rounded to the nearest thousand and displayed with three zeros in place of the rounded digits. The value **876859** will be displayed as **877,000**.

Zeros in masks are ignored if they are to the right of a decimal point or if there are any 9's to the right of the zeros. A mask of **9909** is treated the same as a mask of **9999**; a mask of **9900.00** is treated the same as a mask of **9999.99**.

Alignment of Values

Cell data for which a mask is given will be centered within the column width unless other alignment is specified. Cell data for which no mask is given will be right justified within the column width. The number of characters making up a mask will be used to control the centering of data

within the column. The mask may be thought of as being positioned at the center of the column, with cell values being aligned with the mask from right to left. For example, a mask of **$99,999** used together with a column width of 10 (including the column divider) would give the following results.

Value	Will display as
23567	$23,567
146	$146

Restriction For text tables, if a mask cannot be perfectly centered because of an uneven number of spaces, it is adjusted to the right one position. For example, if there are 9 spaces available for a 6 character mask, the mask will be positioned with 2 spaces to the left and 1 space to the right.

For data centered according to a mask that contains a footnote reference, the footnote symbol will not be included in the centering but will be added in the space to the left of the data.

The keywords **RIGHT** and **CENTER** can be used with a mask to force alignment of values to the right side or center of the column. For example:

```
COMPUTE MAX_SALARY 'Maximum Salary'
        MASK $999,999 RIGHT = MAX(INCOME);
```

Since the default alignment for masks is **CENTER**, you do not need to add this word to a mask to specify centering.

The keyword **RIGHT** will have no effect on a mask containing **only** a character string. The string will always be centered.

Alignment cannot be specified for a mask that contains only a footnote reference. With this type of mask, the data cell would contain only a footnote symbol. To change the alignment of the symbol, include the alignment in the SYMBOL part of the SET FOOTNOTE statement for the footnote.

Tip on Aligning Different Masks within Columns

If you have different masks from row to row in a table, the values from rows with different masks may not be aligned the way you want them to be. For example, if the values are to be centered, each will be centered based on the length of its own mask, without regard to the mask above or below it.

If you wish to make adjustments to align all values in a column, you can sometimes do it simply by adding one or more 9's at the beginning of the shorter mask. Another approach is to add character strings of blanks to one or more of the masks to account for the differences. For example, to align the last digit of values that have a mask of **999** with values that have a mask of **.999**, you can add a blank to the shorter mask as follows: **MASK ' ' 999**. A good way to experiment with the results is to reformat the table output using FORMAT statements to replace the mask for one or more of the variables.

Note that if you are using proportional fonts, the characters do not all have the same width. Blanks and other characters such as decimal points and commas are smaller (more narrow) than numbers. In our example masks above, both have three 9's but one mask is longer by the width of a decimal point, one of the small characters. In this case, adding a single blank to the shorter mask will be sufficient to get alignment. To adjust for an extra 9 in the mask, we would need to add two blanks, because a number takes twice as much space as a blank in a proportional font.

Footnote References and Cell Markers in Masks

A footnote reference can be included in a mask. An example is:

```
MASK $999,999 FOOTNOTE(SOURCE)
```

The symbol for the footnote called **SOURCE** will be inserted in front of the cell value in all table cells affected by the mask. The footnote symbol and text can be provided in a SET FOOTNOTE statement. See the chapter on footnotes for complete details on the use of footnotes in masks.

A Cell Marker is like a footnote symbol without associated footnote text. An example is:

```
REPLACE MASK  WITH 999.99 MARKER ab;
```

ab will be inserted in front of the cell value in all table cells affected by the mask. See the REPLACE MASK MARKER statement in the Format chapter for more detail.

Treatment of Large Cell Values

If a cell value has more digits than shown in its mask, column spaces to the left of the mask space are used, if available. If there is not enough space, the following steps are taken as required to print the value:

1. The value is aligned to the right regardless of the alignment specified by the mask.
2. Leading and trailing mask strings and footnote symbols are removed.
3. Digits to the right of a decimal point are deleted one at a time.
4. If there is still not enough space to print the value, the value is replaced with **nf** and footnoted.

Links and Anchors in HTML Export

HTML provides a way for page viewers to jump between web pages or different locations within a web page. This is accomplished by inserting **Links** and **Anchors** within the web pages. An Anchor is a destination. A Link is an instruction to jump to an anchor or web address when the link is clicked by the viewer. Links are displayed to web page viewers by underlining. Anchors are not visible in the displayed web page. Neither Links nor Anchors affect the appearance or behavior of tables which have not been exported to HTML.

Links can be used in tables for such things as providing a way to move from a footnote symbol in the body of a table to its footnote text or even to link to explanatory text not produced in TPL Tables.

An Anchor is inserted in a mask using the code

HTML ANCHOR *anchor-name*

where *anchor-name* is any string of characters including internal blanks but excluding special characters such as **#**, **''**, **'**, etc. If an anchor name includes a blank, it should be enclosed in quotes. *The names are case sensitive.* Anchors should be unique for any given web page. So if you attach an anchor to a mask, the mask must be for an individual cell, not a row, column or variable.

Example

```
For table 1 row 1 column 1: replace mask with
HTML ANCHOR AA 99.9;
```

A Link is inserted into a mask using the code

```
HTML LINK link-path
```

where *link-path* is a path to a web page or anchor. The path should be relative. If the target file will be in the same directory as the file you are linking from, you should just enter the target file. If you must follow a path it should begin with something like **..** rather than something like **C:**. This is because web pages will typically be moved from a local disk to a web site. Note that you may use forward or backward slashes.

The link should include the full name of the target file; e.g. **table2.htm** not just **table2**. Note that you must know what the target file name will be even if you haven't exported it yet.

If you wish to jump to a specific location on the target page, you must put an anchor at that point and add the anchor name to your link path. You specify this by writing your link path followed by **#** followed by the anchor name; e.g **"table2.htm#start of footnote"**. If the target is on the same page as the link, you can just include the anchor; e.g. **"#start of footnotes"**.

TEXT Masks

You can replace table cell values with text by adding the word TEXT to the mask following the word MASK. A TEXT mask gives you much more flexibility than the simple character strings that can be part of a standard mask. The text can include any of the options associated with other types of labels, such as font specifications, indents and alignments.

You can also include the original numeric cell value in the text by using the word VALUE, but note that the values are not aligned as they would be with a standard mask. Rather, they are included in the text at the specified place. If VALUE is used, you can add an optional decimal indicator in parentheses to specify the number of decimal places for display.

Example

```
POST COMPUTE AVG_AGE 'Age'
        MASK TEXT 'Average ' VALUE (2) = AGE_OBS / PERSONS;
```

If the value of AVG_AGE is 43.5135, it will be rounded to two decimal places and printed in the table cell following the word 'Average' as follows:

```
Average 43.51
```

If the decimal indicator is preceded by a minus sign, the value is rounded to an integer value and displayed with the specified number of trailing zeros. For example, VALUE(-3) applied to the value 85734 rounds it to the nearest thousand and displays it as 85,000.

The default alignment for TEXT masks is CENTER. For text tables, the default is LEFT.

Long cell contents are broken into multiple lines in the same way that a long label is broken into multiple lines. Text tables can contain only one line of information in a cell. So if the line is too long, it is truncated.

TEXT masks can be used to get additional control of the format for cells that contain only footnote symbols. For example, if you do not want the symbol to be enclosed in parentheses as it would be by default, you can put the footnote reference in a TEXT mask without the parentheses:

```
SET FOOTNOTE NP  'NP footnote text';
POST COMPUTE ABC =
        MASK TEXT FOOTNOTE NP  IF PERSONS < 5;
        PERSONS  IF  OTHER;
```

In this example, the footnote symbol will be displayed without parentheses. In addition, the symbol will not be raised unless you add superscript specifications. Superscripts are described in the "Labels" chapter.

Font Control in Masks

The font for data cells is determined by the DEFAULT FONT that you have chosen for table output. Sometimes, however, you may need to use different type styles or sizes for particular variables or data cells. You can do this by including fonts in individual masks. These font specifications are ignored in text tables.

For a complete list of available fonts, see the **FONT** statement in the **FORMAT** section of the manual.

A font specification in a mask takes the same form as in the FORMAT language FONT statement for masks. To change the font for an entire mask, simply insert the FONT specification at the end of the mask. For example,

```
MASK 99,999 FONT TBI 8.
```

Note For a data mask the FONT specification ***must be at the end*** of the mask and the FONT applies to the entire data mask.

If you want to use a variety of fonts within a mask, you may be able to get the desired result by using a TEXT mask. Fonts can be varied within TEXT masks. For example:

```
MASK TEXT FONT TIU 'Average '
        FONT RESET 'Age' FONT HB VALUE;
```

The font size specification is optional. If size is not specified, the size will be determined by the DEFAULT FONT.

The vertical spacing of a data row is not adjusted for the font specifications of individual mask fonts. The spacing is set according to either the largest font in the stub label for the data row OR the DEFAULT FONT -- whichever is larger. If fonts of different sizes are used for different columns and some of the mask font sizes are substantially larger than both the stub label and the DEFAULT FONT sizes, it is possible that the data values with large fonts could overlap those above or below.

Sample Tables Using Masks

The following tables show how various cell values would be displayed with different masks. The variables with masks are first used in the heading, then in the stub.

```
use family codebook;

compute mask1 'No Mask' = gross_income_of_head;

compute mask2 '$999,999' using mask $999,999 =
        gross_income_of_head;

compute mask3 '99,000 right footnote t'
        using mask 99,000 right footnote t =
        gross_income_of_head;

set footnote t text 'Rounded to thousands';

Compute mask4 'right 999999" '" using mask right 999999 "  =
        gross_income_of_head;

compute mask5 '99.99' using mask 99.99 =
```

```
        gross_income_of_head / 100;

compute mask6 '99.9"%"' using mask 99.9'%' =
        gross_income_of_head / 10000;

compute mask7 '99.9% right' using mask 99.9% right =
        gross_income_of_head / 10000;

compute mask8 '99.99%' using mask 99.99% =
        gross_income_of_head / 10000;

table sample1
        'Table showing the effects of various masks used '
                'in heading.':
        stub     heads_class_of_work;
        heading  mask1 then mask2 then mask3 then mask4
                 then mask5 then mask6 then mask7 then mask8;

table sample2
        'Table showing the effects of various masks used in stub':
        stub     mask1 then mask2 then mask3 then
                 mask4 then mask5 then mask6 then mask7
                 then mask8;
        heading  heads_class_of_work;
```

Table showing the effects of various masks used in heading.

	No Mask	$999,999	99,000 right footnote t	right 999999' '	99.99	99.9'%'	99.9% right	99.99%
Head of Family Class of Work								
White collar worker	271,628	$271,628	[1]272,000	271628	2716.28	27.2%	27.2%	27.16%
Blue collar worker	290,948	290,948	[1]291,000	290948	2909.48	29.1%	29.1	29.09
Farm workers	0	0	[1]0	0	0.00	0.0%	0.0	0.00
Service industry workers	85,300	85,300	[1]85,000	85300	853.00	8.5%	8.5	8.53
Armed Forces	0	0	[1]0	0	0.00	0.0%	0.0	0.00
Worker type not reported	0	0	[1]0	0	0.00	0.0%	0.0	0.00

[1] Rounded to thousands

Table showing the effects of various masks used in stub

	Head of Family Class of Work					
	White collar worker	Blue collar worker	Farm workers	Service industry workers	Armed Forces	Worker type not reported
No Mask	271,628	290,948	0	85,300	0	0
$999,999	$271,628	$290,948	$0	$85,300	$0	$0
99,000 right footnote t ...	[1]272,000	[1]291,000	[1]0	[1]85,000	[1]0	[1]0
right 999999' '	271628	290948	0	85300	0	0
99.99	2716.28	2909.48	0.00	853.00	0.00	0.00
99.9'%'	27.2%	29.1%	0.0%	8.5%	0.0%	0.0%
99.9% right	27.2%	29.1%	0.0%	8.5%	0.0%	0.0%
99.99%	27.16	29.09	0.00	8.53	0.00	0.00

[1] Rounded to thousands

CHAPTER 22

Footnotes

FOOTNOTES AND NOTES FOR TABLES

Introduction

The SET FOOTNOTE statement determines the text and symbol for a footnote. This statement can be entered in the codebook, table request, format request or profile (PROFILE.TPL). When the footnote is used in a table, the symbol is printed at the reference point and the footnote text is printed at the end of the table.

You can use a SET NOTE statement anywhere that you can use a SET FOOTNOTE statement. A note is a simple footnote without a symbol. Notes are described later in this chapter.

Footnotes can be referenced in labels and masks. For example, to footnote a table title, we can include the footnote reference in the title part of the TABLE statement:

Example

```
TABLE F1 'Table F1: Population by City ' FOOTNOTE PRELIM:
      STUB  CITY,    HEADING  PERSONS BY SEX;
```

The footnote in this example is called **PRELIM**. We assign a symbol and text using the statement:

```
SET FOOTNOTE PRELIM SYMBOL IS '(P)'
      TEXT IS 'Preliminary Data.';
```

The table title in a text table will print as:

```
Table F1: Population by City(P)
```

The footnote will print at the end of the table as:

```
(P) Preliminary Data.
```

Note that even though the footnote is referenced in the table request, the SET FOOTNOTE statement does not need to be in the table request. The text and symbol can be determined later, for example in a format request. If there are footnotes that you use frequently, you may wish to put their SET FOOTNOTE statements in your profile.

A particular footnote applies to all tables in a job. If the same footnote name is used in more than one SET FOOTNOTE statement, the text and/or symbol information from a later statement will replace those from an earlier statement.

Normally, if a footnote is not referenced or used in a table, it will not be printed. To keep this type of footnote in a table, see the section on "Forcing Printing of Unused Footnotes" or the SET NOTE statement.

TPL TABLES has several built-in footnotes that are automatically included in your tables, regardless of whether they are explicitly referenced in labels or masks. They are described in the section called "Built-in Footnotes".

Entering and Referencing Footnotes

The SET FOOTNOTE Statement

Format SET FOOTNOTE (footnote-id) TEXT label SYMBOL string ;

where **footnote-id** is a name or number. The parentheses around the footnote-id are optional.

The **TEXT** and **SYMBOL** are optional and can be in any order, but there must be at least one of the two in the statement. The footnote TEXT can be any valid TPL TABLES label except that it cannot itself contain a reference to another footnote. The footnote SYMBOL is a character string enclosed in quotes. If you wish, you can use **IS** or = following the words TEXT and SYMBOL.

Example

```
SET FOOTNOTE CONFIDENTIAL SYMBOL IS '(C)'
        TEXT IS 'Confidential Data';
```

If SET FOOTNOTE is used in a codebook, the **';'** at the end of the statement is optional. In the table request, format request or profile, the **';'** is required.

Although a **number** can be used as the **footnote-id**, we recommend that you use names instead. This is especially true if you choose to let TPL TABLES assign numbers as footnote symbols, since the number you give as a footnote-id is unlikely to match the assigned footnote symbol.

Entering Footnote References

Footnotes can be referenced in labels and masks, in the codebook, table request, format request or profile. The format for a footnote reference is:

```
FOOTNOTE (footnote-id)
```

where the footnote-id is the name or number that identifies the footnote. The parentheses around the footnote-id are optional.

Examples are:

```
(codebook)
MONTH  CONTROL  2
 (
        'January'                               = 1
        'February'                              = 2
        'March' FOOTNOTE (REVISED)              = 3
 )

(table request)
COMPUTE WEIGHTED_INCOME
        'Income' FOOTNOTE (WEIGHTED) =  INCOME * WEIGHT;

(format request)
FOR ROW 1 COLUMN 4:
        REPLACE MASK WITH FOOTNOTE CONFIDENTIAL;
```

Note that you can have a label or mask, such as the one above, that consists of nothing but a footnote. In this example, no data would be printed in the table cell at row 1, column 4; only the symbol for the footnote named **CONFIDENTIAL** would be printed in the cell.

You can have as many as 30,000 distinct footnotes (as specified in SET FOOTNOTE statements) in one job. There is no limit on the number of footnote references.

Choosing Footnote Symbols

You can assign a symbol to a footnote in the SET FOOTNOTE statement, or you can let TPL TABLES assign a number as the footnote symbol.

User-Assigned Symbols

The footnote SYMBOL can be a character string of any length, but short symbols (1 to 3 characters) are recommended. For example, if the footnote is inserted in the mask for a data cell of a table, the combination of the data and a long footnote symbol may be too wide for the width of the column. Note that, although a footnote text can include any of the options described in the label chapter, a footnote symbol is only a simple character string and cannot include label options such as **/** or **INDENT**. With the exception of color specifications, the options explicitly described in this chapter are the only ones that can be used with the footnote symbol.

Default Footnote Symbols

When footnotes are specified without symbols, the system assigns numbers to the footnotes. In general, the numbering starts with 1 at the upper left corner of the table and increases with each new footnote use from left to right and top to bottom. Footnote numbering restarts at the beginning of each table.

Note that in the table heading, the numbers are assigned left to right by column, not by hierarchical heading level. This means, for example, that a footnote in a low level heading label in column 1 could be assigned the number 1, while a footnote in a higher level heading label in column 5 could be assigned the number 2.

Note also that the first footnote number assigned to a wafer label footnote will be greater than the numbers assigned to title, headnote or heading footnotes if these exist on the same page.

If you do not like the numbering that is assigned automatically, you can assign your own numbers. For example:

```
SET FOOTNOTE WN  SYMBOL '1'  TEXT 'Footnote for wafer';
SET FOOTNOTE HN  SYMBOL '2'  TEXT 'Footnote for heading';
```

Display of Footnote Symbols in Tables

Display of Footnote Symbols in Labels and Text Masks

When a footnote is referenced in a label, the footnote symbol is inserted in the label in the specified location. The footnote symbol font is used, and the footnote symbols are raised (displayed as superscripts). If multiple footnotes occur at the same point in a label, they are displayed side by side as superscripts separated by commas.

text[3,4]

In a text table, default footnote symbols (numbers) are displayed in parentheses. User-assigned footnote symbols are displayed without parentheses, unless the parentheses are part of the symbol string. If multiple footnotes occur at the same point in the label, the symbols will be displayed side by side. An example of default footnotes symbols displayed side by side is:

text(3)(4)

Occasionally it is desirable to have the footnote symbol even with or even below the text. This can be accomplished by placing NORMAL or SUB immediately before the footnote reference in the label.

Note that if a footnote is referenced in a FORMAT statement following a TITLE CONTINUATION, the footnote symbol will not be displayed on the first page of the table but will follow the title continuation on subsequent pages.

Display of Footnote Symbols in Masks

Table cells can be footnoted by including footnote references in masks. *There can be, at most, one footnote symbol displayed in a particular table cell.* Its location is fixed immediately to the right of any leading string in the mask and to the left of a floating **$** or data value. Note that in the special case where a mask consists of only a character string (no value), this rule will give the correct result of putting the footnote symbol at the end of the string.

If there is data in a footnoted table cell and the data mask specifies centering (centering is the default for masks), the centering is determined before the footnote symbol is added.

If a table cell is footnoted, but there is no data for that cell, the footnote symbol is displayed alone and centered within the cell. The footnote symbol font is used and the symbol is raised. When the footnote symbol is used alone in a data cell, it is enclosed in parentheses. The parentheses are in the font that is in effect for the cell. This is the font that would be used if data were displayed in the cell.

For text tables default footnote symbols (numbers) are enclosed in parentheses when displayed in table cells; user-assigned footnote symbols are not.

Important Note User-assigned footnote symbols are displayed exactly as specified, without the addition of blanks or other separators. Thus, if you are specifying letters or numbers as footnote symbols, you will sometimes need to include one or more separators in the symbols.

Consider the following text table example of a user-specified footnote symbol that is referenced in a mask in a format request:

```
SET FOOTNOTE XXX SYMBOL '1' TEXT 'Footnote example';

FOR TABLE 1 ROW 3 COLUMN 2:
        REPLACE MASK WITH 999 FOOTNOTE XXX;
```

If the value in the data cell for row 3, column 2 is **453**, the value displayed with the **XXX** footnote symbol will be **1453**. This is clearly an undesirable result.

If the footnote symbol is specified as **'1 '**, the value displayed with the footnote symbol will be: **1 453**.

If the footnote symbol is specified as **'1/ '**, the value displayed with the footnote symbol will be: **1/ 453**.

Adjusting Alignment for Footnote Symbols Used Alone in Data Cells

You can control the alignment of a footnote symbol when it is used alone in a table cell by adding **RIGHT**, **LEFT** or **CENTER** when the footnote symbol is described in the SET FOOTNOTE statement. CENTER is the default alignment for cells that contain only a footnote symbol. If you specify RIGHT, the footnote symbol will be displayed at the far right of

the column. If you specify LEFT, it will be displayed at the far left of the column.

RIGHT alignment of footnote symbols may be particularly desirable in cases where the data in a column is right-adjusted.

RIGHT and LEFT specifications will only take effect when footnotes are used in cells that have no data. In all other cases, the footnote symbol will be displayed according to the normal rules: In cells with data, the symbol will be placed to the left of the data; in labels, the footnote symbol will be placed in the label at the point where it is referenced.

Examples

```
SET FOOTNOTE FEW   SYMBOL '(F)' RIGHT
        TEXT  'Fewer than 5 families represented.';

SET FOOTNOTE FEW   SYMBOL RIGHT '(F)'
        TEXT  'Fewer than 5 families represented.';
```

The alignment specification can precede or follow the symbol, so the two statements above will give the same result. The footnote symbol **(F)** will be right-adjusted in any cell where the data value is replaced by the footnote.

```
SET FOOTNOTE FEW  SYMBOL RIGHT
        TEXT  'Fewer than 5 families represented.';
```

TPL TABLES will generate a default numbered footnote symbol and right-adjust it in any cell where the data value is replaced by a footnote.

```
SET FOOTNOTE EMPTY  SYMBOL RIGHT;
```

This example uses a built-in footnote called EMPTY. Built-in footnotes are described later in this chapter. The dash symbol for EMPTY will be right-adjusted in empty data cells.

Note

You cannot change the alignment of the footnote symbol by using LEFT or RIGHT in the MASK. For example:

```
POST COMPUTE POPULATION =
        PERSON_COUNT                    IF  >= 5000;
        MASK RIGHT FOOTNOTE FEW         IF  OTHER;
```

The use of RIGHT or LEFT in a mask that contains only a footnote will either produce an error message or be ignored. To get the desired result for this example, put RIGHT in the SET FOOTNOTE statement as shown above.

Display of Footnotes at End of Table

Footnotes are listed at the end of the last page of each table, unless you use the statement **FOOTNOTES EACH PAGE;** in a format request.

See also the FORMAT statement **MAXIMUM FOOTNOTE SYMBOL WIDTH** for some ways to adjust indentation and alignment of symbols and texts.

Order

The order of the footnotes in the list is determined by the footnote symbols. Footnotes with numeric symbols are printed first, followed by footnotes with alphabetic or special characters in the symbols. These are sorted according to the footnote symbols. Footnotes without symbols are printed last.

You can specify a different display order for the footnotes by using a **FOOTNOTE SEQUENCE** statement in a FORMAT request.

Indentation

If there is a footnote symbol, the footnote text for the first line of the footnote is indented four spaces. The symbol precedes the text with one space between the symbol and the text. If there is a long footnote symbol, the first line of text may be indented further. The symbol will begin at the left edge of the table, followed by one space, then the footnote text.

If there is no footnote symbol, the footnote text is not indented but begins at the left edge of the table. In this case, you can force indentation by adding blanks or an INDENT specification at the beginning of the footnote text.

For multi-line footnote texts, lines after the first are not indented.

The symbols are printed as superscripts using the footnote symbol font. You can also request that footnotes be printed in multiple columns. This feature is described under the FORMAT statement called **FOOTNOTE COLUMNS**.

Adjusting Alignment of Footnote Text

Although footnote text will usually look best if aligned according to the defaults, you may occasionally want a different alignment. To specify alignment, add the alignment specification such as RIGHT to the footnote TEXT. For example:

```
SET FOOTNOTE R_NOTE TEXT RIGHT 'Source:  Census Bureau';
```

A default footnote symbol will be generated and will print at the left. The footnote text will be right-aligned.

In general, this type of alignment will look best for a footnote with no symbol. See the FORMAT statement **SET NOTE** for the best way of adding a simple note to a table.

Footnote Symbol Level

You can raise the symbols more or less than the default amount by using the **RAISE FOOTNOTE SYMBOL** statement described in the FORMAT chapter. You can also use this statement to prevent the footnote symbols from being raised so that they will be printed at the same level as the adjacent numbers or text.

Built-in Footnotes

TPL TABLES has several built-in footnotes. All but one are automatically included in your tables when their display conditions are met, regardless of whether they are explicitly referenced in labels or masks. You can replace the default symbol and/or text for any of these footnotes with SET FOOTNOTE statements in your codebook, table request, format request, or profile (PROFILE.TPL). You can also turn them off (see section below on Deleting Footnotes). The built-in footnotes are:

Footnote name	Symbol	Text
SEE_END	""	"See footnotes at end of table."
EMPTY	"-"	"Data not available."
ERROR	"**"	"Computation error."
NO_FIT	"nf"	"Data does not fit."
SMALL	">0"	"Value is too small to display."
SMALL_NEG	"<0"	"Negative value too near zero to display."
NORANK	blank	No text.

ZERO	No symbol or text so default is "off".

SEE_END The SEE_END footnote will automatically be used in any table that is more than one page long if the table has footnotes. It will appear at the bottom of each page except the last if the page has at least one footnote or if any preceding pages have footnotes. The footnote symbol is the null string "", so no symbol will be displayed in the table. This footnote will not appear if FOOTNOTES EACH PAGE is specified.

EMPTY The EMPTY footnote symbol '**—-**' will automatically appear in any table cell for which there is no data. If there are any empty cells in a table, the footnote symbol and the text 'Data not available.' will be included in the list of footnotes at the end of the table.

Note that a dash wider than hyphen is used as the footnote symbol for EMPTY. This character is not on your keyboard but can be entered by typing **&ENDASH;** See the appendix called "Character Sets" for additional details.

ERROR The ERROR footnote symbol '******' will be displayed in any table cell for which there is no value due to a computation error such as "divide by zero". If there are any table cells with computation errors, the symbol and the text 'Computation error.' will be included in the list of footnotes at the end of the table.

NO_FIT The NO_FIT footnote symbol '**nf**' will replace the data in any table cell in which the data is too large to fit. For example, if the cell value is **2,453,987** and the column width is 7, the correct data value cannot fit is the column. If there are any cells for which the data values are too large to be displayed, the symbol and the text '**Data does not fit**.' will be included in the footnote list at the end of the table.

The NO_FIT footnote is also used when a footnote symbol is too wide to fit in a data cell. For example, if the column width is 4 and the footnote symbol is '********', the footnote symbol cannot fit. It will be replaced with 'nf' the same as if it were a wide data value.

SMALL The SMALL footnote symbol '**>0**' will appear in any table cell where a small, non-zero value would otherwise be displayed as 0. This can happen if the column is too narrow or if the mask doesn't have enough digits after the decimal place. For example, if the mask is **999.9** and the cell value is **.01**, the SMALL footnote symbol will be displayed in the

cell. Likewise, if the column width is **5** and the cell value is **.000006**, the SMALL footnote symbol will be displayed in the cell. If the SMALL footnote symbol appears in any cells of a table, its text '**Value is too small to display.**' will be included in the footnote list at the end of the table.

SMALL_NEG The SMALL_NEG footnote symbol **'<0'** will appear in any table cell where a small, non-zero negative value would otherwise be displayed as **0**. This can happen if the column is too narrow or if the mask doesn't have enough digits after the decimal place. For example, if the mask is **999.9** and the cell value is **-.01**, the SMALL_NEG footnote symbol will be displayed in the cell. Likewise, if the column width is **5** and the cell value is **-.000002**, the SMALL_NEG footnote symbol will be displayed in the cell. If the SMALL_NEG footnote symbol appears in any cells of a table, its text 'Negative value too near zero to display.' will be included in the footnote list at the end of the table.

ZERO The ZERO footnote is the only built-in footnote that is "off" by default. It can be used to call attention to table cells for which the value is exactly zero. You can turn on the ZERO footnote by providing a footnote text with a SET FOOTNOTE statement. If you do not specify a footnote symbol, TPL TABLES will provide a number for the symbol and display it in parentheses. For example,

```
SET FOOTNOTE ZERO TEXT 'Value is exactly 0.';
```

NORANK If you have a table with ranking and a RANK DISPLAY column, you may have rows in the table for which there is no rank number to put in that column. This will happen if not all rows of the table are ranked. By default, the RANK DISPLAY column will be blank for these rows. Use the NORANK footnote to put something other than blank in these rows. You can also specify a footnote text so that the NORANK footnote will appear at the bottom of the table. For example,

```
SET FOOTNOTE NORANK SYMBOL "..."
        TEXT "Rank number not applicable.";
```

Font for Built-in Footnote Symbols

The FOOTNOTE SYMBOL FONT *is not used* for built-in footnote symbols. When a symbol for a built-in footnote is displayed in a data cell, it is printed in the font that is in effect for the cell. This is the font that would be used if data were displayed in the cell. The symbol is not raised. At the end of the table, the footnote symbol is also not raised. It takes on the font that is in effect at the beginning of the footnote text. You can change

the font for the symbol at the end of the table but not in the table cells. To change the symbol font at the end of the table, use a SET FOOTNOTE statement such as:

```
SET FOOTNOTE EMPTY SYMBOL FONT HB 6;
```

Forcing Automatic Numbering for Built-in Footnotes

As discussed in the previous section, the built-in footnotes have built-in footnote symbols. For example, the built-in footnote called EMPTY has a symbol of "–"; the built-in footnote called SMALL has a symbol of '**>0**'. You can get the regular default automatic footnote numbering for a built-in footnote by using a SET FOOTNOTE statement with the word DEFAULT to describe the footnote symbol. The statement

```
SET FOOTNOTE  SMALL  SYMBOL DEFAULT;
```

will replace the built-in symbol for the SMALL footnote with an automatically generated footnote number. Use of DEFAULT also causes the footnote symbol to be raised and appear in the footnote symbol font like other ordinary footnote symbols.

The word DEFAULT can be used for footnotes that are not built-in but will not generally be needed. If no footnote symbol is specified for these footnotes, automatic numbering is assumed.

Conflicts with Other Footnotes in Table Cells

Only one footnote is allowed per table cell. If a user-defined footnote conflicts with a built-in footnote, the following rules apply: for all of the built-in footnotes, except NO_FIT, if the mask applied to a table cell contains a footnote reference and no 9's, the footnote from the mask will override built-in footnotes. If the mask does contain 9's, a footnote in the mask will not override built-in footnotes.

Deleting Footnotes

Using Null Strings

If a footnote has both a null symbol and a null text, the footnote will not be used in any tables. For example,

```
SET FOOTNOTE SMALL    SYMBOL ""     TEXT "";
```

This statement will cause the built-in SMALL footnote to disappear from all tables. The table cells that would have contained the symbol '**>0**' will instead contain 0's and no SMALL footnote symbol or text will be included in the footnote lists.

Using FORMAT Statements

To delete footnotes on a table-by-table basis, you must use the DELETE FOOTNOTE statement in a format request. For example, the following FORMAT statement will delete the footnote named NOTE1 from TABLE A1 but leave it in effect for any other tables in the table request.

```
FOR TABLE A1:  DELETE FOOTNOTE NOTE1;
```

If you wish to delete all footnotes, including the built-in ones, you can use the word **ALL**. Some examples are:

```
FOR TABLE A1:  DELETE FOOTNOTES ALL;
FOR TABLES ALL:  DELETE FOOTNOTES ALL;
```

As with other DELETE actions in the FORMAT language, there are comparable RETAIN clauses. For example, if you wanted to turn off all footnotes, but retain the footnote called NOTE1 for a particular table, you could say:

```
FOR TABLES ALL:  DELETE FOOTNOTES ALL;
FOR TABLE A1:  RETAIN FOOTNOTE NOTE1;
```

Forcing Printing of Unused Footnotes with KEEP

In some cases, you may want the text for a footnote to appear at the end of a table, regardless of whether the footnote is referenced or used in the table. This is the usual case for a footnote that is just a note -- an informative piece of text that applies to the entire table and does not need to be attached to any particular label or table cell. The RETAIN FOOTNOTE statement, used to reverse the effect of DELETE FOOTNOTE, cannot be used to force printing of footnotes that are not used in the table.

A simple way to accomplish this result is to use the SET NOTE statement.

You can also use the FORMAT statement KEEP FOOTNOTE. An example is:

```
SET FOOTNOTE ROUND_NOTE TEXT
        'Note: Components may not add to totals, due to rounding.'
```

```
FOR TABLE B3:  KEEP FOOTNOTE ROUND_NOTE;
```

The text for the footnote called ROUND_NOTE will be printed at the end of table B3, even if the footnote is not used anywhere in the table. Since no footnote symbol is specified and the footnote is not used in the table, no footnote symbol is generated for it. It is printed as a simple note.

Note that the FORMAT statements DELETE FOOTNOTE, RETAIN FOOTNOTE and KEEP FOOTNOTE can apply to the same footnote. If they do, and there is a conflict between them, the last one encountered by the system will win. Also, note that if both the symbol and the text for a footnote are null strings (''), the footnote cannot be kept in the table by either RETAIN FOOTNOTE or KEEP FOOTNOTE.

Example of Table with Footnotes

The following example illustrates the use of many of the footnoting capabilities we have described. In the table request, the footnotes called MULTI, MANDATE, ASSOC, and PLAN are referenced in DEFINE statement labels and in the table title. Since no symbols are specified in the SET FOOTNOTE statements, the default symbols 1 through 4 are assigned by TPL TABLES.

```
use benefit codebook;

define plan_type indent 2 'All participants'/ on plan;
        indent 4 'Total' if 1;
        'Single employer' if 2;
        'Multiemployer' footnote multi if 3;
        'Mandated benefits' footnote mandate if 4;
        'Employer association' footnote assoc if 5;

table one 'Table 86.  Plan' footnote plan
        ' administration:  Percent of full-time participants in selected'
        ' benefits by type of plan sponsor, medium and large firms' :
            heading   health then life then other_ins,
            stub      plan_type;

set footnote plan text  'Does not include supplemental plans.';
```

```
set footnote multi text
        'Individual employers in the same or in a related industry'
        ' contributing a negotiated amount to trust fund providing'
        ' benefits for employees covered under a collective'
        ' bargaining agreement.';

set footnote mandate text
        'The majority of the participants with mandated sickness'
        ' and accident insurance benefits were covered by State'
        ' temporary disability plans.';

set footnote assoc text
        'Band of small employers in a common trade or business,'
        ' for example, savings and loan associations.  The plan'
        ' sponsored by the association is not negotiated with the'
        ' employees.';
```

Three additional footnotes are set in the format request. First, the built-in footnote called SMALL is "reset". The symbol is set to DEFAULT so that a number will be assigned in place of the built-in symbol '**>0**', and the text is set to '**Less than 0.5 percent**' to replace the built-in text '**Value is too small to display**.' Second, the built-in footnote EMPTY is "reset" with a null string so that the built-in text '**Data not available.**" will not print at the end of the table.

The last footnote NOTE_FOOT is a note that will print at the end of the table. It requires a KEEP FOOTNOTE statement, because it is not referenced in the table. Since it is not used in a label or table cell, it is printed as a simple note without a symbol.

```
set footnote small symbol default text 'Less than 0.5 percent.';

set footnote empty text '';

set footnote note_foot text
        indent 2 'NOTE:  Because of rounding, sums of '
        indent 0 'individual items may not equal totals.  '
        'Dash indicates no employees in this category.'/;

keep footnote note_foot;
```

The resulting table is:

Table 86. Plan[1] administration: Percent of full-time participants in selected benefits by type of plan sponsor, medium and large firms

Plan sponsor	Health insurance	Life insurance	Sickness and accident insurance
All participants			
Total	100	100	100
Single employer	96	97	87
Multiemployer[2]	4	3	2
Mandated benefits[3]	–	–	11
Employer association[4]	([5])	([5])	–

[1] Does not include supplemental plans.

[2] Individual employers in the same or in a related industry contributing a negotiated amount to trust fund providing benefits for employees covered under a collective bargaining agreement.

[3] The majority of the participants with mandated sickness and accident insurance benefits were covered by State temporary disability plans.

[4] Band of small employers in a common trade or business, for example, savings and loan associations. The plan sponsored by the association is not negotiated with the employees.

[5] Less than 0.5 percent.

NOTE: Because of rounding, sums of individual items may not equal totals. Dash indicates no employees in this category.

If the table is exported as a text table we get the following. Note the difference in the treatment of footnote symbols.

Table 86. Plan(1) administration: Percent of full-time participants in selected benefits by type of plan sponsor, medium and large firms

Plan sponsor	Health insurance	Life insurance	Sickness and accident insurance
All participants			
Total	100	100	100
Single employer	96	97	87
Multiemployer(2) ...	4	3	2
Mandated benefits(3)	-	-	11
Employer association(4) ..	(5)	(5)	-

1 Does not include supplemental plans.
2 Individual employers in the same or in a related industry contributing a negotiated amount to trust fund providing benefits for employees covered under a collective bargaining agreement.
3 The majority of the participants with mandated sickness and accident insurance benefits were covered by State temporary disability plans.
4 Band of small employers in a common trade or business, for example, savings and loan associations. The plan sponsored by the association is not negotiated with the employees.
5 Less than 0.5 percent.
NOTE: Because of rounding, sums of individual items may not equal totals. Dash indicates no employees in this category.

The SET NOTE Statement

The SET NOTE statement can be used in the codebook, table request or profile (PROFILE.TPL) in addition to the format request. See also the SET NOTE statement in the FORMAT section of the manual for additional details on its use in a format request.

SET NOTE lets you specify note text that can be printed at the end of a table. When a note is printed, it looks like a footnote without a symbol. You do not need to reference the note anywhere or use a KEEP FOOTNOTE statement to force it to be printed as you do with footnotes that do not have symbols. A note will always print at the end of any table(s) for

which it is set. Unless set in the format request, a note will apply to all tables.

Format

```
SET NOTE  (name)  TEXT  label ;
```

where **name** is the name of the note. The parentheses around the note name are optional. The note TEXT can be any valid TPL TABLES label except that it cannot contain a reference to a footnote. The word TEXT is optional. Thus the most simple format for the SET NOTE statement is:

```
SET NOTE  name  label ;
```

Example

```
SET NOTE QQQ 'This table was prepared by QQQ Software, Inc.';
SET NOTE CD 'Confidential Data.';
```

The notes called QQQ and CD will print at the end of each table.

Font Controls in Footnotes

The default fonts for footnotes can be set using the FOOTNOTE TEXT FONT and FOOTNOTE SYMBOL FONT statements in the FORMAT language. This method works well if you want to use the same fonts for all footnotes. If you need to select fonts for individual footnotes, you can enter font choices directly in footnote text and symbols. These font selections are ignored for text table exports.

Rules for inserting font choices in footnote texts are the same as for any other label. For a complete list of available fonts, see the **FONT** statement in the **FORMAT** section of the manual.

To change the font for a footnote symbol, simply add the FONT specification to the SYMBOL part of the SET FOOTNOTE statement. You can use any of the font choices available in the FORMAT language FONT statement. For example,

```
SET FOOTNOTE (A)  SYMBOL FONT HB 6 '*'  TEXT 'Text';
```

To change the font of a default symbol, specify only the font. For example,

```
SET FOOTNOTE (B)  SYMBOL FONT HB 6  TEXT 'Text';
```

If there is a symbol string, the font specification can be on either side of the symbol string. Only one font can apply to a footnote symbol.

The font size specification is optional. If size is not specified, the size will be determined by the default font size for footnote symbols.

If the footnote symbol is attached to a label and is larger than the font sizes in the label, the vertical spacing will be adjusted to allow for the larger footnote symbol. If a large symbol is used in a data cell, the vertical spacing will not be adjusted for the footnote symbol. In this case, it is possible that the large footnote symbol will overlap with the line above.

Matching the Footnote Symbol Font to the Adjacent Font

You can use the word **MATCH** as a font specification for footnote symbols when you want the footnote symbol font to match that of the adjacent text or data value. As with other font specifications, MATCH has no effect on text tables. To use MATCH as the default specification, use the FORMAT statement:

```
FOOTNOTE SYMBOL FONT = MATCH;
```

MATCH can also be specified for individual footnote symbols in the SET FOOTNOTE statement. For example:

```
SET FOOTNOTE PRELIM TEXT 'Preliminary data'
        SYMBOL 'P' FONT = MATCH;
```

When the footnote symbol is used in a data cell, it matches the font in effect for that cell. At the bottom of the page, it matches the font at the beginning of the footnote text. When used in any other label, it matches the font that is active at the point where the footnote is referenced in the label.

Quick Reference Summary of Font Treatment for Symbols

In the following, we use the term "cell font". Cell font is defined as the font that would be used if data were to be displayed in the cell. Initially, it is the DEFAULT FONT.

Non-built-in Footnotes

The symbol font is used for symbols in labels, in cells and at the end of the table.

If a symbol font is entered in SET FOOTNOTE, this font becomes the symbol font for the footnote.

If a symbol is alone in a cell, it is enclosed in parentheses; the parentheses are in the cell font.

Built-in Footnotes

In a data cell, the cell font is always used for the symbol.

At end of a table, the font in effect at the beginning of the footnote text is used for the symbol.

If a symbol font is entered in SET FOOTNOTE, this font is used at the end of the table but the cell font is still used in cells.

MATCH

Match works the same for built-ins and all other footnotes.

MATCH can be set by FOOTNOTE SYMBOL FONT = MATCH; or in individual SET FOOTNOTE statements.

In a label, the font for the symbol matches the font active at the point where the footnote is referenced.

In a data cell, the font for the symbol matches the cell font.

At the bottom of the page, the font for the symbol matches the font in effect at the beginning of the footnote text.

Using Footnotes in TEXT Masks

When a footnote is used in a TEXT mask, the format for the symbol in the data cells is determined by the TEXT mask. See the "Mask" chapter for full details on TEXT masks.

A footnote symbol displayed with a TEXT mask is not raised. Instead, it is displayed at the normal text height unless you precede it with SUP or SUPER to get superscript display. In addition, for data cells containing only a footnote symbol, the symbol is not enclosed by parentheses. If you want parentheses, you need to include them in the TEXT mask.

Example

```
SET FOOTNOTE t_note SYMBOL '***'  TEXT '$100,000 and above.';
POST COMPUTE avg_age 'Age' =
        age / persons MASK 99                    IF  > 17;
        MASK TEXT 'Young' FOOTNOTE t_note        IF  OTHER;
```

Using SYM in Footnote Text for More Control of Symbol Format

In some cases, you may wish to change the format or placement of the symbol at the end of a table where it appears alongside the footnote text but without altering it in the body of the table where it appears in labels or data cells. For example, suppose you want the footnote symbol to be red in the body of the table, but you want it to be green at the end of the table. Or, you want to indent the footnote symbols and text in a certain way at the end of the table.

You can use the special word SYM in footnote text to provide this type of functionality. Footnote text is a normal label with all of the label features such as color specifications, fonts and indentations supported by other labels. If you insert SYM into the footnote TEXT, the footnote symbol will be displayed at that point in the text and will take on the label characteristics that apply at that point in the text. The footnote symbol will retain its original SYMBOL characteristics wherever it is displayed in the body of the table.

Note that references to fonts, color and superscripting in the following examples do not apply to text tables. Label features such as RIGHT IN SPACE are described in detail in the "Labels" chapter.

Example

```
SET FOOTNOTE sample  SYMBOL COLOR  red "x"
        TEXT  COLOR green  FONT hb 10
        SYM " Sample footnote text";
```

In the body of the table, the footnote symbol **x** will appear in red with the default footnote font and will be raised (as a superscript) in the standard way. At the bottom of the table, **x** will be green in Helvetica Bold (hb) font, size 10.

Note that when SYM is used, the footnote symbol at the bottom of the table will, by default, have the footnote text font rather than the footnote symbol font and will not be raised. This is because the symbol takes on the text characteristics in effect. To reproduce the raised format for the symbol within the text, where the footnote text font is h 10 and the footnote symbol font is h 8, we need to specify SUPER in front of SYM to raise the symbol, return to NORMAL level following SYM, and add font specifications as follows:

```
SET FOOTNOTE sample  SYMBOL COLOR  red "x"
```

```
        TEXT  COLOR green  FONT h 8
        SUPER SYM NORMAL  FONT h 10 " Sample footnote text";
```

Since there is one blank at the beginning of " Sample footnote text", there will be one blank space between the symbol and the following footnote text.

When SYM is not used, footnote symbols are right-aligned within a default space of 3 characters. To reproduce this spacing between the footnote symbol and text, we can add RIGHT IN SPACE 3 in front of SYM, remove the blank at the beginning of the text, and then add SPACE TO 4 in front of the text.

```
SET FOOTNOTE sample  SYMBOL COLOR  red "x"
        TEXT  COLOR green
        RIGHT IN SPACE 3  FONT h 8  SUPER SYM NORMAL
        SPACE TO 4  FONT h 10 "Sample footnote text";
```

Using SYM with RIGHT IN SPACE to Align Footnote Symbols and Notes

For footnotes, the alignment at the bottom of a table is determined according to built-in defaults. If you have footnote symbols of different widths, they will be aligned independently, relative to their footnote text. If you wish, you can right-align the symbols within a space of a specific width using a combination of RIGHT IN SPACE, SPACE TO and SYM. See also the "Labels" chapter for additional details on RIGHT IN SPACE and SPACE TO.

To illustrate, we use the following collection of footnotes and notes:

```
set note src 'Source:  Department of Health.';

set note nt
        'Note:  This table shows footnotes with default alignment.';

set footnote sum symbol font h 's'
        text 'The total is less than the sum of the individual items '
        'because many workers participate in plans with more '
        'than one feature.';

set footnote df  text 'Data collected for the month of March.';
```

```
set footnote dental
        text 'Participants who elected dental coverage only
        'were not included in this tabulation.';
```

By default, the footnotes and notes will be displayed as follows:

Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms, 1995[1]

Cost containment feature	All participants	Technical and clerical participants	Production participants
Total[2,s]	100	100	100
Incentive to seek second surgical opinion	35	40	28
Higher payment for generic prescription drugs	7	7	6
Separate deductible for hospital admission	9	9	7
Urging prehospitalization testing	47	52	43
Preadmission certification requirement	16	15	16
Incentive to audit hospital statement	2	2	1

Source: Department of Health.
Note: This table shows footnotes with default alignment.
[s] The total is less than the sum of the individual items because many workers participate in plans with more than one feature.
[1] Data collected for the month of March.
[2] Participants who elected dental coverage only were not included in this tabulation.

As we can see, there is no indentation for notes. For the footnotes with symbols, the symbols are indented within a space of 3, and the footnote texts follow after a space of 1. To align "Source:", "Note:", and the footnote symbols within a space of 7, then start the remaining text at position 10, we can use a combination of RIGHT IN SPACE 7, SYM and SPACE TO 10:

```
set note src  text right in space 7 'Source:'
space to 10 'Department of Health.';

set note nt  text right in space 7 'Note:'
space to 10 'This table shows footnote symbols "right in space".';

set footnote sum symbol 's'  text right in space 7  super sym normal
space to 10 'The total is less than the sum of the individual '
        'items because many'/
space to 10 'workers participate in plans with more than one feature.';
```

```
set footnote df  text right in space 7 super sym normal
space to 10 'Data collected for the month of March.';

set footnote dental  text right in space 7  super sym normal
space to 10;'Participants who elected dental coverage only '
'were not included in'/
space to 10 'this tabulation.';
```

Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms, 1995[1]

Cost containment feature	All participants	Technical and clerical participants	Production participants
Total[2,s]	100	100	100
Incentive to seek second surgical opinion	35	40	28
Higher payment for generic prescription drugs	7	7	6
Separate deductible for hospital admission	9	9	7
Urging prehospitalization testing	47	52	43
Preadmission certification requirement	16	15	16
Incentive to audit hospital statement	2	2	1

Source: Department of Health.
Note: This table shows footnote symbols "right in space".
s The total is less than the sum of the individual items because many workers participate in plans with more than one feature.
1 Data collected for the month of March.
2 Participants who elected dental coverage only were not included in this tabulation.

The footnote symbols, **s**, **1** and **2** are all only a single character. The example would work equally well for any footnote symbol that fits in the space of width 7 specified by RIGHT IN SPACE 7.

CHAPTER 23

Automatic Formatting

DEFAULT FORMAT FOR TABLES

This chapter describes the default rules used when TPL TABLES formats a table automatically. You can easily change most of these rules by using FORMAT statements either in your profile or in a format request.

Page Format

Page size is initially set during installation of TPL TABLES. Within a given page size, a table is printed with margins. The default margin sizes are:

```
top margin = 1 inch;
bottom margin = 1 inch;
left margin = 0.5 inches;
right margin = 0.5 inches;
```

The table is centered within the available space. If the table is too wide for the space, it is divided into as many partitions as necessary with each partition on a new page. These partitions are called banks. The stub labels are repeated in each bank, and each bank is centered.

Pages are not numbered. If you wish to have page numbers, date, time and/or job identification printed on the pages, see the FORMAT statement called PAGE MARKER for details.

Table Title Format

The table title from the TABLE statement appears at the top of the page and is aligned with the left edge of the table. If no title is provided in the TABLE statement, the table name is used as the title. For example:

```
TABLE MANUFACTURING_INDUSTRIES: STATE, UNION_STATUS;
```

would cause the table name MANUFACTURING_INDUSTRIES to be used as the table title.

To center the title, follow the required table name with a title enclosed in quotes and the keyword CENTER. For example:

```
TABLE A CENTER 'Manufacturing Industries' : STATE,
UNION_STATUS;
```

Similarly, use of the word RIGHT with a table title will cause the title to be aligned with the right edge of the table.

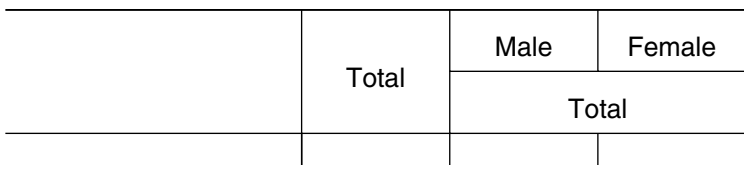
Heading and Column Format

The default column width is about 10 characters including the space taken up by the column divider. Heading labels are centered and segmented over multiple lines automatically when necessary. Each segmented portion is centered. For a text table, if a label segment cannot be perfectly centered because of an uneven number of spaces, it is adjusted to the right one position. For example, if there are 9 spaces available for a 6 character heading label, it will be displayed with 2 spaces to the left and 1 space to the right.

Coalescing of Labels

If two or more adjacent boxes at the same heading level have the same label, the boxes will be automatically merged to a single larger box containing one appearance of the label.

For example, in the printed heading for **TOTAL THEN SEX BY TOTAL**, the boxes for the two lowest level TOTAL columns would be merged:



This coalescing of labels will occur even if the labels are from different variables. If you do not want the label boxes to merge, then you need to make the labels different. The easiest way to accomplish this without affecting the appearance of the individual labels is to put a conditional hyphen at the front of one of the labels. For example,

```
"This is a label"
- "This is a label"
```

will not be merged if they are in adjacent boxes.

See also the Format statement REPLACE LABEL for more information.

Stub and Row Format

The default stub width is about 20 character positions. Stub label indentation for each level of nest is two character positions. If there are enough levels of nest in the stub to cause the indentation to go beyond the middle of the stub, indentation will stop.

A stub label too long for the available space will be automatically segmented over two or more lines. If a stub label must be segmented because it is too long for the stub width, the continued segments will be indented three character positions from the first line segment. As with the indentation to show nesting, no segment will start beyond the middle of the stub.

The following example shows a nesting indentation of two positions and a continuation indentation of three positions:

```
STUB LABEL NUMBER ONE
 STUB LABEL NUMBER
    TWO IS VERY LONG
    EXTENDING OVER
    FOUR LINES
  STUB LABEL NUMBER THREE
```

If a label used in the stub has the word **CENTER** associated with it, the label will be centered within the width of the stub. Similarly, if the label has the word **RIGHT** associated with it, it will be aligned with the right edge of the stub.

Normally a label will appear for each variable used in the stub expression. The one exception is when a null-labeled variable is nested with a variable at the next highest level. For example the stub expression, **CITY BY (TOTAL THEN SEX)** will begin as:

```
BOSTON
  TOTAL..... (total values)
  MALE..... (data values)
  FEMALE..... (data values)
```

Suppose that you want the **"BOSTON"** label to move down one line and replace the **"TOTAL"** label. In other words, you want the city label to be the label for the total line. If the variable producing the total line is created with a null label, **BOSTON** and other city names will collapse down to the total line. For example:

```
BOSTON..... (total values)
  MALE..... (data values)
  FEMALE..... (data values)
```

One way to create a null label is to use the DEFINE statement as follows:

```
DEFINE TOTALS  ON  SEX;
        "     IF  ALL;
```

Then replace **TOTAL** with the DEFINE variable **TOTALS** in the stub expression.

A data row for which there is no data is called an empty row. Empty data rows are not printed.

Wafer Label Format

Wafer labels appear in the upper left corner of each page, two lines below the table title. Wafer labels appear only if there is a wafer expression in the TABLE statement. Empty wafers are not printed. The wafer label format depends on the content of the wafer expression. In the next examples, assume a single variable from the codebook is used in the wafer expression.

Codebook variable	First wafer label will print as:
INCOME OBS 5	INCOME

```
REGION CON                              SOUTH
  (
      SOUTH  =  1
      NORTH  =  2
      EAST  =  3
      WEST  =  4
  )


REGION 'Regions of U.S.A.' CON 1        South
  (
      'South'  =  1
      'North'  =  2
      'East'  =  3
      'West'  =  4
  )


REGION CON 1 (1:4)                      1 REGION
```

This last example illustrates the lack of clarity which can result in a wafer label when condition names or condition labels are not defined for control variable values. However, for certain types of control variables, wafer labels resulting from this format are satisfactory. Assume two control variables representing the size of dwelling (**ROOMS**) and age of household head (**YEARS**) are described in a codebook as follows.

```
ROOMS CON 1
  (
       '1 ROOM'  =  1
       2:9
  )

YEARS  CON 2  (18:99)
```

If **ROOMS** is used as the wafer expression, then the first two wafer labels will be:

```
1 ROOM
2 ROOMS
```

If a wafer expression contains nested control variables, such as **REGION BY ROOMS**, the wafer labels will appear as a succession of control variable condition labels separated by commas, as in:

```
SOUTH, 1 ROOM
```

For a wafer expression such as **YEARS BY INCOME BY ROOMS**, where there is a nested observation variable, the first wafer will have the label:

INCOME FOR (18 YEARS, 1 ROOM)

Data Cell Format

Table cell values that do not have masks are rounded to the nearest whole integer and displayed with no special symbols except commas. The values are right-adjusted in the table columns. If you want a different format or alignment for values, you can specify the format using a print mask.

For any cell that does not have data, a dash is printed in the center of the table cell.

Cell data consisting of too many characters for the column width are handled in a special way. Each cell value that is too long to fit within the column width will be reduced, first by removing from the number any special characters (**, $ %** and footnote symbols), then by truncating decimal places if there are any. A warning message will alert you to the fact that some items have been removed.

If the value is still too long for the entire data value to be displayed, it will be replaced by the footnote symbol **nf**. The **nf** footnote, **Data does not fit.**, will appear at the bottom of the table.

CHAPTER 24

Color and Grey

USING COLOR, COLOR SHADING AND GREY SHADING IN TABLES

General Information on Color and Grey

You can specify color for individual labels, including titles and footnote texts, and masks within codebooks, table requests and FORMAT requests. Color defaults for data values, labels and rules (horizontal and vertical lines) can be specified with FORMAT statements. In addition, you can request color or grey shading for an entire table or for selected parts of a table.

The word **COLOR** can also be spelled **COLOUR**.

COLOR and GREY specifications are ignored in text tables.

Important This chapter is best viewed in the pdf form since colors are not displayed in the paper manual.

Effect on Monochrome Printers

If tables that use color are printed on a non-color printer, the colors will print as shades of grey. See the FORMAT statement COLOR = NO for tips on switching between color and monochrome. See also the special

color GREY. Its best use is for grey shading on either monochrome or color printers.

r g b colors

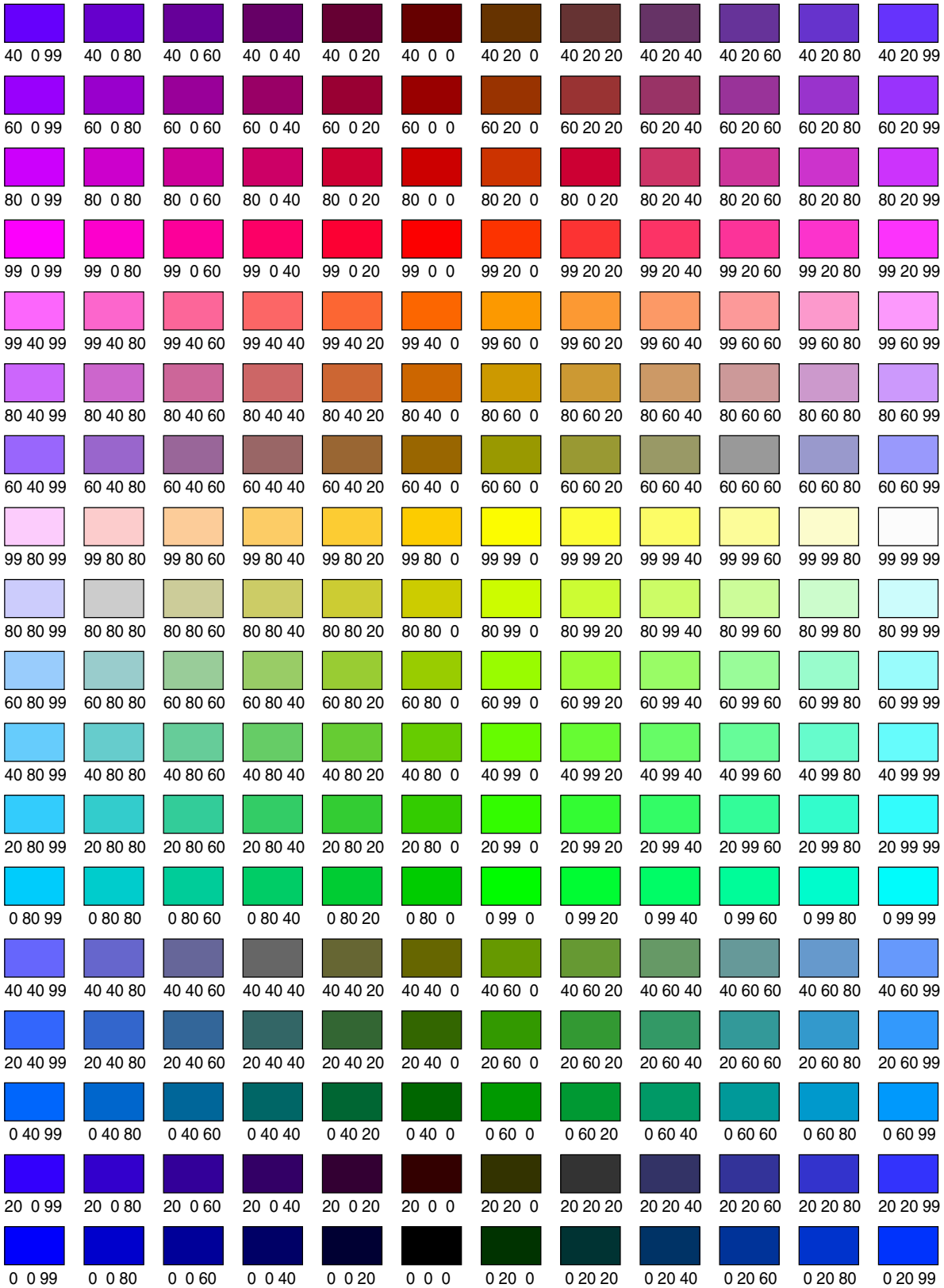
Colors are specified by a combination of red, green and blue. We will refer to them as r g b where the amount of each color in the mix is indicated by a value from 0 to 100. For example, the color blue has the r g b numbers 0 0 100. In other words, there is no red, no green, and the maximum amount of blue.

Color Chart

A color chart file called **colors.ps** is included with the TPL Tables system. It can be found in either the **doc** subdirectory or the **examples** subdirectory of the TPL system directory, depending on the version of TPL TABLES you have. If you have a color PostScript printer, you can print this chart on your color printer to see what colors are printed for a variety of r g b colors.

The color chart is also shown on the next page of this manual. With the PDF version of the manual open in Adobe Acrobat Reader, you can print the page on any color printer.

There is very little consistency between color printers, so the same r g b color printed on one color printer may look quite different when printed on another. With the color chart, you can choose precisely from the colors as printed by your printer.


40 0 99 40 0 80 40 0 60 40 0 40 40 0 20 40 0 0 40 20 0 40 20 20 40 20 40 40 20 60 40 20 80 40 20 99
60 0 99 60 0 80 60 0 60 60 0 40 60 0 20 60 0 0 60 20 0 60 20 20 60 20 40 60 20 60 60 20 80 60 20 99
80 0 99 80 0 80 80 0 60 80 0 40 80 0 20 80 0 0 80 20 0 80 0 20 80 20 40 80 20 60 80 20 80 80 20 99
99 0 99 99 0 80 99 0 60 99 0 40 99 0 20 99 0 0 99 20 0 99 20 20 99 20 40 99 20 60 99 20 80 99 20 99
99 40 99 99 40 80 99 40 60 99 40 40 99 40 20 99 40 0 99 60 0 99 60 20 99 60 40 99 60 60 99 60 80 99 60 99
80 40 99 80 40 80 80 40 60 80 40 40 80 40 20 80 40 0 80 60 0 80 60 20 80 60 40 80 60 60 80 60 80 80 60 99
60 40 99 60 40 80 60 40 60 60 40 40 60 40 20 60 40 0 60 60 0 60 60 20 60 60 40 60 60 60 60 60 80 60 60 99
99 80 99 99 80 80 99 80 60 99 80 40 99 80 20 99 80 0 99 99 0 99 99 20 99 99 40 99 99 60 99 99 80 99 99 99
80 80 99 80 80 80 80 80 60 80 80 40 80 80 20 80 80 0 80 99 0 80 99 20 80 99 40 80 99 60 80 99 80 80 99 99
60 80 99 60 80 80 60 80 60 60 80 40 60 80 20 60 80 0 60 99 0 60 99 20 60 99 40 60 99 60 60 99 80 60 99 99
40 80 99 40 80 80 40 80 60 40 80 40 40 80 20 40 80 0 40 99 0 40 99 20 40 99 40 40 99 60 40 99 80 40 99 99
20 80 99 20 80 80 20 80 60 20 80 40 20 80 20 20 80 0 20 99 0 20 99 20 20 99 40 20 99 60 20 99 80 20 99 99
0 80 99 0 80 80 0 80 60 0 80 40 0 80 20 0 80 0 0 99 0 0 99 20 0 99 40 0 99 60 0 99 80 0 99 99
40 40 99 40 40 80 40 40 60 40 40 40 40 40 20 40 40 0 40 60 0 40 60 20 40 60 40 40 60 60 40 60 80 40 60 99
20 40 99 20 40 80 20 40 60 20 40 40 20 40 20 20 40 0 20 60 0 20 60 20 20 60 40 20 60 60 20 60 80 20 60 99
0 40 99 0 40 80 0 40 60 0 40 40 0 40 20 0 40 0 0 60 0 0 60 20 0 60 40 0 60 60 0 60 80 0 60 99
20 0 99 20 0 80 20 0 60 20 0 40 20 0 20 20 0 0 20 20 0 20 20 20 20 20 40 20 20 60 20 20 80 20 20 99
0 0 99 0 0 80 0 0 60 0 0 40 0 0 20 0 0 0 0 20 0 0 20 20 0 20 40 0 20 60 0 20 80 0 20 99

Color Definitions in color.tpl

Colors can also be referenced by name where the colors have been defined in a file called color.tpl. Establishing color definitions in this way can be very convenient if you have a set of standard colors that you use frequently, because you do not need to remember the r g b values for the colors. Instead, you can reference the colors by name. This approach also allows you to choose different sets of standard color definitions for different printers and adjust your color output to the different printers simply by using a different color.tpl file.

The TPL TABLES installation process creates a file called color.tpl and puts it in the TPL TABLES system directory. Several examples of color definitions are included in this file. You can edit it to change or add to the color definitions. If you want to leave the system color file unchanged but use a different set of color definitions for your own jobs, you can make a copy of color.tpl in the subdirectory where you are working and edit it to include your own set of color definitions. The color definitions in the directory where you are working will override the ones in the color.tpl file in the TPL TABLES system directory.

The format of a color definition in color.tpl is:

Format

```
color r g b
```

where color is a name that you choose to associate with a specific color definition and r, g and b are numbers between 0 and 100 (inclusive) which specify the **r**ed, **g**reen, and **b**lue components of a color.

Note Color definitions entered in color.tpl DO NOT end with a semicolon (;).

Note If you enter a color definition in color.tpl with the color name GRAY or GREY, it will be ignored. These names are reserved for grey characters and shading.

Example Following is an example of a color.tpl file:

```
red            100    0    0
green            0  100    0
blue             0    0  100
brown           60   40    0
cyan             0  100  100
yellow         100  100    0
light_yellow   100  100   20
purple          40    0  100
magenta        100    0  100
orange          90   60    0
black            0    0    0
```

Effect The colors red, green, blue, brown, cyan, yellow, light_yellow, purple, magenta, orange and black are defined in the color.tpl file and can be referenced by name in any TPL TABLES color specifications.

Example To choose the color RED as the default for all characters and rules in a table, you can use the FORMAT statement:

```
DEFAULT COLOR = RED;
```

This statement has the same meaning as the statement:

```
DEFAULT COLOR = 100 0 0;
```

Example To shade the table heading with the color LIGHT_YELLOW, you can use the FORMAT statement:

```
SHADE HEADING LIGHT_YELLOW;
```

This statement has the same meaning as the statement:

```
SHADE HEADING 100 100 20;
```

Average Household Income by Sex of Householder and Region.

Regional Totals	Sex of Householder		
	Total	Male	Female
	Average Family Income		
Total U.S.	$31,830	$36,884	$20,460
New England	35,776	41,825	23,694
Mid Atlantic	33,983	40,217	22,124
East North Central	32,044	37,784	19,174
West North Central	27,947	31,966	16,965
South Atlantic	31,480	36,817	20,673
East South Central	25,151	29,886	14,441
West South Central	28,264	32,728	16,975
Mountain	29,391	33,157	19,281
Pacific	36,429	41,053	24,871

Note on Changing Color Definitions in color.tpl

If you include color names in individual labels, including titles and footnote texts, or masks within codebooks or table requests, you should be aware that these colors are "built into" the labels and masks when the codebooks or table requests are first run. TPL TABLES converts the color names to the literal r g b numbering and saves this numbering as part of the labels or masks as they are processed. Thus, if a codebook is processed with one set of color name specifications and then the color.tpl file is changed, the old color specifications will continue to apply to the labels in that codebook until the codebook is reprocessed. Similarly, changing color.tpl before a TPL TABLES rerun will be effective for FORMAT statement colors but not for table request colors.

Recommendation

If you reference colors by name in a codebook, then change the color.tpl file, you will probably want to reprocess the codebook to switch to the new color definitions.

If you run a table request in which colors are referenced by name, then change color.tpl, you will probably want to run the job over from the beginning to get the new color definitions. Doing a rerun with FORMAT statements will not change the original label or mask colors that were assigned in the table request.

Printing Color Separations for Tables

Color separations cannot be printed directly from TPL TABLES, but you can print them easily using desktop publishing software. First, ask TPL TABLES to convert your tables to Encapsulated PostScript (EPS) files. The tables will be converted into EPS files with one table page per file. You can then bring the resulting EPS table pages into documents created with desktop publishing software. If the tables have color, the EPS files will automatically include the information needed for the desktop publishing software to print CMYK color separations. You do not need to do anything special to make this happen.

Example In PageMaker, after bringing an EPS table page into the document, choose "Print", then "Color", then click on "Separations".

The Special Color GREY

You can specify GREY in any situation where COLOR is allowed. GREY prints equally well on both color and non-color printers. It can be particularly useful for shading if you have a non-color printer. It is less useful for labels or data values, since letters and numbers do not print very well in a grey shade. GREY is specified with a number between 0 (white) and 100 (black).

GREY can also be spelled **GRAY**.

The following examples show some uses of GREY in FORMAT statements.

Examples

```
REPLACE LABEL WITH GREY 30 'A grey label';
SHADE HEADNOTE GREY 5;
```

Since GREY is a special built-in color, the color names GREY and GRAY can only be used with a number that specifies the degree of shading. If there is a definition of GREY or GRAY in the color.tpl file, the definition will be ignored.

Color Specifications for Individual Labels and Masks

Color can be added to labels, including titles and footnote texts, and masks within codebooks, table requests and FORMAT requests. Color specified in individual labels and masks overrides colors specified in the FORMAT statements, DEFAULT COLOR and LABEL COLOR.

The color applies to the printed characters. To shade the background for part of a table, you must use one of the FORMAT statements that begin with the word SHADE.

Labels

COLOR can be used freely within a label in the same way as other types of label characteristics such as fonts, spacing and line breaks.

Example

```
"This is" COLOR RED " a two-tone label." COLOR 20 20 80
```

The label will print as follows:

"This is" will be printed in the default label color.

" a two-tone label." will be printed in RED as defined in the color.tpl file.

If the label is used in a table stub on a line that has a row of data, the trailing dots at the end of the stub label will be printed in color 20 20 80 (a shade of blue).

Masks

COLOR can appear anywhere in a data mask. Note that this is different from a FONT specification which must be at the end of a mask. When used in a data mask, COLOR applies to the entire mask.

Example

```
MASK  COLOR 100 0 0  999.99%
```

The data values and percent sign will be printed in color 100 0 0 (red).

TEXT Masks

If you use a TEXT mask, you can vary colors within a table cell. In the following example, the color is entered with a REPLACE MASK statement in a format request.

Example

```
FOR ROW 1, COLUMN 1:  REPLACE MASK WITH TEXT
      COLOR GREEN '$' COLOR RED VALUE(2);
```

For the table cell in row 1, column 1, the data value will be displayed in red. The value will be preceded by a green $.

Example of Color Mask in Conditional Post Compute

Suppose that you are tabulating average family income by region and you wish to emphasize regions where the average family income is less than 20,000 by printing the table cell value in red for any cell with a value less than 20,000. You can do this with a conditional Post Compute that assigns to these cells a mask with the color red.

```
POST COMPUTE AVG_INCOME 'Average Family Income' =
INCOME/COUNT MASK $999,999  IF INCOME/COUNT >= 20000;
INCOME/COUNT MASK COLOR RED $999,999 IF OTHER;
```

Average Household Income by Sex of Householder and Region.

Regional Totals	Sex of Householder		
	Total	Male	Female
	Average Family Income		
Total U.S.	$31,830	$36,884	$20,460
New England	35,776	41,825	23,694
Mid Atlantic	33,983	40,217	22,124
East North Central	32,044	37,784	19,174
West North Central	27,947	31,966	16,965
South Atlantic	31,480	36,817	20,673
East South Central	25,151	29,886	14,441
West South Central	28,264	32,728	16,975
Mountain	29,391	33,157	19,281
Pacific	36,429	41,053	24,871

Average incomes of less than $20,000 are shown in red.

Color Specifications for Footnotes and Notes

Text

Since the text for a footnote or note can contain any of the features allowed in labels, you can enter one or more color specifications in the text of a SET FOOTNOTE or SET NOTE statement. If you do not enter any color, the text will have the default label color.

Symbols

Footnote symbol color is determined by the following rules.

1. If you enter a symbol color in a SET FOOTNOTE statement, *this color will always take precedence over any other rule* for footnote symbol color.

2 You can choose a **default symbol color** by putting a SYMBOL COLOR statement in your profile or FORMAT request. For example:

```
SYMBOL COLOR = BLUE;
```

If you do not choose a default symbol color, the default label color will be used as the default symbol color.

3. When a footnote is displayed at the bottom of a table, the symbol will have the default symbol color unless rule 1 applies. This is true both for footnotes you create and for built-in footnotes.

4. When a footnote symbol is displayed in a label, it will be in the default symbol color unless rule 1 applies.

5. When a symbol for a built-in footnote is displayed in a table cell, it will be in the cell color unless rule 1 applies. Cell color is the color that would be used if the cell contained a data value.

6. When a symbol for a regular (not built-in) footnote is displayed in a table cell, it will be in the default symbol color unless rule 1 applies. In particular, it will not be affected by color in a mask or a REPLACE MASK COLOR statement.

In a SET FOOTNOTE statement, a footnote symbol can have one color assigned. The color can be entered before or after the symbol. If you are using the default footnote symbol, you can add a color specification before or after the word DEFAULT.

Example

```
SET FOOTNOTE CONFIDENTIAL
        SYMBOL COLOR RED '**'
        TEXT COLOR GREEN  'Confidential data.  Do not release.';
REPLACE TITLE WITH COLOR BLUE
        'Average Household Income by Sex of '
        'Householder and Region' FOOTNOTE CONFIDENTIAL;
```

The table title will be BLUE. The symbol '**' for the footnote called CONFIDENTIAL will be printed in RED both in the table title and at the bottom of the table. The footnote text will be printed in GREEN at the bottom of the table.

Average Household Income by Sex of Householder and Region.[**]

Regional Totals	Sex of Householder		
	Total	Male	Female
	Average Family Income		
Total U.S.	$31,830	$36,884	$20,460
New England	35,776	41,825	23,694
Mid Atlantic	33,983	40,217	22,124
East North Central	32,044	37,784	19,174
West North Central	27,947	31,966	16,965
South Atlantic	31,480	36,817	20,673
East South Central	25,151	29,886	14,441
West South Central	28,264	32,728	16,975
Mountain	29,391	33,157	19,281
Pacific	36,429	41,053	24,871

** Confidential data. Do not release.

If we wanted to use a default numeric footnote symbol, we could get the same RED color for the footnote symbol by using the word DEFAULT with the color:

```
SET FOOTNOTE CONFIDENTIAL
        SYMBOL COLOR RED DEFAULT TEXT COLOR GREEN
                'Confidential data.  Do not release.';
```

For built-in footnotes such as EMPTY, the symbol displayed in a table cell will take on the color in effect for the cell. The footnote symbol at the bottom of the table will have the default symbol color (the same as the default label color if not explicitly set) and the footnote text will have the default label color. If you want to change this treatment, use a SET FOOTNOTE statement to override the built-in treatment.

Example

```
SET FOOTNOTE EMPTY   SYMBOL COLOR RED &endash; ;
        TEXT RED 'Data not available.';
```

Note that a medium-width dash (endash) is used as the footnote symbol for EMPTY. This character is represented by **&endash;** . See the appendix called "Character Sets" for additional details.

Setting COLOR Defaults for Characters and Rules

COLOR defaults can be set for all characters and rules used in tables. Defaults can be set in either the profile or a format request using the following FORMAT statements. Colors can be specified in r g b format or using color names that have been defined in color.tpl.

Format

```
DEFAULT COLOR = color;
LABEL COLOR = color;
RULE COLOR = color;
SYMBOL COLOR = color;
```

Color defaults are applied as described below. For additional details, see the section on "COLOR Defaults" in the FORMAT chapter. In cases where color specifications are entered directly into ***individual*** table elements such as labels, masks or footnotes, these individual specifications will take precedence over the ***default*** COLOR specifications.

DEFAULT COLOR is the print color for the entire table if no other colors are specified. If you do not set DEFAULT COLOR, the default is black.

If RULE COLOR and LABEL COLOR are specified, the DEFAULT COLOR remains as the default color for table cells.

RULE COLOR is the print color for rules. It applies to all rules, including rules added by the FORMAT statement RULE AFTER ROW and rules included with spanner labels. If no explicit RULE COLOR is specified, rules are printed in the default color.

LABEL COLOR is the print color for all text in tables except character strings in cell masks. These strings are printed in the default color. If no explicit LABEL COLOR is specified, all labels, titles and footnote texts are printed in the default color.

SYMBOL COLOR is the print color for all footnote symbols. If SYMBOL COLOR is not set explicitly, the default label color is used for symbols.

COLOR defaults apply only to characters and rules. For background shading in color or grey, see "Background Shading with COLOR or GREY" in this chapter or the FORMAT statement called SHADE.

Example

```
DEFAULT COLOR = 0 20 99;
FOR TABLE 1:  REPLACE TITLE WITH COLOR RED 'Red table title';
FOR TABLE 2:  RULE COLOR = RED;
```

Effect

All tables will be printed in the default color 0 20 99 (a shade of blue) except as follows. The first table will have a red title. The rules in the second table will be red, and the rest of the second table will be printed in the default color.

Replacing Mask Color

With the FORMAT statement, REPLACE MASK COLOR, you can replace the color of a mask without disturbing any other specifications in the mask and without re-entering the entire mask. See the FORMAT chapter for complete details.

Mask color can be replaced for a single cell, a group of cells or the entire table. If you replace the mask color for an entire table, the mask color serves as a default color setting for the table cells without affecting other parts of the table that are colored by the DEFAULT COLOR statement.

Format

```
REPLACE MASK COLOR WITH  color;
```

The **color** can be specifed in **r g b** format or using color names that have been defined in **color.tpl**.

Example

```
FOR ROW 1:  REPLACE MASK COLOR WITH RED;
FOR ROW 1 COLUMN 1:  REPLACE MASK COLOR WITH BLUE;
FOR VARIABLE INCOME:  REPLACE MASK COLOR WITH GREEN;
```

Effect

The mask color for the first row will be red except in column 1 where the mask color will be blue. The rows and/or columns containing INCOME values will have a mask color of green.

Background Shading with COLOR or GREY

Shading is an excellent way to highlight selected parts of a table or to add color in a pleasing way if you have a color printer. Even if you have only a monochrome printer, you can get some excellent effects by using GREY shading to emphasize selected parts of a table.

You can use SHADE statements to specify background shading for an entire table or for different sections of a table. You can choose the color for each element shaded. Shading is specified in the profile or format request with the FORMAT statement called SHADE. Colors can be chosen using r g b numbers, color names that have been defined in color.tpl, or the special color GREY with a number between 0 and 100 (inclusive) that selects the degree of grey shading..

Format

```
SHADE table-element [COLOR] r g b;
SHADE table-element [COLOR] color-name;
SHADE table-element GREY n;
```

table-element can be any of the following:

TABLE	STUB
TITLE	DATA
WAFER LABEL	ROW
HEADNOTE	COLUMN
TOP	CELL
HEADING	FOOTNOTES
STUB HEAD	LABEL

Following are some examples to introduce you to shading. For complete details on shading, see the FORMAT statement called SHADE.

Example

```
FOR TABLE 2:  SHADE HEADING GREEN;
FOR TABLE 2, COLUMN 1:  SHADE CELLS RED;
```

Effect

All tables will be printed without shading except in the second table. The heading in the second table will be shaded in the color GREEN. The first column will be shaded with the color RED.

Example

```
SHADE TABLE PUMPKIN;
REPLACE TITLE WITH  COLOR RED
        'Tableau 15B ' 'Statistiques sommaires selon le genre
d'établissement.'
```

Effect

In this example, we combine background shading with color text in the table title. The entire table is shaded in pumpkin color (99 60 20) and the table title text is red. The other text, numbers and rules are printed in the default color black.

Tableau 15B Statistiques sommaires selon le genre d'établissement.

Dépenses de fonctionnement

	Salaires	Autres	Total
Musées			
Musées	42 357,34	282,36	42 639,70
Parcs naturels	71 628,05	76,25	71 704,30
Lieux d'intérêt historique	61 359,82	62,56	61 422,38
Archives	81 305,43	48,55	81 353,98
Centres d'expositions	511,48	10,53	522,01
Observatoires et planétariums	434,79	1,28	436,07
Zoos et aquariums	67,69	51,66	119,35
Jardins botaniques	74,15	6,98	81,13
Total	257 738,75	540,17	258 278,92

Example

```
FOR ROWS 1 TO 21 BY 2:  SHADE  ROW GREY 10;
```

Effect

Alternate data rows are shaded light grey.

Number of households by type of household and state.

	Type of Household		
	Married couple	Other family	Nonfamily household
Connecticut	57,980	18,666	37,626
Maine	22,535	1,937	11,952
Massachusetts	71,997	14,208	29,184
New Hampshire	11,541	744	3,451
Rhode Island	12,559	719	6,306
Vermont	1,456	360	–
New Jersey	134,095	37,399	70,480
New York	197,547	60,114	110,844
Pennsylvania	218,880	28,678	103,033
Illinois	25,297	1,869	13,212
Indiana	37,912	4,781	7,157
Michigan	2,964	1,437	–
Ohio	–	1,526	3,066
Wisconsin	22,528	–	12,172
Iowa	38,220	1,279	7,824
Kansas	11,169	1,250	1,276
Minnesota	10,192	2,598	7,973
Missouri	20,479	2,975	6,201
Nebraska	13,634	2,514	2,652
North Dakota	4,703	289	295
South Dakota	1,478	251	1,060

– Data not available.

CHAPTER 25

Printing and Export

PRINTING TABLES AND CONVERTING THEM TO DIFFERENT FORMATS

Introduction

By default, TPL tables are created as Adobe PostScript® files. The tables are all contained in a single file, **tables.ps**. which is located in the TPL subdirectory TPL*nnnn*. In the Microsoft Windows environment, tables are displayed, printed and exported using **Ted**. In Unix, Linux and other operating systems, tables are displayed using external programs of the user's choice. Printing and exporting are controlled by user prompts or command-line arguments.

Printing

In Windows, any table may be printed using the **print** command in **Ted**. If you have a printer that supports PostScript, you will get better results, especially if you are scaling or using dotted or dashed rules, by using **Postscript Print**. If you don't have a PostScript printer, you can export to **pdf** and then print from within Acrobat Reader.

In a non-windows environment, you cannot print tables directly unless you have a printer which supports PostScript. Otherwise you should print from within your table display program or export to **pdf** and print from Acrobat Reader or an equivalent program.

How to Export

Instructions for exporting files depend on whether you are running TPL TABLES under Windows or UNIX.

Windows

All table files are exported from TED. See *TED Help* for details on exporting interactively. To produce an exported file in a batch job, you must select the type of exports you want and include the commands in your script see *TPL Help* or the appendix on Scripts.

UNIX

If you run TPL TABLES using the command line prompts, a prompt will ask if you want to export to the various formats. To produce an export file using a command argument, see the appendix with UNIX Run Instructions. If you wish to prevent prompts, you can set values for the export options in format statements.

Autosize

When a table is to be printed, it must fit onto the paper. If the table is too big, it is broken into parts so that it can be printed. Many of the export types are not intended to be printed so they do not have the page printing constraint. Instead the "paper" can be made as wide and long as needed. For example a data table or a CSV file is never broken into parts to accomodate a fixed size page, so autosize is alway set. Web pages can be as large as needed but you may wish to limit their size so users will not have to scroll to see them. So HTML export allows autosize as an option.

Note that autosize does not guarentee that the entire table run will produce a single export file. Table breaks and such commands as EJECT AFTER ROW may cause the output to be spread over multiple files. The exact behavior depends on the export type.

EPS Export

EPS is a form of PostScript designed for exporting into programs such as publication software. It goes into the programs as a graphic but one which can be scaled without loss of resolution. Nearly all of the images in this manual were placed as eps files. Each page of a TPL TABLES run gets

exported as a separate eps file. In addition, each page has a **bounding box w**hich exactly matches the outer edge of the table on the page rather than the entire page. When a TPL TABLES generated eps file is brought into publication software, the image will either be visible or a bounding box will show on the page. So the eps image can be placed exactly where you want it. Page markers are not considered to be part of the table so they are outside the bounding box and will not be displayed in the publication.

PDF Export

Adobe Acrobat PDF format is a commonly used format for exchanging documents including tables. It is often used for web display when the author wishes to display the document with more precise formatting than is available using HTML. The Acrobat Reader, which is used for displaying the PDF file, is available free for most operating systems.

PostScript tables can be distilled to Acrobat PDF format. In the Windows version of TPL TABLES, you can do this as an export from TED if you have Adobe Acrobat Distiller available on your computer. If you do not have Acrobat Distiller, you can install and use Ghostscript (gs815w32.exe) to create PDF files. After TPL TABLES is installed, an installation icon for GPL Ghostscript may be on your desktop. Otherwise, it can be found in C:\program files\gpl. See ***PDF*** in TED Help for more information.

Ghostscript is available for free download for most versions of UNIX, Linux and other operating systems.

HTML Export

You can use TPL TABLES to produce HTML tables that can be directly displayed on the web by any recent browser. The HTML fully conforms to the W3C standard for HTML 4.01. HTML tables are very close in appearance to the PostScript tables. There may be some small differences based on the fonts that are available on the machines viewing the HTML. There are also differences because PostScript is designed for display on fixed page sizes while the HTML "page size" is variable. With the HTML you can specify autosize which means that the HTML page is as large as it needs to be to display the table. This option is not available for PostScript.

Accessible HTML

TPL TABLES can produce HTML tables which conform to the World Wide Web Consortium (W3C) guidelines for HTML Accessibility. These guidelines are part of the Americans with Disabilities Act, Electronic and Information Technology Accessibility (***Section 508***). All HTML tables posted on U.S. government web sites must conform to these guidelines.

By producing your HTML tables with TPL TABLES, you can ensure that the HTML will be conformant and accessible to people who are blind or visually impaired.

The 508 conforming tables are nearly identical to the regular TPL HTML tables so it is unnecessary to put two sets of tables on your website in order to conform to the Section 508 guidelines.

For instructions and additional details about accessible HTML, see the **HTML ACCESS** statement as described in the Format chapter of the manual.

Footnote Display at the End of a Table

All footnote text displayed at the end of a table is included on the last HTML page that has data, even if the PostScript version of the table has footnotes that continue on to one or more pages after the last page of data.

If you have specified that footnotes be displayed in multiple columns, they will be displayed as they are in the PostScript table except that the text for a single footnote will not be broken across multiple columns.

Pages and Navigation

Navigation Bar

By default, HTML tables are saved in a job's **TPLnnnnn** subdirectory in a way similar to EPS tables. One HTML file is created for each table page with a page number and a suffix of **.htm** on the file name.

You can request a navigation bar to string together the HTML pages. If the pages with a navigation bar are to be displayed on the Web, all pages should be stored in the same directory on your Web server.

Links and Anchors

HTML provides a way for page viewers to jump between web pages or different locations within a web page. This is accomplished by inserting **Links** and **Anchors** within the web pages. An Anchor is a destination. A Link is an instruction to jump to an anchor or web address when the link is pressed by the viewer. Links are displayed to viewers by underlining. Anchors are not visible in the displayed web page.

TPL Tables provides language for inserting Links and Anchors into table labels or cells. See the section on Links and Anchors later in this chapter and the chapter on *Labels* and *Masks* for details.

Autosized and Single File HTML

The Autosize option causes the "paper" to expand so that you do not get page breaks because of too many columns or rows in a table. Page breaks will still occur if there are explicit ejects. Breaks will also occur between tables unless you use Format statements to join them, e.g. EJECT AFTER TABLE = NO or SKIP 0 LINES AFTER TABLES.

The Single File option does not affect what gets put on each page of output. It just puts all of the pages together into a single file rather than splitting them across files. All title, heading, and footnote information is retained for each table page and the pages are joined end to end in the HTML file. Since only one HTML file is created, no page number is added to the HTML file name.

It is reasonable to use both the Autosize and Single File options for the same HTML export. If both are used, tables will be formatted without row or column breaks and multiple tables will be joined end to end in a single HTML file.

Note If you are using Autosize with tables that have wafers, you will get best results by putting the wafer labels in one of the spanner positions (WAFER LABEL = DATA SPANNER or ROW SPANNER).

Page Markers

Page Markers are included in HTML tables and are always aligned to the left.

HTML Links and Anchors

HTML links (or hyperlinks) and anchors are used to direct browsers to jump from one web page to another or to jump from one location on a page to another on the same or a different page. A **Link** specifies a path to a page. An **Anchor** marks a location on a page. When you click on a link, the browser jumps to the page specified by the link. If the link includes an anchor, the browser jumps to the location on the target page marked by the anchor.

You can attach links and anchors to masks and labels. They only affect tables that are exported as HTML. A link attached to a mask will cause cells of a table to have links. A link attached to a label will cause the displayed label to have a link.

Formats

```
HTML LINK path-name
HTML LINK path-name#anchor-name
HTML ANCHOR anchor-name
```

where **path-name** is usually a relative path including file name and **anchor-name** is an identifier.

If a link contains blanks, it must be enclosed in ***double*** quotes, for example, "..\north east\my table.htm".

Windows Note If you are entering a link interactively, do not enclose it in quotes. They will be added if they are needed.

Case makes a difference on the web (and on UNIX systems), i.e. the file **Table1.htm** is different from **TABLE1.HTM** or **table1.htm**. For the web, the case of each letter in a link must match the case of each letter of the name on the web. If you use all lower case, you won't have to think about this, but note that the default names of the html files exported from TPL begin with an upper case **T**, e.g. **T**able1.htm, **T**able2.htm, etc. If you retain these names for the web, links to these files must have the upper case **T**.

The link should include the full name of the target file, e.g. **table2.htm** not just **table2**. Note that you must know what the target file name will be even if you haven't exported it yet. You may find it easier to do an export, figure out file names, then add links and export again.

Note that no checking is done to verify the existence of the path-name since the target path-name may not exist when the job is run.

Links

A link is a path to a file. The path should be relative to the location of the file containing the link. If the target file will be in the same subdirectory as the file you are linking from, you can just enter the target file name. If you must follow a path, it should begin with something like **..** rather that something like **c:**. This is because web pages will typically be created on a local disk and then moved from the local disk to a web site. Note that you may use forward or backward slashes.

A link in a label applies to the first label segment that follows it. If your label has more than one segment and you want the link to apply to more of the label, you must set the link for each segment separately. This scheme lets you to attach a link to only a single word or group of words rather than a whole label if that is what you want.

Examples The following link has a target file in the same subdirctory as the file containing the link. No path information is required.

```
replace stub head with html link mw2.htm "Characteristics";
```

The following link has its target file in a different subdirectory of the same directory. In addition, the target subdirectory has a blank in its name, so the link must be enclosed in quotes.

```
replace stub head with
        html link "..\north east\ne2.htm" "Characteristics";
```

The following table title is divided into two segments, so that a link can be applied to only the last segment.

```
replace title with "Population by city, current year.  "
  html link ..\lastyear\citypop.htm "Click here for last year's data.";
```

The same link could be included in a mask:

```
for row 1 column 1:  replace mask with
  html link ..\lastyear\citypop.htm 999.99;
```

Note If HTML ACCESS is set and there is a link in a cell that contains only a footnote symbol, two links will be generated. The HTML ACCESS link will take precedence.

Using Links with Anchors

If you wish to jump to a specific location on the target page, you must put an anchor at that point and add the anchor name to your link. You specify this by writing your link path followed by # followed by the anchor name, e.g **html link table2.htm#start_of_footnotes**. If the target is on the same page as the link, you can just reference the anchor; e.g. **html link #start_of_footnotes**.

An Anchor can be any valid TPL identifier. Anchors should be unique for any given web page. So if you attach an anchor to a mask, the mask must be for an individual cell, not a row, column or variable. Similarly, if an anchor is attached to a label, the label should appear only once on a page.

Examples

In the following example, assume that there are two pages of a table in the same subdirectory. A footnote is displayed at the bottom of the second page **ne2.htm**. One of the labels in the table has a footnote symbol that is linked to an anchor at the beginning of the footnote text.

```
for condition tenure(3): replace label with "No cash rent"
        html link ne2.htm#military_bases footnote cash_rent;

set footnote cash_rent
text html anchor military_bases "Housing units on militay bases are
included in the no cash rent category.";
```

Page **ne1.htm**

Table A1. Households by Type of Residence. See Contents page for links to other tables based on Type of Residence.	
	Total
Connecticut	
Owner	56,750
Renter	33,399
No cash rent[1]	1,832
See footnotes at end of table.	

Page **ne2.htm**

Table A1. Households by Type of Residence. See Contents page for links to other tables based on Type of Residence. - Continued	
	Total
New York	
Owner	98,477
Renter	77,281
No cash rent[1]	5,290
[1]Housing units on military bases are included in the no cash rent category.	

If the path to the page containing the anchor has one or more blanks, enclose the entire link, including the anchor, in quotes.

```
for condition tenure(3): replace label with "No cash rent"
        html link "..\north east\ne2.htm#military_bases"
        footnote cash_rent;
```

HTML Links to External or Absolute URLs

An HTML link to a URL will work for tables that are on the web. Enter the URL as you would enter the path in an HTML link to another table page. It must be enclosed in double quotes.

Example

```
replace title with "Population by state" /
        "This table was created by "
        html link "http://www.tpltables.com" "TPL Tables.";
```

Windows Note If you are entering a link interactively, do not enclose it in quotes. They will be added if they are needed.

How to Request HTML Tables

Instructions for exporting HTML depend on whether you are running TPL TABLES under Windows or UNIX.

Windows

HTML tables are exported from TED. See *TED Help* for details on exporting interactively. To produce HTML in a batch job, see *TPL Help* or the appendix on Scripts.

UNIX

If you run TPL TABLES using the command line prompts, a prompt will ask if you want HTML output. To produce HTML using command arguments, see the appendix with UNIX Run Instructions. See also the Format statement HTML OUTPUT = YES/NO to prevent the prompts.

CSV (delimited) Export

In exported CSV files, each cell value is contained in double quotes and the values are separated by commas. The wafer labels, if any, and the stub labels are added to the data as extra columns at the beginning of each row. If you do not want these label columns, delete the wafer labels and the stub before exporting. The bottom level of the heading is used as the first row of the CSV file. This row provides field names for the columns of the file. If you do not want this row of names, delete the heading before exporting. You may also change the heading labels before exporting if you want better field names.

Footnotes symbols are not included in the labels or values, but other mask items, such as $, %, or mask text, are retained in the data values.

See the Format statement CSV DIVIDER to separate the values with a character other than comma.

Windows Note If you are exporting interactively from TED, there is an option to enter a character other than comma to be used as the divider between the values in the exported file(s), or you can select Tab as the divider.

CSV Files

By default, exported CSV files are saved in a job's **TPLnnnnn** subdirectory. One CSV file is created for each table in the job with a name that includes the table number and a suffix of **.csv** on the file name. For example, if there are three tables in the job, there will be three exported files with the names **Table1.csv**, **Table2.csv**, and **Table3.csv**.

Windows Note If you are exporting interactively from TED, the exported file(s) will be saved using the **File Name** and **Current Directory** shown in the Export screen. You can change the name and directory if you wish.

ODS and XLS Export

ODS and XLS are two different spreadsheet formats. ODS is the current open international standard for spreadsheets. All current versions of Excel (since Excel 2007) and other current spreadsheet programs such as Open Office read ODS files. XLS is an older, Microsoft Excel specific, format for spreadsheets. Other spreadsheet programs will in general read XLS files but tt is recommended that ODS be used instead of XLS because it is the current standard.

TPL spreadsheet export files, when brought into a spreadsheet program, produces a display similar to what the original table looks like. However, when the table display would interfere with the normal use of a spreadsheet, TPL opts for easier use of the spreadsheet. For example, banking is not retained for spreadsheet files.

Text Table Export

A **text table** is a table made up exclusively of fixed-width characters and fixed width blanks. It can be printed on any printer and viewed in most display programs. The text table is often wider than the PostScript table and so may break into more pages. You can specify **autotsize** when you export the table but then the table may be too wide to print.

Below is a PostScript table and its text table equivalent

Table Q1. Selected Characteristics of Households, by Total Money Income
[Numbers in thousands]

	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999
All households	46,333	3,105	5,184	4,846	4,776
Tenure					
Owner	29,791	1,136	2,350	2,494	2,711
Renter	15,672	1,836	2,667	2,229	1,968
No cash rent	871	133	167	123	97
Region					
Northeast	10,020	579	1,190	879	920
Midwest	11,543	812	1,343	1,218	1,239
South	15,469	1,288	1,693	1,778	1,631
West	9,302	425	959	971	987
Type of Household and Sex of Householder					
Male householder					
Married couple	24,967	446	1,114	1,916	2,340
Other family	1,390	87	138	151	162
Nonfamily household	5,686	578	858	736	652
Female householder					
Married couple	1,536	39	96	95	130
Other family	5,283	814	939	775	629
Nonfamily household	7,472	1,142	2,039	1,173	863

Table Q1. Selected Characteristics of Households, by Total Money Income
[Numbers in thousands]

	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999
All households	46,333	3,105	5,184	4,846	4,776
Tenure					
Owner	29,791	1,136	2,350	2,494	2,711
Renter	15,672	1,836	2,667	2,229	1,968
No cash rent	871	133	167	123	97
Region					
Northeast	10,020	579	1,190	879	920
Midwest	11,543	812	1,343	1,218	1,239
South	15,469	1,288	1,693	1,778	1,631
West	9,302	425	959	971	987
Type of Household and Sex of Householder					
Male householder					
Married couple	24,967	446	1,114	1,916	2,340
Other family	1,390	87	138	151	162
Nonfamily household	5,686	578	858	736	652
Female householder					
Married couple	1,536	39	96	95	130
Other family	5,283	814	939	775	629
Nonfamily household	7,472	1,142	2,039	1,173	863

Data Table Export

The Data Table export allows you to turn a table into a data file. It is similar to a text table but with all non-data parts removed. If we use the example from text tables above we get:

```
46,333     3,105     5,184     4,846     4,776
29,791     1,136     2,350     2,494     2,711
15,672     1,836     2,667     2,229     1,968
   871       133       167       123        97
10,020       579     1,190       879       920
11,543       812     1,343     1,218     1,239
15,469     1,288     1,693     1,778     1,631
 9,302       425       959       971       987
24,967       446     1,114     1,916     2,340
 1,390        87       138       151       162
 5,686       578       858       736       652
 1,536        39        96        95       130
 5,283       814       939       775       629
 7,472     1,142     2,039     1,173       863
```

If we add the option to zero-fill we get:

```
046,333   003,105   005,184   004,846   004,776
029,791   001,136   002,350   002,494   002,711
015,672   001,836   002,667   002,229   001,968
0000871   0000133   0000167   0000123   0000097
010,020   0000579   001,190   0000879   0000920
011,543   0000812   001,343   001,218   001,239
015,469   001,288   001,693   001,778   001,631
009,302   0000425   0000959   0000971   0000987
024,967   0000446   001,114   001,916   002,340
001,390   0000087   0000138   0000151   0000162
005,686   0000578   0000858   0000736   0000652
001,536   0000039   0000096   0000095   0000130
005,283   0000814   0000939   0000775   0000629
007,472   001,142   002,039   001,173   0000863
```

There is also an option to retain the stub which is useful in identifying the rows of data

PC-Axis Export (Windows only)

The PC-Axis family consists of a number of programs for the Windows and Internet environments. It supports interactive browsing and downloading from numerical tables and is used by a number of national statistical offices. For more information see the PC-Axis web site at:

www.pc-axis.scb.se

The following rules apply when exporting a table to PC-Axis format:

- You must create the table using FILL EMPTY LINES.
- Each table to be exported must be a single cross-tabulation. It cannot have THEN's in its specification.
- The table cannot contain rank variables.

PC-Axis Files

The files are exported in a standard PC-Axis format. By default, the exported files are saved in a job's **TPLnnnnn** subdirectory. One file is created for each table in the job with a name that includes the table number and a suffix of **.px** on the file name. For example, if there are three tables in the job, there will be three exported files with the names **Table1.px**, **Table2.px**, and **Table3.px**.

Note If you are exporting interactively from TED, the exported file(s) will be saved using the **File Name** and **Current Directory** shown in the Export screen. You can change the name and directory if you wish.

Example Following are a simple Postcript table and the PC-Axis file created from this table.

Table 1. Households by residence and region

	Type of Residence	
	Inside metropolitan areas	Outside metropolitan areas
Region		
Northeast	6,246	1,006
Midwest	4,654	2,622
South	6,438	2,723
West	4,587	1,724

```
CHARSET="ansi";
MATRIX="Q1";
SUBJECT-AREA="area";
SUBJECT-CODE="S1";
TITLE="Table 1. Households by residence and region";
LANGUAGE="en";
CONTENTS="Count";
UNITS="units";
DECIMALS=3;
HEADING="Type of Residence";
STUB="Region";
VALUES("Type of Residence")="Inside metropolitan areas","Outside metro-
politan areas";
VALUES("Region")="Northeast","Midwest","South","West";
DATA=
6246.000 1006.000
4654.000 2622.000
6438.000 2723.000
4587.000 1724.000
;
```

CHAPTER 26

Data Drill (Windows Only)

Looking at the Contributors to Your Table Cells

Data Drilling provides a way to examine the records which contribute to a table cell. This will help you discover such things as which data records are the major contributers to a cell or perhaps help you detect errors in the data.

Data Drilling works in Ted on a finished table. You begin the process by selecting the cells you wish to examine. Once you have selected the cells of interest, press **Data Drill**. You will now be presented with a list of variables from the records to include in your drill report. The list includes most of the variables used in your table request. Post Computed variables are not included in the list since they are not associated with individual fields. Only the numerators used in percent cells are included in the list. If you click **Include variables not in request**, you will also get other codebook variables on the records used by the request. For databases, fields on database tables not used by the request are not available.

Fields are added to your report by highlighting them and pressing the **Add** button. It is best to limit your report to only those fields which are of interest. Otherwise your report is hard to read and print.

Another option you have is to specify the number of decimal places to be used for the observation variables in your report.

After you have completed specifying your drill report, press the **Apply** button. Your data file will be reread and your drill report will be displayed in Ted. You may now add results for additonal cells to your report or create a new drill report.

CHAPTER 27

Statistical Tests (Windows Only)

STATISTICAL TESTING AND DISPLAY

Introduction

TPL Tables is a program for producing statistical tables in presentation quality format. It is not intended as a general purpose statistical package. However, some people wish to produce tables with statistical test results in them. TPL Tables performs these tests and formats the results in a presentation quality output all in a single step. The tests are performed in Ted on "finished" tables.

The calculations specified in this documentation and implemented in TPL Tables code are commonly used calculations for the following statistics. It is the responsiblity of the user to make sure that the assumptions about the samples and distributions required for these calculations are satisfied by the user data. All tests assume normal distributions and that the samples are independent. They do not assume that the number of contributers to each cell match.

The tests currently available are:

Student's T-Test
Z Test
Anova F-Test
F-Test of Standard Deviations
Chi Squared Test
Tukey HSD Test

How Statistics Test Results are Displayed

All of the displays of statistical results in TPL Tables involve marking of the cells on which the tests are performed. In some cases the cells are shaded. In others, footnote symbols or cell markers are used.

In addition, the cells are identified with a footnote text or note at the end of the page or table. The footnotes are ordinary footnotes with names of $STAT_n$ where *n* are successive numbers. The footnote texts for these can be modified as with any other footnote.

Templates

Changing statistics footnote texts individually for each test you perform can be a burden. So TPL Tables provides a more general way to change the text while preserving the individual results. This method involves changing templates. The templates are listed in the *templates.tpl* file in the directory where TPL Tables is installed. Change only the text, not the names. You may change them in place and add an include statement to *profile.tpl*. If you do this, be sure to include the full path name to the templates file. Alternately you can add changed templates to a local *profile.tpl* in the directory where you are running your job. Finally you can add modified template statements to your individual format requests.

Footnote templates are just like regular footnotes with a two exceptions. You should alway use footnotes, never notes. Otherwise the note template will display even when its associated statistical test is not performed. TPL Tables will generate a footnote or note from the template as appropriate for the type of statistic being used. The second distinction of templates is that they may contain the keywords **Value** and **Variable**.

Value is replaced by the significance level selected for the test. For example, if a significance level of .05 is selected, the resulting footnote will contain either **.05** or **5%** where value appears in the template. Which appears depends upon an item in the **Preferences** menu of Ted. If **Confidence as Percent** is checked, **5%** will be used. Otherwise **.05** will be used.

Variable is used only for certain tests such as Chi Squared which are performed using selected control variables. **Variable** is replaced by a list of these variables.

Template Example

Replace the CHI_SQUARED_TWO_SIG template with the text:

```
Set Footnote(CHI_SQUARED_TWO_SIG) Text
    "A 2 tailed Chi Squared test performed on " VARIABLE
    "in the shaded area resulted in a difference at the" VALUE "level.";
```

Perform a 2 tailed Chi Squared test on Group and Sex at a signficance of .10 using the template. **Confidence as Percent** is checked. If the test shows a significant difference, the footnote will be:

A 2 tailed Chi Squared test performed on GROUP and SEX in the shaded area resulted in a difference at the 10% level.

Other Output

The tables themselves contain only the final output of the statistical tests -- whether the results are significant. In some cases you may wish to see additional information. A file, ***stats.log***, is created and displayed in a Ted window It contains additional information about the tests performed. It is located in the TPLnnnn directory where the test was run. The exact content of the log depends upon which tests are performed.

Notes and Restrictions on Statistics Tests

Statistical tests cannot be performed on Post Computed variables in TPL Tables. Some of the tests are performed on Means or Standard Deviations. For these tests, the table should be created using the Mean or Standard Deviation built-in statistics rather than using a Post Compute.

TPL Tables supports weighted means. These can be used in all statistical tests that involve means. TPL Tables implements this by treating a weighted value as multiple records. If a record used in a mean has a weight of 3, TPL Tables treats the statistical calculation as if the data file had 3 identical records with no weight. The extension to fractional weights is straightforward.

The display of statistical tests involves marking of the cells to which they apply with footnote symbols, cell markers or shading. Since a cell cannot have 2 different footnote symbols, cell markers or shades, TPL Tables does not automatically mark the same cell as a contributer to two statistical tests. If you wish to use the same cell in multiple tests, then you can adjust the shading and footnote texts to reflect the actual situation by using standard footnote and shade statements.

Undo

The Ted program has an **undo** command which is especially useful for statistical testing. When a test is applied, several changes may be made in the table. Without **undo**, if you decided to not display the test results you might have to add or remove several format statements. With **undo**, a single click removes all of the new format statements. Note however that applying and undoing statistical tests until you get the result that you want will work but may not be a statistically sound procedure.

Restricting Variables and Conditions in Statistics Testing

Some statistics tests supported by TPL apply to a cross tabulation (*See Cross Tabulation in the Tables chapter*) rather than to 2 individual cells. These tests are restricted to 2 control variables. But TPL supports cross tabulations with more than 2 control variables. For example:

Three Control Variables

	Race							
	White				Black			
	Age							
	Young		Old		Young		Old	
	Sex of Householder							
	Male	Female	Male	Female	Male	Female	Male	Female
Average Income	24,137	16,663	38,068	21,939	19,903	9,608	28,001	16,812

If you choose to run a Chi Squared test against this table, you will be presented with 3 variables to check Race, Age and Sex. You can only check 2. If you check Race and Age, the test will be performed as if the table you ran was:

Restricted to Race by Age

	Race			
	White		Black	
	Age			
	Young	Old	Young	Old
Average Income	21,572	33,496	14,149	22,525

If you choose Race and Sex, the test will be performed as if the table you ran was:

Restricted to Race by Sex

	Race			
	White		Black	
	Sex of Householder			
	Male	Female	Male	Female
Average Income	37,146	21,488	27,420	16,144

Note The cells in the above tables don't "add up" because the cells are averages rather than counts.

Restricting Conditions

You can also restrict the conditions used in a statistics test even for unselected control variables. The effect is the same as running the table request with a select statement to filter out the unwanted conditions.

This feature is useful if your request has a control variable with subtotals or error values or if your control variable includes more categories than you wish to look at in your test. For example, if your control variable is **State** and you are only interested in testing New England states, then you can limit your control variable to only those states without having to create a new table request.

Student's T-Test

Description

A Student's T-Test is used to determine whether the difference between two cells containing means are statistically significant.

Scope

A T-Test is performed on 2 mean cells. Note that the cells must come from a Mean statement rather than a Post Compute.

How is T-Test Calculated?

Based on ***Mathematical Statistics*** by Bickel and Dotsom

For the selected Cells a and b:

n_a is the number of records contributing to a

n_b is the number of records contributing to b

x_{ai} are the values contributing to a

x_{bi} are the values contributing to b

$$\text{Mean } \overline{X}_a = \frac{\sum_i x_{ai}}{n_a}$$

$$\text{Mean } \overline{X}_b = \frac{\sum_i x_{bi}}{n_b}$$

$$S2^2 = \frac{(\sum_i (x_{ai} - \overline{x}_a)^2 + \sum_j (x_{bj} - \overline{x}_b)^2)}{(n_a + n_b - 2)}$$

$$t = \sqrt{n_a n_b / (n_a + n_b)} (|\overline{x}_a - \overline{x}_b|) / S2$$

Footnote Templates

TTEST_ONE_SIG (One tailed T-Test with significant results)
TTEST_TWO_SIG (Two tailed T-Test with significant results)
TTEST_ONE_NOSIG (One tailed T-Test with non-significant results)
TTEST_TWO_NOSIG (Two tailed T-Test with non-significant results)

Z Test

Description

The Z-Test is used to determine if the difference between a sample mean the full population mean is large enough to be statistically significant. Note that the cells must come from Mean statements rather than Post Computes.

Scope

Two table cells are selected. One should be a sample mean and the other a full population mean.

How is Z-Test Calculated?

Based on ***Basic Statistical Analysis***
by Richard C Sprinthall
found at http://en.wikipedia.org/wiki/Z-test

N_p = Population Count
N_s = Sample Count
V_p = Total of values contributing to the Population Cell
V_s = Total of values contributing to Sample Cell
v_{pi} = Individual values contributing to the Population Cell
Standard Deviation for Population σ_p =

$$\frac{\sqrt{N_p \sum_i v_{pi}^2 - (\sum_i v_{pi})^2}}{(N_p - 1)^2}$$

$$SE = \sigma_p / \sqrt{N_s}$$

$$z = (V_s / N_s - V_p / N_p) / SE$$

Footnote Templates

ZTEST_ONE_SIG (One tailed Z-Test with significant results)
ZTEST_TWO_SIG (Two tailed Z-Test with significant results)
ZTEST_ONE_NOSIG (One tailed Z-Test with non-significant results)
ZTEST_TWO_NOSIG (Two tailed Z-Test with non-significant results)

Anova F-Test

Description

An ANOVA test is used to compare the means in a region of a table to see whether there are significant differences among them.

Scope

Select a single Mean cell. Note that the cells must come from a Mean statement rather than a Post Compute. The test will be performed on all of the cells in the Cross Tabulation (*See Cross Tabulation in the Tables chapter*) containing that cell. You can restrict the test to part of the Cross Tabulation if you wish. The test is performed on at most 2 of the control variables used to form the Cross Tabulation

How is Anova F-Test Calculated?

Based on ***Concepts and Applications of Inferential Statistics***
by Richard Lowry
found at http://faculty.vassar.edu/lowry/webtext.html

n_T = Number of Cells in Test
v_T = Total of all values contributing to test
v_{ci} are the individual values contributing to cell c
n_c is the number of cells contributing to cell c

Degrees of Freedom between groups $DF_{bg} = n_T - 1$

Degrees of Freedom within groups $DF_{wg} = \sum_c (n_c - 1)$

Variance for cell c $\sigma_c^2 = \dfrac{n_c \sum_i v_{ci}^2 - (\sum_i v_{ci})^2}{n_c(n_c - 1)}$

Mean for cell c $\bar{v}_c = v_c / n_c$

Mean for all cells $\bar{v}_T = v_T / n_T$

$$MS_{wg} = \frac{\sum_{c} \sigma_c^2 (n_c - 1)}{\sum_{c} (n_c - 1)}$$

$$MS_{bg} = \frac{\sum_{c} n_c (\overline{v_c} - \overline{v_T})^2}{DF_{bg}}$$

$$f = \frac{MS_{bg}}{MS_{wg}}$$

DF_{bg}, DF_{wg}, and f are used to calculate the confidence value

Footnote Templates

ANOVA_ONE_SIG (One tailed ANOVA test with significant results)
ANOVA_ONE_NOSIG (One tailed ANOVA test with non-significant results)

F-Test of Standard Deviations

Description

An F-Test of Standard Deviations compares two cells containing standard deviations to see whether they are statistically different.

Scope

Select two cells containing standard deviations. ***The order of cell selection matters.*** Note that the cells must come from a Stdev or Stdevp statement rather than a Post Compute.

How is F-Test Caculated?

Based on the ***US National Institute of Standards and Technologies (NIST) Engineering Statistical Handbook***:
NIST/SEMATECH e-Handbook of Statistical Methods
http://www.itl.nist.gov/div898/handbook/

For the selected Cells a and b:

n_a is the number of records contributing to a
n_b is the number of records contributing to b
v_{ai} are the values contributing to a
v_{bi} are the values contributing to b

Variance for cell a $\sigma_a^2 = \dfrac{n_a \sum_i v_{ai}^2 - (\sum_i v_{ai})^2}{n_a(n_a-1)}$

Variance for cell b $\sigma_b^2 = \dfrac{n_b \sum_i v_{bi}^2 - (\sum_i v_{bi})^2}{n_b(n_b-1)}$

$f = \sigma_a^2 / \sigma_b^2$

Footnote Templates

FTEST_ONE_SIG (One tailed F-Test with significant results)
FTEST_ONE_NOSIG (One tailed F-Test with non-significant results)

Chi Squared Test

Description

The Chi Squared test is performed on an array of count or weighted count table cells determined by 1 or 2 control variables. The test determines whether the distribution of the values in the cells is significantly different from what would be expected.

Scope

Select a single cell which contains a count or weighted count. The test will be performed on all of the cells in the Cross Tabulation (*See Cross Tabulation in the Tables chapter*) containing that cell. You can restrict the test to part of the Cross Tabulation if you wish. The test is performed on at most 2 of the control variables used to form the Cross Tabulation.

How is Chi Squared calculated?

Based on ***Concepts and Applications of Inferential Statistics***
by Richard Lowry
found at http://faculty.vassar.edu/lowry/webtext.html

Yates' correction Based on
http://en.wikipedia.org/wiki/Yates'_chi-square_test

O_{rc} = The observed value for the cell at row r column c

$$O_r = \sum_c (O_{rc})$$

$$O_c = \sum_r (O_{rc})$$

E_{rc} = The expected value for the cell at row r colum c

$$E_{rc} = \left(\frac{O_c \times O_r}{O_{rc}}\right)$$

$$X^2 = \sum_{r,c} \frac{(O_{rc} - E_{rc})^2}{E_{rc}}$$

In the case where the array of cells is 1 by 2 or 2 by 2 and some observed cell is < 5, a Yates correction is performed by modifying the calculation:

$$X^2 = \sum_{r,c} \frac{(|O_{rc} - E_{rc}| - .5)^2}{E_{rc}}$$

Restrictions

The test is limited to about 100 table cells. More specifically, if N_a is the number of conditions for one control variable and N_b is the number of the other, then (N_a - 1) (N_b - 1) must be less than 100.

Footnote Templates

CHI_SQUARED_ONE_SIG (One tailed Chi Squared test with significant results)
CHI_SQUARED_TWO_SIG (Two tailed Chi Squared test with significant results)
CHI_SQUARED_ONE_NOSIG (One tailed Chi Squared test with non-significant results)
CHI_SQUARED_TWO_NOSIG (Two tailed Chi Squared test with non-significant results)

Tukey HSD Test

Description

The Tukey Honestly Significantly Different (HSD) test is typically a follow-on test to an ANOVA F-test. The ANOVA test may show that the aggregate difference among the means of several samples is significant. The Tukey test attempts to determine which individual pairs of means differ significantly. If you have already run an ANOVA test, you may wish to undo those results before running the Tukey test to avoid cluttering your table.

Scope

Select a single mean or weighted mean cell. The test will be performed on all of the cells in the Cross Tabulation (*See Cross Tabulation in the Tables chapter*) containing that cell. You can restrict the test to part of the Cross Tabulation if you wish. The test is performed on at most 2 of the control variables used to form the Cross Tabulation.

How is Tukey HSD Test Calculated?

Based on ***Concepts and Applications of Inferential Statistics***
by Richard Lowry
found at http://faculty.vassar.edu/lowry/webtext.html

$$\text{Mean } \bar{v}_c = \frac{\sum_i v_{ic}}{n_c}$$

where
n_c is the number of contributers to cell c
v_{ic} are the values which contribute to cell c

$$\text{Harmonic Mean } H = \frac{N}{\sum_c (1/n_c)}$$

where
N is the total number of cells in the calculation
n_c is the number of contributers to cell c

$$\text{Variance } \sigma_c^2 = \frac{n_c \sum_i v_{ic}^2 - \left(\sum_i v_{ic}\right)^2}{n_c(n_c - 1)}$$

where

n_c is the number of contributers to cell c

v_{ic} are the values which contribute to cell c

$$\text{Within Group Mean } \overline{WG} = \frac{\sum_c (\sigma_c^2 (n_c - 1))}{\sum_c (n_c - 1)}$$

$$\text{Tukey value } Q_{ij} = \frac{|\bar{v}_i - \bar{v}_j|}{\sqrt{\overline{WG}/H}}$$

The Q_{ij} value for each pair of cells, *i* and *j*, in the scope is compared with a table to determine whether the values are significantly different. If the values are not significantly different, they get a common Cell Marker.

How are Results Displayed?

A Tukey test involves comparing each cell in a range with every other cell. With even a modest number of cells, the total number of comparisons can be large. If footnotes were generated for each comparison, the table would be crowded and hard to read. Instead a more compact method is used to show results. Markers are used for this. If two cells are statistically similar, they will have a marker letter in common. If two cells are different, they will have no marker letter in common. If for example all pairs of cells in the comparison range are statistically the same, then each of these cells will get the single marker ***a***. If each pair of cells differ, each cell will get a different marker letter.

For a more complicated example, suppose we have 4 table cells we wish to compare. There are 6 comparisons. Suppose we find:

1 - 2 similar so assign a to 1 and 2
1 - 3 similar so assign b to 1 and 3
1 - 4 differ
2 - 3 differ
2 - 4 similar so assign c to 2 and 4
3 - 4 differ

Results

cell 1 gets ab
cell 2 gets ac
cell 3 gets b
cell 4 gets c

Note that we cannot use ***a*** to display both the 1 - 2 comparison and the 1 -3 comparison because then it would appear that 2 and 3 also are statistically similar.

Restrictions

Unlike other statistical tests on Cross Tabulations, you cannot restrict the variables used in a Tukey test. So your cross tabulation must have at most 2 control variables in it. You can restrict the condition values for these variables.

Footnote Templates

TUKEY_NOTE (Note describing output)

CHAPTER 28

TPL-SQL

INTRODUCTION TO THE DATABASE INTERFACE

TPL-SQL is an optional database interface for TPL. It allows TPL Tables and TPL Report to read data directly from a SQL database. When you use the interface, you do not need to first extract the data from the database. So you do not need space to store the extracted data. You also do not need to know SQL. TPL automatically generates the request to extract just the data it needs. It processes the data as it extracts it, so there is not even a need for extra temporary storage. Further, TPL does not write on your database. Anyone with read access to the data can produce tables or reports.

In TPL Tables and TPL Report, there is very little difference between accessing a stand-alone sequential file and accessing one or more relations stored in a database. If you already know TPL Tables or TPL Report and know the structure of your data, you will find it very easy to use TPL-SQL. The primary differences between processing a sequential file and processing a database are found in describing the data in your codebook.

ODBC Note This chapter applies to all versions of TPL-SQL, including the Windows version that accesses databases via ODBC. ***If you are using the Windows version of TPL-SQL, we recommend that you use Codebook Builder to generate a codebook.*** Most of the information contained in this chapter is also included in the Codebook Builder Help, along with instructions on how to use Codebook Builder.

Oracle Note If you are running jobs against an Oracle database on a Unix or Linux machine, you mst set **LD_LIBRARY_PATH=*oracle-path*/lib** where *oracle-path* is where Oracle is installed.

TERMINOLOGY - YES, YOU WANT TO READ THIS

Unfortunately, TPL Tables and relational theory use the same words to mean different things. In relational terminology, a *table* is approximately the equivalent of a flat sequential data file in TPL terminology. In TPL terminology, a *table* is the final product of a TPL Tables run. To avoid confusion, when we refer to a *table* we mean the output of a TPL Tables run. We will refer to the data file used in relational terminology as a *SQL table* or a *relation*. *Variable* and *field* are used interchangeably. The SQL term *column* will not be used as this could be confused with TPL Tables and TPL Report output columns.

TPL-SQL CODEBOOK

The primary difference between using TPL Tables or TPL Report with a sequential file and using them with TPL-SQL can be found in the description of the data contained in the TPL codebook.

Note Codebooks for SQL databases cannot contain repeating groups.

Sequential files may be either flat files or hierarchical files. Each row of a sequential file is called a *record*. If all records are of the same format, the file is flat. If records of different format are interspersed, the file is called *hierarchical*. The order of the records on the sequential file determines the hierarchical membership; i.e. the children immediately follow their parent.

When using TPL-SQL, TPL Tables and TPL Report process data stored in SQL databases as hierarchies. But in the case of a SQL database, records of different types are not interspersed. Instead, the records of each type are stored in separate SQL tables. Since the order of records cannot be used to describe hierarchical relationships, something else must be used. This other thing is a pairing between fields on different SQL tables. We call such a pairing, an *association*, and statements which define them are *association statements*. The fields in association statements are frequently but not always *key* fields. They are the fields that are specified in SQL requests in the *where* clause of *joins*.

A TPL codebook describing a sequential hierarchical file consists of a description of one or more records. Each record description consists of a description of its constituent fields. A TPL codebook which uses TPL-SQL consists of descriptions of one or more SQL tables. Each SQL table

description consists of a description of its constituent fields. In addition, the TPL-SQL codebook includes association statements which define how multiple SQL tables are to be processed together.

A Simple TPL-SQL Codebook Example

Though much of this document discusses hierarchical files, we will begin with a flat file example. Suppose we have a flat sequential file describing a family. The TPL codebook might be:

Sequential File Version

```
Begin Families Codebook
Family  Record
Filler 7
Region control 1
        (
                "Northeast"      = 1
                "North Central" = 2
                "South" = 3
                "West"  = 4
        )
Living_Qrt  "Living Quarters" control 1
        (
                "Owned"          = 1
                "Condominium" = 2
                "Rented"         = 3
                "Unknown"        = " "
        )
Persons_in_family obs 2
Gross_income_of_head obs 7
Gross_income_of_spouse obs 7
End Families Codebook
```

If we now load our data into a SQL database we can describe our data file to TPL using the following:

SQL Database Version

```
Begin Families Codebook SQL
Family defines "family" Table
Region defines "region" control 1
        (
                "Northeast"      = 1
                "North Central" = 2
                "South" = 3
                "West"  = 4
        )
```

```
Living_Qrt  defines "living_qrt" "Living Quarters" control 1
        (
                "Owned"         = 1
                "Condominium" = 2
                "Rented"        = 3
                "Unknown"       = " "
        )
Persons_in_family defines "persons_in_family" obs 2
Gross_income_of_head defines "gross_income_of_head" obs 7
Gross_income_of_spouse defines "gross_income_of_spouse" obs 7
End Families Codebook
```

The only necessary changes are that **SQL** is added after **Codebook**; **Record** becomes **Table**; and the **Filler** field is eliminated. In most cases a **defines** clause is also needed. A change in meaning which is not discernible concerns the order of fields in the record or SQL table description. In a sequential file description, the fields must be listed in the order they occur in the file. In a SQL database description, the order of the fields is arbitrary within a SQL table. Another change is that it is not necessary to describe all of the fields in a SQL table in your codebook. Perhaps some of the fields are confidential and should not be used. Other fields may be text fields which are not appropriate for tables. If you do not have TPL Report, you may wish to omit these fields. For sequential file codebooks you are required to mark the space these fields occupied as **Filler**. In a codebook describing a SQL table, the fields can just be omitted.

Defines Clause

The **defines** clause is used to map TPL variable names into SQL database field names. If the **defines** clause is omitted, TPL assumes that the SQL database field name is the same as the TPL variable name except that it is all uppercase. In the above example, all of the SQL database field names are assumed to be in lowercase. So the **defines** clauses are necessary. Define clauses are discussed more fully later.

A Better Solution - Using Information from the Database

The simple transformation above is not the recommended way of creating a TPL-SQL codebook. The SQL database contains much of the information contained in the TPL codebook. If the TPL codebook information does not match the database information, then errors or incorrect tables will result. Thus instead of requiring the transformation described above, TPL provides a way to automatically query the database for the relevant information

contained within the database. You activate this processing by omitting the information you can obtain from the database. The recommended codebook source is:

You Write

```
Begin Families Codebook SQL
Family Table
Region defines "region" control  get conditions from data
Living_Qrt  defines "living_qrt" "Living Quarters" control from data
Persons_in_family defines "persons_in_family" obs
Gross_income_of_head defines "gross_income_of_head" obs
Gross_income_of_spouse defines "gross_income_of_spouse" obs
End Families Codebook
```

In this codebook source, field widths and obs modifiers such as **float** have been eliminated as have control variable condition values lists. Instead of listing the condition values, we have included **get conditions from data** or its shorter synonym **from data**.

Unix

On computers running Unix, use the **tpl conditions** program to process the above codebook source. The **tpl conditions** program will fill in the field widths, datatype details and control variable conditions to create a new codebook source. The new codebook source shown below can be edited to provide better condition names. It should then be run through **tpl codebook** to produce a compiled codebook. For more details, see the Appendix titled ***TPL Conditions*** and the sections on **tpl conditions** in ***Run Instructions (UNIX)*** for more details.

TPL-SQL Generates

```
Begin Families Codebook SQL
Family defines "family" Table
Region defines "region" control 1
        (
                "1"     = "1"
                "2"     = "2"
                "3"     = "3"
                "4"     = "4"
        )
Living_Qrt  defines "living_qrt" "Living Quarters"  control 1
        (
                "1"     = "1"
                "2"     = "2"
                "3"     = "3"
                " "     = " "
        )
```

```
Persons_in_family  defines "persons_in_family"  obs 2
Gross_income_of_head defines "gross income of head"  obs 7
Gross_income_of_spouse defines "gross_income_of_spouse"  obs 7
End Families Codebook
```

Windows

On computers running Microsoft Windows, the above codebook source is processed directly by the codebook processor to produce the following source. The data source and password if required are also provided to the **codebook** program. See ***Run Instructions (Windows)*** and ***Scripts (Windows)*** for more details.

NOTE: It is important for Windows sources that each variable description start on a new line. Otherwise the new source described below may not be correct.

When the codebook processor has completed its work, a new codebook source is created with **evaluated to** expressions to show the information obtained from the database. The actual conversion depends upon both the data and the database management system. A possible conversion of our example follows:

TPL-SQL Generates

```
Begin Families Codebook SQL
Family defines "family" Table
Region defines "region" control get conditions from data
        evaluated to control 1
        (
                "1"     = 1
                "2"     = 2
                "3"     = 3
                "4"     = 4
        )
Living_Qrt  defines "living_qrt" "Living Quarters" control from data
        evaluated to control 1
        (
                "1"     = 1
                "2"     = 2
                "3"     = 3
                " "     = " "
        )
```

```
Persons_in_family  defines "persons_in_family" obs
        evaluated to obs 2
Gross_income_of_head defines "gross income of head" obs
        evaluated to obs 7
Gross_income_of_spouse defines "gross_income_of_spouse" obs
        evaluated to obs 7
End Families Codebook
```

You may wish to edit and reprocess this new source. This will enable you to do such things as provide better labels for condition values. If the codebook name is *Families*, the new codebook source will be *FAMILIES.S*.

Conversions from Database to TPL Data Types

In our example, we assumed that all of the data was loaded into the database as character fields of the same length as the original data. This is not necessary. SQL databases support many formats of data and conversions of data from one format to another. However, you as a user of TPL do not need to worry about these conversions. In general all you need to do is label the fields as **observation**, **control**, or **character**. The system will correctly determine the exact data type and place it in the **evaluated to** clause. You can use these data types explicity, but this is not recommended.

Restrictions

There are some restraints on what fields you can label **observation** or **control**. For example, if a data field contains alphabetic characters, it should not be used as an observation. TPL will detect and report some of these errors when the codebook is created. Others will only be detected when table or report jobs are processed.

ODBC Data Type Conversions

The following chart shows the acceptable conversions from ODBC to TPL datatypes.

	TPL Types		
ODBC Types	**Obs**	**Con**	**Char[2]**
date	y	y	y
time	y	y	y
timestamp	y	y	y
char	y[1]	y	y
varchar	y[1]	y	y
longvarchar	y[1]	y	y
numeric	y	y	y
decimal	y	y	y
tinyint	y	y	y
smallint	y	y	y
integer	y	y	y
bigint	y	y	y
float	y	y	y
double	y	y	y
binary	y	y	y
varbinary	y	y	y
longvarbinary	y	y	y
bit	y	y	y
real	y	y	y

[1] If a character string described as obs contains non-numeric values, this will be detected when a table or report is processed. It will not be detected by the codebook processor.

[2] **Character** may be a new data category for you. It is similar to control but without the requirement of a list of possible conditions. In TPL Report it can be used in selects, recodes, conditional computes or report statements. In TPL Tables it can be used in selects, defines or conditional computes but not directly in tables

Oracle Data Type Conversions

The following chart shows the acceptable conversions from Oracle to TPL data types.

	TPL Types		
Oracle Types	**Obs**	**Con**	**Char[2]**
date	y	y	y
char	y[1]	y	y
varchar[2]	y[1]	y	y
varchar	y[1]	y	y
number	y	y	y
long	n	n	n
raw	n	n	n
long raw	n	n	n
rowid	n	n	n
mslabel	n	n	n

[1] If a character string described as obs contains non-numeric values, this will be detected when a table or report is processed. It will not be detected by the codebook processor.

[2] **Character** may be a new data category for you. It is similar to control but without the requirement of a list of possible conditions. In TPL Report it can be used in selects, recodes, conditional computes or report statements. In TPL Tables it can be used in selects, defines or conditional computes but not directly in tables

Sybase Data Type Conversions

The following chart shows the acceptable conversions from Sybase to TPL datatypes.

	TPL Types		
Sybase Types	**Obs**	**Con**	**Char[2]**
char	y[1]	y	y
varchar	y[1]	y	y
bit	y	y	y
binary	y	y	y
tinyint	y	y	y
smallint	y	y	y
integer	y	y	y
float	y	n	n
long float	y	n	n
money	y	n	n
datetime	y	y	y
decimal	y	y	y
numeric	y	y	y
image	n	n	n

[1] If a character string described as obs contains non-numeric values, this will be detected when a table or report is processed. It will not be detected by the codebook processor.

[2] **Character** may be a new data category for you. It is similar to control but without the requirement of a list of possible conditions. In TPL Report it can be used in selects, recodes, conditional computes or report statements. In TPL Tables it can be used in selects, defines or conditional computes but not directly in tables

New Data Types

In addition to the usual codebook data types, the following data types may be generated in the **evaluated to** clause. You should not enter these directly. However, you may wish to add a *time-unit* to generated **obs date** fields.

obs varying and **con varying** — These are just regular observation and control variables stored on the database as varying length fields. TPL Tables and TPL Report automatically handle these data types so you can treat them as if they are normal fixed-length fields. "Short" control vari-

able values are right-padded with blanks. This data type is currently implemented for databases only.

character varying — TPL Report does not pad these fields when they are displayed. This data type is currently implemented for databases only.

money or **obs money** — This is a floating point data type. If there is no explicit mask provided, the system defaults to a mask of **$999.99**. This data type is currently implemented for databases only.

control date or **character date** — The format of the date is determined by a database environment variable. If you change the value of this environment variable, you must rerun your codebook against the database before you run a table or report request. Dates are sorted and displayed in chronological order rather than character sort order. This data type is currently implemented for databases only.

Sybase If client Sybase software is not installed on the computer on which TPL is running and installation Option 1 was used to install TPL, **control date** fields will be displayed in US English. If you wish to display dates in a different format, then you can explicitly change the generated labels.

obs date *time-unit* — where *time-unit* may be **year**, **day**, **hour**, **minute**, or **second**. If you chose to explicitly call a field **obs date** and omit *time-unit*, then the system will assume a *time-unit* of **day**. This data type is currently implemented for databases only.

Oracle The field evaluates to the number of units since January 1, 1900 at 00:00:00. Dates are floating point numbers so decimal values will be shown if a mask is provided which specifies them.

Sybase The field evaluates to the number of units since the current time (when job is run). Thus all past dates are negative and future dates are positive. Dates are truncated integer values not decimal fractions.

ODBC The meaning of time units in databases accessed via ODBC depends upon the underlying database system. Consult your database manuals for information on this.

The **obs date** data type is especially useful for computing time intervals; e.g.,

```
Compute Life_span = Death_date_in_years - Birth_date_in years;
```

Note that the time unit for the two terms in the compute must be the same or a trash answer will result.

Label-Code Tables

It is common in SQL databases to have SQL tables which pair code values with longer descriptions or labels. This can save a considerable amount of space in your database. It also allows the description to be changed without changing large numbers of database records. When an extract is made from the database, a SQL join is normally performed so that the label rather than the code is displayed with the other data fields. The TPL codebook can make use of these label-code pairs to create condition sets for control variables.

In our earlier example of codebook generation we included the line:

```
Living_Qrt  "Living Quarters" control from data
```

This generated the following:

```
Living_Qrt  "Living Quarters" control from data evaluated to control 1
        (
                "1"     = 1
                "2"     = 2
                "3"     = 3
                " "     = " "
        )
```

Suppose we had in our database a SQL table **lq_tab** with fields **lq_code** and **lq_lab** and values:

lq_code	lq_lab
1	Owned
2	Condominium
3	Rented
' '	Unknown

The **lq_code** field in the **lq_tab** SQL table has the same range of values as the **Living_Qrt** field of the **family** SQL table. So we can substitute the new TPL codebook statement:

```
Living_Qrt "Living Quarters" control from lq_tab(lq_lab,lq_code)
```

The result is:

```
Living_Qrt  "Living Quarters" control from lq_tab(lq_lab,lq_code)
evaluated to control 1
        (
                "Owned"         = 1
                "Condominium" = 2
                "Rented"        = 3
                " Unknown"      = " "
        )
```

The SQL table containing the label-code pairs can be referenced in this way without itself being described elsewhere in the codebook.

The label and code fields must be put in the parentheses in the order shown above, that is *label first* and *code second*.

Alternate Names - The DEFINES Clause

By default, the TPL name for a field is the same as the name of the field on the database. There are some cases where this is not desirable. For example, in a sequential file we sometimes wish to use the same field as both an observation variable and a control variable. We accomplish this by using a **redefine** clause. TPL-SQL codebooks cannot have **redefine** clauses, but the **defines** clause can be used to accomplish the same result.

```
tpl-name1 defines sql-name  control from data
tpl-name2 defines sql-name obs
```

The *sql-name* is the name for the database field. It may be placed within quotation marks. This is useful if it happens to be a TPL keyword or is otherwise not a valid name for a TPL variable. *tpl-name1* and *tpl-name2* are two TPL variable names. They can be used in table and report requests as well as association statements in the codebook. The only place the SQL name can be used is in the special SQL SELECT statements discussed later.

A codebook description with a **defines** may include all of the standard codebook field qualifiers; e.g.,

```
Living_quarters  "Living Quarters" defines Living_Qrt control
condition label is "Housing type = " value  from data
```

Sybase Sybase is case sensitive; that is, lower-case letters and upper-case letters are not treated as equal. TPL is case sensitive only for items within quotes. Thus if Sybase fields were given lower-case names when the fields were created in the database, **defines** must be used to reference them. For example, if a Sybase field is "Last_name" then consider the following **TPL codebook** statements:

1) Last_name CONTROL GET CONDITIONS FROM DATA;

2) LAST_NAME DEFINES Last_name CONTROL
GET CONDITIONS FROM DATA;

3) LAST_NAME DEFINES "Last_name" CONTROL
GET CONDITIONS FROM DATA;

Statement (1) is not acceptable since TPL will convert Last_ name to LAST_NAME which will not match the database field name.

Statement (2) is also unacceptable since again TPL will convert Last_name to LAST_NAME

Statement (3) is correct because TPL will not change the case of the quoted item.

ODBC Some databases accessed via ODBC in the Windows version of TPL-SQL are case sensitive in the same way as described for Sybase above. If you get error messages saying that your database fields cannot be found, it may be because you need to enclose the database field names in quotes. If you use the Codebook Builder to prepare your codebook, you do not need to be concerned with this. Codebook Builder will provide the names from the database and enclose them in quotes.

Fields within a single SQL table must have unique names. However, it is common practice for fields in different SQL tables within the same database to have the same name. This is especially true for fields used for TPL associations or SQL joins. When processing a request, TPL Tables

and TPL Report must know which SQL table should be used to retrieve data for a variable. There are two options available in TPL Tables and TPL Report. First, you can use the SQL technique of qualifying a name when there is an ambiguity. For example suppose both the **Company** SQL table and the **Employee** SQL table have a field called **company_id**. A TPL Tables request can include a **Table** statement such as:

```
Table T1: Company.company_id by region, total;
```

An alternate approach using TPL would be to use **defines** clauses in your codebook to give the two fields different TPL names.

Creating Subfields with Substr

The **substr** feature lets you describe variables that are subparts of fields in your database. In non-database codebooks, this funtionality is provided by **Redefine**.

The list of data types supported for **Substr** should be the same as the list supported for Control variables.

The syntax for a subfield is:

```
SUBSTR(sql-name, start-position, length-of-subfield)
```

The *sql-name* is the name of the field in the database. If the *sql-name* is not a valid TPL name or is in a database that is case sensitive, it must be enclosed in quotes. In addition, quotes are required for lower case **sql-names**. **Start-position** is relative to the **sql-name** field

An example from a codebook source with subfields is:

```
COMPANY_CODE "Company Code" Defines "company_code" Con
        from Data
company_type Defines Substr("company_code", 1, 2) Con from Data
Numeric_part Defines Substr("company_code", 3, 3) Con from Data
```

An example of a resolved codebook source with all database-derived information filled in is:

```
COMPANY_CODE "Company Code" Defines "company_code" Con
Right Blank Fill from Data evaluated to Con 5 (
"AM703" = "AM703"
```

```
"AP001" = "AP001"
etc.
)
 company_type Defines Substr("company_code", 1, 2) Con from Data
evaluated to Con 5 (
"AM" = "AM"
"AP" = "AP"
etc.
 )
Numeric_part Defines Substr("company_code", 3, 3) Con
Right Blank Fill from Data evaluated to Con 5 (
 "001"  =  "001"
.
.
"703" = "703"
etc.
)
```

Multiple SQL Tables and Association Statements

A SQL database typically has many SQL tables in it. A TPL codebook for a SQL database need not describe all of these SQL tables. However a TPL codebook will usually describe more than one SQL table. The method for describing multiple SQL tables is basically the same as that used for describing multiple record types for a sequential hierarchical file. Separate descriptions are included for each SQL table. In addition, in the SQL case, one or more association statements must be provided to relate the separate SQL tables.

An Example

Consider the Families codebook we discussed earlier. Suppose our SQL database also contains a SQL table of family member data. Our combined codebook source might look like the following:

```
Begin Families Codebook SQL
Family  defines "family" Table
Family_id defines "family_id" obs
Region defines "region" control  get conditions from data
Living_Qrt  defines "living_qrt" "Living Quarters" control from data
Persons_in_family defines "persons_in_famly" obs
Gross_income_of_head  defines "gross_income_of_head" obs
Gross_income_of_spouse defines "gross_income_of_spouse" obs

Member  defines "member" Table
Family_id  defines "family_id" obs
```

```
Age defines "age" obs
Sex  defines "sex" control from data
Education defines "education" control from data
Favorite_car defines "favorite_car" from car(car_id,car_name)

Family is parent of member where Family.Family_id =
    Member.Family_id;
End Families Codebook
```

The example consists of two descriptions of individual SQL tables plus one association statement near the end which relates the two SQL tables. Notice that each of the SQL table descriptions has a field, **Family_id**, on it. This field is used by the association statement to connect the members with their family. The association statement asserts that a **Member** record belongs to a particular **Family** whenever the **family_id** on the **Member** record matches the **Family_id** on the **Family** record. The **parent** tells the system that **Family** is above **Member** in a hierarchical relation. In other words, each **Family** may have multiple **Member**s. TPL imposes few restrictions on the data types of the terms on each side of the equal sign in a **where** clause. However a database error will result if one of the terms is an integer or floating point number in the database and the other contains a non-numeric value.

When TPL Tables or TPL Report processes a request using this association statement, it will read each **Family** record, get the **Family_id** from the record and then use this **Family_id** to retrieve from the **Member** SQL table each of the **Member** records which have this **Family_id**. It will then read the next **Family** record, get its **Family_id** and proceed in the same way.

Warning **Fields on the right side of an association statement should be indexed. Otherwise performance may be unaccepable.** See the section on "Optimizing Performance" for more information.

By default, if a **Family** record has a **Family_id** which appears on no **Member** records, the **Family** record is rejected with an error message. The TPL codebook statements **Process incomplete hiearchies = yes** or **no** and **Report incomplete hierarchies = yes** or **no** will alter this behavior in the same way as it does with sequential hierarchies.

Now suppose we wish to add the **Car** SQL table to our codebook. Each **Member** has one **Favorite_car**. Then we do not have a **parent** relation but rather what we call a **sibling** or **sib** relation. The addition to our codebook might be:

```
Car defines "car" table
Car_id defines "car_id" control from data
Car_name defines "car_name" character
Car_cost defines "car_cost" obs
Car_weight defines "car_weight" obs

Member is sib of Car where Member.Favorite_car = Car.Car_id
```

When TPL Tables or TPL Report processes a database using this new association statement, for each **Member** record it reads, it will determine the **favorite_car** for that **Member**. This **Favorite_car** code will be searched for in the **Car_id** field of the **Car** SQL table. If exactly 1 match is found, processing will proceed normally. If no record in the **Car** SQL table has the appropriate **Car_id** an error message will result and both the **Member** and **Car** records will be discarded. If multiple **Car** records are found to have matching **Car_id**s, an error message will result and the "duplicate" **Car** records will be rejected. Which **Car** record is kept is undefined. If **Report incomplete hierarchies = no** is specified, no error message will be reported for either the case of no **Car** records or multiple **Car** records.

The effect of **Tabulate incomplete hierarchies = yes** depends upon whether a table or report is being created. In a TPL Tables job, **Tabulate incomplete hierarchies = yes** causes a higher level record with no lower level records to contribute to cross tabulations which only involve the higher level records. But when we are using **sib** associations, both records are at the same level. So there can be no cross tabulations to which the **Member** record contributes and the **Car** record does not. Thus **Tabulate incomplete hierarchies = yes** has no effect on records in an incorrect **sib** association in TPL Tables job.

In a TPL Report job, **Tabulate incomplete hierarchies = yes** results in a report row being generated for a record which does not have a sibling. The sibling columns for that row are marked as missing data.

It is not an error if multiple **Members** have the same favorite car or if some car is the favorite of no members. All that is required for a correct **sib** association is that each record from the SQL table to the left of the **sib** association have exactly one match in the SQL table to the right of the **sib** association. This brings up an important point that will be discussed more fully later. When a table or report request is processed, the resulting output will depend upon which associations are used. In our example, if we compute the average cost of **favorite_car**s by using our **sib** association to get from the **Member** SQL table to the **Car** SQL table we will not get the same result as we would if we just calculated the average cost of cars

in the **Car** SQL table. In the former case, the average is weighted by how many members have a car as their favorite. In the latter case, the average is unweighted.

More on Association Statements

In the above example, **Member** and **Car** were associated using a single pair of fields. In some cases multiple pairs of fields must be used to specify an association. These pairs are connected by **and**. Consider a database of **Employer**s and **Employee**s. Assume some of the **Employer**s have many branches. An **Employee** is employed at a single branch. The branch is designated by both a **Company_id** and a **Branch_id**. Then we might have two different association statements to relate the **Employee** SQL table to the **Employer** SQL table:

```
Employer is parent of Employee where
Employer.Company_id = Employee.Company_id

Employer is parent of Employee where
Employer.Company_id = Employee.Company_id
and Employer.Branch_id = Employee.Branch_id
```

A table or report request may use either of these associations. If the first one is used and we calculate the average number of employees per employer we will get the average number of employees per company. If we use the latter association statement we will get the average number of employees per branch.

As a notational convenience, the association statements can be written more concisely as:

```
Employer is parent of Employee where Company_id = Company_id

Employer is parent of Employee where Company_id = Company_id
and Branch_id = Branch_id
```

The SQL table name to the left of an equal sign in a **where** clause is assumed to be the same as the SQL table name to the left of **parent** or **sib** and the SQL table name to the right of the equal signs is assumed to be the same as the SQL table name to the right of **parent** or **sib**.

Use of %INCLUDE in Codebooks

A TPL codebook describing a large complex database can become very long. An easy way to make the codebook more manageable is to create separate files for descriptions of each SQL table. You can then use the **%INCLUDE** feature to assemble these files into a single TPL codebook. This technique is especially useful for corporate databases which are used by many different categories of users. Certain users will produce tables or reports from only some of the SQL tables while other users will use a different set of SQL tables for their work. Separate TPL codebooks may be made for these different users by including only those SQL table descriptions that they will need for their work.

Codebook Abstract

When the TPL codebook processor is run against a SQL database, multiple files are produced. For MS Windows, one is the new codebook source with **evaluation**s in it. This was discussed earlier. The second file is the **.K** file which is the codebook executable. This is a file that is not intended for reading. The third file is the **.L** file which contains error messages for an unsuccessful codebook or the **Abstract** of a successful codebook run. The Abstract for a TPL-SQL codebook is similar to the Abstract produced from a sequential file with a few exceptions.

A codebook for a sequential file contains columns for the number of bytes described, the level, and the parent of each record. None of these are relevant to a SQL database record. The sequential file codebook also contains information on the location of each field within a record. This is also not relevant for a SQL description.

The TPL-SQL codebook abstract contains two things not found in a sequential file codebook. One is the SQL column name. This name will match the TPL variable name unless a **defines** has been used to change the TPL name.

The second addition to a TPL-SQL codebook abstract is a list of the Association statements that have been specified. Each of these association statements has a number assigned to it. This number is important. It may be needed in creating a plan for processing the data in a table or report request.

TPL CODEBOOK Copyright(C) 2005 QQQ Software, Inc. All Rights Reserved.Version 6.0 of CODEBOOK compiled on Wed May 11 18:03:09 EST 2005.

TPLDB ABSTRACT FOR DATABASE tpldb
Created 6/17/05 at 4:49:16 PM from codebook source tpldb.cbk

The records and variables described in your codebook are listed below in alphabetical order. For non-database codebooks, the first position of each record is location 1.

RECORD SQL TABLE

OFFICE	BRANCH
COMPANY	COMPANY
EMPLOYEE	PERSON

SQL DATABASE ASSOCIATIONS

1:	OFFICE is parent of EMPLOYEE	where ID = ID
2:	COMPANY is parent of OFFICE and BRANCH = BRANCH	where COMPANY_ID = COMPANY_ID

VARIABLE	SQL COLUMN	SIZE	TYPE	TPL RECORD
NUMBER_EMP	SIZE	4	OBS	OFFICE
OFFICE	BRANCH	—	RECORD OBS	OFFICE
BIRTH	BIRTH	—	CON	EMPLOYEE
BIRTH_C	BIRTH	—	CHAR	EMPLOYEE
BIRTH_O	BIRTH	—	OBS	EMPLOYEE
BRANCH	BRANCH	4	OBS	COMPANY
BRANCH	BRANCH	4	OBS	OFFICE
COMPANY	COMPANY	—	RECORD OBS	COMPANY
COMPANY_CHAR	COMPANY_NAME	20	CHAR	COMPANY
COMPANY_ID	COMPANY_ID	4	OBS	COMPANY
COMPANY_ID	COMPANY_ID	4	OBS	OFFICE
COMPANY_NAME	COMPANY_NAME	20	CON	COMPANY
ID	IDX	1	OBS	OFFICE
ID	ID	10	OBS	EMPLOYEE
PERSON	FULLNAME	20	CON	EMPLOYEE
SALARY	SALARY	8	OBS	EMPLOYEE
SEX	SEX	1	CON	EMPLOYEE
EMPLOYEE	PERSON	—	RECORD OBS	EMPLOYEE

End CODEBOOK processing

TABLE AND REPORT REQUESTS FOR SQL DATABASES

A TPL Tables or TPL Report request run against a SQL database looks very much like a request run against a sequential file. There are five main differences:

1) The command line for invoking the job is slightly different. (The command line options appropriate for your database system are described in the run instructions appendices.)

2) A job run against a SQL database may require that some variable names be qualified with the name of their SQL table.

3) A SQL database request may include Association statements.

4) A request run against a SQL database may require a plan to specify how the data is to be read.

5) Statements can be included to optimize performance. A table or report request run against a SQL database may include a **SQL Select** statement in addition to or instead of a regular TPL **Select** statement. You can use a **SQL Fetch** statement in your profile or format request for additional performance tuning.

Qualified Names

All TPL variable names for sequential file fields must be unique. As was stated earlier, for a SQL database, the TPL names need be unique only for a given SQL table. TPL must know which SQL table it should retrieve data from. If there is an ambiguity, the variable names must be qualified by the TPL name for the SQL table; e.g. **Employer.Branch_id** or **Employee.Branch_id**. TPL Tables and TPL Report will produce an error message whenever an ambiguity exists.

The need for qualified names can be completely eliminated by using **defines** clauses in the codebook to assign unique TPL names to each database field. If you have not done this, TPL still minimizes the need for qualified names by the following procedure.

Suppose you wish to use **Branch_id** which is found on both the **Employee** SQL table and the **Employer** SQL table. The first

time you use **Branch_id** in your request you must qualify it; e.g. **Employee.Branch_id**. From that point forward, TPL will assume that when you use **Branch_id** you mean **Employee.Branch_id**. If in fact you wish to use **Employer.Branch_id**, you must explicitly qualify the name. From that point forward, you must qualify all occurrences of **Branch_id**. If this rule seems complicated, don't worry. You may choose to always qualify ambiguous TPL names. Also, if you fail to qualify a name that requires qualification, TPL will produce an error message rather than risk making an incorrect assumption about what you intend.

Association Statements in Table or Report Requests

In some cases, you may wish to create a table or report which requires an association of SQL tables which was not anticipated when the TPL codebook was created. Rather than require that the codebook be recreated, TPL allows you to add association statements to a TPL Tables or TPL Report request. Association statements in table and report requests are the same as association statements in the codebook except that a semicolon (**;**) is required at the end of each association statement. Association statements, if included in a request, must be located immediately after the **Use** statement of the table or report request. They are assigned numbers in order of occurrence beginning after the last number used by the codebook association statements. For example, if the codebook has 3 association statements and a table request has 2 more, the two table request association statements will be numbered 4 and 5.

The Processing Plan

By far the most important difference between a TPL Tables or TPL Report request run against a SQL database and against a sequential file concerns the sequence of records delivered to TPL for processing. In processing a sequential file the data records are delivered to TPL in precisely one possible order — the order the records appear on the file. In the database case, the records may be delivered to TPL in a variety of orders. The "same" request can produce radically different output if a different sequence of records is delivered to TPL.

The sequence of records delivered to TPL is determined by a **Plan**. A Plan is a list of association statements which "chain" together the SQL tables. A plan is **valid for a request** if it satisfies a collection of conditions:

1) All variables used in the request must be on SQL tables chained together in the plan.

2) No SQL table may be visited more than once in following the plan chain.

3) The plan must define a single hierarchical path through the database.

The next few sections will discuss these conditions.

What is a Chain?

SQL tables are chained together by a set of associations if starting from some node SQL table, we can get from it to each of the other SQL tables in the chain by going from the SQL table on the left of an association to a SQL table on the right of an association. For example suppose we have the following associations (the where clauses have been omitted for clarity):

```
1:      A is parent of B where ...
2:      B is parent of C where ...
3:      B is sib of G where ...
4:      G is parent of F where...
5:      C is parent of D where ...
6:      E is parent of D where...
7:      D is parent of F where ...
```

Starting with **A** as our node we can get to **B** (using 1) then **C** (using 2) then **D** (using 5) then **F** (using 7). So **A→B→C→D→F** form a chain. We can also form the chains **E→D→F** and **A→B→G→F** and several others. There is no chain which includes both **A** and **E**. So a table request which uses data from SQL table **A** and SQL table **E** could not be processed using the collection of Association statements listed.

How Can A SQL Table Be Chained to Itself?

The requirement that a plan not pass through the same SQL table twice is because TPL would not know which pass should be used to evaluate a variable. In some database formats people do wish to violate this rule. Suppose for example you have all employees including supervisors in the same file. You wish to have a count of how many employees have supervisors who earn particular salaries. You would like to use the Association:

```
Employee is parent of Employee where employee_id =
is_supervised_by;
```

The solution to this problem is to use the **defines** construct in the codebook. In our previous examples we have assigned a new TPL name to a SQL field. We can also use a **defines** to assign a new name for a SQL table. Our codebook would include the following statements:

```
Employee Table
    description of employee fields
Supervisor defines Employee Table
    same description of fields
Supervisor is parent of Employee where employee_id =
is_supervised_by
```

What is a "Single Hierarchical Path"?

Probably the easiest way to identify a "single hierarchical path" is to assign level numbers to the SQL tables in a plan. The node of the plan is the SQL table on the left of the first association statement. It is assigned level 0. Follow the plan chain. If the association statement contains **sib** the SQL table on the right is given the same level number as the SQL table on the left. If the association statement contains **parent** then the SQL table on the right is given a level number 1 higher than the SQL table on the left. Now look at your completed list. If any two **parent** associations have SQL tables with the same level number on the left, the plan does not define a single hierarchical path. Consider the association statements we looked at earlier:

```
A is parent of B where ...
B is parent of C where ...
B is sib of G where ...
G is parent of F where...
C is parent of D where ...
E is parent of D where...
D is parent of F where ...
```

One plan we specified was **A→B→G→F**. If we apply our test to this plan we get:

A **[level 0]** is parent of B **[level 1]**
B **[level 1]** is sib of G **[level 1]**
G **[level 1]** is parent of F **[level 2]**

No two parent associations have the same level number so the plan is a single hierarchical path.

Another valid plan is: **A→B→G→C** which uses the rules

A **[level 0]** is parent of B **[level 1]**
B **[level 1]** is sib of G **[level 1]**
B **[level 1]** is parent of C **[level 2]**

Compare this with the chain **A→B→C→G→F**. This chain uses

A **[level 0]** is parent of B **[level 1]**
B **[level 1]** is parent of C**[level 2]**
B **[level 1]** is sib of G **[level 1]**
G **[level 1]** is parent of F **[level 2]**

In this example, the second and fourth associations both are parent relations starting at level 1 so we do not have a single hierarchical path.

Why Does TPL Need a Single Hierarchical Path?

An example is probably the easiest way to answer this question. Suppose we have a database with 3 SQL tables. The SQL tables are **Company**, **Company_car**, and **Employee**. These SQL tables are connect by:

```
Company is parent of Company_car where company_id = owner_id
Company is parent of Employee where company_id = employer_id
```

These association statements jointly define a plan which violates the single hierarchy requirement. Now consider the following table request:

```
Compute Ratio = Car_price / Employee_salary;
Post compute Ave_car_salary_ratio =Ratio / Company_car;
Table T1: Company_location, Ave_car_salary_ratio;
```

This table request should tell us something about the cost of keeping employees happy in different locations. Unfortunately, TPL Tables cannot process this request. The problem is that there is no way to compute **Ratio** since we cannot pair an **Employee_salary** with a particular **Car_price**. We don't know who drives which car.

Let's change the example slightly. Suppose each company buys only one model of car. Then our association statements become:

Company is **sib** of Company_car where company_id = owner_id
Company is **parent** of Employee where company_id = employer_id

Now our plan is valid and our table request works. We know which car the employee has because we know his company and which car the company buys. So we can compute **Ratio**.

TPL doesn't allow multiple hierarchical paths in a single table or report request because the TPL system then does not have enough information to combine fields from the different paths.

Plan Selection

When the TPL Tables or TPL Report request processor encounters a request which is to be run against a database, the system analyzes the request and the association statements in the request and codebook. If you have specified a plan, it tests whether the plan you specified is valid for your request. If you have not specified a plan, it determines all valid plans for the request. If there is exactly one valid plan, the plan is reported at the end of the translation step and processing continues. If there is no valid plan or there are multiple valid plans, processing stops.

When there is no valid plan for your request, you have two choices. First you may add additional association statements to your request or codebook so that all required SQL tables are chained into a single plan. If this cannot be done, you must modify your request by eliminating all references to fields on the SQL tables which cannot be linked into the plan chain. Sometimes splitting a job into two separate jobs will enable you to get all of the data you want from the database while using valid plans.

If your request can be processed using more than one valid plan, the TPL system will list all valid plans and stop at the end of the translation step. At this point you must examine the listed plans and determine which if any correctly capture the desired meaning of your tables or reports. You must then add a plan statement to your request and reprocess the request. The following is an example of the output at the end of the translation step:

```
PLANS:

Read:   COMPANY
  3:    COMPANY is parent of OFFICE     where COMPANY_ID = COMPANY_ID and
        BRANCH = BRANCH
  2:    OFFICE     is sibling of OFFICE_1    where COMPANY_ID = COMPANY_ID
        and BRANCH = BRANCH
  1:    OFFICE     is parent of EMPLOYEE     where ID = ID

Read:   OFFICE
  2:    OFFICE     is sibling of OFFICE_1    where COMPANY_ID = COMPANY_ID
        and BRANCH = BRANCH
  1:    OFFICE     is parent of EMPLOYEE     where ID = ID
  4:    EMPLOYEE   is parent of COMPANY      where ID = OWNER

Read:   EMPLOYEE
  4:    EMPLOYEE   is parent of COMPANY      where ID = OWNER
  3:    COMPANY is parent of OFFICE     where COMPANY_ID = COMPANY_ID and
        BRANCH = BRANCH
  2:    OFFICE     is sibling of OFFICE_1    where COMPANY_ID = COMPANY_ID
        and BRANCH = BRANCH

*** ERROR:  Since there is more than one possible plan for processing
this request, you must select one of the above plans by inserting a
plan statement in your request.  Suppose the plan you wish to use has
the numbers 3,1,2 in that order next to the association statements.
Then your plan statement would be:

                PLAN 3 1 2;
```

How to Specify a Plan

A plan specification is just the word **PLAN** followed by a list of association statement numbers (in processing order) followed by a semicolon (**;**). An example of a plan statement is:

```
PLAN  3 5 12;
```

Association statement numbers may be obtained from the list of valid plans as in the example above. Alternately, they may be obtained from the codebook abstract. As mentioned before, if you add new association statements to your table or report request, their statement numbers are just the next unused numbers. A **PLAN** statement may occur anywhere in a table or report request after the **USE** statement and any association statements.

Plans and the COUNT Variable

Count is a built-in variable in TPL Tables. If a cross-tabulation has no explicit observation variable, **Count** is implicitly taken to be the observation variable. In a sequential hierarchical file **Count** gives a count of the number of records at the lowest level of the hierarchy. In a table request run against a SQL database, **Count** gives the equivalent result — a count of the number of records accessed from the SQL table on the right of the last association statement in the plan. The danger inherent in this is that if the plan changes, the count will also change.

Suppose we have a database with **Industry**, **Company**, and **Employee** SQL tables. The SQL tables use the associations:

```
1:  Industry is parent of Company where industry_id = industry_id
2:  Company is parent of Employee where company_id = company_id
```

We produce a table statement:

```
Table T1: Industry_category, Company_location;
```

Assuming **Industry_category** is on the **Industry** SQL table and **Company_location** is on the **Company** SQL table, the plan is just:

```
Plan 1;
```

Thus the table will be a count of companies for each location and industry category. Now suppose we add a second table to our request:

```
Table T2: Industry_category, Education_level;
```

where **Education_level** is on the **Employee** SQL table. Our request now requires the plan:

```
Plan 1 2;
```

The implicit **Count** now counts employees. Without changing the first table, we have changed its meaning. We now get a count of employees for each industry category and location instead of a count of companies.

The safest way to avoid problems in counting is to **always explicitly include the SQL table name in any cross-tabulations that do not already have an observation variable**. Then you will know exactly what you are counting.

Optimizing Performance

Indexing for Multi-Table Processing

Fields on the right side of an association statement should be indexed in the database. This is true for sibling, one-to-one associations as well as for hierarchical, one-to-many associations.

TPL does not use joins to process multiple SQL tables. Instead it processes the data in a hierarchical fashion. Suppose you have a database with **Employer** and **Employee** SQL tables. The tables are in a parent-child relationship where matches are on the basis of **employer_id** on each of the SQL tables. You wish to produce a TPL table using both of these SQL tables. TPL will read the first **Employer** record and find the value from that record for the **employer_id** field. It will then search through the **Employee** SQL table for each **Employee** record with the desired **Employer_id**. If your database does not have an index built on **Employer_id** on the **Employee** SQL table, then TPL will have to read through the entire **Employee** SQL table for each **Employer**. This can produce unacceptable performance!

In order to avoid this performance problem, your database must have indexes built on the key fields used on the "child" or right-hand side of the Association statement. If multiple key fields are needed to relate two SQL tables, you will get the best performanace if your database has an index based on the combined fields.

SQL Select

A table or report request run against a SQL database may include a **SQL Select** statement in addition to or instead of a regular TPL **Select** statement. The **SQL Select** statement provides support for an optimization which sometimes produces significantly improved performance. If you use a regular TPL **Select** statement in your table or report request with a SQL database, all records which follow the plan are delivered to TPL for processing. Those which fail the **Select** are rejected by TPL. If you use a **SQL Select** statement, records are rejected within the database software. Use of this statement improves performance by reducing network traffic and by saving TPL from processing data which it does not need.

Importance of Indexing and an Efficient SQL Select Statement

If the **SQL Select** statement excludes a large share of the data, a significant time savings can result. However, you must be careful that your **SQL Select** statement is efficient. TPL passes the **SQL Select** statement to the database system "as is" without attempting to optimize it.

SQL Select improves performance only if doing selection within the database is as fast as doing it within TPL. If you are selecting on a non-indexed field, the database selection is usually much slower than TPL selection. So don't use **SQL Select** if the field being selected on is not indexed.

In one case a user had 300,000 establishment records in his database. He had an indexed field **cycle** on **Establishment**. He first tried the **SQL Select** statement:

```
SQL Select on establishment "cycle between 115 and 123";
```

Using this statement, his request took 2 hours.

He replaced his **SQL Select** statement with:

```
SQL Select on establishment "cycle in (115,116,117,118,119,120,121
,122,123)";
```

This TPL request produced the same results but took only 5 minutes to process.

The reason why the second one was so much faster is that since **cycle** was indexed, the individual values in the **in** clause could be found quickly while the **between** construction required the database to do a sequential search for values in the range.

Description of SQL Select

The syntax of a **SQL Select** statement is:

```
SQL Select on SQL-Table "selection-string";
```

SQL-Table is the TPL name for a SQL table. *selection-string* is a string of text which is appended unchanged to the **where** clause of a **SQL Select** statement. Since the *selection-string* is not modified by TPL, it should contain SQL field names rather than TPL variable names.

Suppose the **Age** field is on the SQL table **Person**. Our TPL codebook has used a **defines** clause to assign the TPL name **Age_obs** to **Age**. Then the following two statements should produce the same result:

```
Select if Age_obs < 50;
SQL Select on Person "Age < 50";
```

Sybase and ODBC String values passed in SQL select statements must be in single quotes; e.g.

```
SQL Select  on Company  "name = 'QQQ Software' ";
```

If TPL and the database software are both running on the same computer, there will be little difference in performance between having TPL reject records and having the database software reject the records. Bigger differences will occur if TPL and the database software are running on different machines and if the selection is to be done at the bottom level of the hierarchy defined by the plan. In such cases data will be rejected before it travels across your network. If the machine running the database software is faster than the machine running TPL, additional performance improvements will be realized.

In most table and report requests, far more records are retrieved at the bottom level of the processing hierarchy than at higher levels. If a record fails a select at a level above the bottom level of the hierarchy, then no records will be retrieved from the database from lower levels regardless of whether a TPL **Select** or a **SQL Select** is used. Thus if selection is done above the bottom level of a hierarchy, there is unlikely to be much difference in performance between using a regular TPL **Select** and a **SQL Select**.

Difference in Results between Regular Select and SQL Select

In rare cases, **SQL Select** and regular **Select** can produce different results. The differences only occur when **Tabulate Incomplete Hierarchies = Yes;** has been specified.

Suppose we have a database with **Family** and **Member** data. If a family has no members, it will still contribute to the table if **Tabulate Incomplete Hierarchies = Yes;** is specified. Instead, suppose the database family does have members but a **SQL Select** is used to remove all of its members. To TPL, the cases are the same and the family will contribute to the table if **Tabulate Incomplete Hierarchies = Yes;** is specified.

Now suppose the database family does have members but a regular **Select** is used to remove all of them. TPL requires that an entire hierarchical unit pass a **Select** in order for any part of it to be included in the tabulation (see the section on the Select statement in the Hierarchies chapter). So the family is excluded regardless of the setting for **Tabulate Incomplete Hierarches**. In this rare case, **SQL Select** allows a family to be included which a regular **Select** excludes.

SQL Fetch

The **SQL Fetch** statement is a tuning parameter which should be placed in your **profile.tpl** file or in your format statements. It affects the amount of data that is moved from the SQL Server to TPL on each request for data.

The syntax of a **SQL Fetch** statement is:

```
SQL Fetch count = n;
```

where *n* is an integer. The default is 10.

In cases where TPL is executing on one machine and your database is on another, the choice of value can strongly affect network traffic and performance. In a typical example, changing the **SQL Fetch Count** value from 1 to 10 caused the job to run in 1/3 of the time!

Choosing too high a value for **SQL Fetch Count** will cause the job to use more memory than needed. This can actually slow down performance. Too small a value will degrade performance. If your table or report request uses a single SQL relation, then there is no theoretical limit to how high a value you can use. However there is probably little to be gained by using a value greater than 100. If you are processing a database hierarchically, select a **SQL Fetch Count** value no larger than the largest number of records at the bottom of the hierarchy associated with any given record immediately above. For example, if you are processing a family-member hierarchy and no family has more than 12 members, then 12 is the ideal choice for **SQL Fetch Count**. The exact choice is not critical. There will be little performance difference if you use 15 or 8.

SUMMARY

TPL-SQL provides TPL Tables and TPL Report with direct access to data stored on a SQL database. No intermediate storage is required. You do not need to know SQL in order to user the interface.

A TPL-SQL codebook is a simplified standard TPL codebook. It differs from a standard codebook in that information such as field widths can be omitted because these can be obtained from the database itself. The one important addition found in TPL-SQL codebooks is association statements which specify how different SQL tables are to be processed together. These association statements are chained together to form a plan for reading through the data during processing of a table or report request.

A TPL-SQL table or report request is also very similar to a standard TPL Tables or TPL Report request. The primary difference is that you may need to select the plan that is to be used in processing the data.

CHAPTER 29

Format

THE FORMAT LANGUAGE

Introduction

The FORMAT language gives you precise control over the format of your tables. The automatic formats provided by TPL TABLES are usually acceptable for analysis and for some types of publications. However, publication standards in your organization may require that you adjust your table formats in ways that cannot be achieved by using TPL statements alone. In other cases, you may find that the default values for such things as column widths or page size are not appropriate for the types of tables you are doing. These defaults can be changed with FORMAT statements.

You can use FORMAT statements along with your table request when you first produce a set of tables, or you can quickly reformat a set of tables without reprocessing your data. For example, after running a TPL job, you may see that the numbers in your tables are too large to be displayed with the default column size, or you may want to change the wording of a table title. Size and label details such as these can be easily changed with FORMAT.

A special FORMAT statement called **DATA TABLES** can be used to format your tables as a data file that can be used as input to other types of software, such as spread sheets or graphics programs.

Where to Put FORMAT Statements

FORMAT statements are prepared using an editor and saved in a file called a format request. The format request can be used along with the TPL table request file or separately as part of a rerun process.

Windows Note If you have the Windows version of TPL TABLES, you have the option of editing your tables interactively. See *TED Help* for instructions.

FORMAT statements can also be included in your TPL TABLES profile. This is a good approach if you want certain statements to apply to all of your tables whenever you run a TPL TABLES job.

Composition of FORMAT Statements

A FORMAT statement consists of two parts: a FOR clause and an ACTION clause. The ACTION clause specifies what is to be done to the tables. The FOR clause specifies where the ACTION clause should take effect.

A typical FOR clause is:

```
FOR TABLES 1 TO 3  ROWS 1, 3 AND 5  COLUMN 5 :
```

Some typical ACTION clauses are:

```
STUB WIDTH = 25;
COLUMN WIDTH = 14;
REPLACE MASK WITH $99,999.99;
PAGE WIDTH = 140;
```

FOR clauses are optional. If there is no FOR clause before an ACTION, the FOR clause from a previous statement applies to the new ACTION. If there is no previous FOR clause, the ACTION applies to the entire set of tables in the request.

The following set of FORMAT statements shows how ACTIONS can be grouped with FOR clauses. The first two statements apply to all tables unless specifically changed by subsequent FOR clauses.

```
PAGE WIDTH = 120;
STUB WIDTH = 25;

FOR TABLES 2 AND 3:
        DELETE EMPTY COLUMNS;
```

```
        REPLACE STUB HEAD WITH 'Average Income';
        STUB WIDTH = 20;

FOR TABLE 1 COLUMNS 1, 3, AND 5:
        COLUMN WIDTH = 12;
        REPLACE MASK WITH 999.9;

FOR TABLE 2 VARIABLE TOTAL:
         REPLACE LABEL WITH 'All Employees';
```

Action Levels

Different types of ACTION clauses take effect at different levels: request, table, wafer, column, row or cell.

For example,

```
STUB WIDTH = 30;
```

is applicable at the table level; that is, you cannot specify one stub width for wafer 1 of a table and a different stub width for wafer 2 of the same table. You can specify different stub widths for different tables in the same request.

If any part of a FOR clause is inapplicable for an associated ACTION because of the level of the ACTION, the term in the FOR clause is ignored. Consider the FORMAT statement:

```
FOR TABLE 3  ROWS 3 TO 10  COLUMNS 1 TO 6  WAFER 1 :
        DELETE COLUMNS;
```

This statement will cause columns 1 through 6 to be deleted from table 3. The row and wafer restrictions are inapplicable, so they will be ignored. If TABLE 3 were omitted from the FOR clause, columns 1 through 6 would be deleted from all tables in the request.

When an ACTION is specified with a FOR clause that does not apply, TPL TABLES follows the statement with a message in the OUTPUT file. If you find that some of your FORMAT statements are applied (or not applied) in the way that you expect, check the OUTPUT file for messages.

Action Conflicts

If two or more conflicting actions are specified for the same part of a request, the last one specified will win. An example of actions in conflict is two column widths specified for the same column:

```
COLUMN WIDTH = 12;
FOR TABLE 1  COLUMNS 1 TO 5:  COLUMN WIDTH = 8;
```

In this case, all columns in the request will have a width of 12 except columns 1 to 5 in table 1. Those columns will have a width of 8.

Action Size Specifications

For any action that specifies a size, the size is specified by

```
amount  [ unit ]
```

where amount is a number and unit is optional. If no unit is specified, characters are assumed. If a unit is specified, the amount can be a decimal number and the unit can be expressed as inches, centimeters or points using any of the following words or abbreviations:

```
inch
inches
in
ins
cm
points
pt
pts
```

Fractional sizes must be specified as decimal numbers. For example,

```
STUB WIDTH = 2.5 IN;
```

What can be in the FOR Clause?

The following elements can be referenced in a FOR clause:

```
ALL
TABLE
WAFER
ROW
COLUMN
VARIABLE
CONDITION
```

- To apply an action to an entire table request, you can specify:

  FOR ALL:

- Tables can be referenced by number, name or the word ALL:

  FOR TABLE table name(s): or

  FOR TABLE table number(s): or

  FOR TABLES ALL:

- Wafers, rows and columns can be referenced by numbers or the word ALL.

Note If any **rows** of a table do not appear in the table because they are **empty** (do not have any data) or because the rows are **ranked**, you cannot determine row numbers by counting data rows in the printed table. You can find the row numbers for **PRINTED ROWS** in the OUTPUT file.

- Variables can be specified by

  FOR VARIABLE variable name:

- Control variable conditions can be specified by

  FOR CONDITION variable name(condition number): or

  FOR CONDITION variable name(condition name):

Multiple variables or conditions can be referenced in the same FOR clause, with or without commas between them. Examples are:

```
FOR VARIABLE A VARIABLE B :
FOR VARIABLES A, B, C :
FOR CONDITIONS VAR(1), VAR(2), VAR1(1) :
FOR CONDITIONS VAR1(1,2), VAR2(1) :
```

Variable references only have meaning when used with the actions REPLACE LABEL and REPLACE MASK; condition references only have meaning when used with REPLACE LABEL.

Ranges of values can be expressed in the FOR clause using the word TO. Commas, equal signs, and the word AND are optional. For example,

```
FOR TABLE C3, ROWS 3 TO 10, COLUMNS = 1, 3, 6,
        WAFERS 1 AND 2:
```

means the same as

```
FOR TABLE C3  ROWS 3 TO 10  COLUMNS 1  3  6  WAFERS 1 2:
```

FOR clauses can include increments. For example,

```
FOR ROWS 5 TO 40 BY 5: .....
```

This clause means: In the range 5 to 40, begin with row 5 and take every 5th row. It means the same as the following clause.

```
FOR ROWS 5 10 15 20 25 30 35 40: .....
```

The Format Actions

The FORMAT ACTIONS are grouped by type and listed below. In the FORMAT reference section of this chapter, statements are ordered alphabetically and described in detail.

Note All FORMAT statements must end with a semicolon (;).

Control Page Size

```
PAGE LENGTH = size;
PAGE LENGTH = AUTOMATIC;
PAGE WIDTH  = size;
PAGE WIDTH  = AUTOMATIC;
PAPER = type;
```

Change Stub and Column Widths

```
COLUMN WIDTH = size;
COLUMN WIDTH = AUTOMATIC;
COLUMN WIDTH AUTOMATIC =  NO;                (or YES)
COLUMN WIDTH = AUTOMATIC MAXIMUM = size;
COLUMN WIDTH AUTOMATIC MAXIMUM = NO;         (or YES)

STUB WIDTH = size;
STUB WIDTH = AUTOMATIC;
STUB WIDTH AUTOMATIC = NO;                   (or YES)
STUB WIDTH = AUTOMATIC MAXIMUM = size;
STUB WIDTH AUTOMATIC MAXIMUM = NO;           (or YES)
```

Delete or Retain a Table or Part of a Table

DELETE	ALL RULES;
or	BANK DIVIDER
RETAIN	COLUMNS;
	DOWN RULES;
	EMPTY COLUMNS;
	EMPTY LINES;
	END RULE;
	FOOTNOTE;
	HEADING;
	HEADNOTE;
	LAST RULES;
	LEADING ZEROS;
	LEADING ZEROS EXCEPT FIRST;
	ROWS;
	SPANNER RULES;
DELETE	STUB;
or	TABLES;
RETAIN	TITLE;
	WAFER;
	WAFER LABEL;

Rules and Spanners

RULE WEIGHT = n;
BOLD RULE WEIGHT = n;
ROW SPAN;
DATA SPAN;
BANK DIVIDER; [1]
TOP RULE; [1]
END RULE; [1]
WAFER LABEL = DATA SPANNER;
WAFER LABEL = ROW SPANNER;
WAFER LABEL = HEADNOTE;
RULE AFTER ROW; [1]
BOTTOM RULE = DATA SPAN; [1]
BOTTOM RULE = ROW SPAN; [1]
DOWN RULE WEIGHT = n and/or DOUBLE; [1]
BOTTOM RULE = BOLD ROW SPAN; [1]
RULE AFTER ROW RULE WEIGHT = n and/or DOUBLE; [1]
REPLACE MASK FONT WITH font;
UNDERLINE ROW; [1]

[1] This command is no longer recommended. Go to the command to learn about it's replacement.

Column Banking

BANK AFTER COLUMN;
BANK AFTER COLUMN = NO; (or YES)
BANKS PER PAGE = n;

Row Banking

BANK AFTER ROW;
BANK AFTER ROW = NO; (or YES)
ROW BANKS PER PAGE = n;
BANK DIVIDER DELETE; [1]
BANK DIVIDER WEIGHT = n, DOUBLE or SINGLE;[1]

Control Stub Indentation and Placement

STUB CONTINUATION = size;
STUB INCREMENT = size;
STUB RIGHT; (or LEFT)
STUB START = size;
STUB STOP = size;

Mark Pages with Page Numbers and Other Information

PAGE MARKER marker specifications;
BOTTOM PAGE MARKER = marker specifications;

Page Margin Sizes *(see MARGIN)*

LEFT MARGIN = size;
RIGHT MARGIN = size;
TOP MARGIN = size;
BOTTOM MARGIN = size;

Column Margins

RULE MARGIN = size;
DATA RULE MARGIN = size;

Align Left, Right or Center

ALIGN COLUMN HEAD direction;
ALIGN HEADING LABEL direction;

[1] This command is no longer recommended. Go to the command to learn about it's replacement.

```
ALIGN HEADNOTE  direction;
ALIGN STUB HEAD  direction;
ALIGN STUB LABEL  direction;
ALIGN TABLE  direction;
ALIGN TITLE  direction;
ALIGN WAFER LABEL  direction;
```

Control Page Breaks and Contents

```
SKIP n LINES AFTER BANK;
SKIP n LINES AFTER TABLE;
SKIP n LINES AFTER WAFER;
EJECT AFTER ROW;
EJECT AFTER ROW  = NO;          (or YES)
EJECT AFTER TABLE;
EJECT AFTER TABLE = NO;         (or YES)
EJECT AFTER WAFER;
EJECT AFTER WAFER = NO;         (or YES)
```

Spacing and Presentation

```
SKIP amount AFTER ROW;
ROTATE;
EXTRA LEADING = n;
SCALE = n;
HEADING SPACE = n;
TABLE SPACE = n;
FONT = type/size;
```

Replace Labels, Masks and Values

```
REPLACE LABEL WITH label;
REPLACE MASK WITH mask;
REPLACE MASK WITH TEXT mask;
REPLACE MASK COLOR WITH color;
REPLACE MASK FONT WITH font;
REPLACE MASK FOOTNOTE WITH footnote;
REPLACE MASK MARKER WITH string;
REPLACE VALUE WITH value;
REPLACE VALUE WITH null;
REPLACE HEADNOTE WITH label;
REPLACE STUB HEAD WITH label;
REPLACE STUB CONTINUATION WITH label;
REPLACE TITLE WITH label;
REPLACE TITLE CONTINUATION WITH label;
REPLACE WAFER LABEL WITH label;
```

Replace Values with Rank Numbers

```
RANK ON VALUES;
```

Replace Column Divide Character and Stub Filler Character

```
REPLACE FILLER CHARACTER WITH 'char';
REPLACE DIVIDE CHARACTER WITH 'char'; [1]
```

Choose Rounding Method for Final Data Values

```
ROUND = UP;  (or EVEN)
```

Format Tables as a Data File

```
DATA TABLES; [1]
DATA TABLES ZERO FILL; [1]
```

Footnotes and Notes

```
SET FOOTNOTE footnote;
KEEP FOOTNOTE footnote;
KEEP DATA FOOTNOTE;
FOOTNOTE SEQUENCE = list;
FOOTNOTES ON EACH PAGE or WAFER;  (or ON LAST PAGE)
FOOTNOTE COLUMNS = n;
FOOTNOTE COLUMNS = n JUSTIFIED;          (or UNJUSTIFIED)
REPLACE MASK FOOTNOTE WITH footnote;
REPLACE FOOTNOTE / NOTE
RAISE FOOTNOTE SYMBOL = n;
MAXIMUM FOOTNOTE SYMBOL WIDTH = n;
SET NOTE note;
```

Heading Compression

```
COMPRESS HEADING;
SPANNER HEADING;
```

PostScript

```
POSTSCRIPT = YES;  (or NO)
```

[1] This command is no longer recommended. Go to the command to learn about it's replacement.

Shading and Color

```
COLOR defaults:
        DEFAULT COLOR = color;
        LABEL COLOR = color;
        RULE COLOR = color;
        SYMBOL COLOR = color;
SHADE table-element color/GREY;
COLOR = NO;  (or YES)
REPLACE COLOR color WITH FONT font;
REPLACE MASK COLOR WITH color;
```

Accessible HTML

```
HTML ACCESS;
```

Retain Extra Files

```
RETAIN TABLES FILE; [1]
RETAIN CELLFILE;
```

Print and Export Control (UNIX only)

Normally (in default mode), the system will prompt you at the end of a job to find out whether you want to print outputs or export files to other formats. You can use the following statements to select the print options in advance.

```
CSV OUTPUT = YES or NO or PROMPT;
EPS OUTPUT = YES or NO or PROMPT;
HTML OUTPUT = YES or NO or PROMPT;
ODS OUTPUT = YES or NO or PROMPT;
XLS OUTPUT = YES or NO or PROMPT;
PDF OUTPUT = YES or NO or PROMPT;
DATATABLE OUTPUT = YES or NO or PROMPT;
TEXT TABLE = YES or NO or PROMPT;
PRINT OUTPUT = YES or NO or PROMPT;
PRINT TABLES = YES or NO or PROMPT;
```

The default for all statements is PROMPT.

[1] This command is no longer recommended. Go to the command to learn about it's replacement.

Other Print Controls

```
CSV DIVIDER = divider;
```

Use of FORMAT Statements in Profile

FORMAT statement can be included in your TPL TABLES profile (the file called **profile.tpl**). These statements will be the default values for all table runs, or, if you have a profile in your current directory, the statements from that profile will determine the defaults for all jobs run from that directory.

Profile-only Statements

In addition to the standard FORMAT statements, there are a few statements that are used only in the TPL TABLES profile.

If TPL TABLES is already running when you insert or change a profile-only statement in profile.tpl, ***you must restart TPL TABLES*** to make the change effective.

Choosing Character Sets, Non-English Alphabets, and International Formats

```
CODEPAGE = name;
COUNTRY = name;
```

Memory Setting

```
CELL MEMORY = amount;
```

The following statements are only relevant to UNIX systems. With the exception of PRINT COMMAND and DISPLAY NAME, they are initially set when you install TPL TABLES.

PostScript Display (UNIX only)

```
DISPLAY NAME = PostsScript-displayer;
```

Printer Selection (UNIX only)

```
PRINT COMMAND = 'command';
```

Editor Specifications (UNIX only)

```
EDITOR NAME = editor_name;
EDITOR FILE = editor_file;
```

FORMAT LANGUAGE REFERENCE

Introduction

In the preceding section, we have provided an overview of the FORMAT language. In this section, we describe each FORMAT statement in detail. The statement descriptions are arranged in alphabetical order.

ALIGN COLUMN HEAD

Format There are three possible alignments for a column head.

```
ALIGN COLUMN HEAD LEFT;
ALIGN COLUMN HEAD RIGHT;
ALIGN COLUMN HEAD CENTER;
```

Meaning The COLUMN HEAD is the bottom level heading label for a column. The ALIGN COLUMN HEAD statement can be used to specify alignment for this level of heading labels without regard to the alignment of the heading labels above. A column head can be aligned left, right or center within its column. If a column head is more than one line long, all lines of the column head label will be aligned the same way.

You will find this statement particularly useful if you wish to change the default alignment for all column heads without entering an alignment specification into each individual column head label. If you have included an alignment of LEFT, RIGHT or CENTER in an individual label that is used as a column head, the alignment specified within the label will override the ALIGN COLUMN HEADS statement.

Level ALIGN COLUMN HEAD can be specified for selected columns.

Default

```
ALIGN COLUMNS HEAD CENTER;
```

The default for all heading labels is CENTER unless a different default has been set with the ALIGN HEADING LABELS statement.

Example

```
ALIGN COLUMN HEAD RIGHT;
FOR TABLE 2 COLUMNS 1 TO 3:  ALIGN COLUMN HEAD CENTER;
```

Effect All column heads will be aligned to the right edge of the columns, except in table 2 where the columns heads will be centered in the first 3 columns.

ALIGN HEADING LABELS

Format There are three possible alignments for heading labels.

```
ALIGN HEADING LABELS LEFT;
ALIGN HEADING LABELS RIGHT;
ALIGN HEADING LABELS CENTER;
```

The following are equivalent to the sequence of words ALIGN HEADING LABELS:

```
ALIGN HEADING
ALIGN HEAD LABELS
ALIGN HEAD
```

Meaning The labels in the table heading can be aligned to the left, right or center. If a heading label is more than one line long, all lines of the heading label will be aligned the same way.

You will find this statement particularly useful if you wish to change the default alignment for all heading labels without entering an alignment specification into each individual label. If you have included an alignment of LEFT, RIGHT or CENTER in an individual label that is used as a heading label, the alignment specified within the label will override the ALIGN HEADING LABELS statement.

Default alignment of stub heads is the same as the default alignment for heading labels unless you use a separate statement called ALIGN STUB HEAD.

See also the statement called ALIGN COLUMN HEAD. This statement can be used to set a different default alignment for the labels at the lowest level of the heading.

Level ALIGN HEADING LABELS can be specified for individual tables.

Default

```
ALIGN HEADING LABELS CENTER;
```

Example

```
ALIGN HEADING LABELS RIGHT;
FOR TABLE 3:  ALIGN COLUMN HEAD CENTER;
```

Effect All heading labels will be aligned to the right, except the column head labels in table 3. These will be centered.

ALIGN HEADNOTE

Format There are three possible alignments for a headnote.

```
ALIGN HEADNOTE LEFT;
ALIGN HEADNOTE RIGHT;
ALIGN HEADNOTE CENTER;
```

Meaning The headnote is a label that can be placed immediately above the table heading with a REPLACE HEADNOTE statement. This label can be aligned left, right or center. If it is more than one line long, all lines will be aligned the same way.

You will find this statement particularly useful if you wish to change the default alignment for all headnotes without entering an alignment specification into each individual headnote. If you have included alignment of LEFT, RIGHT or CENTER in an individual headnote label, that alignment will override any label alignment statements.

Level Headnote alignment can be controlled at the table level.

Default

```
ALIGN HEADNOTE LEFT;
```

Example

```
REPLACE HEADNOTES WITH 'Standard Headnote';
ALIGN HEADNOTES RIGHT;
FOR TABLE 2:  REPLACE HEADNOTE WITH
        CENTER 'Special Headnote';
```

Effect All tables will have the same right-aligned headnote except for Table 2 where the headnote has a different text and an alignment of CENTER. Since the CENTER alignment is part of the headnote label for Table 2, this alignment overrides the ALIGN HEADNOTE statement.

ALIGN STUB HEAD

Format There are three possible alignments for a stub head.

```
ALIGN STUB HEAD LEFT;
ALIGN STUB HEAD RIGHT;
ALIGN STUB HEAD CENTER;
```

Meaning The stub head is a label that can be placed in the corner area above the table stub and beside the heading. This label can be aligned left, right or center within its area. If the stub head is more than one line long, all lines of the stub head will be aligned the same way.

You will find this statement particularly useful if you wish to change the default alignment for all stub heads without entering an alignment specification into each individual stub head. If you have included an alignment of LEFT, RIGHT or CENTER in an individual stub head label, that alignment will override any label alignment statements.

Level Stub head alignment can be controlled at the table level.

Default

```
ALIGN STUB HEAD CENTER;
```

The default alignment for all labels in the heading area, including the stub head, is CENTER unless a different default has been set with the ALIGN HEADING LABELS statement.

Example

```
ALIGN HEADING LABELS RIGHT;
ALIGN STUB HEAD LEFT;
FOR TABLE 2:  ALIGN STUB HEAD RIGHT;
FOR TABLE 4:  REPLACE STUB HEAD WITH
        'Total Population' CENTER;
```

Effect The first statement ALIGN HEADING LABELS RIGHT; changes the default alignment for the heading area, including the stub head, to RIGHT. The next statement ALIGN STUB HEAD LEFT; overrides the first alignment statement so that all stub heads will default to LEFT alignment. However, in the second table, the stub head will be aligned RIGHT. In the fourth table, the stub head contains an explicit specification of CENTER in the label. This specification will override the default stub head alignment that was set at LEFT, so that the stub head in the fourth table will be centered.

ALIGN STUB LABELS

Format There are three possible alignments for stub labels.

```
ALIGN STUB LABELS LEFT;
ALIGN STUB LABELS RIGHT;
ALIGN STUB LABELS CENTER;
```

Meaning Stub labels can be aligned to the left, right or center of the stub. You will find the ALIGN STUB statement particularly useful if you wish to change the default alignment for all tables without entering an alignment specification into each individual stub label. If you have included an alignment of LEFT, RIGHT or CENTER in an individual stub label, the ALIGN STUB LABELS specification will not apply to that stub label.

Note ALIGN STUB LABEL does not affect the alignment of stub labels that have the SPANNER attribute.

Left Alignment

If the stub labels are aligned to the LEFT (the default alignment), standard indentation rules apply to nested and multi-line labels unless you override the standard treatment with other FORMAT statements. The left-most position for the beginning of a stub label is normally the first stub position if the stub is on the left. If you have specified STUB RIGHT; the left-most stub position is normally indented 5 positions from the left edge of the stub. The left-most position can be changed with a STUB START statement.

Right and Center Alignment

If you choose CENTER or RIGHT aligment for the stub labels, all lines of stub labels will be aligned the same way. All other indentations, either standard or user-specified, will be ignored.

Level The default alignment for stub labels can be specified at the table level.

Default

```
ALIGN STUB LABELS LEFT;
```

Example

```
ALIGN STUB LABELS RIGHT;
FOR VARIABLE JOBS:  REPLACE LABEL WITH
        'Occupations' CENTER;
```

Effect The stub labels will be aligned at the right edge of the stub except for labels such as 'Occupations' where a different alignment, included in the label, will override the default that was set with ALIGN STUB LABELS RIGHT.

ALIGN TABLE

Format There are three possible table alignments.

```
ALIGN TABLE LEFT;
ALIGN TABLE RIGHT;
ALIGN TABLE CENTER;
```

Meaning A table is aligned (LEFT, RIGHT, or CENTER) between the left and right margins. If the table is too wide to fit on the page, it will automatically be divided into as many sections as necessary (one page per section) with each section aligned the same way. If a table is divided into sections by a BANK statement, each section will be aligned the same way.

Level Table alignment can be controlled at the table level. Table alignment cannot change within a table.

Default

```
ALIGN TABLE CENTER;
```

Example

```
ALIGN TABLES LEFT;
FOR TABLE 3:  ALIGN TABLE CENTER;
```

Effect All tables except the third will be aligned with the left margin on the page. The third table will be centered.

ALIGN TITLE

Format There are three possible alignments for table titles.

```
ALIGN TITLES LEFT;
ALIGN TITLES RIGHT;
ALIGN TITLES CENTER;
```

Meaning The table title can be aligned with the left or right edges of the table, or it can be centered within the width of the table. If the title is more than one line long, all lines of the title will be aligned the same way.

You will find this statement particularly useful if you wish to change the default title alignment for all tables without entering an alignment specification into each individual table title. If you have included an alignment of LEFT, RIGHT or CENTER in an individual table title, the ALIGN TITLE specification will not apply to that table title.

Level Title alignment can be controlled at the table level.

Default

```
ALIGN TITLE LEFT;
```

Example

```
ALIGN TITLES CENTER;
FOR TABLE 2:  ALIGN TITLE RIGHT;
```

Effect The table title will be centered for all tables except the second. For the second table, the table title will be aligned with the right edge of the table.

ALIGN WAFER LABELS

Format There are three possible alignments for wafer labels.

```
ALIGN WAFER LABELS LEFT;
ALIGN WAFER LABELS RIGHT;
ALIGN WAFER LABELS CENTER;
```

Meaning A wafer label can be aligned with the left or right edges of the table, or it can be centered within the width of the table. If the wafer label is more than one line long, all lines of the wafer label will be aligned the same way.

You will find this statement particularly useful if you wish to change the default alignment for all tables without entering an alignment specification into each individual wafer label. If you have included an alignment of LEFT, RIGHT or CENTER in an individual wafer label, the ALIGN WAFER LABELS specification will not apply to that wafer label.

Level The default alignment for wafer labels can be controlled at the table level.

Default

```
ALIGN WAFER LABELS LEFT;
```

Example

```
ALIGN WAFER LABELS RIGHT;
FOR TABLE 2 WAFER 1:  REPLACE WAFER LABEL
        WITH 'All regions' LEFT;
```

Effect The wafer labels will be aligned with the right edge of the tables except in the first wafer of table 2 where the specification of LEFT in the REPLACE WAFER LABEL statement overrides the default that was set with ALIGN WAFER LABELS RIGHT.

AUTOMATIC STUB AND COLUMN WIDTHS

Format

```
STUB WIDTH = AUTOMATIC;
STUB WIDTH = AUTOMATIC  MAXIMUM = n [ unit ];
COLUMN WIDTH = AUTOMATIC;
COLUMN WIDTH = AUTOMATIC  MAXIMUM = n [ unit ];
```

where n is a number that specifies a width, and unit is optional. If no unit is specified, characters are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as inches, cm or points.

The word IS can be used in place of =. Both are optional and can be left out altogether. AUTO is a synonym for AUTOMATIC and MAX is a synonym for MAXIMUM.

See also the statements STUB WIDTH and COLUMN WIDTH to set specific widths for stub and columns.

The following options are also available and are generally used with NO to reverse the effect of earlier automatic width statements.

```
STUB WIDTH AUTOMATIC = NO;                  (or YES)
STUB WIDTH AUTOMATIC MAXIMUM = NO;          (or YES)
COLUMN WIDTH AUTOMATIC =  NO;               (or YES)
COLUMN WIDTH AUTOMATIC MAXIMUM = NO;        (or YES)
```

Meaning

You can use these statements if you wish to "stretch" tables to the full width of the page minus the margins; if you wish to have all tables be the same width, regardless of the number of columns; or if you wish to have all banks of a table be the same width, even if they have different numbers of columns.

If you specify COLUMN WIDTH AUTOMATIC; extra space is added to the columns so that the table takes up the full width of the page minus the left and right margins and the stub width.

If you specify STUB WIDTH AUTOMATIC; the stub will be expanded so that the table takes up the full width of the page after deducting the left and right margins and the space required for the columns.

In general, only one of these two statements would be applied to the same table. If you specify AUTOMATIC for both the stub and the column

widths, space will be added only to the columns, unless you also provide a MAXIMUM clause to limit the amount that can be added to the columns. In that case, extra space will be added to the stub.

In some cases, COLUMN WIDTH AUTOMATIC; will make columns wider than is desirable. This is especially likely when a banked table has only a small number of columns in the last bank. COLUMN WIDTH AUTOMATIC MAXIMUM n; can be used to limit the effect of automatic column widths so that no column will have a width greater than n.

Note If you have set an explicit width for a column that is wider than the maximum automatic value, this column will not have its width reduced.

Note If you specify a maximum automatic value for columns, the final width of the table may not be as wide as the page width minus the margins and stub.

STUB WIDTH AUTOMATIC MAXIMUM n; can be used to limit the expansion of the stub to a maximum of n.

Level These statements can be specified at the individual table level.

Default

```
STUB WIDTH = 20; and

COLUMN WIDTH = 10;
```

unless explicitly specified otherwise.

Example Assume that for a table with 5 columns, we have the following FORMAT statements:

```
PAGE WIDTH = 100;
LEFT MARGIN = 5;
RIGHT MARGIN = 5;
STUB WIDTH = 15;
COLUMN WIDTH AUTOMATIC;
```

Effect The available space for the columns will be 100 - 5 - 5 - 15 = 75 (i.e. the page width minus the margins and the stub). Each column will be expanded from the default width of 10 to a width of 15 so that the full page width will be used.

Example

```
COLUMN WIDTH = 20;
COLUMN WIDTH AUTOMATIC;
FOR TABLES 2 and 3: COLUMN WIDTH AUTOMATIC NO;
```

Effect Column width is first set to 20 but then is set to automatic for all tables. The last statement turns off the automatic column width for tables 2 and 3, so tables 2 and 3 will have column width of 20.

Restrictions Text tables may not produce exactly equal column widths for COLUMN WIDTH AUTOMATIC; If the available space divided by the number of columns yields a fractional number, some adjustments must be made to allow for the fact that all characters and spaces have the same width.

If you specify STUB WIDTH AUTOMATIC; for a banked table, you may get some undesirable effects. All banks of a table are formatted with the same number of lines. If one bank has a relatively narrow stub, a stub label may need to be broken into two lines to fit in the stub. Another bank with fewer columns may have a wider stub, but the stub labels will be broken into multiple lines in the same way as they were for the bank with the narrow stub. This could produce a strange-looking result.

If these statements are used with PAGE WIDTH AUTOMATIC; they will be ignored, regardless of the order of the statements.

BANK AFTER COLUMN

Format A FOR clause is required to identify the column(s) where banking should take place.

```
FOR COLUMN n [optional additional column numbers]:
BANK AFTER COLUMN;
```

Meaning BANK allows you to specify a break point for a table so that a wide table can be split into sections called ***column banks***. Each section is printed on a separate page with all necessary labels (including all stub labels) repeated for each section of the table.

The following option is also available and is generally used with NO to reverse a previous BANK.

```
BANK AFTER COLUMN = NO;          (or YES)
```

Level Bank points are specified by column but are controlled at the table level. Bank points cannot change within a table.

Default If the table is too wide for the page, it is banked automatically into as many sections as necessary.

Example

```
FOR COLUMNS 4, 8:  BANK AFTER COLUMNS;
```

Effect The first page of the table will contain columns 1-4; the second page of the table will contain columns 5-8. The remaining columns (assuming there are few enough to fit) will be on the third page of the table.

Example

```
FOR TABLES ALL COLUMNS 4, 8:  BANK AFTER COLUMNS;
FOR TABLE 3: BANK AFTER COLUMNS = NO;
```

Effect All tables except table 3 will be banked after columns 4 and 8.

Restrictions If you specify a break point beyond the end of the table, the BANK statement will be ignored. For example, if a table has ten columns, the statement

```
FOR COLUMN 15:  BANK AFTER COLUMN;
```

will be ignored.

BANK AFTER ROW

Format FOR *row specification*: BANK AFTER ROW;

Meaning When ROW BANKS PER PAGE is specified for a table, banking occurs at the bottom of a page. With BANK AFTER ROW, you can choose the bank points by specifying the data rows where the banking should occur.

Note If you use the word EJECT instead of BANK, the results will be the same.

Precise control of bank points can be useful when you want to prevent banking in the middle of logical groupings of rows. It can also be useful for balancing the number of rows in each bank.

Note that if any rows of the table are not printed because they are empty (do not have any data) or because the rows are ranked, you cannot determine row numbers by counting data rows in the printed table. You can find the row numbers for **PRINTED ROWS** in the OUTPUT file. If you reference an empty row in the FOR clause, the bank will occur before the next row that has data.

Note If you have ranked rows and reference an empty row, the BANK AFTER ROW statement will have no effect. For ranked rows, you need to reference a row that has data.

The following option is also available and is generally used with NO to reverse a previous BANK AFTER ROW.

```
BANK AFTER ROW = NO;          (or YES)
```

Level Row banking applies to entire tables, but bank points can be specified for individual rows.

Default Bank points are determined automatically based on the number of rows that can fit on a page.

Example

```
ROW BANKS PER PAGE = 3;
FOR TABLE 1 ROW 35:  BANK AFTER ROW;
```

Effect The table will break after row 35 and the table will continue with another bank on the same page if there is enough space for another bank. If there is not enough space, the new bank will start on the next page. All other bank points will be determined automatically.

Note The default for row banking is ROW BANKS PER PAGE = 1. Thus, if BANK AFTER ROW is used without a ROW BANKS PER PAGE statement to specify multiple banks per page, there will be no row banking and the table will simply break and go to a new page instead of banking after the specified row(s).

Note Row numbering restarts with each wafer. For a table with multiple wafers, if no wafer specification is included in the BANK AFTER ROW statement, the statement will apply to all wafers.

Restrictions You must include a FOR clause to specify the rows where the banking should occur. Otherwise, the BANK AFTER ROW statement will be ignored.

If you have multiple wafers in a row-banked table and there is room for more than one wafer on a page, you cannot eject or bank at the end of one wafer to force the beginning of the next wafer to start on a new page.

BANK DIVIDER

This statement has been replaced by RETAIN BANK DIVIDER which has more options.

Format

```
BANK DIVIDER  WEIGHT n;
BANK DIVIDER  DOUBLE or SINGLE;
BANK DIVIDER  DOUBLE or SINGLE  WEIGHT n;
BANK DIVIDER  DELETE;
```

where n is a number in points (1 point = 1/72 of an inch).

Meaning When a table is row banked with multiple banks side by side on a page, rules are usually added between the banks to divide them. These commands control the weight of the divider, whether the divider is a double rule, or whether it is deleted entirely.

Level Bank dividers can be controlled at the individual table level.

Default

```
BANK DIVIDER DOUBLE WEIGHT .5;
```

Example

```
For Table ONE: BANK DIVIDER SINGLE WEIGHT 2.0;
```

Effect Banks are divided by a single bold rule.

BANKS PER PAGE

Format BANKS PER PAGE = n;

where n is a number. The word IS can be used in place of =. Both are optional and can be left out altogether.

Meaning If a table is too wide to fit on a page, it is automatically broken into sections called ***column banks***. Banking can also be requested explicitly with the BANK AFTER COLUMN statement. By default, each bank begins on a new page. The BANKS PER PAGE statement can be used to print multiple banks on a page.

One line is skipped between banks unless you request a different spacing with the SKIP AFTER BANKS statement. With the statement SKIP 0 LINES AFTER BANKS; the banks will be joined with no space between banks, and the stub head will be deleted for banks after the first. See the SKIP AFTER BANKS statement for details.

Example FOR COLUMN 2: BANK AFTER COLUMN;
BANKS PER PAGE = 2;

U.S. Waterborne Exports

By Country

Country of Ultimate Destination	Short Tons of 2000 Lbs.	
	Liner	Tanker
Total	248,745	4,146,065
MEXICO, CENTRAL AMER. & CARIB.		
Total	3,410	527,313
0 BULK	1,932	526,500
1 GENERAL	1,478	813
SOUTH AMERICA		
Total	104,772	1,348,266
0 BULK	59,425	1,319,128
1 GENERAL	45,347	29,138

Country of Ultimate Destination	Short Tons of 2000 Lbs.	
	Tramp	Total
Total	15,334,746	19,729,556
MEXICO, CENTRAL AMER. & CARIB.		
Total	584,649	1,115,372
0 BULK	576,483	1,104,915
1 GENERAL	8,166	10,457
SOUTH AMERICA		
Total	2,664,773	4,117,811
0 BULK	2,587,432	3,965,985
1 GENERAL	77,341	151,826

Table alignment is determined for each individual page of a banked table and depends on the width of the widest bank on the page. Narrower banks are aligned with the stub of the widest bank.

Table titles, wafer labels, headnotes and footnotes will appear only once on a page regardless of the number of banks. They will be aligned with the widest bank on the page.

Note If you are requesting multiple banks per page, we recommend that you use equal width banks whenever possible. In general, unequal width banks on the same page do not look good. This is especially true if you have specified **SKIP 0 LINES AFTER BANKS.**

Note All banks on a page will take up the same amount of vertical space. If you have different width banks on the same page, you may find that the page ends sooner than you expect, especially if you have a very narrow bank that requires the table title to be broken into several lines. In addition, if one bank has a very long heading label that must be broken into several lines, the space requirement for that label will apply to all of the banks. Each bank may then take more vertical space than you expect.

Level BANKS PER PAGE can be specified at the individual table level.

Default

```
BANKS PER PAGE = 1;
```

Example

```
BANKS PER PAGE = 3;
SKIP 3 LINES AFTER BANKS;
```

Effect Each page of the table will contain 3 banks with 3 blank lines between the banks. If the number of banks in the table is not a multiple of 3, then the last page will contain fewer than 3 banks.

Restrictions There must be enough vertical space on the page for each bank to contain at least one line of data.

BOLD RULE

Format BOLD RULE WEIGHT = *n*;

Meaning The statement controls the thickness of bold rules. The rules at the start (TOP RULE) and end (END RULE) of a table are by default bold rules. Other rules may be designated as bold or these rules can be assigned a different weight. Otther properties of these rules should be set using the RETAIN RULE options.

Level BOLD RULE WEIGHT can be specified at the individual table level.

Default BOLD RULE WEIGHT = 1.2; (in **pts** where 1 pts = 1/72 inches)

Example

Default Bold Rule Weight

	Race of Householder	
	White	
	Hispanic Origin of Householder	
	Hispanic	Not hispanic
Average Income		
Regions		
Northeast	21,358	36,708
Midwest	23,091	31,161
Southeast	24,598	31,954
West	24,944	33,865

Bold Rule Weight = .5

	Race of Householder	
	White	
	Hispanic Origin of Householder	
	Hispanic	Not hispanic
Average Income		
Regions		
Northeast	21,358	36,708
Midwest	23,091	31,161
Southeast	24,598	31,954
West	24,944	33,865

BOTTOM RULE SPAN

This Statement has been replaced by
RETAIN LAST RULE ***rule-options***

Format

BOTTOM RULE = [BOLD] ROW SPAN;
BOTTOM RULE = DATA SPAN;

The word IS can be used in place of =. Both are optional and can be left out altogether.

Meaning

This statement applies to the horizontal rule at the bottom of a table on pages other than the last page of the table. Normally, the last bottom rule on the last page of a table is a bold line that spans the entire table. Other bottom rules are normal thickness and span only the data part (DATA SPAN) to indicate that the table is not finished. An exception occurs if you have specified FOOTNOTES EACH PAGE. In this case all bottom rules span across the entire table.

With BOTTOM RULE = ROW SPAN; you can request that all bottom rules span across the entire table. If you want all of these rules to be bold, you can add the optional word BOLD to the statement.

Level

The BOTTOM RULE statement can be specified for individual tables.

Default

BOTTOM RULE = DATA SPAN;

unless you have specified FOOTNOTES EACH PAGE.

Example

FOR TABLE 1: BOTTOM RULE = ROW SPAN;

Effect

For all pages of the first table, the bottom rule will span across the entire table, but it will be bold only on the last page of the table.

Note

If you are also using the statement SKIP 0 LINES AFTER BANKS; the table will look better with the default of BOTTOM ROW = DATA SPAN. If you are using SKIP 1 (or more) LINES AFTER BANKS; the statement BOTTOM RULE = BOLD ROW SPAN; will produce a more pleasing result.

CELL MEMORY (PROFILE only)

Format CELL MEMORY = n;

where **n** is the number of bytes to be used for cell memory.

Meaning *In general, you do not need to concern yourself with the amount of memory being used by TPL Tables*, but certain types of jobs may run more efficiently if extra memory is available for the generation of table cells. If the cell generation or subcell generation step of a tables run contains the statement:

The cell buffer was emptied *n* times.

where ***n*** is greater than **1** or **2**, then your job will probably run faster if you increase the value in the statement. The messages regarding unloading of the cell buffer are contained in the **output** file. For all jobs, the cell buffer is unloaded at least once.

Examples of situations in which the extra cell memory may be beneficial to you include:

- if you have jobs that must calculate many medians or other quantiles,
- if you are producing extremely large tables in which there is data for all or most of the table cells.

Default CELL MEMORY = 2,000,000;

The default value (about 2 megabytes) is set at installation time. This is a reasonable value for most jobs.

Example CELL MEMORY = 4,000,000;

Note If you are using the optional COUNTRY statement, you should enter the value without commas or according to the convention set for your country.

Restrictions *The value chosen must be below the actual memory available on your computer.* Otherwise, it will impact other processes running on your computer without speeding up TPL TABLES and/or will cause TPL Tables to run very slowly. Pick the smallest value that keeps the cell buffer from being emptied many times.

CODEPAGE (PROFILE only)

CODEPAGE determines the character set and sort order for your requests and tables. It is most often used for non-English languages that have alphabetic characters not available in the default CODEPAGE.

See also the statement called COUNTRY to set standards for features such as currency symbols and date/time formats.

Important If you add a CODEPAGE or COUNTRY statement to your profile, change a CODEPAGE or COUNTRY statement in your profile, or make changes to **country.tpl**, ***you need to restart TPL*** to activate the changes. When running a table request, you must use the same CODEPAGE and COUNTRY statements that you used when processing your codebook. Otherwise, you will have conflicting standards. In particular, conflicts in CODEPAGE will cause the sort order to be scrambled.

Note If you use CODEPAGE and/or COUNTRY statements, place them at the beginning of your profile.

Format

```
CODEPAGE = cp-name;
```

where **cp-name** is a CODEPAGE name. The word IS can be used in place of =. Both are optional and can be left out altogether.

Level CODEPAGE applies to all tables.

Windows Default CODEPAGE = WIN88591;

UNIX Default CODEPAGE = ISO88591;

Meaning The CODEPAGE statement determines the following:

1. the character set that includes the alphabet you wish to use in names and labels,
2. the character set for text tables, and
3. the sort sequence for the character set.

The character sets associated with different CODEPAGES are contained in files that are installed in the TPL system directory. These files have names that end with **.cp**. The names correspond to the supported codepages. For example, for CODEPAGE = WIN88591; the character set information is

contained in the file **WIN88591.cp**. For CODEPAGE = ISO88591; the character set information in contained in the file **ISO88591.cp**.

The **.cp** files are ASCII text files that can be printed or displayed on the screen but you should not change them. Please tell us if you think that a change should be made for your alphabet.

Alphabet for Names

The TPL alphabet depends on the CODEPAGE. The default CODEPAGE is adequate for many, but not all, languages. If you need additional letters, look at the CODEPAGES in the Appendix to find an alphabet that you can use.

The Character Set for Printing

TPL automatically provides the printable characters for the selected CODEPAGE. For text tables, the characters must be stored on the printer.

The Sort Sequence

The proper order for sorting depends on the character set used. TPL will use the sequence that goes with the character set selected by the CODEPAGE statement.

The sort sequences for all character sets are stored in a file called **sort.tpl** that is installed in the TPL system directory. This is an ASCII text file that you can print or display on the screen. It can be edited according to the instructions included at the beginning of the file. We would appreciate your telling us if you think that changes should be made in the **sort.tpl** file that we distribute with the TPL software.

If You Need to Select a CODEPAGE

Consult the character set tables in the Appendix called "Character Sets" and use the codepage name from any table that contains the characters that you need.

COLOR Defaults

Format

```
DEFAULT COLOR = r g b;
LABEL COLOR = r g b;
RULE COLOR = r g b;
SYMBOL COLOR = r g b;
```

where r, g and b, are numbers between 0 and 100 (inclusive) which specify red, green, and blue components of color.

Meaning

COLOR defaults can be set for all characters and rules (lines) used in tables. The defaults are applied as follows.

DEFAULT COLOR is the print color for the entire table if no other colors are specified. If RULE COLOR and LABEL COLOR are specified, the DEFAULT COLOR remains as the default color for table cells.

RULE COLOR is the print color for rules. It applies to all rules, including rules added by the FORMAT statement RULE AFTER ROW and rules included with spanner labels. If no explicit RULE COLOR is specified, rules are printed in the default color.

LABEL COLOR is the print color for all text in tables except character strings in cell masks. These strings are printed in the default color. If no explicit LABEL COLOR is specified, all labels, titles and footnote texts are printed in the default color.

SYMBOL COLOR is the print color for all footnote symbols. If SYMBOL COLOR is not set explicitly, the default label color is used for symbols.

In cases where color specifications are entered directly into ***individual*** table elements such as labels, masks or footnotes, these individual specifications will take precedence over the ***default*** COLOR specifications.

COLOR defaults apply only to characters and rules. For background shading in color or grey, see the SHADE statement (or COLOR shading).

Note on Cell Color

There is no default color statement that applies only to table cells. If you wish to change the cell color without affecting other table elements that are to be displayed in the default color, you can use the FORMAT statement, REPLACE MASK COLOR, to replace the mask color for entire tables. This statement affects only cell color and does not change the other mask characteristics of a cell.

Note on Underlining

The color for underlining is determined by the labels or masks to which the underlining applies if it is part of the font for the labels or masks. If you have used the FORMAT statement UNDERLINE ROW, the underlining will be in the color of the DEFAULT COLOR.

Level All COLOR statements can be specified for individual tables.

Default The default color is black.

Example

```
DEFAULT COLOR = 0 20 99;
FOR TABLE 1:  REPLACE TITLE WITH
        COLOR 100 0 0 'Red table title';
FOR TABLE 2:  RULE COLOR = 100 0 0;
```

Effect All tables will be printed in the default color 0 20 99 (a shade of blue) except as follows. The first table will have a title in the color 100 0 0 (red). The rules in the second table will be printed in the color 100 0 0 (red). The rest of the second table will be printed in the default color.

Alternate Format for the COLOR Statements

Colors can also be referenced by name where the colors have been defined in a file called **color.tpl**.

Format

```
DEFAULT COLOR = color;
LABEL COLOR = color;
RULE COLOR = color;
SYMBOL COLOR = color;
```

where **color** is a user-selected name that has been assigned to a color definition in the **color.tpl** file.

The **color.tpl** file is installed as part of the TPL TABLES system with several colors already defined. You can customize this file to add the colors of

your choice. For complete details, see the section called "General Information about Color" in the Color chapter.

Example

```
DEFAULT COLOR = BLUE;
FOR TABLE 2:  RULE COLOR = BROWN;
FOR TABLE 2, COLUMN 1:  REPLACE MASK WITH
        COLOR RED 999.9;
```

Effect All tables will be printed in the default color BLUE except the second table. The rules in the second table will be printed in the color BROWN. Since the color RED is included within the mask for the first column of the second table, the data values in that column will be printed in RED. The rest of the second table, the labels and the other data values, will be printed in the default color.

COLOR = NO

Format

```
COLOR = NO;
REPLACE COLOR c WITH FONT f;
```

where **c** is a color name or r g b specification and **f** is a font name and optional font size.

Meaning In some cases a table is designed to be printed on a color printer but must be previewed on a monochrome printer. Colors are printed on a monochrome printer as shades of grey. The resulting table is often hard to read, and different colors sometimes convert to the same shade of grey. These two additional statements may be added to your FORMAT request to deal with this.

Level These statements apply to the entire request.

Default COLOR = YES; is the default. If you have not specified COLOR = NO; the REPLACE COLOR statements are ignored.

If COLOR = NO; is selected, and no REPLACE COLOR statement is provided, all color information is ignored and the tables are printed in black and white.

If COLOR = NO and REPLACE COLOR statements are included, the tables are still printed in black and white but the REPLACE COLOR statements are used to substitute special fonts for color.

Example Suppose you have a table intended for printing on a color printer and suppose that you wish to use a RED data mask to emphasize certain table cells. You need to preview this table on a monochrome printer but you want the red cells to stand out. You can accomplish this by adding the following two statements to your FORMAT request or profile.

```
COLOR = NO;
REPLACE COLOR RED WITH FONT HB;
```

Now all red cells will be displayed in Helvetica Bold. When you are ready to print on your color printer, just replace COLOR = NO; with COLOR = YES; The REPLACE statements need not be removed since they will be ignored.

Restrictions Changing the color of a string of characters does not change its length, but changing the font and especially the font size of the string does change its

length. REPLACE COLOR statements are intended for use in previewing tables before final printing on a color printer. To aid in doing this, all table layout is done before the font changes are applied. Consequently, if a wider font is specified in a REPLACE COLOR statement, some character strings may be too wide to fit in their space. Thus, they may overlay some of the adjacent rules or stub filler dots. If the font size is not changed, this will rarely happen but it is possible.

Alternate Approach

If you want to produce a publication-quality table that emphasizes certain table cells or labels with color and can be printed on both a monochrome and color printer, you can assign both a color and a distinguishable font to the cells or labels that you want to emphasize. Do not use a REPLACE COLOR statement but do use COLOR = NO; when printing on a monochrome printer. On a color printer, both the color change and the font change will highlight the emphasized cells or labels. On a monochrome printer, the font change will show the emphasis.

COLUMN WIDTH

Format

COLUMN WIDTH = amount [unit];

where amount is a number and unit is optional. If no unit is specified, characters are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as inches, cm or points.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning

For specified columns, make the table columns n characters wide. The column width includes the column divide character. If the column width is n, then the column will have one less than n positions for data plus one position for the divide character.

If the column width action is not restricted by a FOR clause, the column width will be the same for all columns in all tables. Column widths can vary within a table.

See also AUTOMATIC STUB AND COLUMN WIDTHS in this chapter for automatic adjustment of columns widths to fill the available space.

Level

Column width can be specified for individual columns but is controlled at the table level. For a particular column, the column width will be the same throughout any one table.

Default

COLUMN WIDTH = 10;

Example

```
COLUMN WIDTH = 15;
FOR TABLE 3 COLUMNS 2 AND 4: COLUMN WIDTH = 8;
```

Effect

The first FORMAT statement sets column width to 15 for all tables. If the second statement follows in the same format request, it sets column width to 8 for columns 2 and 4 of the third table. For other columns of the third table and for all other tables, column width will still be 15.

Restrictions

The minimum column width is 3. The page must be wide enough to hold the stub + the margins + the widest column.

COMPRESS HEADING

Format COMPRESS HEADING;

Meaning The table heading will be formatted to take up the minimum possible vertical space after allowing for space above and below the label in each heading box.

Note that, even with the COMPRESS HEADING; statement, heading boxes will always contain at least 1/2 line of space above and below the label.

See also the statement SPANNER HEADING for another way to reduce vertical heading space.

Level Heading compression can be specified for individual tables.

Default The heading is not compressed. It is built from the top down, and heading boxes are aligned with adjacent boxes, except at the bottom level where the box heights may need to vary to fill the space above.

Example The following table has two levels of heading labels. The heading is very tall, because each level of labels has one long label. The default formatting for the heading makes the boxes at each level tall enough to contain the longest label.

Movie attendance and video rental by sex

	How often do you go to the movies with a friend or a member of your family?		Rent videos?		
	Rarely	Often	Never	Only on winter week ends	Other
Total	90	70	79	38	71
Sex					
Female	48	40	38	21	42
Male	42	28	39	15	29
No response	–	2	2	2	–

– Data not available.

If we use the statement COMPRESS HEADING; we will shrink the height of the heading. Note that the boxes no longer align horizontally.

Movie attendance and video rental by sex

	How often do you go to the movies with a friend or a member of your family?		Rent videos?		
			Never	Only on winter week ends	Other
	Rarely	Often			
Total	90	70	79	38	71
Sex					
Female	48	40	38	21	42
Male	42	28	39	15	29
No response	–	2	2	2	–

– Data not available.

Example A common use of COMPRESS HEADING; is to force the bottom level of heading boxes to line up (have the same height) even though adjacent boxes may have different numbers of heading boxes above them. Following is a table of this type:

Movie attendance and video rental by sex

	How often do you go to the movies with a friend or a member of your family?		VCR owners only	
	Rarely	Often	Rent videos?	
			Rarely	Often
Total	90	70	79	81
Sex				
Female	48	40	38	50
Male	42	28	39	31
No response	–	2	2	–

– Data not available.

If we apply the statement COMPRESS HEADINGS; without changing the table in any other way, the heading will be compressed, but the lowest level boxes will not line up.

Movie attendance and video rental by sex

	How often do you go to the movies with a friend or a member of your family?		VCR owners only	
			Rent videos?	
	Rarely	Often	Rarely	Often
Total	90	70	79	81
Sex				
Female	48	40	38	50
Male	42	28	39	31
No response	–	2	2	–

– Data not available.

To force alignment of the boxes at the lowest level, we must add lines to the boxes above. One way to do this is to add a / in front of the labels above: / 'VCR owners only' and / 'Rent videos?'. This technique would give the desired alignment.

Movie attendance and video rental by sex

	How often do you go to the movies with a friend or a member of your family?		VCR owners only	
			Rent videos?	
	Rarely	Often	Rarely	Often
Total	90	70	79	81
Sex				
Female	48	40	38	50
Male	42	28	39	31
No response	–	2	2	–

– Data not available.

COUNTRY (PROFILE only)

The COUNTRY setting determines standards for such things as date/time formats, data formats, and currency symbols. If you are using U.S. standards for these things, you would normally need a COUNTRY statement only if you wish to select a particular date format, for example to specify 4-digit year. For additional information about displaying year with 4 digits, see the PAGE MARKER statement.

If the default TPL character set does not provide all of the characters you need, for example accented letters needed for some languages other than English, see also the statement called CODEPAGE.

Important If you make changes to **country.tpl**, add a COUNTRY or CODEPAGE statement to your profile, or change a COUNTRY or CODEPAGE statement in your profile, ***you need to restart TPL*** to activate the changes. When running a table request, you must use the same CODEPAGE and COUNTRY statements that you used when processing your codebook. Otherwise, you will have conflicting standards. In particular, conflicts in CODEPAGE will cause the sort order to be scrambled.

Note If you use COUNTRY and/or CODEPAGE statements, place them at the beginning of your profile. If you are selecting a country other than the U.S., you should check the profile to see if any decimal numbers are used. If they are, you may need to edit them so that they conform to your country's standards as described below in the section called "Separators in Masks and Decimal Constants".

Format

```
COUNTRY = country-name;
```

where **country-name** is the name of a country listed in the **country.tpl** file. The word IS can be used in place of =. Both are optional and can be left out altogether.

Level COUNTRY applies to all tables.

Default

```
COUNTRY = US;
```

Meaning The COUNTRY statement lets you select appropriate standards for items such as thousand separator, decimal separator, currency symbol, and date/time formats. The standards are set in the file called **country.tpl**. This file is installed in the TPL system directory. It is an ASCII text file that can be printed or displayed on the screen. The **country-name** that you enter in

the COUNTRY statement must match a name in **country.tpl**. Associated with each country in the file, you can see the standards that have been set for that country.

The countries that currently have entries in the file are listed below. If your country name is not on this list, look in **country.tpl** to see if it has been added. If your country is not in **country.tpl**, you will need to edit the file to add the standards for your country. If you add a country to **country.tpl**, please send your entry to us. We will add it to our **country.tpl** file so that your country will always be included when you get new versions of TPL software.

AUSTRALIA
AUSTRIA
BELGIUM_DUTCH
BELGIUM_FRENCH
BRAZIL
CANADA
CANADA_FRENCH
DENMARK
FINLAND
FRANCE
GERMANY
HUNGARY
ICELAND
IRELAND
ITALY
JAPAN
MEXICO
NETHERLANDS
NZEALAND
NORWAY
POLSKA
PORTUGAL
SKOREA
SPAIN
SWEDEN
SWITZERLAND
TAIWAN
UK
US

If there is an entry for your country but you wish to change the standards, you can edit the **country.tpl** file. Follow the directions contained in the file accurately. Errors in this file could cause your TPL jobs to fail. As with **profile.tpl**, you can edit the file in the TPL system directory, or you can make a customized copy to use in the directory where you are running your TPL jobs.

Separators in Masks and Decimal Constants

For the default country US, the decimal separator is "." and the thousands separator is "," . The COUNTRY statement lets you choose different defaults so that you can enter masks and numeric constants and have values print with the separators that are customary in your country.

If you wish to display data values with no comma separator, you can edit your country in the **country.tpl** file or add a country entry with a unique name. In the line for the country, enter **N** in the thousands separator field.

Enter the country name in the COUNTRY statement in **profile.tpl**. The thousands separator will be suppressed.

Enter **masks** according to your country standard. For example, if your thousands separator is "." and your decimal separator is "," then a valid mask would be **MASK 9.999,99** .

Note that if your country uses a period "." as the decimal separator you must enter comma "," as the thousands separator in masks. In the output, the data will use the correct symbol for the thousands separator if it differs from comma. Similarly, if your country uses a comma "," as the decimal separator, you must enter a period as the thousands separator in masks.

For example, some countries, such as France and Finland, use comma "," as the decimal separator and blank as the thousands separator. Blank can be entered in **country.tpl**, but a mask with a blank will cause a syntax error. If you use blank as the thousands separator, you must enter the mask using a period as the thousands separator. The data values will be printed correctly with blank as the thousands separator.

Enter **numeric constants** according to your country standard. For example, if your decimal separator is "," then **457,22** is a valid entry. The thousands separator is not relevant, because thousands separators cannot be entered in numeric constants.

Note This feature does not extend to observation variable values in a data file. If they have decimal or thousands separators, they must conform to US standards, i.e. period for decimal and comma for thousands.

Effect on Currency Formats

If you have entered the COUNTRY statement in your profile, the currency symbol will be taken from the corresponding country entry in **country.tpl**. This entry also determines whether the symbol should be displayed before or after the money value, with or without a blank between. As already noted, you can edit **country.tpl** if you wish to change the symbol or its placement.

The rules for entering currency symbols in masks in codebooks and statements vary depending on the type of currency symbol for your country. In any case, if your country name and codepage specifications are correct, a Postscript output will have the correct currency symbol inserted in the correct place in your data values. If you are using a non-Postscript printer, what will be printed depends on your printer. Changes to the **country.tpl**

file can usually fix non-Postscript printer problems. If not, exporting to pdf and then printing will fix the problems.

1. For some countries, the currency symbol is a single special character such as the UK Sterling symbol. For this type of currency symbol, you may enter either $ or the special symbol in your print masks.
2. For other countries such as France, the currency symbol is a regular letter in the alphabet. In such cases you must use the US $ in your print masks, but the letter will be used as the currency symbol in formatting data values.
3. For currency symbols such as Kr and Cr$ that contain more than one character, you must use the US $ in your print masks, but the correct combination of symbols will be used in formatting the data.

Examples

```
COUNTRY = Denmark;
User specified mask is: $999,99
Output for 332.76 is:  332,76Kr

COUNTRY = UK;
User specfied mask is: £999.99
Output for 332.76 is: £332.76

COUNTRY = CANADA_FRENCH;
User specfied mask is: $99.999,99
Output for 54332.76 is: $54 332,76
```

In some countries it is customary to insert a blank space between the currency symbol and the value. Because space is often limited in columns of data, we have not included blank characters for currency symbols in our **country.tpl** file. If you wish to have the blank space inserted, you can change the currency style field in your **country.tpl** file to request the blank space for your country's entry.

Note TPL software does not support currencies that put the currency symbol at the decimal point place. Please contact us if you have a requirement for this format.

Special Treatment for Currency Symbols in Output

TPL provides special treatment for masks that contain the US currency symbol $. For example, in a column of numbers with a mask of **$999.99**, only the first number in the column will be displayed with a $. (See the MASK chapter for a more detailed description of the $ treatment.) Non-US currency symbols are given similar treatment.

Date and Time Formats

Whenever date and time are displayed by TPL, the formats are determined by COUNTRY. For example, if you use DATE or TIME in the FORMAT statement called PAGE MARKER and have specified **COUNTRY = SWITZERLAND**, the statement:

```
PAGE MARKER = "Page " NUMBER "  Job run on "
        DATE " at " TIME;
```

will produce tables with page numbers, date and time in the following format:

```
Page 1  Job run on 31.12.07 at 14,24,38
```

CSV DIVIDER

Format

CSV DIVIDER = *divider*;

where ***divider*** can be:

SEMICOLON
COMMA
TAB
SPACE
or any single character in quotes.

The word IS can be used in place of =. Both are optional and can be left out altogether.

Meaning

For exported CSV files, the default divider (delimiter) between values is comma. CSV DIVIDER can be used to specify a different divider character.

Level

The divider is controlled at the request level. The same divider applies to all tables within the same table request.

Default

CSV DIVIDER = ",";

Example

CSV DIVIDER = " ";

Effect

When the tables are exported to CSV format, the values will be separated by a blank character.

Restriction

If a Tab is entered as the divider, it will be treated the same as a blank. If you are using the Windows version of TPL and do the export interactively from Ted, you can select Tab as the divider.

CSV OUTPUT (UNIX only)

Format CSV OUTPUT = YES or NO or PROMPT;

Normally, when you have created tables, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for CSV and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**. Use **CSV DIVIDER** to specify the divide character.

DATA SPAN

The SPAN specification controls the width of both SPANNER labels and horizontal lines that have been inserted with the RULE AFTER ROW statement. DATA SPAN is the default. Alternate SPAN specifications are described under the ROW SPAN statement.

DATA TABLES

This statement has been replaced by the Data Table export option.

Format

```
DATA TABLES;
DATA TABLES ZERO FILL;
```

Meaning

The DATA TABLES statement is especially useful if you are summarizing information for subsequent use in other software such as spread sheet or graphics programs.

It formats the tables so that they can be used as a data file. All title and label information is removed along with the horizontal lines (rules) and the space between pages and tables. The vertical lines (down rules) are replaced with blanks. Footnote symbols and text are removed, and the content of cells that would have special footnotes such as the dash (**-**) for "**Data not available.**" is converted to 0.

Only masks remain in effect. If you need decimal points in the numbers or if you plan to use the data tables as input to software that does not allow commas in the data, you may need to add your own mask for the data since the default mask inserts commas in numbers that are longer than 3 digits and does not include decimal points.

Any other display information that you want to keep in the tables can be preserved with RETAIN statements.

The page width of a data table is automatically calculated to hold all columns of the table so that all values in a data row will be on the same line. In other words, wide data tables are not automatically banked.

If more than one table is included in the table request, you will probably want them to have the same columns widths so that all columns of the data file output are aligned together. You will probably also want the heading expressions to be identical so that each column has the same information throughout.

If you have several tables in a request but want to use only one of them as a data table, you can use FORMAT statements to delete all tables except the one you want as a data file. For example, if you have several tables in

a request but want to use only the second as a data file, you can accomplish the desired result with the following FORMAT statements:

```
DATA TABLES;
FOR TABLES = ALL: DELETE TABLE;
FOR TABLE = 2: RETAIN TABLE;
```

ZERO FILL

By default, any space not occupied by the data values and associated mask characters will be filled with blanks. If you use the ZERO FILL option, columns will be filled with zeros to the left of each value instead of blanks. If the values are not right-aligned, then blanks will fill any unused space to the right of the values. If a value has a mask that includes a character such as $, this character will remain with the data or will be replaced with a blank in cells where the $ would not print in a non-data table.

By default, columns are separated by a blank character. If you want to have zeros fill this space, you must use RETAIN DOWN RULES; and REPLACE DIVIDE CHARACTER WITH '0' ; *These statements must **follow** the DATA TABLES; statement.*

Level Data tables can be controlled at the individual table level.

Default Tables are formatted with complete title, label and footnote information, margins, horizontal and vertical rules and pagination.

Example

```
FOR TABLE 1: REPLACE MASK WITH 999999 RIGHT;
        DATA TABLE;
```

Effect Format the table as a data file with the data right-adjusted within each column and no commas added.

Example

```
DATA TABLES;
REPLACE MASK WITH 999999.9;
RETAIN DOWN RULES;
REPLACE DIVIDE CHARACTER WITH ',';
```

Effect The tables are formatted as a data file; the numbers are formatted with a decimal point. Retaining the down rules and replacing the divide character with comma causes the numbers to be separated by commas within each data row.

Example

```
REPLACE MASK WITH 999 RIGHT;
DATA TABLES ZERO FILL;
RETAIN DOWN RULES;
```

```
REPLACE DIVIDE CHARACTER WITH '0';
```

Effect All values will be right-aligned in the columns, with no commas or decimal points. The values will be zero filled on the left and the space between columns will contain a zero, so there will be no blanks or other non-numeric characters in the data file output.

Note By default, TPL TABLES will not include rows that have no data. You may find it useful to use RETAIN EMPTY LINES; with the DATA TABLES; statement when you need to have a predictable number of lines in the output. See the RETAIN EMPTY LINES statement for details.

Note The DATA TABLES statement is equivalent to the following collection of FORMAT statements:

```
TOP MARGIN = 0;
LEFT MARGIN = 0;
PAGE LENGTH = AUTOMATIC;
PAGE WIDTH = AUTOMATIC;
DELETE TITLE;
DELETE HEADNOTE;
DELETE HEADING;
DELETE STUB;
DELETE FOOTNOTES ALL;
DELETE ALL RULES;
DELETE WAFER LABEL;
PAGE MARKER = '';
```

You must take care that you do not unintentionally override parts of the DATA TABLES effect by following the DATA TABLES statement with a FORMAT statement that conflicts with one of the above. For example, the sequence

```
DATA TABLES;
PAGE LENGTH = 66;
```

would override the PAGE LENGTH AUTOMATIC statement that is built into DATA TABLES and cause gaps to appear between pages.

In addition, stub deletion is accomplished within TPL TABLES by setting the stub width to zero. If you follow the DATA TABLES statement with a STUB WIDTH statement that sets the stub to a width greater than zero, the effect may be to retain the stub in the tables.

Restrictions There is one exception to the treatment of footnotes in a data table. In a regular table, if a number is too large to fit within the column, it is footnoted with the symbol "nf" appearing in the cell. In a data table, if a number is too large, it is replaced with a blank, because a truncated value or a 0 would be wrong. If you get a data table with blanks where data values should be, expand the column width with a COLUMN WIDTH statement.

DATA TABLE OUTPUT (UNIX only)

Format

```
DATA TABLE OUTPUT = YES or NO or PROMPT;
DATA TABLE OUTPUT = STUB;
DATA TABLE OUTPUT = ZERO FILL;
```

Normally, when you have created tables, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for **DATA TABLE** and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**. **STUB** will cause the stub to be retained. **ZERO FILL** will cause leading blanks to be replaced by zeros.

DELETE

The following statements correspond to RETAIN statements. See the like-named RETAIN statements for details.

```
DELETE ALL RULES;
DELETE BANK DIVIDER;
DELETE BOTTOM RULE;
DELETE COLUMNS;
DELETE DOWN RULES;
DELETE EMPTY COLUMNS;
DELETE END RULE;
DELETE HEADING BOTTOM RULE;
DELETE HEADING CROSS RULE;
DELETE HEADING;
DELETE HEADNOTE;
DELETE LAST RULE;
```

DELETE LEADING ZEROS;
DELETE ROW;
DELETE RULE AFTER ROW;
DELETE RULE AFTER STUB;
DELETE SPANNER RULE;
DELETE STUB;
DELETE TABLES;
DELETE TITLE;
DELETE TOP RULE;
DELETE WAFER;
DELETE WAFER LABEL;

DISPLAY NAME (UNIX/Linux Profile only)

Windows Note The Windows version of TPL TABLES uses TED, the TPL editor, to display tables. **DISPLAY NAME** is ignored.

Format

DISPLAY NAME = *PostsScript-displayer;*

where *PostScript-displayer* can be either just the program name or the name including a full path if needed.

Meaning When a TPL TABLES job run in PostScript mode completes successfully, you will be asked if you wish to display the tables. If you answer "yes", the tables will be displayed using the *PostScript-displayer* as a separate process.

Examples

Sun Solaris:

```
DISPLAY NAME = pageview;
DISPLAY NAME = /usr/openwin/bin/pageview;
```

Linux (using KDE):

```
DISPLAY NAME = kghostview;
```

DO NOT RANK ON VALUES

See RANK ON VALUES

DO NOT REPORT ROWS

See RANK ON VALUES

DOWN LINE

This statement has been replaced by RETAIN DOWN RULE rule-options

Format DOWN LINE WEIGHT = n; or
DOWN LINE DOUBLE; or
DOWN LINE SINGLE; or
DOWN LINE WEIGHT = n DOUBLE or SINGLE;

Meaning RULE is a synonym for LINE in these statements. See the RETAIN DOWN RULE statements for more information.

DOWN RULE

This statement has been replaced by RETAIN DOWN RULE rule-options

Format

DOWN RULE WEIGHT = n; or
DOWN RULE DOUBLE; or
DOWN RULE SINGLE; or
DOWN RULE WEIGHT = n DOUBLE or SINGLE;

where **n** is a number. The word IS can be used in place of =. Both are optional and can be left out altogether. If both a WEIGHT value and one of the words DOUBLE or SINGLE are used, they can be in any order. The word LINE can be used in place of the word RULE.

Meaning DOWN RULE statements do not affect exported text tables. They apply to the vertical lines that extend from within the heading to the bottom of the table between the columns. These lines are called down rules. You can provide a WEIGHT value and/or specify whether the rules should be DOUBLE or SINGLE. The WEIGHT value increases or decreases the thickness of the rules. It is expressed in points where each point is 1/72 inches.

Note The appearance for a particular rule weight on the printed page will vary from printer to printer. This is especially true with printers of different dpi (dots per inch).

DOWN RULE specifications can be restricted to specific columns. This is useful, for example, if you wish to emphasize the division between certain columns with double rules or a thicker rule. When columns are specified in the FOR clause, the specifications are applied to the rules that follow the specified columns.

The down rule that separates the stub from the body of the table is specified as column 0. This is true even if you have specified STUB RIGHT.

See also the RULE statement to change the appearance of rules other than DOWN RULES.

Note If both a DOWN RULE statement and a RULE statement are present, the DOWN RULE specifications take precedence for the vertical rules regardless of the order of the statements in the format request.

Level DOWN RULE can be specified at the individual column level.

Default

```
DOWN RULE WEIGHT = .5  SINGLE;
```

Example

```
FOR TABLE 3 COLUMNS 2, 4:  DOWN RULE WEIGHT = 1.5;
```

Effect All rules in the tables will have the default rule weight except the vertical rules after columns 2 and 4. These two rules will be thicker than the others.

Example

```
FOR COLUMN 0:  DOWN RULE WEIGHT = 2;
```

Effect Since column 0 refers to the tale stub, the rule between the stub and the first data column will be thicker than the rules between the other columns.

Example

```
FOR TABLE THREE_A COLUMN 0: DOWN RULE DOUBLE;
```

Effect There will be a double rule between the stub and the first data column.

Table 2. Households by sex and education level of householder

	Total	Sex of Householder	
		Male	Female
Educational Attainment of Householder			
8 years or less	3,986	2,517	1,469
High school, 1 to 3 years	3,699	2,352	1,347
High school, 4 years	10,875	7,512	3,363
College, 1 to 3 years	5,059	3,554	1,505
College, 4 years	3,466	2,629	837
College, 5 or more years	2,915	2,257	658

Referring to a Rule that Follows a Deleted Column

If you have deleted a column or range of columns and want to change the appearance of the following down rule, you will need to include the deleted column(s) in a range in the FOR clause for the DOWN RULE statement.

Examples Column 5 is deleted and and a weight is specified for the down rule following column 4.

```
FOR COLUMN 5: DELETE COLUMN;
FOR COLUMNS 4 TO 5: DOWN RULE WEIGHT = 1;
```

Columns 6 to 9 are deleted and a weight is specified for the down rule following column 5.

```
FOR COLUMNS 6 TO 9: DELETE COLUMNS;
FOR COLUMNS 5 TO 9: DOWN RULE WEIGHT = 1;
```

Restrictions The DOWN RULE statement has no effect if the divide character is set to something other than the default divide character of | .

The rule weight value must be greater than or equal to zero. A rule weight of 0 does not make the rule disappear. Instead, it results in the thinnest line that is possible on your output device.

If the rule weight value is too large, the rules will be so thick that they will make broad bands that overlay the columns. This is usually undesirable. For example, the statement:

```
DOWN RULE WEIGHT = 72;
```

will create broad black bands that are 1 inch wide (1 pt. = 1/72", so 72 points = 1").

Table Title

Count

	Hispanic Origin of Householder		
	Hispanic		Not hispanic
Number of Earners			
None	466		5,830
1	854		9,085
2	864		9,577
3	189		2,137
4	86		691
5	12		151
6	5		41
7	3		7
8	1		1

EDITOR (UNIX Profile only)

Windows The Windows version of TPL TABLES is linked to TED, the TPL editor. EDITOR statements have no effect.

UNIX There are two EDITOR statements. They are used only in **profile.tpl** and are initially set at installation time if you have indicated that you would like to have TPL TABLES linked to your editor. They are described below in case you need to change or add them after installation.

Format

```
EDITOR NAME = editor_name;
EDITOR FILE = editor_file;
```

Meaning TPL TABLES has been designed so that you can use the text editor of your choice to create codebooks, table requests and format requests. Any editor that creates stand-alone ASCII text files is acceptable.

If you choose to link TPL TABLES to your editor, TPL TABLES will automatically transfer to your editor when a job stops because of an error.

Editor Name

The **editor_name** should be the name you use to start your editor. For example, if you start your editor by entering **ED**, the editor name statement in your profile should be:

```
EDITOR NAME = ED;
```

Path names are allowed but not required.

Editor File

TPL TABLES assumes that you can start your editor with a command that includes the name of the file to be edited. TPL TABLES uses a file name of **TPLTEMP** when it transfers to your editor, so the EDITOR FILE statement is:

```
EDITOR FILE = TPLTEMP;
```

Some editors require a special extension such as **DOC** or **TXT** for any file to be edited. If this is the case with your editor, include the required extension in your EDITOR FILE statement. For example, if the required extension is **TXT**, use the statement

```
EDITOR FILE = TPLTEMP.TXT;
```

EJECT

The following EJECT statements are the default settings for table and wafer pagination. They are sometimes used explicitly in conjunction with the **SKIP AFTER TABLE** and **SKIP AFTER WAFER** statements. See the SKIP statements for details.

```
EJECT AFTER TABLE [ = YES ];
EJECT AFTER WAFER [ = YES ];
```

EJECT with NO gives the same result as the SKIP 0 LINES statement described in **SKIP AFTER TABLE** and **SKIP AFTER WAFER**.

```
EJECT AFTER TABLE = NO;
EJECT AFTER WAFER = NO;
```

EJECT AFTER ROW

Format FOR *row specification*: EJECT AFTER ROW;

Meaning This statement can only be used with a FOR clause that specifies the rows where the page breaks should occur. The ROWS are data rows. At each specified data row, the page will be completed and the table will continue on the next page.

Precise control of page breaks can be useful when you want to prevent page breaks in the middle of logical groupings of rows. In other cases, you may want to start a new page for particular variables or values.

If you have a table that ends with a very small number of rows on the last page, you may wish to improve the appearance of the last page by moving a few rows of data from the next-to-last page to the last page. You can achieve this by specifying the row where you want the page break to occur.

Note that if any rows of the table are not printed because they are empty (do not have any data) or because the rows are ranked, you cannot determine row numbers by counting data rows in the printed table. You can find the row numbers for **PRINTED ROWS** in the OUTPUT file. If you reference an empty row in the FOR clause, the page break will occur before the next row that has data.

Note If you have ranked rows and reference an empty row, the EJECT AFTER ROW statement will have no effect. For ranked rows, you need to reference a row that has data.

If you have used the command **RETAIN EMPTY LINES** and there is no ranking, you can determine row numbers by looking at the tables. See RETAIN EMPTY LINES for more information.

The following option is also available and is generally used with NO to reverse a previous EJECT.

```
EJECT AFTER ROW = NO;          (or YES)
```

Level Page breaks can be specified for individual rows.

Default Page breaks are determined automatically based on the number of rows that can fit on a page.

Example FOR TABLE 1, WAFER 2, ROW 35: EJECT AFTER ROW;

Effect The page containing row 35 of the second wafer in the first table will end after row 35. If row 35 is not printed because it has no data, then the page eject will occur before the first printed row following row 35. The table will continue on the next page. All other page breaks will be determined automatically.

Example
```
FOR ROWS 20 to 200 by 20: EJECT AFTER ROW;
FOR ROW 100 : EJECT AFTER ROW = NO;
```

Effect A page break will occur after rows 20, 40, 60 and so on to row 200 except after row 100.

Note Row numbering restarts with each wafer. For a table with multiple wafers, if no wafer specification is included in the EJECT statement, the statement will apply to all wafers.

Restrictions If you use EJECT AFTER ROW without a FOR clause, you will get an error message.

EPS OUTPUT (UNIX only)

Format EPS OUTPUT = YES or NO or PROMPT;

Normally, when you have created tables, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for EPS, and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**.

EXTRA LEADING

Format EXTRA LEADING = n;

where n is a number. The word IS can be used in place of =. Both are optional and can be left out altogether.

Meaning This statement can be used to regulate the amount of space between lines in tables. It has no effect on exported text tables.

Note If you simply want to scale down the size of your table, ***see the SCALE statement***. With the SCALE statement, you can reduce the overall size of everything in a table to a percentage of its original size and fit more of your table on a page or other smaller space.

Leading (rhymes with "heading") is the space between lines of text or data. TPL TABLES automatically adjusts this space in proportion to the font size you have chosen. If this amount is not appropriate for your tables, you can change the spacing with the EXTRA LEADING statement. Increasing the leading number will increase the amount of space between lines; decreasing the leading number will decrease the amount of space between lines.

A value of 0 for EXTRA LEADING will give the default spacing. Since the default spacing is often too close for tables of data, TPL TABLES uses an extra leading value of .15 to increase the spacing by a small amount.

The font sizes include the leading. For each line, most of the vertical space will be occupied by the printed characters and part will be reserved for the space between lines. For example, if the font size is 12 (points), each line, including the leading, will take 12/72 = 1/6 inch of vertical space. Thus, there will be 6 lines per inch. (A point is 1/72 inch).

In the EXTRA LEADING statement, the extra leading value is multiplied by the font size to determine the extra amount of spacing. For example, **EXTRA LEADING = .5;** will add .5 * font size to line spacing. If the font size is 12, the extra leading will be .5 * 12 points or 6 points, and each line will take 12 + 6 = 18 points of vertical space. 18 points = 1/4 ", so there will be 4 lines per inch. The characters will be the standard character size for a font size of 12, but there will be much more space between the lines.

If there is more than one font size specification for a line, the extra leading calculation will be based on the largest font size for the line.

Level Extra leading can be controlled at the individual table level.

Default EXTRA LEADING = .15;

Example FOR TABLE 1: EXTRA LEADING = .2;

Effect For the first table, line spacing will be increased beyond the standard PostScript leading by .2 * font size. If the font size for a line is 10, the extra space will be 2 points.

Restrictions The EXTRA LEADING value cannot be less than 0.

FONT

Print style and size can be specified with the FONT statement.

Note The FONT statement is not supported for text table export.

Format table-element FONT = fontname fontsize;

where **fontname** is the TPL TABLES abbreviation for the font name, and **fontsize** is a number.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether. **Fontsize** is optional for all except the DEFAULT FONT statement.

Individual masks and labels, including titles, footnotes and page markers, can contain FONT specifications that will override the FONT statements. For more information, see the chapters on Masks, Labels and Footnotes.

Example

```
TITLE FONT = HB 10;
```

Effect The title font will be 10 pt Helvetica Bold.

Level The DEFAULT font applies to the entire request, but fonts for other table elements can be specified at the table level.

Table Elements

Fonts can be specified for the following table-elements. Note that there is no font specification that applies to table cells only. To change the cell font only, see the statement REPLACE MASK FONT.

DEFAULT	The DEFAULT font applies to the table cells and any table elements not otherwise specified.
TITLE	
TITLE CONTINUATION	
FOOTNOTE TEXT	
FOOTNOTE SYMBOL (or SYMBOL)	
HEADNOTE	
WAFER LABELS	
STUB HEAD	
CONDITION LABELS	
CONDITION LABELS IN HEADING	
CONDITION LABELS IN STUB	
VARIABLE LABELS	

```
VARIABLE LABELS IN HEADING
VARIABLE LABELS IN STUB
```

Font Names

You can choose from any of the following fonts. Refer to them by TPL TABLES abbreviation. For example, the following specifies Times-BoldItalic.

```
TITLE FONT TBI;
```

Abbreviations	Font names
C	Courier
CB	Courier-Bold
CI	Courier-Oblique
CBI	Courier-BoldOblique
T	Times-Roman
TB	Times-Bold
TI	Times-Italic
TBI	Times-BoldItalic
H	Helvetica
HB	Helvetica-Bold
HI	Helvetica-Oblique
HBI	Helvetica-BoldOblique
N	Helvetica-Narrow
NB	Helvetica-Narrow-Bold
NI	Helvetica-Narrow-Oblique
NBI	Helvetica-Narrow-BoldOblique
A	AvantGarde-Book
AB	AvantGarde-Demi
AI	AvantGarde-BookOblique
ABI	AvantGarde-DemiOblique
B	Bookman-Light
BB	Bookman-Demi
BI	Bookman-LightItalic
BBI	Bookman-DemiItalic
S	NewCenturySchlbk-Roman
SB	NewCenturySchlbk-Bold
SI	NewCenturySchlbk-Italic
SBI	NewCenturySchlbk-BoldItalic
P	Palatino-Roman

PB	Palatino-Bold
PI	Palatino-Italic
PBI	Palatino-BoldItalic
Z	ZapfChancery-MediumItalic
D	ZapfDingbats*
Y	Symbol*

* The Symbol and ZapfDingbats fonts are mainly useful for footnote symbols.

Font Sizes

Font sizes are specified in points. A point is 1/72", so there are 72 points in one inch. The font size includes the space between the lines.

Examples If the font size is 12, each line takes 12 points of vertical space or 12/72 = 1/6". With font size 12, there are 6 lines per inch.

If the font size is 8, each line takes 8 points of vertical space or 8/72 = 1/9". With font size 8, there are 9 lines per inch.

General Rule To get larger characters in your table, increase the font size; to get smaller characters, decrease the font size.

A font size must be specified for the DEFAULT FONT. For all other fonts, if no size is specified, it will be whatever size is already in effect for that table element. For example, if the title font is set at H 12 (Helvetica 12) in the profile and the statement **TITLE FONT TBI;** (Times-BoldItalic) is included in the format request, the title will be printed in Times-BoldItalic with a size of 12.

Default Font defaults are initially set at installation time. A sample profile after installation is:

```
Postscript = yes ;
Default font = H 8;
Footnote text font = T 8;
Footnote symbol font = H 8;
Title font = HB 10;
paper = LETTER;
```

You can edit the profile to change the defaults and/or you can override them in your format request. If you usually use fonts that are different from those that were established at installation time, you will probably want to change the font statements in the profile.

Example Following is a sample format request with font specifications:

```
postscript = yes;
default font H 10;
title font TB 12;
footnote text font T 8;
footnote symbol font Y 7;
page width = 8.5 inches;
page length = 11 inches;
right margin = .7 in;
left margin = .7 in;
```

Adding Underline to Fonts

You can add underlining to any of the fonts by adding a **U** to the font specification.

Example In the following example, the default font is set to **HIU** for **H**elvetica **I**talic **U**nderline and the title font is set to **HBIU** for **H**elvetica **B**old **I**talic **U**nderline. The footnote text and symbol fonts are set to non-underlined fonts.

```
Default font = HIU 10;
Title font = HBIU 12;
Footnote text font = H 8;
Footnote symbol font = H 8;
```

Table F10: Amount of training and average age by sex.

	Total	*Sex*		
		Female	*Male*	*No response*
Total				
Average Age	*44*	*44*	*44*	*48*
Employer Training	*High*	*High*	*High*	*Low*
Manufacturing				
Average Age	*46*	*([1])*	*46*	*48*
Employer Training	*High*	*Medium*	*Medium*	*Low*
Other				
Average Age	*43*	*43*	*43*	*–*
Employer Training	*High*	*High*	*Medium*	*–*

[1] Confidential
– Data not available.

Note that when built-in footnotes such as EMPTY apply to table cells, they are set to the default font. Thus, the dash character for the EMPTY footnote is underlined in the table cells.

To underline rows across the entire data section of a table, see the statement RETAIN RULE AFTER ROW UNDERLINE.

Using the Symbol and Zapf Dingbats Fonts

The character sets for the Symbol and ZapfDingbats fonts are shown in the Appendix. As you will see if you look at these character tables, the Symbol font includes numbers but the ZapfDingbats font does not. Neither of these fonts can be used to print alphabetic characters.

For Footnote Symbols

If you use Symbol as the font for footnote symbols (**FOOTNOTE SYMBOL FONT Y;**), you can use the special characters in the Symbol font for some footnote symbols and numbers for other footnote symbols.

If you choose ZapfDingbats characters for footnote symbols (**FOOTNOTE SYMBOL FONT D;**), you cannot also have numbered footnote symbols. In addition, some of the symbols used for built-in footnotes do not exist is the ZapfDingbats font. Unless you want to select all footnote symbols yourself, the best approach is to choose one of the other fonts in the FOOTNOTE FONT statement and include the ZapfDingbats font specifications in the footnote symbol part of the SET FOOTNOTE statement. See the footnote chapter for additional information on including font specifications in SET FOOTNOTE.

For Labels

Since the Symbol and ZapfDingbats fonts do not contain the usual alphabetic characters, these fonts should not be used for FONT statements other than FOOTNOTE SYMBOL. To use characters from these fonts in labels, add the FONT specifications to individual labels. This technique is described in the labels chapter.

Although most of the characters in these special fonts are not on your keyboard, you can enter them in labels and footnote symbol strings by typing \nnn where nnn is the 3 digit decimal code for the character. The character set tables in the Appendix show the 3 digit code for each character.

Example

```
FOOTNOTE SYMBOL FONT = D 10;
FOOTNOTE TEXT FONT = T 12;
SET FOOTNOTE 1 SYMBOL '\098' TEXT "Sample footnote";
```

Effect [b] Sample footnote

Note that most Symbol and ZapfDingbats characters cannot be printed when a text table export is selected. They will print as blanks or other characters, depending on your printer's character set.

Matching the Footnote Symbol Font to the Adjacent Font

You can use the word MATCH as a font specification for footnote symbols when you want the footnote symbol font to match that of the adjacent text or data value. The format for this special FONT statement is:

```
FOOTNOTE SYMBOL FONT = MATCH;
```

When the footnote symbol is used in a data cell, it matches the font in effect for that cell. When used in a label, it matches the font that is active at the point where the footnote is referenced in the label. When used with the footnote text at the bottom of a page, the symbol matches the font used at the beginning of the footnote text.

Spaces in Proportional Fonts

With a proportional font, a blank space cannot be the same width as all other characters, because the character widths vary. In general, a blank is approximately one half the width of a number in the same font -- or one half the average width of a letter. Thus, with a proportional font, you will need about twice the number of blanks to get the same amount of blank space you would get with a non-proportional font.

Recommendation

Helvetica is the recommended font for the body of a table, especially when a small font (below 10 point) is used. Other fonts can be used effectively for the title and footnote text. Bold or italics may be used to emphasize certain data values.

FOOTNOTE COLUMNS

Format

FOOTNOTE COLUMNS = n [JUSTIFIED }; or

FOOTNOTE COLUMNS = n [UNJUSTIFIED };

where **n** is a number. The word **IS** can be used in place of =. Both are optional and can be left out altogether. The specification of JUSTIFIED or UNJUSTIFIED is optional. JUSTIFY and UNJUSTIFY are synonyms for these words.

For text tables, this statement is ignored.

Meaning

If you specify the number of columns for footnotes, the footnotes printed at the end of a table will be justified within the number of columns you request. By "justified", we mean that blank space will be added between words so that all lines in a column will have the same width.

Justification will not take place for the last line of a footnote text (even if followed by one or more slash characters), because the last line of text is often short and does not look good if justified. For the same reason, justification will not take place for lines that are followed by a blank line. This would be the case, for example, where a segment of footnote text was followed by two slash characters (//) to cause double spacing after the line .

If you do not want to have the footnotes jusified, add the word UNJUSTIFIED to the statement.

FOOTNOTE COLUMNS also applies to notes created with SET NOTE statements.

Level

FOOTNOTE COLUMNS can be specified at the individual table level.

Default

FOOTNOTE COLUMNS = 1 UNJUSTIFIED;

Example

FOOTNOTE COLUMNS = 1;

Effect

For all tables, footnotes will be justified in a single column that spans across the full table width.

Example

FOOTNOTE COLUMNS = 2 UNJUSTIFIED;

Effect Footnotes will be displayed unjustified in double column format for all tables.

Tip If you have a short line that is not at the end of a footnote text and you wish to prevent justification of the line, you can do so by calculating the width of the footnote column and adding SPACE TO and a blank character at the end of the line. This forces unused space at the end of the line to be filled with blanks and prevents justification of the text. The following example illustrates this technique. The SPACE TO location is slightly less that the footnote column width to allow for the blank at the end of the line.

```
FOOTNOTE COLUMNS = 2;

SET FOOTNOTE multi TEXT "Sorry.  This footnote doesn't "
        "really go with the accompanying table, but it does illustrate "
        "the point!.....  Incidence rates for number of injuries and "
        "illnesses per 10,000 workers can be calculated as: "
        "((N/EH) x 20,000,000) where" //
INDENT .25 in "N"
SPACE TO 1 in "="
SPACE TO 1.15 in "number of injuries"
SPACE TO 2.7 in " " / "EH"
SPACE TO 1 in "="
SPACE TO 1.15 in "total hours worked"
SPACE TO 2.7 in " " /
SPACE TO 1.15 in "in the calendar year"
SPACE TO 2.7 in " " / "20,000,000"
SPACE TO 1 in "="
SPACE TO 1.15 in "base for 10,000"
SPACE TO 2.7 in " " /
SPACE TO 1.5 in "working hours per week." / ;
```

Table 86. Plan[1] administration: Percent of full-time participants in selected benefits by type of plan sponsor, medium and large firms

Plan sponsor	Health insurance	Life insurance	Sickness and accident insurance	Dental insurance
All participants	100	100	100	100
Single employer	96	97	87	43
Multiemployer[2]	4	3	2	34
Mandated benefits[3]	–	–	11	23
Employer association[4]	([5])	([5])	–	–

[1] Does not include supplemental plans.

[2] Sorry. This footnote doesn't really go with the accompanying table, but it does illustrate the point!..... Incidence rates for number of injuries and illnesses per 10,000 workers can be calculated as: ((N/EH) x 20,000,000) where

N	=	number of injuries
EH	=	total hours worked in the calendar year
20,000,000	=	base for 10,000 working hours per week.

[3] The majority of the participants with mandated sickness and accident insurance benefits were covered by State temporary disability plans.

[4] Band of small employers in a common trade or business, for example, savings and loan associations. The plan sponsored by the association is not negotiated with the employees.

[5] Less than 0.5 percent.

NOTE: Because of rounding, sums of individual items may not equal totals. Dash indicates no employees in this category.

FOOTNOTES ON EACH PAGE / WAFER

Format

```
FOOTNOTES ON EACH PAGE;        or
FOOTNOTES ON EACH WAFER;       or
FOOTNOTES ON LAST PAGE;
```

The word **ON** is optional.

This statement also applies to notes created with SET NOTE statements.

Meaning All footnotes that apply to a page or wafer of the table will be printed at the bottom of that page or wafer. For footnotes that have automatically assigned footnote numbers, the numbering will start at the beginning of the table and continue to the end. For example, if a footnote in the table title is assigned the number 1 for the footnote symbol, that footnote will keep the number 1 throughout the table.

Default

```
FOOTNOTES ON LAST PAGE;
```

All footnotes are saved for display on the last page of the table. Pages other than the last page have the built-in footnote SEE_END with the text:

```
See footnotes at end of table.
```

Level Footnote placement can be controlled at the individual table level.

Example

```
FOOTNOTES EACH PAGE;
FOR TABLE 3:  FOOTNOTES ON LAST PAGE;
```

Effect For all tables except the third, footnotes will be displayed at the bottom of each page. For the third table, footnotes for the entire table will be displayed on the last page.

Note If a title continuation contains a footnote reference and you have requested FOOTNOTES EACH PAGE, the footnote from the continuation will be displayed at the bottom of the first page even though the continuation is not added to the title until the second and subsequent pages.

FOOTNOTE SEQUENCE

Format FOOTNOTE SEQUENCE = f1, f2, ... fn;

where f1, f2, ... fn are footnote names.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether. Commas between footnote names are also optional.

This statement also applies to notes created with SET NOTE statements.

Meaning The FOOTNOTE SEQUENCE statement can be used to control the order in which footnotes are displayed at the end of the table. The listed footnotes will be displayed in the order shown in the statement rather than in default order. Footnotes that are not listed in the statement will be displayed in default order, but following all of the listed footnotes.

Level Footnote sequence can be controlled for individual tables.

Default Footnotes are displayed in the default footnote order. Footnotes with numeric symbols are printed first. Footnotes with alphabetic or special characters in the symbols are sorted according to the footnote symbols and printed next. Footnotes that do not have symbols are printed last.

Example

```
FOR TABLE 1: FOOTNOTE SEQUENCE =
        SMALL, FN_ONE,FN_TWO;
FOR TABLE 2: FOOTNOTE SEQUENCE =
        EMPTY, SMALL, SOURCE;
```

Effect At the end of the first table, footnotes will be displayed in the order: SMALL, FN_ONE, FN_TWO, then any other footnotes in default order. Similarly, at the end of the second table, footnotes will be displayed with the footnotes EMPTY, SMALL and SOURCE preceding any other footnotes. For all other tables, footnotes will be displayed in default order.

GAP IN HEADER

Format GAP IN HEADER;
GAP IN HEADER = FALSE;
GAP IN HEADER EXCEPT BOTTOM;

Meaning When down rules are deleted in a table, it is sometimes difficult to match up the columns with their header labels. This is especially true if the header is complicated. GAP IN HEADER leaves gaps in the cross rules in the headings where the deleted down rules would have crossed. If GAP IN HEADER EXCEPT BOTTOM is used, the results are the same as GAP IN HEADER except the bottom rule of the header does not have gaps. This usually produces a better looking table than having gaps in the bottom rule.

Level GAP IN HEADER can be specified for individual tables.

Default GAP IN HEADER = FALSE;

Examples

The following is a typical table with the bottom row of labels right aligned

	Race of Householder						Type of Household		
	White		Black		Other		Married couple	Other family	Nonfamily household
	Hispanic Origin of Householder								
	Hispanic	Not hispanic	Hispanic	Not hispanic	Hispanic	Not hispanic			
Average Income									
Regions									
Northeast	21,358	36,708	19,330	24,514	38,577	37,267	44,222	25,561	21,666
Midwest	23,091	31,161	24,466	20,306	13,724	30,468	37,722	21,376	18,403

If we also use DELETE DOWN RULES START IN HEADER we get:

	Race of Householder						Type of Household		
	White		Black		Other				
	Hispanic Origin of Householder								
	Hispanic	Not hispanic	Hispanic	Not hispanic	Hispanic	Not hispanic	Married couple	Other family	Nonfamily household
Average Income									
Regions									
Northeast	21,358	36,708	19,330	24,514	38,577	37,267	44,222	25,561	21,666
Midwest	23,091	31,161	24,466	20,306	13,724	30,468	37,722	21,376	18,403

If we add GAP IN HEADER we get

	Race of Householder						Type of Household		
	White		Black		Other				
	Hispanic Origin of Householder								
	Hispanic	Not hispanic	Hispanic	Not hispanic	Hispanic	Not hispanic	Married couple	Other family	Nonfamily household
Average Income									
Regions									
Northeast	21,358	36,708	19,330	24,514	38,577	37,267	44,222	25,561	21,666
Midwest	23,091	31,161	24,466	20,306	13,724	30,468	37,722	21,376	18,403

If we change to GAP IN HEADER EXCEPT BOTTOM we get

	Race of Householder						Type of Household		
	White		Black		Other				
	Hispanic Origin of Householder								
	Hispanic	Not hispanic	Hispanic	Not hispanic	Hispanic	Not hispanic	Married couple	Other family	Nonfamily household
Average Income									
Regions									
Northeast	21,358	36,708	19,330	24,514	38,577	37,267	44,222	25,561	21,666
Midwest	23,091	31,161	24,466	20,306	13,724	30,468	37,722	21,376	18,403

HEADING SPACE

Format HEADING SPACE = n;

where **n** is a decimal number that is a multiplier of the font size.

The word HEAD can be used in place of the word HEADING.

Meaning HEADING SPACE can be used to increase or decrease the space above and below the heading labels. The most common use is to decrease the space so that the heading will take less vertical space on the page. This statement has no effect on text tables.

Note If you simply want to scale down the size of your table, ***see the SCALE statement***. With the SCALE statement, you can reduce the overall size of everything in a table to a percentage of its original size and fit more of your table on a page or other smaller space.

The HEADING SPACE number is a multiplier of the font size. It changes the space between the top label characters in a heading box and the top of the heading box and between the bottom label characters and the bottom of the heading box. The number **n** is split between the top and bottom of the heading box.

The minimum recommended heading space is .2.

If a heading box contains a label that splits into multiple lines, the space *between* the lines is not affected by the HEADING SPACE statement. See the statement EXTRA LEADING to change the space between lines.

See also TABLE SPACE to change the vertical space between elements throughout the table rather than only in the heading.

Level HEADING SPACE can be specified for individual tables.

Default HEADING SPACE = 1.15;

The default value of 1.15 is split between the top and bottom so that the space is a little more than 1/2 character above and below the label characters in the heading.

Example Following are two tables, the first with the default heading space of 1.15 and the second with heading space of .3:

Table with default heading space.

	Total	Sex		
		Female	Male	No response
Total	160	88	70	2
Age				
14	72	45	27	–
15	69	33	36	–
16	19	10	7	2

– Data not available.

FOR TABLE 2: HEADING SPACE = .3;

Table with heading space reduced to .3.

	Total	Sex		
		Female	Male	No response
Total	160	88	70	2
Age				
14	72	45	27	–
15	69	33	36	–
16	19	10	7	2

– Data not available.

Example See also the SPANNER HEADING statement for another illustrated example.

HTML ACCESS

Format HTML ACCESS;

Meaning When this option is set, HTML generated by TPL TABLES can be read by HTML screen readers. This makes the tables accessible to visually impaired or blind individuals and fulfills the requirements of ***Section 508*** of the US Government Rehabilitation Act.

When HTML ACCESS is specified, the appearance of an HTML table in a browser differs very little from that of an HTML table created without the ACCESS statement. The biggest differences are that the stub and heading labels are always bold and footnote references become links to their footnote text. If you click on a footnote symbol on a page, you will jump to the footnote text at the bottom of the page or to the footnotes at the end of the table as appropriate. The HTML source also changes. Each label in the table is given an identifier and each cell of the table is given a list of the label identifiers which are logically associated with it. When a person selects a table cell, the screen reader reads all of the labels associated with the cell.

Level This option applies at the request level only.

Default The default HTML tables do not have the special accessibility features.

Note TPL TABLES makes the associations on the basis of the TPL code rather than on the basis of the physical appearance of the table. So a TPL user can produce misleading results. Consider the following table and the circled data cell:

	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999
All households	46,333	3,105	5,184	4,846	4,776
Type of Residence					
Inside metropolitan areas	35,752	2,160	3,708	3,517	3,413
Outside metropolitan areas	10,581	946	1,476	1,329	1,362
Race and Hispanic Origin of Householder					
White	39,969	2,082	4,172	4,059	4,115
Black	5,162	932	872	671	560
Hispanic	2,908	308	474	435	330
Type of Household and Sex of Householder					
Male householder					
Married couple	24,967	446	1,114	1,916	2,340

It appears as though the cell depends on the three labels:

	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999
All households	46,333	3,105	5,184	4,846	4,776
Type of Residence					
Inside metropolitan areas	35,752	2,160	3,708	3,517	3,413
Outside metropolitan areas	10,581	946	1,476	1,329	1,362
Race and Hispanic Origin of Householder					
White	39,969	2,082	4,172	4,059	4,115
Black	5,162	932	872	671	560
Hispanic	2,908	308	474	435	330
Type of Household and Sex of Householder					
Male householder					
Married couple	24,967	446	1,114	1,916	2,340

In fact TPL exports that the cell depends upon only the two labels:

	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999
All households	46,333	3,105	5,184	4,846	4,776
Type of Residence					
Inside metropolitan areas	35,752	2,160	3,708	3,517	3,413
Outside metropolitan areas	10,581	946	1,476	1,329	1,362
Race and Hispanic Origin of Householder					
White	39,969	2,082	4,172	4,059	4,115
Black	5,162	932	872	671	560
Hispanic	2,908	308	474	435	330
Type of Household and Sex of Householder					
Male householder					
Married couple	24,967	446	1,114	1,916	2,340

To see why this is the case we must look at the table request. In the table it appears as though *White*, *Black* and *Hispanic* are all conditions of *Race and Hispanic Origin of Householder*. When we look at parts of the table request, we have:

```
define hh_race /'Race and Hispanic Origin of Householder' on race;
   'White'  if 1;
   'Black'   if  2;

define hh_hispanic on hispanic_origin;
   'Hispanic' if 1;
...

stub all_hh_lab then residence  then hh_race then
   hh_hispanic then hh_sex by hh_type then hh_age ;
```

Hispanic is in fact not a category of **hh_race**. So the variable label for **hh_race** does not apply. The solution to this is to modify the request so that *White*, *Black* and *Hispanic* are all conditions of the same variable:

```
Compute combined:
   1 if race = 1 and hispanic_origin = 1; /* White Hispanic */
   2 if race = 1 and hispanic_origin = 2; /* White non-Hispanic */
   3 if race = 2 and hispanic_origin = 1; /* Black Hispanic */
   4 if race = 2 and hispanic_origin = 2; /* Black non-Hispanic */
   5 if hispanic_origin = 1;    /* Other Hispanic */
```

```
 define hh_race1 /'Race and Hispanic Origin of Householder'
          on combined;
   'White'      if 1 to 2;
   'Black'      if 3 to 4;
   'Hispanic'   if 1;
                if 3;
                if 5;
...

stub all_hh_lab then residence  then hh_race then
    hh_sex by hh_type then hh_age ;
```

HTML OUTPUT (UNIX only)

Format HTML OUTPUT = YES or NO or PROMPT;

Normally, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for HTML, and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**.

In addition to specifying whether HTML should be output, there are several options to specify how the HTML is to be formatted.

Format HTML OUTPUT = SINGLE or MULTIPLE;

If **Single** is specified, all of the tables are place in a single HTML file though the table page breaks occur as they would normally. Default is **Multiple**.

Format HTML OUTPUT = NAVIGATION or NONAVIGATION;

If Navigation is specified, each page of a multi-page table has a navigation bar placed at the top which allows a user of the tables to move from one

table to the next. The HTML files must be placed in the same directory for this feature to work.

Format HTML OUTPUT = AUTOSIZE;

A paper table is normally limited to the size of the paper. Web pages are assumed to be of unlimited size. This option takes advantage of this by creating a single "page" which is wide enough to hold all columns of the table and long enough to hold all rows. Page breaks caused by change of table do result in page breaks unless Skip 0 lines between ... is specified. Other breaks such as Eject After Row are ignored. Also requests for banking are ignored. When a table has multiple wafers, the wafer labels are switched to spanner labels and the wafers are combined into a single page.

KEEP

The following statements without *rule-options* are defaults. They are identical to the RETAIN statements. See the RETAIN statements for details.

KEEP ALL RULES *rule-options*;
KEEP BANK DIVIDER *rule-options*;
KEEP BOTTOM RULE *rule-options;*
KEEP COLUMNS;
KEEP DOWN RULES *rule-options*;
KEEP EMPTY COLUMNS;
KEEP END RULE *rule-options;*
KEEP HEADER BOTTOM RULE *rule-options;*
KEEP HEADER CROSS RULE *rule-options*;
KEEP HEADING;
KEEP HEADNOTE;
KEEP LAST RULE *rule-options*;
KEEP LEADING ZEROS;
KEEP ROW;
KEEP RULE AFTER ROW *rule-options;*
KEEP RULE AFTER STUB *rule-options*;
KEEP STUB;
KEEP SPANNER RULE *rule-options*;
KEEP TABLES;
KEEP TITLE;
KEEP TOP RULE *rule-options;*
KEEP WAFER;
KEEP WAFER LABEL;

KEEP DATA FOOTNOTE

KEEP DATA FOOTNOTE can be used with REPLACE MASK to change the format of data values without removing footnotes. See REPLACE MASK for details.

KEEP FOOTNOTE

Format KEEP FOOTNOTE (name);

where **name** is a footnote name. The parentheses around the footnote name are optional.

Meaning The KEEP FOOTNOTE statement is used to force printing of a footnote at the end of a table, even though the footnote does not apply to any label or mask that is used in the table. If the footnote has no explicit footnote symbol, only the footnote text is printed.

The most common use of KEEP FOOTNOTE is to print a note at the end of a table without referencing the footnote in a label or mask. See also the SET NOTE statement as an alternate way of accomplishing the same result.

Footnotes that do not have symbols are printed before any other footnotes, unless you have changed the order of footnote printing with a FOOTNOTE SEQUENCE statement.

Level Footnotes can be kept for individual tables.

Default If a footnote is not referenced in a table, it is not printed. If a footnote has no explicit footnote symbol, a footnote number is generated for use as the symbol.

Example

```
SET FOOTNOTE COMMERCE
        TEXT  "Source:  Department of Commerce";
KEEP FOOTNOTE COMMERCE;
```

Effect The text of the footnote called COMMERCE will print at the end of all tables. No footnote symbol will be printed. Since there is no footnote symbol and no indentation in the text, the footnote text will begin printing at the left edge of the table with no indentation.

Restrictions The FORMAT statements **DELETE FOOTNOTE**, **RETAIN FOOTNOTE** and **KEEP FOOTNOTE** can apply to the same footnote. If they do, and there is a conflict between them, the last one encountered by the system will win.

If both the symbol and the text for a footnote are null strings (''), the footnote cannot be kept in the table.

LINE

This statement has been replaced by
BOLD RULE WEIGHT *= n*
and
RULE *rule-options;*

Format

[BOLD] LINE WEIGHT = n;	or
[BOLD] LINE DOUBLE;	or
[BOLD] LINE SINGLE;	or
[BOLD] LINE WEIGHT = n	DOUBLE or SINGLE;

Meaning RULE is a synonym for LINE in these statements. See the RULE statements for more information.

MARGINS (LEFT, RIGHT, TOP, BOTTOM)

Format There are four MARGIN actions.

```
LEFT MARGIN = amount  [ unit ];
RIGHT MARGIN = amount  [ unit ];
TOP MARGIN = amount  [ unit ];
BOTTOM MARGIN = amount  [ unit ];
```

where **amount** is a number and **unit** is optional. If no unit is specified, **characters** are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as **inches**, **cm** or **points**.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning Leave a margin of the size indicated by n. One or more of the margins can be changed for a table. The margins do not have to be the same size. The table is positioned (centered or aligned left or right) within the space remaining after the left and right margins sizes are subtracted from the page width. The table begins on the first line after the top margin and breaks at the bottom margin if it is longer than one page.

Level Margins can be controlled at the individual table level. Margins cannot change within a table.

Default

```
LEFT MARGIN = 5;
RIGHT MARGIN = 5;
TOP MARGIN = 6;
BOTTOM MARGIN = 6;
```

TED Note There are additional options for adjusting margins when printing tables directly from TED. In the Print menu, you can set different print margins for odd and even pages. This can be particularly useful if you need wider margins on alternate pages for binding the table pages together.

Example

```
TOP MARGIN = 2 cm;
BOTTOM MARGIN = 3 cm;
LEFT MARGIN = 0;
RIGHT MARGIN = 0;
```

Effect Set the top margin to 2 cm and the bottom margin to 3 cm. Remove the left and right margins by setting them to 0.

To remove ALL margins, set all margins to 0. This is not recommended

Restrictions The page must be wide enough to hold the stub + the margins + the widest column.

The default margins will work correctly in most cases. If you want to change the margins with the MARGIN statement, we recommend that you express margin sizes in terms of inches, cm or points so that their absolute sizes for printing will not depend on the font size in effect. If margin sizes are expressed in terms of characters, the results will sometimes be acceptable, but they will often be something other than what you intended and, at worst, you will get table output that looks "buggy". If parts of a table are "lost" at the top or right edges of the paper, check your margin specifications.

For most laser printers, a margin size of at least .25 inches is required. If you try to print something that fills the paper to the edges, you may lose part if it.

MAXIMUM FOOTNOTE SYMBOL WIDTH

Format MAXIMUM FOOTNOTE SYMBOL WIDTH = n;

where **n** is an integer that indicates a number of characters. MAX can be substituted for the word MAXIMUM.

Meaning MAXIMUM FOOTNOTE SYMBOL WIDTH controls the spacing between the left edge of a table and the footnote symbols at the bottom of the table.

If you have footnote symbols of different widths in the same table, they may look best at the bottom of the table if the footnote symbols are right-aligned with each other. In order to assure this, the maximum number of characters in footnote symbols must be known. If the maximum number specified is too large, you will get more space between the symbols and the left edge of the table than you probably want. If the maximum number specified is too small, the footnote symbols will not be right-aligned.

Note Most fonts have characters of varying widths. For example, the letter **O** is wider than the letter **I**. If you have several wide characters in a footnote symbol, you may need to use a larger number (greater than the number of characters) for your maximum symbol width.

By default, footnote symbols are right-aligned within a space of 3. Thus, if all of your footnote symbols are only one character wide, the symbols will be indented when displayed at the bottom of a table. If you do not want them to indent, you can set the maximum footnote symbol width to 1.

Level Footnote symbol width can be specified for individual tables.

Default MAXIMUM FOOTNOTE SYMBOL WIDTH = 3;

Aligning Footnote Symbols of Varying Widths

Examples Following is a set of footnote statements and a table that displays these footnotes using the default footnote symbol width of 3. Note that KEEP FOOTNOTE causes two of the footnotes to be displayed even though they are not referenced in the table.

```
set footnote sum symbol font h 's'
    text 'The total is less than the sum of the individual '
'items because many workers participate in plans with more '
```

```
'than one feature.';

set footnote dental
    text 'Participants who elected dental coverage only '
'were not included in this tabulation.';

set footnote df  text 'Data collected for the month of March.';

set footnote src  symbol 'Source:'  text 'Department of Health.';

set footnote nt symbol 'Note:'
    text 'This table shows footnotes with defaul alignment.';

keep footnote src;
keep footnote nt;
footnote sequence src nt sum df dental;
```

Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms, 2005[1]

Cost containment feature	All participants	Technical and clerical participants	Production participants
Total[2,s]	100	100	100
Incentive to seek second surgical opinion	35	40	28
Higher payment for generic prescription drugs	7	7	6
Separate deductible for hospital admission	9	9	7
Urging prehospitalization testing	47	52	43
Preadmission certification requirement	16	15	16
Incentive to audit hospital statement	2	2	1

Source: Department of Health.
Note: This table shows footnotes with default alignment.
s The total is less than the sum of the individual items because many workers participate in plans with more than one feature.
1 Data collected for the month of March.
2 Participants who elected dental coverage only were not included in this tabulation.

The footnote symbols vary from 1 to 7 characters in width, so the texts are starting at different points, giving a somewhat ragged effect. We can add the following statement to the width of the widest symbol:

```
MAXIMUM FOOTNOTE SYMBOL WIDTH = 7;
```

Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms, 2005[1]

Cost containment feature	All participants	Technical and clerical participants	Production participants
Total[2,s]	100	100	100
Incentive to seek second surgical opinion	35	40	28
Higher payment for generic prescription drugs	7	7	6
Separate deductible for hospital admission	9	9	7
Urging prehospitalization testing	47	52	43
Preadmission certification requirement	16	15	16
Incentive to audit hospital statement	2	2	1

Source: Department of Health.
Note: This table shows footnotes with a maximum footnote width setting.
s The total is less than the sum of the individual items because many workers participate in plans with more than one feature.
1 Data collected for the month of March.
2 Participants who elected dental coverage only were not included in this tabulation.

If you have widely varying widths for symbols and also have footnote texts that wrap to multiple lines, you may wish to indent the wrapped lines of the long footnote texts so that they line up with the others. You can do this by inserting an INDENT in the texts near but after the beginning of the text. The extra indent will apply to all lines following the first line of the text. Use an INDENT amount that is one higher than the maximum symbol width.

```
MAXIMUM FOOTNOTE SYMBOL WIDTH = 7;

set footnote sum symbol font h 's'
   text 'The total ' INDENT 8 'is less than the sum of the individual '
'items because many workers participate in plans with more '
'than one feature.';

set footnote dental
   text 'Participants' INDENT 8 ' who elected dental coverage only '
'were not included in this tabultion.';
```

Following is the table with all lines of footnote text starting at the same point.

Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms, 2005[1]

Cost containment feature	All participants	Technical and clerical participants	Production participants
Total[2,s]	100	100	100
Incentive to seek second surgical opinion	35	40	28
Higher payment for generic prescription drugs	7	7	6
Separate deductible for hospital admission	9	9	7
Urging prehospitalization testing	47	52	43
Preadmission certification requirement	16	15	16
Incentive to audit hospital statement	2	2	1

Source: Department of Health.
Note: This table shows footnotes with a maximum footnote width setting.
s The total is less than the sum of the individual items because many workers participate in plans with more than one feature.
1 Data collected for the month of March.
2 Participants who elected dental coverage only were not included in this tabulation.

Aligning Footnotes to the Left

Example Following is a set of NOTE and FOOTNOTE statements. All of the footnote symbols are one character wide. At the bottom of the table, the notes will be aligned to the left edge of the table and the footnotes will be indented to allow for the default maximum symbol size of 3. If we want to align all footnotes to the left so that they line up with the notes, we can specify a maximum symbol width of 1.

Note If you want the footnotes to be slightly indented but less than the default amount, you can use a maximum symbol size of 2.

```
set footnote sum symbol font h 's'
    text 'The total is less than the sum of the individual '
'items because many workers participate in plans with more '
'than one feature.';

set footnote dental
    text 'Participants who elected dental coverage only '
'were not included in this tabulation.';

set footnote df  text 'Data collected for the month of March.';
```

```
set note src  'Source: Department of Health.';

set note nt 'Note: This table shows footnotes with a maximum footnote '
'width setting.';

footnote sequence src nt sum df dental;

MAXIMUM FOOTNOTE SYMBOL WIDTH = 1;
```

Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms, 2005[1]

Cost containment feature	All participants	Technical and clerical participants	Production participants
Total[2,s]	100	100	100
Incentive to seek second surgical opinion	35	40	28
Higher payment for generic prescription drugs	7	7	6
Separate deductible for hospital admission	9	9	7
Urging prehospitalization testing	47	52	43
Preadmission certification requirement	16	15	16
Incentive to audit hospital statement	2	2	1

Source: Department of Health.
Note: This table shows footnotes with a maximum footnote width setting.
s The total is less than the sum of the individual items because many workers participate in plans with more than one feature.
1 Data collected for the month of March.
2 Participants who elected dental coverage only were not included in this tabulation.

ODS OUTPUT (UNIX only)

Format ODS OUTPUT = YES or NO or PROMPT;

Normally, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for **ODS**, and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**.

PAGE LENGTH

Format

```
PAGE LENGTH = amount  [ unit ];
```

where **amount** is a number and **unit** is optional. If no unit is specified, lines are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as **inches**, **cm** or **points**.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning The table is divided into pages according to the number of lines per page specified by **amount**. The number of lines available for the table is determined by subtracting the top and bottom margins from the page length. For the second and following pages of a table, the title, wafer label (where applicable) and heading labels are repeated. Stub labels may be repeated for the first data row if required to properly identify the data.

Level Page length is controlled at the request level. All tables within the same table request will use the same page length specification.

Default The system default is set at installation time and stored in the file called **profile.tpl**. When the system is installed, **profile.tpl** is stored in the TPL TABLES system directory.

You can change the system default by editing the PAGE LENGTH specification in **profile.tpl** and saving the result back in the system directory.

If you want to leave the system default as is but change the default for a set of table requests, you can do so by making a copy of **profile.tpl** with a different PAGE LENGTH specification and saving it in the directory where you are running your table jobs.

Example

```
PAGE LENGTH = 50;
TOP MARGIN = 3;
BOTTOM MARGIN = 2;
```

Effect Tables will be divided into pages of length 50. The top and bottom margins will use a total of 5 lines, leaving 45 lines per page for the tables.

Restrictions The page must be long enough to hold 1 row of data plus the margins, table title, wafer label (where applicable), heading, footnotes (where applicable), and stub label nesting required to identify a row of data.

You should express page length in something other than lines. This is because you can choose different character sizes. If page length is expressed in lines, the length of the page will vary as the character size changes. This result is usually undesirable. Specify page length in inches, cm or points, or use the PAPER statement to select a page size.

PAGE LENGTH AUTOMATIC

This statement has been replaced by text export with the autosize option.

Format

PAGE LENGTH = AUTOMATIC;

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether. **AUTO** can be used as an abbreviation for **AUTOMATIC**.

Meaning

This statement can only be used with text tables. The page length will be set at the length needed to contain all tables in the request without page breaks. This statement should not be used with banked tables. If your table is wide, make sure that the page width is large enough to hold all columns without banking.

PAGE LENGTH AUTOMATIC; is most useful in the following two cases:

1. If you are customizing your table output for use with other software and want to get an unbroken stream of data rows.

2. If you are reviewing a table on the screen and want to look at it as one long page, uninterrupted by the extra space, title and heading labels that would otherwise appear at each page break. For this purpose, you may find it convenient to use **PAGE WIDTH = AUTOMATIC;** to prevent banking of a wide table. The combination of automatic page length and automatic page width will allow you to review tables on the screen without the tables being split into sections as would be required for printing on a particular size of paper.

Level

Page length is controlled at the request level. All tables within the same table request use the same page length specification.

Default

Tables break at the end of each page.

Example

PAGE LENGTH = AUTOMATIC;

Effect

Each table will be formatted as one long page. If there are multiple tables in the request, there will be one long page containing all of the tables.

Restrictions

This statement should not be used with banked tables.

If this statement is used for tables that have wafers, all wafer titles will be retained. If you do not want them to intersect the table, you can nest the wafer expression into the top of the stub instead of using a separate wafer specification.

PAGE MARKER

Format

```
PAGE MARKER = marker specification;
BOTTOM PAGE MARKER = marker specification;
```

Normally, there can be only one PAGE MARKER for a table. If both PAGE MARKER and BOTTOM PAGE MARKER are used, there can be one marker at the top of the page and another at the bottom. For text tables BOTTOM PAGE MARKER is not supported.

The **marker specification** can be one or more of the following:

```
NUMBER
COUNT
START = n
TOP
BOTTOM
RIGHT
LEFT
RIGHT THEN LEFT
LEFT THEN RIGHT
DATE
TIME
JOB
ODD
EVEN
label segments
```

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether. The specifications can be in any order.

Meaning PAGE MARKER is used to add identifying information to table pages. The page marker can contain any combination of page number, date, time, job id and label segments. Starting page number can also be set.

Level PAGE MARKER can be specified for each table separately. However, start number and marker location will carry forward to following tables unless they are explicitly reset. Further, the start number and marker location are independent of marker text even though they may appear in the same FORMAT statement.

Example

```
PAGE MARKER = TOP LEFT NUMBER START 3;
FOR TABLE 2: PAGE MARKER = START 5 "Page " NUMBER;
```

The result will be that all page markers will be in the top left corner of the page. Table 1 will just have numbers starting at 3. Table 2 will have "**Page** " and numbers starting at 5. Table 3 and following tables will just have numbers, but the numbers will start where table 2 left off since there is no new START term for table 2.

Default Tables do not have page markers. If PAGE MARKER is specified, the default page start is **START = 1** and the default marker location is **TOP CENTER**. A BOTTOM PAGE MARKER is always at the bottom of the page with a default alignment of **CENTER**.

Page Numbering

PAGE MARKER = NUMBER; will produce page numbers that are centered vertically and horizontally within the top margin of a page. **PAGE MARKER = NUMBER START 5;** will cause the first table to start numbering at 5. Succeeding tables will continue the numbering unless a new **PAGE MARKER = START n;** is specified.

Example

```
FOR TABLE 1:  PAGE MARKER  RIGHT  'A'  NUMBER;
FOR TABLES 2 AND 3:
        PAGE MARKER  RIGHT  'B' NUMBER  START 1;
```

Effect This example uses a combination of a letter and a page number to mark groups of table pages. Thus, for example, if the first table is to be inserted in Section A of a document, the page markers can be "A1", "A2", "A3", etc. If the second and third tables are to be inserted in Section B of the same document, they can have markers of "B1", "B2", "B3", etc. Since START 1 is included in the marker, the numbering for the second and third tables will restart at 1 on the first page of the second table; the numbering will continue on through the third table. All page markers will be in the top right corner of the page.

ODD and EVEN

When a page marker for a table includes both NUMBER and ODD, the page numbers for that table will all be odd. For example, if ODD is specified for the first table, its pages will be numbered 1,3,5,7,...

Use of EVEN will result in even numbered table pages.

If a table would normally begin with an even number but ODD is specified, then one is added to its starting number so that it will begin with an odd number.

Example

```
FOR TABLE 2:
        PAGE MARKER = "PAGE " NUMBER START 7 EVEN;
```

Effect Page numbering for table 2 will begin with 8, the first even number after the specified start number.

ODD and EVEN are useful when a document has tables on every other page or when wide tables are displayed as facing page pairs. This latter is done in TPL TABLES by creating two tables with corresponding stubs and the heading split across the two tables. (See the entry for STUB RIGHT for more details on this.) The first table should have a page marker which includes EVEN while the second should have the same starting page but should include ODD in its page marker.

Page Count

COUNT can be used to get a page count. It specifies the total number of table pages produced by a job. It is not affected by START or by the presence of multiple Page Markers in the job. An example of a statement using COUNT is:

Example

```
PAGE MARKER = "Page " NUMBER " of "  COUNT;
```

If there are 10 pages of table output in the job, the marker for the first page will be "Page 1 of 10"; the marker for the second page will be "Page 2 of 10"; and so on to the last page with a marker of "Page 10 of 10".

Marker Location

The location of page markers can be controlled by using TOP, BOTTOM, LEFT, RIGHT, or CENTER. CENTER is the default location. If LEFT is specified, the marker will start on the left page margin. If RIGHT is specified, the marker will end on the right margin. If TOP is specified, the marker will be placed 1/2 of the top margin down from top of page. If BOTTOM is specified, the marker will be placed 1/2 of the bottom margin above the bottom of the page.

If you are using TOP or BOTTOM, you may wish to increase the top or bottom margin specification beyond the standard 1 inch to keep the markers from appearing too near the top or bottom edge of the paper. This is especially important if you are using a multiline marker with a laser printer, since laser printers do not print on the top and bottom 1/4 inches of the paper.

If you want marker locations to alternate between left and right pages, use LEFT THEN RIGHT or RIGHT THEN LEFT.

As with START number, the marker location will continue across tables unless a new location is specified. For example, if you begin with RIGHT THEN LEFT for the first table, the marker location will alternate between right and left pages for all following tables unless an explicit LEFT, RIGHT, or CENTER is specified in a FORMAT statement for a later table.

Multiple Page Markers

If there are multiple PAGE MARKER statements for a table, each one will override the preceding one so that there will be only one marker. For PostScript tables, you can have two markers, one at the top of the table and one at the bottom, by using both a PAGE MARKER and a BOTTOM PAGE MARKER. The same options are available for both the top and bottom markers.

Example

```
PAGE MARKER RIGHT THEN LEFT "Research Bulletin ATN-05";
BOTTOM PAGE MARKER "Page " NUMBER;
```

Effect The text "Research Bulletin ATN-05" will be displayed at the top right of the first page, the top left of the second page, and so on. The first page will also have "Page 1" centered at the bottom, the second page "Page 2", and so on.

Notes If you specify BOTTOM in a regular page marker and also have a BOTTOM PAGE MARKER statement for the same table, both markers will go at the bottom with one possibly overlaying part of the other.

Alignments and Spacing within Page Markers

LEFT, RIGHT, or CENTER can only be used at the beginning of a marker, before any label segments. This alignment applies to the entire page marker. If you want more control of spacing within the marker, see SPACE and SPACE TO in the chapter called "Labels".

Other Options

In addition to or instead of page numbers, a page marker can contain anything allowed in a label except a footnote. Also there are some built-in special items. These are DATE, TIME, and JOB.

Example

```
PAGE MARKER = TOP LEFT THEN RIGHT
"Page " NUMBER " for job " JOB
" run on " DATE " at " TIME;
```

Effect If job TPL873 is run on March 4, 2003 at 11:24 A.M., the output will be:

```
Page 1 for job TPL873 run on 3/4/03 at 11:24:00 AM
```

which will appear in the upper left corner of Page 1. Page 2 will have its marker in the upper right corner of the page.

Note that the format of date and time will be affected by the COUNTRY statement. The examples in this section are shown in the format for the default country, **COUNTRY = US;**.

Since page markers are vertically centered within their margin, use of slashes at the start of a page marker specification will push the marker text down while slashes at the end of the page marker specification will raise the marker.

Example If you request that TPL TABLES convert your tables to encapsulated PostScript for inclusion in a desktop publishing document, TPL TABLES will create one file for each table page. Assume a table request with 3 tables, where the first table has two pages and the other tables have a single page. The following will produce page markers which contain the job number and the file names created for the table pages.

```
FOR TABLE 1: PAGE MARKER = JOB " P" NUMBER "T1.EPS";
FOR TABLE 2: PAGE MARKER = JOB " P" NUMBER "T2.EPS";
FOR TABLE 3: PAGE MARKER = JOB " P" NUMBER "T3.EPS";
```

Effect If the job is TPL345, then the markers for the four pages will be:

```
TPL345 P1T1.EPS
TPL345 P2T1.EPS
TPL345 P3T2.EPS
TPL345 P4T3.EPS
```

These page markers will be automatically deleted if the pages are imported into a desktop publishing system.

Windows Note

If you are exporting encapsulated PostScript from TED in a script or using the ENCAPS program, the default file names are as shown in the above example. If you are exporting encapsulated PostScript interactively from

TED, the naming convention for the exported files is different. See the Script appendix or TED Help for naming conventions and options.

UNIX Note

Under UNIX, the encapsulated PostScript files have the lower case suffix **.eps**.

4-Digit Year

You can choose to have year displayed with 4 digits by editing the file called **country.tpl**. This file is installed in the TPL system directory, but you may also have customized copies in other directories where you run TPL jobs.

The **country.tpl** file is a simple ascii text file. The following options for date format are shown near the top of the file:

```
*  date format code
*    0 -> mm/dd/yy
*    1 -> dd/mm/yy
*    2 -> yy/mm/dd
*    3 -> mm/dd/yyyy
*    4 -> dd/mm/yyyy
*    5 -> yyyy/mm/dd
```

To choose 4-digit year as your standard, edit the entry for your country by changing the value in the fourth column to the number that matches the date format you want.

All current dates displayed in your TPL jobs will show the year in four digits. These include run dates in the output file as well as dates that are specified with the PAGE MARKER statement.

UNIX Users: Add a COUNTRY statement in your **profile.tpl** file if you do not already have one. For example:

```
COUNTRY = US;
```

Windows Users: If you make changes to **country.tpl**, add a COUNTRY statement to your profile, or change a COUNTRY statement in your profile, you need to ***restart TPL*** to activate the changes.

PAGE WIDTH

Format

```
PAGE WIDTH = amount  [ unit ];
```

where **amount** is a number and **unit** is optional. If no unit is specified, **characters** are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as **inches**, **cm** or **points**.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning The table is formatted to fit within a page width of n characters. The table is aligned within the space remaining after the left and right margins are subtracted. If the table is too wide for the space, it is divided into as many partitions as necessary with each partition beginning on a new page. The stub labels are repeated in each partition.

Level Page width is controlled at the request level. All tables within the same table request will use the same page width specification.

Default The system default is set at installation time and stored in the file called **profile.tpl**. When the system is installed, **profile.tpl** is stored in the TPL TABLES system directory.

You can change the system default by editing the PAGE WIDTH specification in **profile.tpl** and saving the result back in the system directory.

If you want to leave the system default as is but change the default for a set of table requests, you can do so by making a copy of **profile.tpl** with a different PAGE WIDTH specification and saving it in the directory where you are running your table jobs.

Example

```
PAGE WIDTH = 100;
LEFT MARGIN = 2;
```

Effect The table will be formatted within a page width of 100 characters. It will be centered within the 96 character space remaining after the left and right margins are subtracted from the page width.

Restrictions You should express page width in something other than characters. This is because you can choose different character sizes. If page width is expressed in characters, the width of the page will vary as the character size

changes. This result is usually undesirable. Specify page width in inches, cm or points, or use the PAPER statement to select a page size.

PAGE WIDTH AUTOMATIC

This statement has been replaced with text table export with the autosize option.

Format PAGE WIDTH = AUTOMATIC;

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether. **AUTO** can be used as an abbreviation for **AUTOMATIC**.

Note This statement can only be used with text tables.

Meaning TPL TABLES calculates the page width to be the sum of the stub width, the widths of all columns, and the left and right margins.

Level Page width is controlled at the request level. All tables within the same table request will use the same page width specification.

Default The system default is set at installation time and stored in the file called **profile.tpl**. When the system is installed, profile.tpl is stored in the TPL TABLES system directory.

Example (for a table with 6 columns):

```
PAGE WIDTH = AUTOMATIC;
LEFT MARGIN = 4;
RIGHT MARGIN = 4;
STUB WIDTH = 30;
COLUMN WIDTH = 15;
```

Effect The table will be formatted for a page width of 128 characters (4 + 4 + 30 + 15 * 6).

PAPER

Format PAPER = papersize;

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning PAPER can be used to select one of the standard built-in paper sizes. Options are:

LETTER	(8.5 in x 11 in)
LEGAL	(8.5 in x 14 in)
A3	(42.0 cm x 29.7 cm)
A4	(21.0 cm x 29.7 cm)
B5	(18.2 cm x 25.7 cm)

The PAPER statement is used in place of the combination of PAGE WIDTH and PAGE LENGTH. For example, the statement

```
PAPER = LETTER;
```

gives the same result as the pair of statements

```
PAGE WIDTH = 8.5 IN;
PAGE LENGTH = 11 IN;
```

To determine the amount of space available for the table, deduct the margins from the page size. The tables will begin on the first line after the top margin and will be aligned within the left and right margins.

The default paper size is set at installation time and stored in the file called **profile.tpl** in the TPL TABLES system directory. You can change the default paper size after installation by editing **profile.tpl**.

Level Paper size is controlled at the request level. All tables within the same table request will be formatted for the same page size.

Windows Default PAPER = LETTER;

UNIX Default At installation time, you will be given a choice of paper sizes.

Example PAPER = A4;

Effect All tables will be formatted for the A4 paper size (21.0 cm x 29.7 cm).

PDF OUTPUT (UNIX only)

Format PDF OUTPUT = YES or NO or PROMPT;

Normally, when you have created tables, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for **PDF**, and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**.

POSTSCRIPT

Postscript = no; *is no longer recommended.*
Text table output can be obtained in Postscript mode by using an export option.

Format POSTSCRIPT = YES; or

POSTSCRIPT = NO;

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning If you specify

POSTSCRIPT = YES;

table output will be coded in PostScript.

If you specify

POSTSCRIPT = NO;

your tables will be formatted as ASCII text. FORMAT statements that apply only to PostScript will be ignored.

In PostScript mode, you can choose from any of the PostScript fonts listed in the FONT statement section of this chapter. Most of the fonts are proportional. This means that the character widths vary from character to character. For example, the letter "**i**" takes up less space than the letter "**m**". TPL TABLES will do all of the format adjustments needed for proper alignment with proportional fonts.

You can change the overall line spacing with the EXTRA LEADING statement. Line spacing for a particular line will also be affected by the fonts used for different parts of the line. For labels, spacing is determined by the largest font used in the label. For data rows, if the DEFAULT FONT is larger than any of the label fonts used for the row, the DEFAULT FONT will determine the line spacing.

You can also print your tables sideways on the page. See the ROTATE statement for details.

Interaction of Size Specifications with PostScript

Following is a list of the size specifications that can be affected by a change to PostScript mode. If these sizes are expressed in terms of characters or lines, rather than in centimeters, inches or points, the absolute sizes for printing will depend on the font size in effect.

```
PAGE LENGTH = size;
PAGE WIDTH = size;
COLUMN WIDTH = size;
STUB WIDTH = size;
STUB CONTINUATION = size;
STUB INCREMENT = size;
STUB START = size;
STUB STOP = size;
TOP MARGIN = size;
BOTTOM MARGIN = size;
LEFT MARGIN = size;
RIGHT MARGIN = size;
```

Size can be specified as a number followed by:

```
inch
inches
in
```

```
ins
cm
points
pt
pts
```

Fractional sizes must be specified as decimal numbers. For example,

```
STUB WIDTH = 2.5 IN;
```

In general, if you will be switching between PostScript and non-PostScript modes, sizes other than page and margin size will work well in both modes if they are expressed in characters. If you are using a proportional font in PostScript, you will often be able to get more characters within a given width. The most common exception is when you have a label in upper case letters. Upper case letters are often wider in a proportional font.

Sizes specified in inches, centimeters or points will work for text tables as well as in PostScript tables. If you are not requesting PostScript output, the measures will be converted to 12 pt equivalents in characters. With 12 pt type, 1 inch can contain 10 characters in the horizontal direction and 6 lines in the vertical direction.

Page and Margin Sizes

In PostScript mode, page and margin sizes should be expressed in terms of centimeters, inches or points so that their absolute sizes for printing will not depend on the font size in effect. In the case of page size, you can also use the PAPER statement to pick a standard paper size.

The system default margins will work correctly in PostScript mode. If you want to change the margins with the MARGIN statement, we recommend that you express margin sizes in terms of inches, cm or points.

If your table is not positioned properly on the paper or if parts of the table are "lost" at the top or right edges of the paper, check your page and margin specifications.

Treatment of Footnote Symbols in PostScript

Built-in Footnotes

(EMPTY, SMALL, ERROR, NO_FIT, ZERO and SEE_END)

At the bottom of a table, the footnote symbol font is used and the symbol is raised. In data cells, the default font is used so that the symbol has the same font as the data, and the symbol is not raised.

All Other Footnotes

The footnote symbol font is always used and the symbol is always raised. When the footnote symbol is alone in a data cell, it is enclosed in parentheses. The parentheses are always in the default font.

If multiple footnotes occur at the same point in a label, the symbols are displayed side by side, separated by commas.

Level PostScript is controlled at the request level. If PostScript is specified, all tables in the request will be created in PostScript mode.

Windows Default POSTSCRIPT = YES;

The statement is put in **profile.tpl** at installation time along with some FONT defaults. You can change any of these defaults after installation by editing **profile.tpl**, or you can override them with FORMAT statements in your format requests.

UNIX Default The default is set at installation time.

If you are working with a PostScript printer and would like to have PostScript defaults entered in your system profile, you can set these defaults when you install TPL TABLES. See the UNIX Installation Instructions for details. You can change any of these defaults after installation by editing **profile.tpl**, or you can override them with FORMAT statements in your format requests.

Example

```
POSTSCRIPT = YES;
FOR TABLE 2:  ROTATE;
```

Effect All tables will be prepared in PostScript format. The second table will be rotated to print sideways on the page. All other tables will be printed upright.

Restrictions Most laser printers require a margin. If you try to print something that fills the paper to the edges, you may lose part if it. Therefore, we do not recommend margin sizes of 0 when using PostScript.

The DATA TABLES and PAGE LENGTH AUTOMATIC statements cannot be used in PostScript mode.

PRINT (UNIX only)

Normally, TPL TABLES will prompt you at the end of a job to find out whether you want to print your OUTPUT file or your tables. You can use the following statements to select the print options in advance.

Format

```
PRINT OUTPUT = YES or NO or PROMPT;
PRINT TABLES = YES or NO or PROMPT;
```

The default for both statements is **PROMPT**.

PRINT COMMAND (UNIX profile only)

Format

```
PRINT COMMAND = 'command' ;
```

where **command** is a UNIX print command.

Meaning TPL TABLES will direct its output to the default printer for your computer. If you wish to change this, you may modify the PRINT COMMAND statement in the profile.

Level The command takes effect for the entire table request.

Default

```
PRINT COMMAND = 'lp';
```

Example

```
PRINT COMMAND = 'lp -dpost';
```

where **post** is the name of your PostScript printer.

Effect Output and tables will be directed to the PostScript printer.

Note Unlike most FORMAT statements, PRINT COMMAND will only work if it is placed in the profile, not in the FORMAT request. If different people wish to use different printers, they should create local profiles with different print statements.

RAISE FOOTNOTE SYMBOL

Format RAISE FOOTNOTE SYMBOL = amount;

where **amount** is a number. The word **IS** can be used in place of =. Both are optional and can be left out altogether.

Meaning This statement can be used to adjust the height of a footnote symbol relative to the footnote text and the labels or data values being footnoted. The amount specifies a fraction of the height of adjacent numbers or text. The statement is ignored in text table export.

RAISE FOOTNOTE SYMBOL 0; will prevent the footnote symbols from being raised. In other words, they will be printed at the same level as the adjacent numbers or text.

Level Footnote height can be specified for individual tables.

Default RAISE FOOTNOTE SYMBOL .3;

Example RAISE FOOTNOTE SYMBOL .2;

Effect All footnote symbols will be raised 2/10 of the height of adjacent numbers or text. This is slightly lower than the default height.

Restrictions The amount must be greater than or equal to zero.

In general, you will not want to raise the footnote *above* the default amount unless you have added extra leading (see the FORMAT statement called **EXTRA LEADING**) or have selected a very small footnote symbol font relative to the other fonts used in your tables. If the footnote symbols are raised substantially, they will overlay the lines above in an undesirable way.

RANK ON VALUES

Format

```
RANK ON VALUES;
RANK ON VALUES SMALLEST IS 1;
RANK ON VALUES BIGGEST IS 1;
DO NOT RANK ON VALUES;
```

where VALUES can be singular or plural. The = sign can be used in place of IS. Both are optional and can be left out altogether. ONE, FIRST, or any other word you prefer can be used in place of 1.

Meaning

TPL TABLES has two different ways of specifying and displaying ranking in tables. Both apply only to values in a column.

The first is specified in the table request using a statement similar to a DEFINE to create a RANK control variable which is nested in the stub. The results are displayed by rearranging the rows of the table in rank order. The rank variable may be nested below another control variable so that the table will have separate subrankings. It is also possible to rank only the largest (or smallest) values and then aggregate all of the other values into a single subcategory. See the RANK statement for more details.

The second method of displaying ranking is specified by the Format statement RANK ON VALUES. One or more columns of the table are picked, and the values in those columns are replaced by a rank number for the values. Values can be excluded from the ranking but the values cannot be broken into subrankings. This representation of ranking is especially useful if you wish to rank more than one column of values in the same table.

If there is no specification of SMALLEST or BIGGEST in the RANK ON VALUES statement, it is assumed that BIGGEST IS 1 and the biggest value in the column will be replaced with the number 1.

Notes

Empty cells are not included in the ranking. If you wish to exclude cells from the ranking you may do so by setting the cell values for these cells to NULL. (See the REPLACE VALUE statement.)

For a table with multiple wafers, the ranking will be applied to all wafers if there is no wafer specification in the RANK ON VALUES statement. The ranking will restart at the beginning of each wafer.

When two or more values are the same, they will be replaced by the same rank number and the rank number following will be adjusted upward. For

example, if a column has the values 15, 18 and 18, the rank numbers for biggest to smallest will be 3, 1, and 1.

Level Ranking can be specified for individual columns.

Default DO NOT RANK ON VALUES;

Example The following table statement produces a table with three identical columns.

```
Table R1 'Average Income Per Person with Ranking':
Stub TOTAL then HOUSEHOLD_SIZE;
Head AVG_INCOME then AVG_INCOME then AVG_INCOME;
```

The following format statements leave the 1st column unchanged. They rank the 2nd and third columns in opposite directions and label them appropriately. Total values in the first row are replaced with NULL so that they will not be included in the rankings.

```
For Table R1 Row 1 Columns 2 and 3: Replace value with NULL;

For Table R1 Column 2: Rank on Values;
For Table R1 Column 3: Rank on Values Smallest is 1;

For Table R1 Column 2 Variable AVG_INCOME:
        Replace label with "Rank Top to Bottom";
For Table R1 Column 3 Variable AVG_INCOME:
        Replace label with "Rank Bottom to Top";
```

Effect The values in columns 2 and 3 are replaced by rank numbers. For column 2, the biggest value gets a rank of 1. For column 3, the smallest value gets a rank of 1.

Average Income Per Person with Ranking

	Avg	Rank Top to Bottom	Rank Bottom to Top
Total	13,841	–	–
Size of Household			
1 in Household	17,617	1	16
2 in Household	15,860	2	15
3 in Household	12,213	3	14
4 in Household	10,455	4	13
5 in Household	8,027	5	12
6 in Household	6,422	6	11
7 in Household	5,532	7	10
8 in Household	5,414	8	9
9 in Household	4,283	9	8
10 in Household	2,908	12	5
11 in Household	3,345	10	7
12 in Household	1,886	14	3
13 in Household	3,202	11	6
15 in Household	609	16	1
18 in Household	1,512	15	2
21 in Household	2,269	13	4

Restrictions

RANK ON VALUES can only be applied to columns.

Only one ranking can apply to a column. Subrankings are not supported.

Column rankings may be restricted by setting the cells to NULL, but the FOR clause used for the restrictions must be at the cell (or perhaps row or column) level. It cannot be specified for Variables or Control variable conditions.

If RANK ON VALUES is applied to a table that was created using a RANK variable in the table request, RANK ON VALUES will override the ranking specified in the table request and rows will be displayed in the original, unranked order instead of being reordered by rank value.

REPLACE COLOR

REPLACE COLOR is useful when pre-viewing color tables on a monochrome printer, because it lets you replace colors with special fonts. See the FORMAT statement called COLOR = NO for details.

REPLACE DIVIDE CHARACTER

This statement has largely been replaced by:
DELETE DOWN RULES;

Format

```
REPLACE DIVIDE CHARACTER WITH 'char';
REPLACE DIVIDE CHARACTER WITH 'char' EXCEPT ZERO;
```

Meaning

Replace the column dividers from the top of the heading to the bottom of the table with the single character enclosed in quotes.

The most common use of the statement is to replace the column dividers with blank. In this case, the horizontal lines in the heading will be "broken" with a blank at any point where the column divider passes through the heading.

If you want to have a solid line at the bottom of the heading, add the words **EXCEPT ZERO** (for row 0 of the table) to the statement. For example:

```
REPLACE DIVIDE CHARACTER WITH ' ' EXCEPT ZERO;
```

If you want all horizontal heading lines to be solid, replace the divide character with a null character as follows:

```
REPLACE DIVIDE CHARACTER WITH '';
```

The divide character can only be replaced with a blank or null character. If any other character is specified, the statement is ignored and the default character '|' is used. For text tables, other characters may be used.

Level

The divide character can be controlled at the individual table level. The divide character cannot change within a table.

Default

```
REPLACE DIVIDE CHARACTER WITH '|';
```

Example

```
REPLACE DIVIDE CHARACTER WITH '*';
```

Effect

If you are producing a text table, the column dividers will be replaced with vertical lines of the character *. Otherwise the statement will be ignored.

Example

```
REPLACE DIVIDE CHARACTER WITH ' ';
```

Effect The column dividers will be replaced with blanks from the top of the heading to the bottom of the table. This statement is similar to **DELETE DOWN RULES;** but applies to the heading in addition to the data section of the table.

Restrictions If this statement is used with DELETE DOWN RULES, the divide character will only apply to the heading.

REPLACE FILLER CHARACTER

Format REPLACE FILLER CHARACTER WITH 'characters';

The word **FILL** can be used in place of the word **FILLER**.

Meaning The character or multiple matching characters enclosed in quotes will be repeated between the stub label and the data section of the table for each line that has data.

If a **single** dot or other non-blank filler character is included within the quotes, then the character is repeated to fill the stub line. If there is not enough space for at least 2 filler characters, then no filler character is added to the line.

If a dot or other non-blank filler character **is repeated n times**, then the stub will be broken to a new line if necessary to allow enough space for **n** filler characters. In other words, there will always be a minimum of **n** filler characters between the stub label and the data section.

Level The filler character can be controlled at the individual table level. The filler character cannot change within a table.

Default REPLACE FILLER CHARACTER WITH '.';

The default '.' character repeated in the stub is sometimes called a **dot leader**.

Example REPLACE FILLER CHARACTER WITH ' ';

Effect The dotted line that normally follows the stub label on data lines will be replaced with blank space.

Example REPLACE FILLER CHARACTER WITH '....';

Effect There will always be a minimum of four dots between the end of the stub label and the data section of the table.

REPLACE FOOTNOTE / NOTE

The REPLACE FOOTNOTE statement allows you to modify the text or symbol for a footnote for an individual table.

Format

REPLACE FOOTNOTE *footnote-name* TEXT WITH *label*;
REPLACE NOTE *footnote-name* TEXT WITH *label*;
REPLACE FOOTNOTE SYMBOL WITH *symbol-specification*;

where *symbol-specification* may include a footnote symbol string, font, color and an alignment.

Example

```
Set Footnote Prelim text "Preliminary Results" symbol 'p';
For Table 2 : Replace footnote Prelim text "Final Results";
Replace footnote prelim symbol right color red 'f' font HB 6;
```

Effect

For all tables except table 2, cells or labels containing footnote prelim will be marked with a p and Preliminary Results will appear at the end of the tables. For table 2, the marked cells and labels will have an f and Final Results will appear at the end of the table.

Level

Replace Note and Replace Footnote apply at the wafer level if Footnotes Each Wafer is set. Otherwise it applies at the table level.

Note

REPLACE FOOTNOTE is like SET FOOTNOTE except:

- A footnote must already have been created by a SET FOOTNOTE before a REPLACE FOOTNOTE can be used.
- REPLACE FOOTNOTE can be applied to an individual table.
- REPLACE FOOTNOTE TEXT and REPLACE FOOTNOTE SYMBOL cannot be combined into a single statement

REPLACE HEADNOTE

Format

REPLACE HEADNOTE WITH label; or
REPLACE HEAD NOTE WITH label;

Meaning

The headnote is placed at the top of the table between the table title and the top of the heading. If there is also a wafer label, the headnote is placed below the wafer label. The order is shown in the table below:

Table Title

Wafer Label

Headnote

Stub Head	Heading
Row label	Data Value

If no alignment is specified, the headnote will be aligned with the left edge of the table. Alignment can be controlled by adding LEFT, RIGHT or CENTER to the headnote.

If multiple banks or wafers appear on the same page, the table title and headnote are not repeated after the first bank or wafer on the page.

Level

Headnotes can be specified at the table level.

Default

There are no headnotes. The default alignment when a headnote is specified is LEFT.

Example

```
REPLACE HEADNOTE WITH  '[Numbers in thousands.]';
```

Effect

The headnote **[Numbers in thousands.]** will be placed above the heading at the left edge of the table.

Note

You can choose type style and size for headnotes by including FONT specifications in individual headnote labels. See the chapter on labels for details. A default headnote font can be set with the HEADNOTE FONT statement. For example,

```
HEADNOTE FONT  HB 6;
```

REPLACE LABEL

Replacing a Variable Label

Format A FOR clause is required to identify the **variable** for which the label will be replaced.

```
(1) FOR VARIABLE variable-name:  REPLACE LABEL WITH label;
(2) FOR ROW n VARIABLE variable-name:
                REPLACE LABEL WITH label;
(3) FOR COLUMN n VARIABLE variable-name:
                REPLACE LABEL WITH label;
```

Meaning

(1) For all occurrences of the named variable, replace its label with a new one.
(2) For the named variable, if it occurs in the stub in row **n**, replace its label with a new one.
(3) For the named variable, if it occurs in the heading starting in column **n**, replace its label with a new one.

Level

(1) Variable labels can be controlled at the table level.
(2) An individual label in the stub can be changed. The change can be applied to selected wafers. If the table has banks, the corresponding label will be changed in each bank.
(3) An individual occurrence of a variable label can be changed in the heading. If the table has wafers, the corresponding occurrence of the label will be changed in all wafers, even if one or more specific wafers are specified in the FOR clause.

Note If a row is specified and the variable does not appear attached to that row, the format statement will be silently ignored. If a column is specified and the variable does not start in that column, the format statement will be silently ignored. It is sometimes tedious to figure out the exact row or column desired. You may specify a range of rows or columns which include the required row; e.g. instead of specifying

```
FOR COLUMN 7 VARIABLE AGE: REPLACE LABEL
        WITH 'Age of Person';
```

you may specify

```
FOR COLUMNS 5 TO 10 VARIABLE AGE: REPLACE LABEL
        WITH 'Age of Person';
```

Default The default variable label is determined when the variable is described in the codebook or defined in the table request.

Example

```
FOR TABLE 1 VARIABLE Employees:
        REPLACE LABEL WITH 'Count of Workers';
FOR TABLE 2 VARIABLE TOTAL:
        REPLACE LABEL WITH 'All Workers';
```

Effect In the first table, the label for the variable Employees will be replaced with the label **Count of Workers**. In the second table, the label for the variable TOTAL will be replaced with the label **All Workers**.

Example Normally, if adjacent labels in the heading of a table exactly match, the labels are collapsed into a single label. In the following heading, the variable SEX has a label "Sex of Householder" in adjacent heading boxes and they are collapsed into a single spanning box.

```
HEADING TENURE BY SEX;
```

TABLE 1

	Tenure					
	Owner		Renter		No cash rent	
	Sex of Householder					
	Male	Female	Male	Female	Male	Female

This is usually an acceptable format for the heading but, in some cases, you might want to prevent this collapsing. You can do so by modifying one of the labels so that it does not exactly match the others. A change which will not affect the appearance of the table is to add an optional hyphen to the label. In this example, if we change the middle instance of the SEX label, it will not match the SEX labels on either side of it. The middle instance of the SEX label begins in column 3.

```
FOR COLUMN 3 VARIABLE SEX:
        REPLACE LABEL WITH "Sex of Householder" -;
```

The resulting heading is:

TABLE 1

	Tenure					
	Owner		Renter		No cash rent	
	Sex of Householder		Sex of Householder		Sex of Householder	
	Male	Female	Male	Female	Male	Female

Replacing a Condition Value Label

Format A FOR clause is required to identify the **control variable condition** for which the label will be replaced.

```
(1) FOR CONDITION variable-name(n):
        REPLACE LABEL WITH label;
    FOR CONDITION variable-name(condition name):
        REPLACE LABEL WITH label;
(2) FOR ROW n CONDITION variable-name(n):
        REPLACE LABEL WITH label;
    FOR ROW n CONDITION variable-name(condition name):
        REPLACE LABEL WITH label;
(3) FOR COLUMN n CONDITION variable-name(n):
        REPLACE LABEL WITH label;
    FOR COLUMN n CONDITION variable-name(condition name):
        REPLACE LABEL WITH label;
```

Meaning

(1) For all occurrences of the named control variable, replace the label of the referenced condition with a new one. In the first format, it is the **n**th condition; in the second format, the condition is referenced by condition name.

(2) For the condition value specified, if its label occurs in row **n** of the stub, replace the label.

(3) For the condition value specified, if its label begins in column **n**

of the heading, replace the label.

Note If the control variable is described in the codebook with the clause **DISPLAY AS SORTED**, then the condition numbers will be determined by the sort order of the condition values.

Level

(1) Condition labels can be controlled at the table level.
(2) An individual label in the stub may be changed. The change can be applied to selected wafers. If the table has banks, the corresponding label will be changed in each bank.
(3) An individual occurrence of a condition label can be changed in the heading. If the table has wafers, the corresponding occurrence of the label will be changed in all wafers, even if one or more specific wafers are specified in the FOR clause.

Note If a row is specified and the condition does not appear attached to that row, the format statement will be silently ignored. If a column is specified and the condition does not start in that column, the format statement will be silently ignored. It is sometimes tedious to figure out the exact row or column desired. You may specify a range of rows or columns which include the required row. For example, instead of specifying:

```
FOR COLUMN 7 CONDITION SEX(2): REPLACE LABEL WITH "Men";
```

you may specify:

```
FOR COLUMNS 5 TO 10 CONDITION SEX(2):
        REPLACE LABEL WITH "Men";
```

Default The default condition labels are determined when the variable is described in the codebook or defined in the table request.

Example

```
FOR CONDITION sex(2):  REPLACE LABEL WITH 'Second Sex';
FOR CONDITION sex(3):  REPLACE LABEL WITH 'Unknown';
FOR VARIABLE avg_age:  REPLACE LABEL WITH 'Average Age';
FOR VARIABLE TOTAL:  REPLACE LABEL WITH 'Student Count';
COLUMN WIDTH = 15;
```

Effect The second and third condition labels for the variable **Sex** will be replaced with the labels **Second Sex** and **Unknown**. The variable labels for

avg_age and **TOTAL** will also be replaced. In addition, the COLUMN WIDTH statement is used to increase the column width as shown below.

Before

	Total	AVG AGE
Total	160	15
Sex		
Female	88	15
Male	70	15
No response	2	16

After

	Student Count	Average Age
Student Count	160	15
Sex		
Female	88	15
Second Sex	70	15
Unknown	2	16

Example Consider the following table statement:

```
TABLE A1  'Table A1.  Average pay by industry for each state':
        WAFER STATE, STUB INDUSTRY, HEADING AVG_PAY;
```

This table has a wafer for each state. For most states, mining and construction appear as separate industries. In Hawaii, mining and construction are combined and the data are contained within mining. Thus, for the Hawaii wafer, there will be no row for construction, and we would like to label the condition for "Mining" as "Mining and construction". We would also like to add an explanatory footnote to that one instance of the label and display it at the bottom of that wafer. Assuming that Hawaii has a STATE code of 39, we can do this with the following statements:

```
SET FOOTNOTE COMB  TEXT 'In Hawaii, mining and construction are
combined and the data are contained within mining.';
FOOTNOTES EACH PAGE;
FOR TABLE A1 WAFER 39 ROW 1 CONDITION INDUSTRY(1):
        REPLACE LABEL WITH
        'Mining and Construction' FOOTNOTE COMB;
```

Wafers for three states, including Hawaii, are shown below.

Table A1. Average pay by industry for each state

Alabama

	Average Pay
Mining	$21.50
Construction	25.35
Manufacturing	18.72

Table A1. Average pay by industry for each state — Continued

Alaska

	Average Pay
Mining	$22.43
Construction	28.80
Manufacturing	23.62

Table A1. Average pay by industry for each state — Continued

Hawaii

	Average Pay
Mining and Construction[1]	$24.20
Manufacturing	19.25

[1] In Hawaii, mining and construction are combined and the data are contained within mining.

REPLACE MASK

Format

```
REPLACE MASK WITH mask  [ DISPLAY DECIMAL direction n ];

KEEP DATA FOOTNOTE;  (can be paired with REPLACE MASK)
```

The DISPLAY DECIMAL clause is optional. The **direction** can be LEFT or RIGHT; **n** is a number.

Meaning

Replace the mask with a new one. The new mask can be any valid TPL TABLES mask. If no FOR clause is used, the mask will apply to all cells in all tables. You can restrict the application of the mask either by location OR by variable but not both. In addition to replacing standard data masks, you can replace masks with TEXT masks.

You can add a DISPLAY DECIMAL clause to move the decimal point to the left or right before values are formatted for output. This clause can be added to a mask in a codebook or table request but is most commonly used in a REPLACE MASK statement.

Keeping Data Footnotes

If you are replacing the mask for cells that contain footnotes assigned in conditional Post Compute statements, the new mask may override these footnotes. If you use KEEP DATA FOOTNOTE; with the REPLACE MASK statement, you can change the format of the data values without losing the footnotes.

KEEP DATA FOOTNOTE; must immediately follow the REPLACE MASK statement that would remove the footnotes.

Example

```
POST COMPUTE THOUSANDS '' =
WEIGHT / 1000 MASK 9,999 IF WEIGHT / 1000 >= 100;
WEIGHT / 1000 MASK 9,999 FOOTNOTE LESS_100 IF OTHER;

SET FOOTNOTE LESS_100
        TEXT 'Values less than 100 should not be published.';
```

The statements above would assign a footnote to any cell that had a value of less than 100. The following statements will replace the mask with another that will right-align the data, but also retain the footnotes assigned in the conditional Post Compute.

```
REPLACE MASK WITH RIGHT 9,999;
KEEP DATA FOOTNOTE;
```

Restriction KEEP DATA FOOTNOTE; cannot be used when replacing the mask for a variable. It can only be used when replacing a mask by location, such as for all tables, for particular rows or columns, or for a particular cell.

Replacing Mask by Location

To replace masks for particular table cells, use a FOR clause with the appropriate row, column and/or wafer location.

If there are 9's in the replacement mask, built-in footnotes such as EMPTY or ERROR will not be replaced. If there are no 9's in the replacement mask, these footnotes will be replaced.

Level The location for mask replacement can be specified at the individual cell level.

Example

```
FOR COLUMN 2:  REPLACE MASK WITH $999,999.99;
```

Effect The mask will be replaced to show dollars and cents in the second column of all tables.

Example

```
FOR TABLES 2 AND 3 ROWS 3 TO 6:
REPLACE MASK WITH 999.99 RIGHT;
```

Effect The values in rows 3 through 6 of tables 2 and 3 will be right-adjusted in the columns and displayed to show two decimal places.

Example

```
FOR TABLE B1 COLUMN 2:  REPLACE MASK WITH 99.9 RIGHT;
FOR TABLE B1 COLUMN 1 ROW 4:
        REPLACE MASK WITH 'Secret';
```

Effect The values in column 2 of table B1 will be right-adjusted and displayed showing one decimal place. The value in the cell found at the intersection of row 2 and column 3 will be replaced by the word **Secret**.

Before

B1

	Total	AVG AGE
Total	160	15
Sex		
Female	88	15
Male	70	15
No response	2	16

After

B2

	Total	AVG AGE
Total	160	14.7
Sex		
Female	88	14.6
Male	70	14.7
No response	Secret	15.8

Replacing Mask by Variable

To replace the mask for an observation variable, use a FOR clause with the variable name.

Level Variable masks can be replaced for individual tables.

Example

```
FOR VARIABLE INCOME:  REPLACE MASK WITH $999,999.99;
```

Effect The mask for the observation variable INCOME will be replaced in any tables where INCOME is used.

Example

```
FOR TABLES 2 AND 3, VARIABLE INCOME:
        REPLACE MASK WITH 9,999.99 RIGHT;
```

Effect The INCOME values in tables 2 and 3 will be right-adjusted in the columns and displayed to show two decimal places. If INCOME is used in any other tables, the mask will not be replaced for those tables.

Restrictions Mask replacement by variable will not override:

1. built-in footnotes such as EMPTY or ERROR;
2. conditional masks or footnotes that have been assigned in conditional post compute statements.
3. masks specified by row, column or wafer.

Since masks can only be used with observation variables, masks cannot be replaced for control variables and conditions.

Default The default mask is established when an observation variable is described in the codebook or computed in a table request. If no mask is associated with an observation variable, its final cell values are displayed right-aligned and rounded to the nearest whole integer with no other special symbols except commas.

Treatment of Conflicting Masks

Mask replacement cannot be specified both by variable and by row, column or wafer location:

1. If the two types of specification are used in the same FOR clause, any location specification other than table will be ignored and the mask will be replaced wherever the variable is used.

2. If the same table location would be affected by two different REPLACE MASK statements, where one is specified by variable and the other is specified by row, column and/or wafer, the statement with the variable specification will be ignored. This rule applies regardless of the order of the statements.

Moving the Decimal Point before Display

You can add a DISPLAY DECIMAL clause to to move the decimal point to the left or right before values are formatted for output.

Example

```
FOR TABLE 1:  REPLACE MASK WITH 999
        DISPLAY DECIMAL LEFT 3;
```

Assume that the table contains average income values in dollars. For each value, the decimal point will be shifted left three positions and the value will be displayed as a whole number. The effect is to show the average income values in thousands of dollars. For the value 75724.36, the decimal point will be moved left three positions. The resulting value of 75.72436

will then be rounded to a whole number according to the mask of 999 and will be displayed as 76.

DISPLAY DECIMAL can be added to any mask, in the codebook, table request or format request. The mask can be a regular mask or a TEXT mask. Regardless of where it is entered, it is used only for display purposes and does not affect tabulation or other computations.

Restriction The DISPLAY DECIMAL clause will not be applied in any cell where you have replaced the value using the FORMAT statement REPLACE VALUE.

Replacing Masks with Text

TEXT masks give you much more flexibility than the simple character strings that can be part of a standard data mask. The text can include any of the options associated with other types of labels, such as font specifications, indents and alignments. You can also include the original numeric cell value in the text by using the word VALUE, but note that the values are not aligned as they would be with a standard mask. Rather, they are included in the text at the specified place. If VALUE is used, you can add an optional decimal indicator in parentheses to specify the number of decimal places for display.

Example

```
FOR ROW 10 COLUMN 1:  REPLACE MASK WITH
        TEXT 'No Response ' VALUE(2) '%' LEFT;
```

Effect If the value in row 10, column 1 is 5.148, cell contents will be displayed as:

No response 5.15%

where the value is rounded to two decimal places.

Effect The cell in column 1, row 2 will contain '50 % responding' .

If we reverse the order of the two REPLACE statements, the "new" value of 50 will subsequently be replaced by the TEXT mask that contains the original cell value.

Interaction with REPLACE VALUE

If you are both replacing the mask for a cell with a TEXT mask that contains the word VALUE and replacing the value for the cell in the same format request, you must put the REPLACE MASK statement before the REPLACE VALUE statement. Otherwise, the "new" value will not be used.

Example

```
FOR COLUMN 1 ROW 2:
        REPLACE MASK WITH TEXT  VALUE ' % responding';
        REPLACE VALUE WITH  50;
```

Effect The cell in column 1, row 2 will contain '50 % responding' .

If we reverse the order of the two REPLACE statements, the "new" value of 50 will subsequently be replaced by the TEXT mask that contains the original cell value.

REPLACE MASK COLOR

Format

```
REPLACE MASK COLOR WITH  color-name;
REPLACE MASK COLOR WITH  r g b;
```

where

r, **g** and **b** are numbers between 0 and 100 (inclusive) which specify red, green, and blue components of color;

color-name is the name of a color defined in the color.tpl file.

The word CELL is a synonym for the word MASK.

Meaning This statement lets you replace the color of a mask without disturbing any other specifications in the mask and without re-entering the entire mask.

Level Mask color can be replaced at the level of individual cells or for observation variables.

Example

```
FOR ROW 1:  REPLACE MASK COLOR WITH RED;
FOR ROW 1 COLUMN 1:  REPLACE MASK COLOR WITH BLUE;
FOR VARIABLE INCOME:  REPLACE MASK COLOR WITH GREEN;
```

Effect The mask color for the first row will be red except in column 1 where the mask color will be blue. The rows and/or columns containing INCOME values will have a mask color of green.

Note A REPLACE MASK statement that follows a REPLACE MASK COLOR statement will nullify the MASK COLOR setting if applied to the same cell(s).

Restrictions This statement cannot be used to replace the mask color for TEXT masks, since TEXT masks can contain multiple colors.

REPLACE MASK FONT

Format REPLACE MASK FONT WITH font-name [n];

where

font-name is a font identifier such as H or TB and

n is a number indicating a font size.

Meaning This statement lets you replace the font of a mask without disturbing any other specifications in the mask and without re-entering the entire mask. If you replace the mask font for all cells, you get the effect of changing the DEFAULT FONT for cells without affecting the DEFAULT FONT as applied to any other parts of tables.

Level Fonts can be replaced at the level of individual cells or for observation variables.

Example

```
FOR ROW 1:  REPLACE MASK FONT WITH B 12;
FOR ROW 1 COLUMN 1:  REPLACE MASK FONT WITH HB 10;
FOR VARIABLE INCOME:  REPLACE MASK FONT WITH B;
```

Effect The font for the first row will be Bookman 12 except in column 1 where the font will be Helvetica Bold 10. The rows and/or columns containing INCOME values will have a mask font of Bookman, and the size will be whatever font size is already specified for these cells.

Example

```
DEFAULT FONT = H 12;
REPLACE MASK FONT WITH H 11;
```

Effect The font Helvetica 12 will be used for all parts of the tables except the cells. The MASK FONT Helvetica 11 will be used for all cells.

Note A REPLACE MASK statement that follows a REPLACE MASK FONT statement will nullify the MASK FONT setting if applied to the same cell(s). If a font is specified in the REPLACE MASK statement, that font will be used. If not, the default font will be used.

Restrictions This statement cannot be used to replace the mask font for TEXT masks, since TEXT masks can contain multiple fonts.

REPLACE MASK FOONOTE

Format REPLACE MASK FOOTNOTE WITH footnote-name;

where

footnote-name is the name of a previously created footnote.

Meaning This statement lets you replace the footnote part of a mask without disturbing any other specifications in the mask The table cells, rows, columns or variables need not already have a mask.

Level Mask footnotes can be replaced at any level.

Example

```
FOR COLUMNS 1: REPLACE MASK WITH $9,999.99;
FOR ROWS 3 COLUMNS 1: REPLACE MASK FOOTNOTE WITH
REVISED;
FOR ROWS 4 COLUMNS 1: REPLACE MASK WITH FOOTNOTE
REVISED;
```

Effect The REPLACE MASK WITH FOOTNOTE REVISED replaces the entire mask while REPLACE MASK FOOTNOTE WITH REVISED only replaces (adds) the footnote to the mask.

TABLE 1

	Average Income
Educational Attainment of Householder	
8 years or less	$16,786.96
High school, 1 to 3 years	21,337.58
High school, 4 years	[r] 28,977.97
College, 1 to 3 years	([r])
College, 4 years	$44,858.95
College, 5 or more years	55,087.44

[r] revised

REPLACE MASK MARKER

Format

REPLACE MASK MARKER WITH "*string*";

Meaning

A Mask Marker is a raised string which is placed to the left of the value displayed in a cell. It does not affect the centering of the values in a cell. This statement lets you add or replace the marker string of a mask without disturbing any other specifications in the mask The table cells, rows, columns or variables need not already have a mask.

Level

Mask Markers can be replaced at the level of rows, columns, individual cells or for observation variables.

Notes

Mask markers are similar to footnote symbols except they do not have footnote text associated with them. They are useful when you wish to mark multiple cells with different symbols but want common footnote text to refer to all of them. They are used to display the results of some statistical tests.

Example

```
Keep footnote STAT0;
For row 1 column 2: Replace mask marker with "a";
For row 2 column 2: Replace mask marker with "b";
For row 3 column 2: Replace mask marker with "b";
For row 4 column 2: Replace mask marker with "a";

Set Footnote(STAT0) Symbol default Text = "Different letters indicate
significant differences between mean soores at the " "5% level.";
```

TABLE 1

	Value	Mean
Test 1	107	$^{a}5$
Test 2	154	$^{b}8$
Test 3	171	$^{b}9$
Test 4	110	$^{a}6$
Total	542	7

Different letters indicate significant differences between mean scores at the 5% level.

REPLACE STUB CONTINUATION

Format REPLACE STUB CONTINUATION WITH label;

CONTINUE and CONTINUED are synonyms for CONTINUATION.

Meaning Replace the default continuation indicator for stub labels with the specified label. The continuation indicator can be any valid TPL TABLES label.

For pages after the first page of a table, the continuation indicator will be added to all variable and condition labels that are "in the nest" when a page break occurs. These labels are repeated from the previous page because of the page break. If any of them contain the keyword CONTINUATION, the continuation indicator will be inserted at that point. Otherwise, it will be added at the end of the label.

Level Stub continuation can be controlled at the table level.

Default The default is no continuation indicator.

Example

```
REPLACE STUB CONTINUATION WITH ' (Cont.)';
```

Effect The continuation indicator ' **(Cont.)**' will be added to all stub labels "in the nest" on pages following the first page. If the stub label

```
Industry Code
```

applies to data rows at the end of one page and also to lines at the beginning of the next page, it will be repeated with the continuation indicator as follows:

```
Industry Code (Cont.)
```

Example

```
REPLACE STUB CONTINUATION WITH ' - Continued';
FOR TABLE 2:  REPLACE STUB CONTINUATION WITH '';
```

Effect The stub continuation indicator will be suppressed for table 2 but the indicator ' - Continued' will be added to continued stub labels in all other tables.

Note If a stub continuation contains a footnote reference and you have requested FOOTNOTES EACH PAGE, the footnote from the continuation will be displayed at the bottom of the first page even though the continuation is not added to stub labels until the second and subsequent pages.

REPLACE STUB HEAD

Format REPLACE STUB HEAD WITH label;

Meaning The box that precedes the heading labels in the upper left corner of the table can contain a STUB HEAD label. The label can be any valid TPL TABLES label and will appear in the stub head box for each page of the table. It is always centered within the box.

Level Stub head can be controlled at the wafer level.

Default The stub head box is empty.

Example REPLACE STUB HEAD WITH 'All' / 'Employees';

Effect The stub head box will contain the label as shown below.

Sample Table

All Employees	BOSTON	ST LOUIS
WHITE COLLAR	2	2
BLUE COLLAR	1	3

REPLACE TITLE

Format REPLACE TITLE WITH label;

Meaning Replace the table title established in the TABLE Statement with a new title. The title can be any valid TPL TABLES label.

Level The table title can be controlled at the table level.

Default The default table title is the one specified in the TABLE Statement. If no table title is specified in the TABLE Statement, the table name is used as the default table title.

Example REPLACE TITLE WITH CENTER 'Average Family Income by City.';

Effect The table title is replaced by a centered title

Average Family Income by City

REPLACE TITLE CONTINUATION

Format REPLACE TITLE CONTINUATION WITH label;

CONTINUE and CONTINUED are synonyms for CONTINUATION.

Meaning Replace the default continuation indicator for the table title with a label. It will be added to the title on all pages after the first page of the table. The continuation indicator can be any valid TPL TABLES label.

If the title contains the keyword CONTINUATION, the continuation indicator will be inserted at that point. Otherwise, it will be added at the end of the title.

Level Title continuation can be controlled at the table level.

Default The default continuation indicator is ' **- Continued**'.

Example REPLACE TITLE CONTINUATION WITH ' (Cont.)';

Effect The continuation indicator ' **(Cont.)**' will be added to the table title on pages following the first page. If the table title is:

Average Family Income by City.

then the result on the second and following pages will be:

Average Family Income by City (Cont.)

Example REPLACE TITLE CONTINUATION WITH '';

Effect The title continuation indicator will be suppressed.

Note If a title continuation contains a footnote reference and you have requested FOOTNOTES EACH PAGE, the footnote from the continuation will be displayed at the bottom of the first page even though the continuation is not added to the title until the second and subsequent pages.

REPLACE VALUE

Format REPLACE VALUE WITH n;

where **n** is a number, or

REPLACE VALUE WITH NULL;

Meaning This statement can be used to replace a tabulated cell value with a number or with a value of NULL. If NULL is used, the result will be the same as for an empty table cell. The symbol for the built-in footnote called EMPTY will be placed in the cell unless the EMPTY footnote has been deleted. In that case, a 0 value will be placed in the cell.

The replacement value is displayed according to the mask in effect for the table cell.

Level Values can be replaced for individual table cells.

Default The original tabulated value is displayed in the cell.

Example

```
REPLACE MASK WITH 9,999;
FOR TABLE 2 COLUMN 2, ROW 5:
        REPLACE VALUE WITH 5023.2;
```

Effect All table cells will be displayed using the mask 9,999. In table 2, column 2, row 5, the cell value will be replaced with the value 5023.2 . Since the mask is 9,999 with no decimal places, the replaced cell value will be rounded and displayed as 5023.

Example

```
FOR TABLE 1 COLUMN 3:  REPLACE VALUE WITH NULL;
```

Effect All table cells in table 1, column 3 will be displayed as if they were empty. The symbol for the built-in footnote called EMPTY will be displayed in the cell unless this footnote has been deleted. If it has been deleted, the cell value will be 0.

Interaction with VALUE in TEXT Mask

If you are both replacing the mask for a cell with a TEXT mask that contains the word VALUE and replacing the value for the cell in the same format request, you must put the REPLACE MASK statement before the REPLACE VALUE statement. Otherwise, the "new" value will not be used.

Example

```
FOR COLUMN 1 ROW 2:
        REPLACE MASK WITH TEXT  VALUE ' % responding';
        REPLACE VALUE WITH  50;
```

Effect The cell in column 1, row 2 will contain '50 % responding' .

If we reverse the order of the two REPLACE statements, the "new" value of 50 will subsequently be replaced by the TEXT mask that contains the original cell value.

Restrictions REPLACE VALUE WITH NULL; cannot be used to make an entire row empty such that it will be deleted from the table as an empty row unless the row is already empty. Likewise, if all values in a column are replaced with NULL using this statement, the column cannot be deleted with the statement, DELETE EMPTY COLUMNS; Although the displayed values have been replaced with NULL, the underlying original tabulated values still exist. To delete rows or columns, see the statements DELETE ROW; and DELETE COLUMN;

REPLACE VALUE cannot be used to put a value in an empty cell.

REPLACE WAFER LABEL

Format REPLACE WAFER LABEL WITH label;

Meaning Replace the wafer label with a new one. If this action is applied to a table with more than one wafer, a FOR clause should be used to specify which wafer is to have its label replaced. Otherwise, all wafers in the table will have the same label.

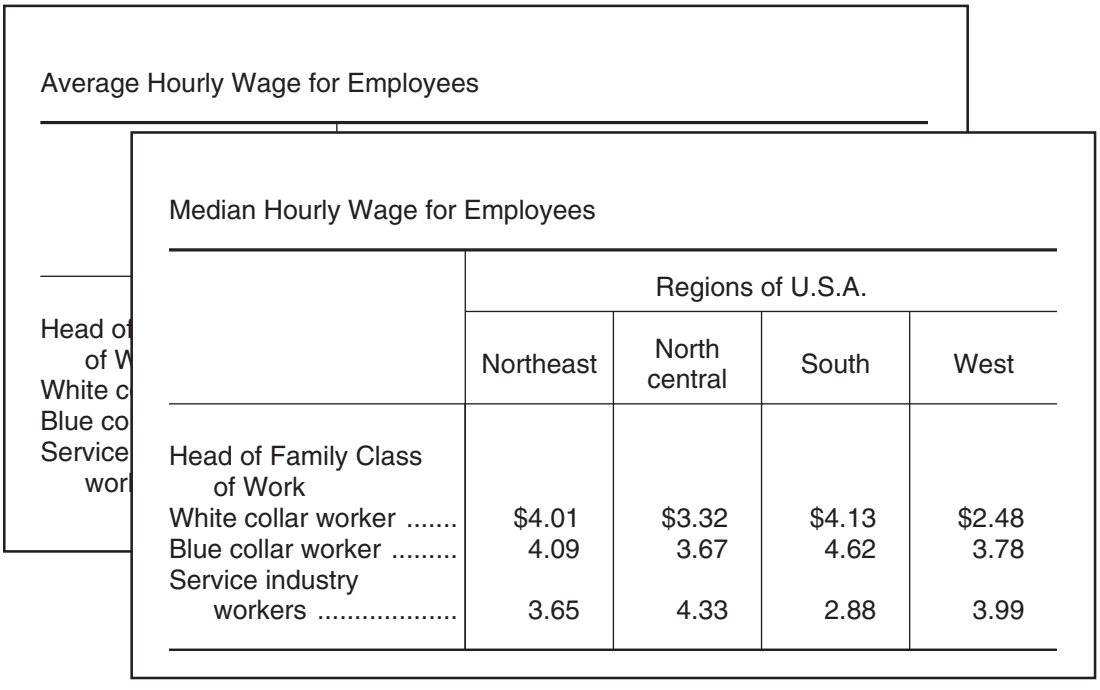
Level Wafer labels can be controlled at the individual wafer level.

Default The default wafer label is determined by the TABLE Statement.

Example

```
FOR TABLE 2 WAFER 1:  REPLACE WAFER LABEL WITH
        'Median Hourly Wage for Employees';
FOR TABLE 2 WAFER 2:  REPLACE WAFER LABEL WITH
        'Average Hourly Wage for Employees';
```

Effect The wafer labels for the first two wafers of table 2 will be replaced with the new labels specified. All other wafer labels will be determined by the TABLE Statement(s).



Average Hourly Wage for Employees

Median Hourly Wage for Employees

	Regions of U.S.A.			
	Northeast	North central	South	West
Head of Family Class of Work				
White collar worker	$4.01	$3.32	$4.13	$2.48
Blue collar worker	4.09	3.67	4.62	3.78
Service industry workers	3.65	4.33	2.88	3.99

REPORT ROWS

Format REPORT ROWS;
DO NOT REPORT ROWS

Meaning For requests which produce a very large number of rows, the **output** file can be quite large because of the list of printed rows. DO NOT REPORT ROWS suppresses this list and makes the output file more easily reviewed.

Level This statement works at the request level.

Default REPORT ROWS;

RETAIN ALL RULES

Format DELETE / RETAIN ALL RULES *rule-properties;*

Meaning If DELETE all horizontal and vertical lines (called rules) will be deleted from the table, including the table heading. Any rules added with the RULE AFTER ROW statement will be deleted. Rules associated with SPANNER labels will be deleted.

Level Deletion of rules can be controlled at the individual table level.

Default RETAIN ALL RULES;

All rules are displayed unless the current divide character is blank. In that case, the vertical lines will be replaced with blanks.

Example FOR TABLE 3: DELETE ALL RULES;

Effect All rules will be deleted from the third table, but they will be retained for other tables.

Characteristics of Households, by Income [Numbers in thousands]

Characteristics	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999	$20,000 to $29,999
All households	46,333	3,105	5,184	4,846	4,776	8,470
Tenure						
Owner	29,791	1,136	2,350	2,494	2,711	5,341
Renter	15,672	1,836	2,667	2,229	1,968	2,986
No cash rent	871	133	167	123	97	143
Region						
Northeast	10,020	579	1,190	879	920	1,736
Midwest	11,543	812	1,343	1,218	1,239	2,080
South	15,469	1,288	1,693	1,778	1,631	2,918
West	9,302	425	959	971	987	1,736

Data values printed in red should be suppressed before publication.

The same table with rules removed.

Characteristics of Households, by Income [Numbers in thousands]

Characteristics	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999	$20,000 to $29,999
All households	46,333	3,105	5,184	4,846	4,776	8,470
Tenure						
Owner	29,791	1,136	2,350	2,494	2,711	5,341
Renter	15,672	1,836	2,667	2,229	1,968	2,986
No cash rent	871	133	167	123	97	143
Region						
Northeast	10,020	579	1,190	879	920	1,736
Midwest	11,543	812	1,343	1,218	1,239	2,080
South	15,469	1,288	1,693	1,778	1,631	2,918
West	9,302	425	959	971	987	1,736

Data values printed in red should be suppressed before publication.

RETAIN BANK DIVIDER

Format DELETE / RETAIN BANK DIVIDER *rule-properties*
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD
STANDARD

(See rule properties for details)

Meaning When a table is row banked with multiple banks side by side on a page, rules are usually added between the banks to divide them. This command controls the properties of that divider.

Level Bank dividers can be controlled at the individual table level.

Default

```
RETAIN BANK DIVIDER DOUBLE WEIGHT .5;
```

Example Below is a banked table followed by the same table with the format statements:

```
KEEP BANK DIVIDER COLOR RED DOT;
RULE MARGIN = 1;  (This just makes output look better)
```

Table Q1. Selected Characteristics of Households, by Total Money Income
[Numbers in thousands]

Characteristics	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	Characteristics	Under $5,000	$5,000 to $9,999	$10,000 to $14,999
All households	3,105	5,184	4,846				
				Type of Household and Sex of			
Tenure				Householder			
Owner	1,136	2,350	2,494	Male householder			
Renter	1,836	2,667	2,229	Married couple ..	446	1,114	1,916
No cash rent	133	167	123	Other family	87	138	151
				Nonfamily			
Region				household ...	578	858	736
Northeast	579	1,190	879	Female			
Midwest	812	1,343	1,218	householder			
South	1,288	1,693	1,778	Married couple ..	39	96	95
West	425	959	971	Other family	814	939	775

Data values printed in red should be suppressed before publication.

Table Q1. Selected Characteristics of Households, by Total Money Income
[Numbers in thousands]

Characteristics	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	Characteristics	Under $5,000	$5,000 to $9,999	$10,000 to $14,999
All households	3,105	5,184	4,846				
				Type of Household and Sex of			
Tenure				Householder			
Owner	1,136	2,350	2,494	Male householder			
Renter	1,836	2,667	2,229	Married couple ..	446	1,114	1,916
No cash rent	133	167	123	Other family	87	138	151
				Nonfamily			
Region				household ...	578	858	736
Northeast	579	1,190	879	Female			
Midwest	812	1,343	1,218	householder			
South	1,288	1,693	1,778	Married couple ..	39	96	95
West	425	959	971	Other family	814	939	775

Data values printed in red should be suppressed before publication.

RETAIN BOTTTOM RULE

Format

DELETE BOTTOM RULES;
RETAIN BOTTOM RULES *rule-properties*;

Meaning

BOTTOM is a synonym for LAST in these statements. See the RETAIN LAST RULE statements for more information.

RETAIN CELLFILE

Important *This statement only works* if it is included in the profile or in a FORMAT request that is used *when the tables are first produced*. If it is added for a rerun, it will have no effect.

Format RETAIN CELLFILE;

Meaning When the statement is used, the cellfile output from the job is retained so that it can be merged with the output from one or more other jobs.

See the section called "Merging Output from Multiple Runs to Create a Single Output" in the chapter called "Data" for complete details on using this statement.

Level This statement works at the request level. It applies to all tables in the request.

Default The cellfile is deleted at the end of the job.

RETAIN COLUMNS

Format DELETE / RETAIN COLUMNS;

Meaning DELETE COLUMNS usually only makes sense if used with a FOR clause that restricts the number of columns deleted. The heading structure is adjusted so that it looks as if the deleted columns were never there.

Deletion of columns will be reflected in the listing of printed columns in the OUTPUT file of the TPL subdirectory created for your job. If you have deleted some columns and you need to do other column-specific changes to a table, you can use this listing to identify the remaining columns.

Level Deletion of columns can be specified for individual columns but is controlled at the table level. When a column is deleted from a table, it is deleted throughout the table.

Default RETAIN COLUMNS;

Example FOR TABLE 3 COLUMN 1: DELETE COLUMN;

Effect The first column of the third table will be deleted. The format of the table heading will be adjusted accordingly.

RETAIN DOWN RULES

Format

RETAIN DOWN RULES *rule-properties*;
DELETE DOWN RULES;
DELETE DOWN RULES START IN HEAD
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD
STANDARD
(See rule properties for details)

Meaning

DELETE DOWN RULES: The column dividers will be deleted from the bottom of the heading to the bottom of the table.

DELETE DOWN RULES START IN HEAD: The column dividers will be deleted from the top of the heading to the bottom of the table.

DELETE DOWN RULES; can be restricted to specific columns. This is useful, for example, if you wish to delete most of the down rules but retain them between sections of a table. When columns are specified in the FOR clause, the down rules that follow the specified columns are the ones that are changed.

The down rule that separates the stub from the body of the table is specified as column 0. Alternately it can be controlled by RETAIN RULE AFTER STUB. This is true even if you have specified STUB RIGHT.

Level

Deletion of down rules can be specified at the individual column level.

Default

RETAIN DOWN RULES;

All down rules are displayed.

Example

FOR TABLE 3: DELETE DOWN RULES;

Effect All column dividers will be deleted from the third table, except in the heading. Down rules will be retained for other tables.

Example

```
FOR TABLE 2 COLUMN 3:  DELETE DOWN RULE;
```

Effect In the second table, the down rule following column 3 will be deleted.

Example

```
STUB RIGHT;
DELETE DOWN RULES;
FOR COLUMNS 2 4 AND 0:  RETAIN DOWN RULES;
```

Effect The stub will be on the right side of the table. All down rules will be deleted except those following columns 2, 4 and 0. In deleting or retaining down rules, column 0 always refers to the down rule between the table body and the stub. Since, in this example, the stub is on the right, the down rule after the last column is retained by the reference to column 0.

RETAIN EMPTY COLUMNS

Format DELETE / RETAIN EMPTY COLUMNS;

Meaning If DELETE selected, delete the table columns that do not have data. Adjust the structure of the table heading so that it looks as if the empty columns were never there.

Note that there is a listing of printed columns in the OUTPUT file in the TPL subdirectory created for each job. If you have deleted empty columns and you need to do other column-specific changes to a table, you can use this listing to identify the remaining columns.

Level Deletion of empty columns can be controlled at the individual table level.

Default RETAIN EMPTY COLUMNS;

Example FOR TABLES 2 AND 3: DELETE EMPTY COLUMNS;

Effect For the second and third tables, columns that do not have data will be deleted. For all other tables in the request, empty columns will not be deleted.

Restrictions In order for a column to be considered "empty", it must not have any data in any part of the table. For example, if a column has no data in wafers 1 to 5 of a table but does have data in one row of wafer 6, the column is not considered to be empty. It will be retained.

RETAIN EMPTY LINES

Important *This statement only works* if it is included in the profile or in a FORMAT request that is used *when the tables are first produced*. If it is added for a rerun, it will have no effect.

Format RETAIN EMPTY LINES;

Meaning When TPL TABLES formats a table, it does not include data rows that do not have data. We call these "empty lines". If you want the empty lines to be included in your tables, you can use the statement **RETAIN EMPTY LINES;**

All empty lines will be retained, even for wafers in which all lines are empty. If you find that you have wafers that have no data and wish to remove them, you can reformat the table output using the DELETE WAFER statement to remove the wafers that have no data.

You may find it useful to use **RETAIN EMPTY LINES;** with the DATA TABLES; statement when you need to have a predictable number of lines in the output.

You can also use **RETAIN EMPTY LINES;** to get a preview of table formats without processing a full data file. You need only one valid record of each record type in your data file to successfully complete a job.

Note If your table stub has nestings of control variables with many values (e.g. state by city), the number of lines generated by **RETAIN EMPTY LINES;** could be huge.

Level This statement works at the request level. It applies to all tables in the request.

Default DELETE EMPTY LINES;

RETAIN END RULE

Format

RETAIN / DELETE END RULE *rule-properties;*
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD
STANDARD
--
ROW SPAN
DATA SPAN
(See rule properties for details)

Meaning

Use the DELETE option of this statement to delete the horizontal rule at the end of the last page of a table. RETAIN LAST RULE controls the rule at the bottom of pages other than the last page of a table.

Level

Deletion of rules can be controlled at the individual table level.

Default

RETAIN END RULE;

The table finishes with a horizontal line all the way across the bottom.

Example

FOR TABLE 3: DELETE END RULE;

Effect

The horizontal line at the end of the third table will be deleted. For all other tables, it will be retained.

Example

See also the SPANNER HEADING statement for an illustrated example.

RETAIN FOOTNOTE

Format DELETE / RETAIN FOOTNOTE (name); or
DELETE / RETAIN FOOTNOTES ALL;

where name is a footnote name. The parentheses around the footnote name are optional.

Meaning The named footnote will be deleted. If there is no FOR clause specifying tables, the footnote will be deleted from all tables. If you wish to delete all footnotes, including the built-in ones, use **DELETE FOOTNOTES ALL;**

If you delete built-in footnotes that apply to table cells, the values in those cells will be zero.

Level Footnotes can be deleted for individual tables.

Default RETAIN FOOTNOTES;

Example FOR TABLE 1: DELETE FOOTNOTE SMALL;

Effect The built-in footnote **SMALL** will be deleted from the first table. Zero values will be printed in any cells that would have been footnoted as containing small values (i.e. values that round to zero).

RETAIN HEADER BOTTOM RULE

Format DELETE HEADER BOTTOM RULE;
RETAIN HEADER BOTTOM RULE *rule-properties;*
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD

STANDARD
(See rule properties for details)

Meaning Use this statement to delete the horizontal line (called a rule) at the bottom of the header

Level The header bottom rule can be deleted for individual tables.

Default RETAIN HEADER BOTTOM RULE;

Example FOR TABLE 1: RETAIN HEADER BOTTOM RULE DASH COLOR = RED;

Table Title

	Hispanic Origin of Householder	
	Hispanic	Not hispanic
Number of Earners		
None	466	5,830
1	854	9,085
2	864	9,577
3	189	2,137
4	86	691
5	12	151
6	5	41
7	3	7
8	1	1

RETAIN HEADER CROSS RULE

Format DELETE HEADER CROSS RULES;
RETAIN HEADER CROSS RULES *rule-properties;*
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD
STANDARD
(See rule properties for details)

Meaning Use this statement to delete the horizontal line (called a rule) in the header except the top and bottom rule of the header

Level All of the header cross rules can be deleted for individual tables. Individual cross rules in the header cannot be modified.

Default RETAIN HEADER CROSS RULE;

Example RETAIN HEADER CROSS RULE COLOR = RED;

TABLE 1

	Race of Householder				Type of Household	
	White		Black		Married couple	Other family
	Hispanic Origin of Householder					
	Hispanic	Not hispanic	Hispanic	Not hispanic		
Average Income						
Regions						
Northeast	21,358	36,708	19,330	24,514	44,222	25,561
Midwest	23,091	31,161	24,466	20,306	37,722	21,376
Southeast	24,598	31,954	41,863	19,019	36,981	20,122
West	24,944	33,865	15,421	22,543	38,856	22,709

RETAIN HEADING

Format DELETE / RETAIN HEADING; or
DELETE / RETAIN HEADER; or
DELETE / RETAIN HEAD;

Meaning DELETE HEADER results in the table heading being removed from the table. The table title is immediately followed by the horizontal line at the top of the data portion of the table.

Level Heading deletion can be specified for individual wafers.

Default RETAIN HEADING;

Example FOR TABLE 1: DELETE HEADING;

Table Title

	Hispanic Origin of Householder	
	Hispanic	Not hispanic
Number of Earners		
None	466	5,830
1	854	9,085
2	864	9,577
3	189	2,137
4	86	691
5	12	151
6	5	41
7	3	7
8	1	1

Table Title

Number of Earners		
None	466	5,830
1	854	9,085
2	864	9,577
3	189	2,137
4	86	691
5	12	151
6	5	41
7	3	7
8	1	1

Effect The table heading will be removed from the first table in the request All of the other tables will be formatted with the heading labels present.

RETAIN HEADNOTE

Format DELETE / RETAIN HEADNOTE;

Meaning The tables are formatted without headnotes. This statement would usually be used with a FOR statement so that it only applies to one or more selected tables.

Level Headnote deletion can be controlled at the table level.

Default RETAIN HEADNOTE;

Example REPLACE HEADNOTE WITH 'Final tabulations for New York.'
FOR TABLES 2 AND 3: DELETE HEADNOTES;

Effect The headnote '**Final tabulations for New York**' will appear above the table heading for all tables except tables 2 and 3.

RETAIN LAST RULES

Format

DELETE LAST RULE;
RETAIN LAST RULE *rule-properties*
DELETE BOTTOM RULE;
RETAIN BOTTOM RULE *rule-properties*
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD
STANDARD

ROW SPAN
DATA SPAN

(See rule properties for details)

Meaning

Use this statement to delete the horizontal line (called a rule) at the bottom of each page of the table *except the last page* of the table. See the statement DELETE END RULE; to delete the rule on the last page of a table.

By default, the bottom rule on the last page of the table extends all the way across the table. On pages before the last, the bottom rule extends only across the data part of the table. Thus, there is automatically a distinction between the last page and the earlier pages.

For text tables, the bottom rule for each page of table is a horizontal line that extends all the way across the bottom of the table. If you specify DELETE LAST RULES; they will be deleted for all but the last page of the table. This provides an additional way of indicating that the table is continued on the next page.

Level

Deletion of rules can be controlled at the individual table level.

Default

RETAIN LAST RULE;

Each page of table finishes with a horizontal line.

Example

FOR TABLE 3: DELETE LAST RULE;

Effect The horizontal line at the bottom of each page of the third table will be deleted for all pages except the last. For all other tables, it will be retained.

RETAIN LEADING ZEROS

Format

DELETE / RETAIN LEADING ZEROS;
DELETE LEADING ZEROS EXCEPT FIRST;

ZEROES can be used in place of ZEROS.

Meaning

When decimal values between 0 and 1 are printed, no leading zeros are printed to the left of the decimal point. For example, the number 0.45 will print as .45 with no zero to the left of the decimal point. Similarly, when decimal values between -1 and 0 are printed, no leading zeros are printed to the left of the decimal point. For example, the number -0.45 will print as -.45 with no zero to the left of the decimal point.

You can use the **EXCEPT FIRST** option, if you want to retain the leading zero for the first value in each column. More precisely, if the first non-empty cell in a column is a decimal value less than 1, the leading zero will be displayed for that value. For example, the value .53 will be displayed as 0.53 if it is the first value in the column. Leading zeros will not be displayed for other values in the column unless you have SPANNER labels or have used FORMAT statements to insert horizontal rules (lines) in the data section of the tables. If spanners or horizontal rules are present, the treatment of leading zeros will restart after each spanner or rule. (**Note** that this treatment is similar to that of the characters $ and % when they are specified in print masks associated with heading variables.)

The counterpart statement RETAIN LEADING ZEROS EXCEPT FIRST; is treated the same as the statement RETAIN LEADING ZEROS; In other words, all leading zeros are retained.

Level

Deletion of leading zeros can be specified for individual tables.

Default

RETAIN LEADING ZEROS;

Example

DELETE LEADING ZEROS EXCEPT FIRST;

Effect

0.16	2.05
2.05	.16
.53	.53
.94	.94

RETAIN ROWS

Format DELETE / RETAIN ROWS;

Meaning DELETE ROWS usually makes sense only when used with a FOR clause that restricts the number of rows deleted. The ROWS are data rows. When a ROW is deleted, the stub labels are adjusted accordingly. Since a data row can have more than one line of stub labels associated with it, deletion of a data row may result in deletion of more than one line from the table.

Note that empty rows (rows that do not have any data) are automatically removed from tables by default. Thus if you are referencing rows by row number in the FOR clause, you may need to check the OUTPUT file listing of **PRINTED ROWS** in order to pick the correct row numbers.

Level Row deletion can be specified for individual rows but is controlled at the wafer level. The rows to be deleted can vary from one wafer to another.

Default Rows that have data are retained. Empty rows are deleted.

Example

```
FOR TABLE 2 ROWS 3 6 9 12:  DELETE ROWS;
```

Effect In the second table, data rows 3, 6, 9, and 12 will be deleted. All other data rows will be retained unless they are empty (have no data).

RETAIN RULE AFTER ROW

Format

RETAIN/DELETE RULE AFTER ROW; or
RETAIN/DELETE RULE AFTER ROW **RULE *rule specs***;
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inch]
BOLD
STANDARD

ROW SPAN
DATA SPAN

SPACE = *n* [*n* unit is standard height of a data row.]
UNDERLINE
(See rule properties for details)

Meaning RETAIN RULE AFTER ROW can be used to insert horizontal rules (lines) in a table. If used with a FOR clause, it will insert rules after selected rows. If used without a FOR clause, it will insert a rule after every row of every table.

Note If the **$** or **%** characters are used in masks that apply to the data columns, then, for any column with these masks, the **$** or **%** character will be repeated in the first non-empty cell following each inserted rule.

Note If any rows of a table do not appear in the table because they are **empty** (do not have any data) or because the rows are **ranked**, you cannot determine row numbers by counting data rows in the printed table. You can findthe row numbers for **PRINTED ROWS** in the OUTPUT file. If you reference an empty row, the RETAIN RULE AFTER ROW statement will have no effect.

Level RETAIN RULE AFTER ROW can be specified for individual rows.

Default Tables are formatted without extra rules. If rules are specified, they are single with a default rule weight of .5, which is the same as the default weight for other non-bold rules in tables.

Example FOR ROW 3: RETAIN RULE AFTER ROW DOUBLE WEIGHT = .75 ROW SPAN;

Effect A double horizontal rule will be inserted after row 3. The rule will be somewhat thicker than the default weight of .5 and will span across the entire width of the table.

Households by sex and education of head of household

Educational Attainment of Householder	Total	Sex of Householder	
		Male	Female
8 years or less	3,986	2,517	1,469
High school, 1 to 3 years	3,699	2,352	1,347
High school, 4 years	10,875	7,512	3,363
College, 1 to 3 years	5,059	3,554	1,505
College, 4 years	3,466	2,629	837
College, 5 or more years	2,915	2,257	658

Example FOR TABLE 1, ROW 6, 13,1,28,36,43: RETAIN RULE AFTER ROW;

Percentile Distribution of Salaries for Senior High Principals, 1988-89

	ALL REPORTING SYSTEMS	ENROLLMENT GROUP			
		25,000 OR MORE	10,000 TO 24,999	2,500 TO 9,999	300 TO 2,499
	AVERAGE SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$65,747	$71,470	$65,479	$67,432	$60,121
80TH PERCENTILE	63,798	65,588	63,782	63,660	55,818
75TH PERCENTILE	62,829	65,579	63,434	62,277	55,325
70TH PERCENTILE	62,016	64,152	62,829	60,763	53,887
60TH PERCENTILE	60,058	61,233	61,973	59,549	51,480
50TH PERCENTILE	58,527	59,829	59,822	58,209	48,484
40TH PERCENTILE	$55,858	$58,985	$57,633	$56,269	$47,500
30TH PERCENTILE	53,887	57,084	55,583	54,416	42,225
25TH PERCENTILE	51,624	55,116	54,426	53,740	42,056
20TH PERCENTILE	49,843	54,621	52,638	51,500	42,000
10TH PERCENTILE	45,317	46,901	48,765	45,317	40,312
NUMBER RESPONDING	149	26	65	36	22
MEAN	57,159	59,839	58,586	57,644	48,983
LOW	$33,966	$43,235	$43,594	$42,115	$33,966
HIGH	74,428	73,526	68,208	74,428	65,747
	LOWEST SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$65,448	$71,470	$64,260	$67,432	$60,121
80TH PERCENTILE	61,464	65,448	62,016	60,505	55,818
75TH PERCENTILE	60,505	61,656	61,464	60,308	55,325
70TH PERCENTILE	59,674	60,452	60,690	59,674	51,624
60TH PERCENTILE	57,805	56,789	59,099	56,269	50,849
50TH PERCENTILE	55,504	55,377	57,924	55,489	48,484
40TH PERCENTILE	$53,870	$53,991	$55,583	$54,216	$47,500
30TH PERCENTILE	50,849	51,610	53,094	51,687	42,225
25TH PERCENTILE	48,826	49,788	50,974	50,572	42,056
20TH PERCENTILE	47,517	48,714	49,650	48,826	42,000
10TH PERCENTILE	43,411	42,778	46,861	44,712	40,312
NUMBER RESPONDING	158	29	71	36	22
MEAN	55,032	56,052	56,224	55,639	48,844
LOW	$33,966	$37,843	$37,685	$42,115	$33,966
HIGH	74,428	71,910	68,178	74,428	65,747
	HIGHEST SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$68,178	$71,910	$67,498	$68,543	$60,121
80TH PERCENTILE	65,627	70,552	65,799	65,205	56,000
75TH PERCENTILE	64,637	67,121	65,485	64,608	55,818
70TH PERCENTILE	63,716	65,595	63,864	63,448	55,325
60TH PERCENTILE	62,016	64,443	62,477	60,763	51,480
50TH PERCENTILE	60,465	63,500	61,678	59,536	48,484
40TH PERCENTILE	$57,711	$61,857	$60,690	$57,186	$47,500
30TH PERCENTILE	55,583	59,712	57,662	54,827	42,225
25TH PERCENTILE	54,128	59,544	55,971	53,809	42,056
20TH PERCENTILE	51,015	57,041	55,470	53,581	42,000
10TH PERCENTILE	47,500	50,880	50,613	45,317	40,312
NUMBER RESPONDING	158	29	71	36	22
MEAN	58,842	62,629	60,387	58,687	49,122
LOW	$33,966	$45,535	$44,466	$42,115	$33,966
HIGH	76,269	76,269	69,621	74,428	65,747

RETAIN RULE AFTER STUB

Format

RETAIN RULE AFTER STUB *rule-options*;
DELETE RULE AFTER STUB RULES;
DELETE RULE AFTER STUB START IN HEAD
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD
STANDARD
(See rule properties for details)

Meaning

DELETE RULE AFTER STUB: The rule between the stub and the first column will be deleted from the bottom of the heading to the bottom of the table. This applies even when STUB RIGHT is specified.

DELETE RULE AFTER STUB START IN HEAD: The rule between the stub and the first column will be deleted from the top of the heading to the bottom of the table. This applies even when STUB RIGHT is specified.

Level

Deletion rule after stub can be specified at the individual table level.

Default

RETAIN RULE AFTER STUB;
The rule between the stub and the body of the table is displayed.

Example

```
STUB RIGHT;
RETAIN RULE AFTER STUB COLOR = RED;
```

Table Q1. Selected Characteristics of Households [In thousands]

Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999	$20,000 to $29,999	$30,000 to $39,999	Characteristics
46,333	3,105	5,184	4,846	4,776	8,470	6,751	..... All households
							Tenure
29,791	1,136	2,350	2,494	2,711	5,341	4,788	..... Owner
15,672	1,836	2,667	2,229	1,968	2,986	1,868	..... Renter
871	133	167	123	97	143	95	..... No cash rent

RETAIN SPANNER RULES

Format DELETE / RETAIN SPANNER RULES;

The word SPAN can be used in place of the word SPANNER.

Meaning If **Delete** is used, the horizontal rules above and below spanner labels in the body of the table. These can be labels in the stub that have the SPANNER attribute, or they can be wafer labels used with one of the WAFER LABEL SPANNER statements in the format request or profile.

Note *If the spanning labels are created with the statements*:

```
WAFER LABELS = DATA (or ROW) SPANNER;
SKIP 0 LINES AFTER WAFERS;
```

you must also add the statement:

```
DELETE LAST RULES;
```

to delete the rule after each spanning wafer label.

Note *You may also need to use the statement:*

```
DELETE DOWN RULES;
```

especially with wafer label spanners. Otherwise, the down rules may intrude on the spanner space

Level DELETE SPANNER RULES can be specified for individual tables.

Default

```
RETAIN SPANNER RULES;
```

Example

```
WAFER LABEL = ROW SPANNER;
SKIP 0 LINES AFTER WAFERS;
DELETE SPANNER RULES;
DELETE LAST RULES;
DELETE DOWN RULES;
```

Does relationship with the mother affect nervousness with members of the opposite sex?

	Nervous with opposite sex?		
	Total	Nervous	Not nervous
Female			
Get along with Mom?			
Yes	60	37	23
No	3	1	2
So-So	23	12	11
N/A	1	1	–
Total	87	51	36
Male			
Get along with Mom?			
Yes	54	22	32
No	2	–	2
So-So	11	4	7
N/A	2	2	–
Total	69	28	41
Both sexes			
Get along with Mom?			
Yes	114	59	55
No	5	1	4
So-So	34	16	18
N/A	3	3	–
Total	156	79	77

– Data not available.

See also the SPANNER HEADING statement for another illustrated example.

RETAIN STUB

Format DELETE / RETAIN STUB;

Meaning If **Delete** is chosen, the table stub, including SPANNER labels, the stub head and the down rule that normally separates the stub labels from the rest of the table, are removed.

Level Stub deletion can be controlled at the table level.

Default RETAIN STUB;

Example DELETE STUB;

Effect The stub labels will be removed from all tables in the request.

RETAIN TABLES FILE

This statement is not needed. You can use export to add a text table to a directory which already has a tables in other formats.

Format RETAIN TABLES FILE;

Meaning Normally, text table output is saved in the output subdirectory in either text table format (a file called tables) or default format (a file called tables.ps). With the statement RETAIN TABLES FILE; you can save both. First, run in one of the two modes and then rerun in the other mode so that both types of table output are saved.

As long as this FORMAT statement is present, both the tables and the tables.ps files will be retained in the output subdirectory once they have been created. If you subsequently remove the statement and do a rerun with the same output subdirectory, the system will revert to the default and keep only one of the two files.

Level This statement applies to all tables in a request.

Default DELETE TABLES FILE;

Example RETAIN TABLES FILE;

Effect If you run the job in PostScript mode, then rerun it in non-PostScript mode), both the **tables.ps** file and the **tables** file will be saved in the output subdirectory.

Note If you have also created encapsulated PostScript files (.eps), they will be retained as long as the .ps file is retained.

RETAIN TABLES

Format DELETE TABLES;

Meaning If not used with a FOR clause, DELETE TABLES will cause all tables to be deleted from the output. If it is used with a FOR clause that specifies which tables should be deleted, only the specified tables will be deleted from the table output.

Level Table deletion can be controlled at the individual table level.

Default RETAIN TABLES;

Example FOR TABLES 2 AND 3: DELETE TABLES;

Effect The second and third tables will be deleted. All others will be retained.

Example
```
DATA TABLES;
FOR TABLES ALL: DELETE TABLES;
FOR TABLE 3: RETAIN TABLE;
```

Effect The third table will be formatted as a data file. All other tables will be deleted.

RETAIN TITLE

Format DELETE TITLE;

Meaning Tables are formatted without title lines at the top of each page.

Level Title deletion can be controlled at the table level.

Default RETAIN TITLE;

Example FOR TABLE 2: DELETE TITLE;

Effect The second table will be formatted without title lines at the top of each page.

RETAIN TOP RULE

Format

DELETE TOP RULE;
RETAIN TOP RULE *rule-properties*
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]
BOLD
STANDARD
(See rule properties for details)

Meaning

Use this statement to delete the horizontal line (called a rule) at the top of each page of the table. By default, the top rule of the table extends all the way across the table.

Level

Deletion of rules can be controlled at the individual table level.

Default

RETAIN TOP RULE;

Each page of table starts with a horizontal line.

Example

FOR TABLE 3: DELETE TOP RULE;

Effect

The horizontal line at the top of each page of the third table will be deleted. For all other tables, it will be retained.

RETAIN WAFER

Format

DELETE WAFER;

Meaning

DELETE WAFER usually only makes sense if used with a FOR clause. The wafers specified in the FOR clause will be deleted.

If the first wafer of a table is deleted, the first printed wafer will not have a title continuation. In other words, it will look like it is the first wafer of the table.

If there are footnotes with automatic numbering in the table, the numbering will apply only to the footnotes in printed wafers. If footnotes are to be printed at the end of the table, they will be printed at the end of the last printed wafer. Footnotes that apply only to deleted wafers will be omitted.

Level Wafer deletion can be specified for individual wafers.

Default RETAIN WAFER;

Example FOR TABLE 2, WAFERS 2 TO 4: DELETE WAFERS;

Effect The second through fourth wafers will be deleted from the second table.

RETAIN WAFER LABEL

Format DELETE WAFER LABEL;

Meaning Tables are formatted without wafer labels at the top of each wafer.

Level Wafer labels can be deleted for individual wafers.

Default RETAIN WAFER LABEL;

Example FOR TABLE 3 WAFER 2: DELETE WAFER LABEL;

Effect The second wafer of the third table will be formatted without a wafer label.

ROTATE

Note ROTATE has no effect for exports other than **eps**.

Format ROTATE;

Meaning The tables are formatted to print sideways on the page. This format is sometimes called "Landscape". Table margins are rotated along with the tables, so that the terms, top, bottom, left and right are relative to the orientation of the table rather than the page.

If you have a table with many columns, you may be able to get all the columns on one page by using the ROTATE statement to turn the table sideways.

Level Rotation can be specified at the individual table level.

Default Tables are printed upright.

Example FOR TABLES 2: ROTATE;

Effect Table 2 will be rotated to print sideways. All other tables will print upright.

ONE

	BOSTON	ST LOUIS
WHITE COLLAR	2	2
BLUE COLLAR	1	3

TWO

	BOSTON	ST LOUIS
WHITE COLLAR	2	2
BLUE COLLAR	1	3

ROUND

Format ROUND = UP; or

ROUND = EVEN;

The = sign is optional and can be omitted.

Note *Do not use a FOR clause with ROUND.* If you do, you will get a syntax error message.

Meaning By default, when final data values need to be rounded, the rounding is done according to the "round even" rule as described in the "Masks" chapter. A value that ends with 5 is rounded up or down depending on the digit to the left of the 5. If the digit to the left of the 5 is even, the value is rounded down. If the digit to the left is odd, the value is rounded up.

To override the default and always round up, use the ROUND = UP; statement.

Level The ROUND specification applies to the entire request.

Default ROUND = EVEN;

Example ROUND = UP;

Effect All data values ending with 5 will be rounded up.

ROW BANKS PER PAGE

Format ROW BANKS PER PAGE = n;

where n is a number. The word IS can be used in place of =. Both are optional and can be left out altogether.

Meaning This statement does not affect to text tables. If you have a narrow table, you can use ***row banking*** to break the table into sections that can fit side by side on a page. For a table that is longer than a page, the table will be broken at the bottom of the page and continue at the top of the same page instead of going to a new page. Optionally, you can specify an earlier break point with a BANK AFTER ROW statement (equivalent to EJECT AFTER ROW). If there is no ROW BANKS PER PAGE statement, the BANK AFTER ROW statement will cause a page break but there will be no banking.

If there is not enough space on the page to contain the specified number of banks or if the number of banks is set to 1, no banking will take place.

Banks are separated by a double line. For other options, see BANK DIVIDER.

On the last page of banking, for example the end of a wafer or end of a table, if there are not enough rows to fill out all the banks, the banks will not be balanced automatically. This is illustrated in the first example below. There are only enough rows for about 1 1/4 banks, so one bank is shorter than the other. See the next sections for ways to balance the number of rows in each bank. It's also possible to have no right bank. In the example below, if we asked for 3 banks per page, we would only get 2, because there are not enough rows to make a third bank. In that case the title and the footnotes will still be formatted at the full width of 3 banks.

Level The number of banks is controlled at the table level.

Default ROW BANKS PER PAGE = 1;

Example ROW BANKS PER PAGE = 2;

Effect The following table, that would normally be broken into two pages, is banked on a single page.

Table Q1. Median family incomes for selected characteristics

	Median Income
U.S. Total	$25,800
Owner	31,400
Renter	17,610
No cash rent	15,500
Male householder	30,925
Female householder	15,000
New England	30,461
Owner	37,480
Renter	21,233
No cash rent	16,100
Male householder	36,490
Female householder	18,574
Mid Atlantic	28,150
Owner	35,230
Renter	19,415
No cash rent	18,600
Male householder	34,300
Female householder	16,673
North Central	25,438
Owner	30,538
Renter	15,693
No cash rent	18,711
Male householder	30,352
Female householder	13,075
South Atlantic	22,974
Owner	27,200
Renter	16,760
No cash rent	13,780
Male householder	28,476
Female householder	14,555
South Central	22,500
Owner	28,256
Renter	14,964
No cash rent	11,829
Male householder	28,060
Female householder	11,787
Mountain	24,452
Owner	29,178
Renter	17,051
No cash rent	11,000
Male householder	27,869
Mountain	
Female householder	$16,112
Pacific	28,134
Owner	36,257
Renter	19,715
No cash rent	24,050
Male householder	32,836
Female householder	18,259

Balancing Banks of Unequal Length

In the next example, the BANK AFTER ROW statement is used to break the first bank before the bottom of the page.

Example

```
ROW BANKS PER PAGE = 2;
FOR ROW 24:  BANK AFTER ROW;
FOR CONDITION REGION (4) :
        REPLACE LABEL WITH "South Atlantic";
```

Effect The table is balanced with the same number of rows in each bank. Note that the REGION labels in the original table all had a slash (/) at the beginning to leave a blank line before each one. The REPLACE LABEL statement is used to remove the extra space from the "South Atlantic" label at the top of the second bank so that the rows will line up between banks.

Table Q1. Median family incomes for selected characteristics

	Median Income		Median Income
U.S. Total	$25,800	South Atlantic	$22,974
Owner	31,400	Owner	27,200
Renter	17,610	Renter	16,760
No cash rent	15,500	No cash rent	13,780
Male householder	30,925	Male householder	28,476
Female householder	15,000	Female householder	14,555
New England	30,461	South Central	22,500
Owner	37,480	Owner	28,256
Renter	21,233	Renter	14,964
No cash rent	16,100	No cash rent	11,829
Male householder	36,490	Male householder	28,060
Female householder	18,574	Female householder	11,787
Mid Atlantic	28,150	Mountain	24,452
Owner	35,230	Owner	29,178
Renter	19,415	Renter	17,051
No cash rent	18,600	No cash rent	11,000
Male householder	34,300	Male householder	27,869
Female householder	16,673	Female householder	16,112
North Central	25,438	Pacific	28,134
Owner	30,538	Owner	36,257
Renter	15,693	Renter	19,715
No cash rent	18,711	No cash rent	24,050
Male householder	30,352	Male householder	32,836
Female householder	13,075	Female householder	18,259

Lining Up Rows with SKIP AFTER ROW

The SKIP AFTER ROW statement can be very useful in lining up rows between row banks. For example, if you have a table with several wafers and all wafers do not have the same number of data rows, you may wish to skip space after some data rows so that the wafers can line up across the banks.

See the SKIP AFTER ROW statement for more information and an example.

Wafer Labels in Banked Wafers

The default location for wafer labels is the "headnote" position. When more than one wafer is banked on a page, only the label for the first wafer can be displayed in the headnote position as shown below. Even though there are two wafers, one in each bank, the "New England" region label is displayed in the headnote position and the wafer label for the second region is missing.

Table W1. Wafer labels in headnote position

New England

	Median Income		Median Income
Total	$30,461	Total	$28,150
Owner	37,480	Owner	35,230
Renter	21,233	Renter	19,415
No cash rent	16,100	No cash rent	18,600
Male householder	36,490	Male householder	34,300
Female householder	18,574	Female householder	16,673

To display all wafer labels, move them to the data spanner or row spanner positions.

Example WAFER LABEL = DATA SPANNER;

Table W2. Wafer labels in DATA SPANNER position

	Median Income		Median Income
	New England		Mid Atlantic
Total	$30,461	Total	$28,150
Owner	37,480	Owner	35,230
Renter	21,233	Renter	19,415
No cash rent	16,100	No cash rent	18,600
Male householder	36,490	Male householder	34,300
Female householder	18,574	Female householder	16,673

Balancing Banks with Joined Wafers

ROW BANKS PER PAGE can be used with joined wafers. Note, however, that EJECT or BANK AFTER ROW will not work for balancing banks if you are trying to break after the last row of a wafer. Instead, use the statement EJECT AFTER WAFER.

Example

```
WAFER LABEL = DATA SPANNER;
EJECT AFTER WAFERS = NO;
COLUMN WIDTH = 1 INCH;
ROW BANKS PER PAGE  =  2;
FOR WAFER 4: EJECT AFTER WAFER;
```

Effect This table is similar to the one in the previous table, but the REGION variable is in the wafer. The wafers are joined together with 2 banks on the page, and EJECT is specified for the 4th wafer so that the banks will be equal in length.

Table Q2. Median family incomes for selected characteristics

	Median Income		Median Income
	U.S. Total		South Atlantic
Total	$25,800	Total	$22,974
Owner	31,400	Owner	27,200
Renter	17,610	Renter	16,760
No cash rent	15,500	No cash rent	13,780
Male householder	30,925	Male householder	28,476
Female householder	15,000	Female householder	14,555
	New England		South Central
Total	$30,461	Total	$22,500
Owner	37,480	Owner	28,256
Renter	21,233	Renter	14,964
No cash rent	16,100	No cash rent	11,829
Male householder	36,490	Male householder	28,060
Female householder	18,574	Female householder	11,787
	Mid Atlantic		Mountain
Total	$28,150	Total	$24,452
Owner	35,230	Owner	29,178
Renter	19,415	Renter	17,051
No cash rent	18,600	No cash rent	11,000
Male householder	34,300	Male householder	27,869
Female householder	16,673	Female householder	16,112
	North Central		Pacific
Total	$25,438	Total	$28,134
Owner	30,538	Owner	36,257
Renter	15,693	Renter	19,715
No cash rent	18,711	No cash rent	24,050
Male householder	30,352	Male householder	32,836
Female householder	13,075	Female householder	18,259

Restrictions ROW BANKS PER PAGE can be used on joined tables, but you cannot balance the banks by breaking between tables. The statements EJECT or BANK AFTER ROW and EJECT AFTER TABLE will not give the desired result.

ROW SPAN

Format ROW SPAN;

Meaning The SPAN specification controls the width of both SPANNER labels and horizontal lines that have been inserted with the RETAIN RULE AFTER ROW statement.

ROW SPAN; causes the SPANNER labels and lines to extend across the entire table, including the stub.

DATA SPAN; is the default SPAN specification. It causes the SPANNER labels and lines to extend across the data columns only.

Level The SPAN specification applies to the entire request. All tables will be formatted with the same SPAN style.

Default DATA SPAN;

Example ROW SPAN;

Effect Any SPANNER labels or horizontal lines created with the RETAIN RULE AFTER ROW statement will span across the entire table, including the stub.

Row Span

	ALL REPORTING SYSTEMS	ENROLLMENT GROUP			
		25,000 OR MORE	10,000 TO 24,999	2,500 TO 9,999	300 TO 2,499
	AVERAGE SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$65,747	$71,470	$65,479	$67,432	$60,121
80TH PERCENTILE	63,798	65,588	63,782	63,660	55,818
75TH PERCENTILE	62,829	65,579	63,434	62,277	55,325
70TH PERCENTILE	$62,016	$64,152	$62,829	$60,763	$53,887
60TH PERCENTILE	60,058	61,233	61,973	59,549	51,480
50TH PERCENTILE	58,527	59,829	59,822	58,209	48,484

Data Span

	ALL REPORTING SYSTEMS	ENROLLMENT GROUP			
		25,000 OR MORE	10,000 TO 24,999	2,500 TO 9,999	300 TO 2,499
	AVERAGE SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$65,747	$71,470	$65,479	$67,432	$60,121
80TH PERCENTILE	63,798	65,588	63,782	63,660	55,818
75TH PERCENTILE	62,829	65,579	63,434	62,277	55,325
70TH PERCENTILE	$62,016	$64,152	$62,829	$60,763	$53,887
60TH PERCENTILE	60,058	61,233	61,973	59,549	51,480
50TH PERCENTILE	58,527	59,829	59,822	58,209	48,484

Restrictions This statement has no effect on the statement WAFER LABEL = ROW SPANNER or DATA SPANNER or HEADNOTE.

RULE

Format

RULE *rule-properties*
where available *rule-properties* are:

SOLID
DOT
DASH
DOUBLE

COLOR = *color*

WEIGHT = *weight* [*weight* in pts - 1/72 inches]

(See rule properties for details)

Meaning

The RULE statement sets default properties for an entire table. If properties are set for specific rules or categories of rules, the more specific property will override the values set by this statement.

Level

RULE statements can be specified for individual tables.

Default

RULE SOLID WEIGHT = .5 COLOR = BLACK;

Example

RULE DOUBLE COLOR = MAGENTA;

Table Q1. Selected Characteristics of Households [In thousands]

Characteristics	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999	$20,000 to $29,999	$30,000 to $39,999
All households	46,333	3,105	5,184	4,846	4,776	8,470	6,751
Tenure							
Owner	29,791	1,136	2,350	2,494	2,711	5,341	4,788
Renter	15,672	1,836	2,667	2,229	1,968	2,986	1,868
No cash rent	871	133	167	123	97	143	95

RULE AFTER ROW

This statement has been replaced by:
RETAIN RULE AFTER ROW rule-properties;

Format

RULE AFTER ROW; or
RULE AFTER ROW **RULE** ***rule specs***;

where *rule specs* can be one of the words DOUBLE or SINGLE, a weight amount WEIGHT = *n*, or both.

Meaning

RULE AFTER ROW can be used to insert horizontal rules (lines) in a table. If used with a FOR clause, it will insert rules after selected rows. If used without a FOR clause, it will insert a rule after every row of every table.

The word RULE can be repeated followed by a WEIGHT and/or DOUBLE specification.

DOUBLE (or SINGLE) can be used to specify whether the rules should be double or single.

WEIGHT= *n* can be used to increase or decrease the thickness of the rules. The value ***n*** is the number of points of thickness where each point is 1/72 inches. Double rules have the same weight as they would have if single.

The rules can be restricted to the data area or they can span the entire width of the table including the stub. You can choose the **style** you want using a FORMAT statement:

DATA SPAN; will cause the rules to extend only across the data columns. DATA SPAN is the system default.

ROW SPAN; will cause the rules to extend across the entire table, including the stub.

Note

The SPAN style will apply to both RULE AFTER ROW and SPANNER labels.

If the **$** or **%** characters are used in masks that apply to the data columns, then, for any column with these masks, the **$** or **%** character will be repeated in the first non-empty cell following each inserted rule.

Note If any rows of a table do not appear in the table because they are **empty** (do not have any data) or because the rows are **ranked**, you cannot determine row numbers by counting data rows in the printed table. You can find the row numbers for **PRINTED ROWS** in the OUTPUT file. If you reference an empty row, the RULE AFTER ROW statement will have no effect.

See also the statement RETAIN RULE AFTER ROW UNDERLINE if you want to underline rows in the data area and do not want any space inserted above the lines.

Level RULE AFTER ROW can be specified for individual rows.

Default Tables are formatted without extra rules. If rules are specified, they are single with a default rule weight of .5, which is the same as the default weight for other non-bold rules in tables.

Example

```
ROW SPAN;
FOR ROW 3: RULE AFTER ROW  RULE DOUBLE WEIGHT = .75;
```

Effect A double horizontal rule will be inserted after row 3. The rule will be somewhat thicker than the default weight of .5 and will span across the entire width of the table.

Households by sex and education of head of household

Educational Attainment of Householder	Total	Sex of Householder	
		Male	Female
8 years or less	3,986	2,517	1,469
High school, 1 to 3 years	3,699	2,352	1,347
High school, 4 years	10,875	7,512	3,363
College, 1 to 3 years	5,059	3,554	1,505
College, 4 years	3,466	2,629	837
College, 5 or more years	2,915	2,257	658

Example

```
FOR TABLE 1, ROWS  6, 13, 1, 28, 36, 43:  RULE AFTER ROW;
```

Effect A horizontal rule will be inserted after rows 6, 13, 21, 28, 36, and 43.

Percentile Distribution of Salaries for Senior High Principals, 1988-89

	ALL REPORTING SYSTEMS	ENROLLMENT GROUP			
		25,000 OR MORE	10,000 TO 24,999	2,500 TO 9,999	300 TO 2,499
	AVERAGE SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$65,747	$71,470	$65,479	$67,432	$60,121
80TH PERCENTILE	63,798	65,588	63,782	63,660	55,818
75TH PERCENTILE	62,829	65,579	63,434	62,277	55,325
70TH PERCENTILE	62,016	64,152	62,829	60,763	53,887
60TH PERCENTILE	60,058	61,233	61,973	59,549	51,480
50TH PERCENTILE	58,527	59,829	59,822	58,209	48,484
40TH PERCENTILE	$55,858	$58,985	$57,633	$56,269	$47,500
30TH PERCENTILE	53,887	57,084	55,583	54,416	42,225
25TH PERCENTILE	51,624	55,116	54,426	53,740	42,056
20TH PERCENTILE	49,843	54,621	52,638	51,500	42,000
10TH PERCENTILE	45,317	46,901	48,765	45,317	40,312
NUMBER RESPONDING	149	26	65	36	22
MEAN	57,159	59,839	58,586	57,644	48,983
LOW	$33,966	$43,235	$43,594	$42,115	$33,966
HIGH	74,428	73,526	68,208	74,428	65,747
	LOWEST SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$65,448	$71,470	$64,260	$67,432	$60,121
80TH PERCENTILE	61,464	65,448	62,016	60,505	55,818
75TH PERCENTILE	60,505	61,656	61,464	60,308	55,325
70TH PERCENTILE	59,674	60,452	60,690	59,674	51,624
60TH PERCENTILE	57,805	56,789	59,099	56,269	50,849
50TH PERCENTILE	55,504	55,377	57,924	55,489	48,484
40TH PERCENTILE	$53,870	$53,991	$55,583	$54,216	$47,500
30TH PERCENTILE	50,849	51,610	53,094	51,687	42,225
25TH PERCENTILE	48,826	49,788	50,974	50,572	42,056
20TH PERCENTILE	47,517	48,714	49,650	48,826	42,000
10TH PERCENTILE	43,411	42,778	46,861	44,712	40,312
NUMBER RESPONDING	158	29	71	36	22
MEAN	55,032	56,052	56,224	55,639	48,844
LOW	$33,966	$37,843	$37,685	$42,115	$33,966
HIGH	74,428	71,910	68,178	74,428	65,747
	HIGHEST SALARY PAID				
SALARY DISTRIBUTION					
90TH PERCENTILE	$68,178	$71,910	$67,498	$68,543	$60,121
80TH PERCENTILE	65,627	70,552	65,799	65,205	56,000
75TH PERCENTILE	64,637	67,121	65,485	64,608	55,818
70TH PERCENTILE	63,716	65,595	63,864	63,448	55,325
60TH PERCENTILE	62,016	64,443	62,477	60,763	51,480
50TH PERCENTILE	60,465	63,500	61,678	59,536	48,484
40TH PERCENTILE	$57,711	$61,857	$60,690	$57,186	$47,500
30TH PERCENTILE	55,583	59,712	57,662	54,827	42,225
25TH PERCENTILE	54,128	59,544	55,971	53,809	42,056
20TH PERCENTILE	51,015	57,041	55,470	53,581	42,000
10TH PERCENTILE	47,500	50,880	50,613	45,317	40,312
NUMBER RESPONDING	158	29	71	36	22
MEAN	58,842	62,629	60,387	58,687	49,122
LOW	$33,966	$45,535	$44,466	$42,115	$33,966
HIGH	76,269	76,269	69,621	74,428	65,747

RULE MARGIN

Format

```
RULE MARGIN = n;
DATA RULE MARGIN = n;
```

where **n** is a decimal number.

Meaning RULE MARGIN statements let you adjust the amount of space between the contents of columns and the rules that divide the columns. The value **n** indicates a number of characters. For example, 1.5 indicates 1.5 characters. For a right-adjusted mask, **n** is the number of characters between the mask and the down rule to its right. For a left-adjusted mask, **n** is the number of characters between the mask and the down rule to its left.

If RULE MARGIN is used, the same margin applies to the columns labels. In other words, both cell values and column labels will be spaced away from the down rules by at least the distance of the rule margin.

If DATA RULE MARGIN is used instead of RULE MARGIN, then the change in spacing occurs only in the part of the table where the data is. No change in margin occurs in the heading.

One character is defined to be the width of the number 5 in the default font and size for the request.

The space is actually the number of characters between the masks or labels and the center of the down rule, but this detail is normally not significant.

Note For text tables, the behavior of this statement as described here will be approximated within the constraints of fixed-width spacing.

Level The statement applies at the table level.

Default

```
RULE MARGIN = .5;
```

Example

```
REPLACE MASK WITH 99,999 RIGHT;
RULE MARGIN = 1.5;
```

Effect Data values and column labels will have a margin of at least 1.5 characters from the down rules. In particular, there will be a 1.5 character space between the right-aligned data and the down rule to the right.

Example Assume that we want to align the data right with a rule margin of 1 in the data part of the table but leave the heading unchanged. In addition, for

cells that have no data, we want to align the dash symbol for the EMPTY footnote with the right edge of any data values. By default, in a cell that contains only a footnote symbol, the symbol is centered. We can align the dash symbol with the right edge of any data values by specifying RIGHT for the EMPTY footnote symbol.

```
REPLACE MASK WITH 999.99 RIGHT;
SET FOOTNOTE EMPTY SYMBOL RIGHT;
DATA RULE MARGIN = 1;
```

All cell contents will be right-aligned but separated from the down rule by 1 character of space.

RULE PROPERTIES

Format

rule-styles category
SOLID
DOT
DASH
DOUBLE
rule-color category
COLOR = *color*
rule-weight category
STANDARD
BOLD
WEIGHT = *weight (in pts where 1 pt = 1/72 inches)*
span-type category
ROW SPAN
DATA SPAN
vertical-space category
UNDERLINE
ROW SPACE = *n* (Value is standard height of a data row).

Meaning

The various rule properties apply to rules in a table. The RETAIN statements tell which properties can be assigned to different rules. For example ***span-type*** is not meaningful for down rules so it is not available for RETAIN DOWN RULES. If an unsupported rule property is used, no warning message is issued. The rule property is just ignored. Only one property from each category may be used. If multiple properties from the same category are specified, the last one entered is applied.

SOLID: This is the default for most table rules.

DOT: The rule is a dotted line. Dotted rules may be used to group parts of a table which are closely related. The spacing of the dots is not controllable by TPL. If you are using TED in the Windows version of TPL, the spacing will be affected by the rule weight. If the rule weight is thick, the rule may display in TED as a solid line. If you export to pdf or html, the dots will appear.

DASH: The rule is a dashed line. Neither the length of the dashes nor the space between is controllable by TPL. If you are using TED in the Windows version of TPL, the spacing will be affected by the rule weight. If the rule weight is thick, the rule may display in TED as a solid line. If you export to pdf or html, the dashes will appear.

DOUBLE: The rule is doubled. The width and height of a table is not affected by a double rule. Instead the space available following the double line is reduced. Double lines are the default for the rule between banks or a row banked table. They can also be used to separate major sections of a table.

COLOR: COLOR specifies the color of a rule. Available colors are listed in the **color.tpl** file where TPL is installed. Additional colors may be added to this file. For complete details, see the section called "General Information about Color" in the **Color** chapter.

STANDARD: STANDARD specifies the standard weight or thickness of a rule. The default for a standard rule is **.5 pts** where a **pt** is 1/72 inches.

BOLD: BOLD specifies a rule which is normally thicker than a standard rule. The default for a bold rule is **1.2 pts**.

WEIGHT = *n*: WEIGHT allows you to specify the thickness for a specific rule. *n* is in **pts**.

ROW SPAN: A row span rule runs the entire length of a table.

DATA SPAN: A data span rule begins after the stub and spans the data area only.

ROW SPACE: Row space controls the amount of space a horizontal rule occupies. It is only used with RETAIN RULE AFTER ROW. ROW SPACE = 1 means that the rule occupies the standard height of a table row. The space is divided evenly above and below the rule. UNDERLINE is a synonym for ROW SPACE = 0; An underlined row adds no extra space. An underlined row takes up the same vertical space as row which is not underlined.

Defaults SOLID for all rules except bank dividers where the default is DOUBLE.

COLOR = BLACK for all rules

STANDARD for all rules except the top and end rules of a table which are BOLD. The default for a standard rule is **.5 pts** where a **pt** is 1/72 inches. The default for a bold rule is **1.2 pts**.

ROW SPACE = 1

Example

For Rows 3: Keep rule after row Color = RED space = 3.0 Row Span;
For Columns 3: Keep down rule Dash;
For Columns 1: Keep down rule Bold;
For Rows 5: Keep rule after row underline Data Span;

Table Q1. Selected Characteristics of Households [In thousands]

Characteristics	Total	Under $5,000	$5,000 to $9,999	$10,000 to $14,999	$15,000 to $19,999
All households	46,333	3,105	5,184	4,846	4,776
Tenure					
Owner	29,791	1,136	2,350	2,494	2,711
Renter	15,672	1,836	2,667	2,229	1,968
No cash rent	871	133	167	123	97
Region					
Northeast	10,020	579	1,190	879	920
Midwest	11,543	812	1,343	1,218	1,239

SCALE

Format SCALE = n%;

where **n** is a number. Decimal numbers such as 99.4 are allowed.

Meaning The SCALE statement can be used when a table is just a bit too big to fit on a page or in the available space in a document. It provides much finer control of sizes than you can get with font size changes. For tables with more than one page, you may find that you can fit more rows and/or another column on each page, depending on the table and the degree of scaling you choose. Page markers, if present, are scaled the same as the tables.

Level The statement applies to the entire table request.

Default SCALE = 100%;

Scaled tables will display and print accurately when distilled to PDF. EPS versions of scaled tables will display and print accurately when inserted in documents using software that accepts EPS.

Note Page markers are not scaled so they will appear like page markers on unscaled tables.

Windows/TED Note

Because of limitations in display of fractional font sizes, scaled tables may not display perfectly in TED. This is true also if you print from TED using the regular Print command in the TED File menu. If you have access to a PostScript printer, you can use the Postscript Print option in TED. With PostScript Print, the scaled tables will print perfectly. See ***PostScript Print*** in TED Help for more information. If you have Adobe Acrobat Distiller, you can configure TED to do an Export to PDF from TED's File menu. The exported PDF will display and print accurately in Adobe Acrobat Reader even when the printer does not support PostScript. See ***Export*** in TED Help for more information.

Example Assume that we are inserting a table into a document that mixes text and tables. Assume also that we want all of the tables to be formatted with the same font sizes, for example titles in font size 10 and the rest in font size 8. One of the tables doesn't quite fit in the allotted space. If we format this tables in a smaller font, the difference will be noticeable, so we would prefer to reduce the table just enough to fit. We can do this by scaling the table to less than 100%, for example

```
SCALE =95%;
```

The table is shown below. On the left is the unscaled version. On the right is the result of scaling to 95%.

Table B2: Single and plural births by birth month and sex of child

	Total	Single Births	Plural Births
January			
Male	5,156	5,002	154
Female	4,790	4,636	154
February			
Male	4,726	4,561	165
Female	4,376	4,235	141
March			
Male	5,037	4,885	152
Female	4,697	4,514	183
April			
Male	4,728	4,557	171
Female	4,624	4,483	141
May			
Male	4,942	4,750	192
Female	4,751	4,588	163
June			
Male	4,885	4,719	166
Female	4,675	4,515	160
July			
Male	5,346	5,157	189
Female	5,046	4,867	179
August			
Male	5,398	5,237	161
Female	5,136	4,967	169
September			
Male	5,054	4,903	151
Female	4,944	4,808	136
October			
Male	5,129	4,970	159
Female	4,901	4,749	152
November			
Male	4,704	4,564	140
Female	4,442	4,291	151
December			
Male	4,914	4,726	188
Female	4,901	4,738	163

Table B2: Single and plural births by birth month and sex of child

	Total	Single Births	Plural Births
January			
Male	5,156	5,002	154
Female	4,790	4,636	154
February			
Male	4,726	4,561	165
Female	4,376	4,235	141
March			
Male	5,037	4,885	152
Female	4,697	4,514	183
April			
Male	4,728	4,557	171
Female	4,624	4,483	141
May			
Male	4,942	4,750	192
Female	4,751	4,588	163
June			
Male	4,885	4,719	166
Female	4,675	4,515	160
July			
Male	5,346	5,157	189
Female	5,046	4,867	179
August			
Male	5,398	5,237	161
Female	5,136	4,967	169
September			
Male	5,054	4,903	151
Female	4,944	4,808	136
October			
Male	5,129	4,970	159
Female	4,901	4,749	152
November			
Male	4,704	4,564	140
Female	4,442	4,291	151
December			
Male	4,914	4,726	188
Female	4,901	4,738	163

Example Assume that we have a table that almost fits on one page but breaks to a new page for the last 5 rows of data. We can scale it down a little and get it to fit on one page with the following SCALE statement.

```
SCALE = 93%;
```

The table is shown below. On the left is the unscaled version. On the right is the scaled version. The 5 rows that were originally on a second page are highlighted with grey shading.

Top 30 counties ranked by number of plural births for total births and each sex[1]

	Total	Single Births	Plural Births
Total			
Lincoln	12,296	11,860	436
Taylor	10,924	10,501	423
Eisenhower	5,831	5,638	193
Comanche	4,517	4,341	176
Cheyenne	5,433	5,270	163
Clinton	3,900	3,762	138
Coral	2,586	2,494	92
Bannock	2,564	2,474	90
Massachusett	3,114	3,025	89
Maricopa	2,039	1,956	83
Harrison	1,809	1,732	77
Cameron	2,079	2,004	75
Blackfoot	2,190	2,117	73
Mohawk	1,987	1,917	70
Chippewa	1,903	1,833	70
Norfolk	2,032	1,970	62
Vermillion	1,659	1,602	57
Adams	1,814	1,759	55
Holland	2,116	2,062	54
McKinley	1,356	1,303	53
Male			
Lincoln	6,404	6,177	227
Taylor	5,623	5,413	210
Eisenhower	2,983	2,888	95
Cheyenne	2,778	2,690	88
Comanche	2,285	2,202	83
Clinton	1,959	1,887	72
Bannock	1,329	1,282	47
Coral	1,309	1,263	46
Massachusett	1,576	1,532	44
Mohawk	1,044	1,005	39
Blackfoot	1,143	1,104	39
Norfolk	1,031	992	39
Maricopa	1,046	1,007	39
Harrison	908	870	38
McKinley	711	674	37
Holland	1,145	1,111	34
Vermillion	861	828	33
Chippewa	965	934	31
Cameron	1,022	992	30
Manhattan	616	588	28
Female			
Taylor	5,301	5,088	213
Lincoln	5,892	5,683	209
Eisenhower	2,848	2,750	98
Comanche	2,232	2,139	93
Cheyenne	2,655	2,580	75
Clinton	1,941	1,875	66
Coral	1,277	1,231	46
Massachusett	1,538	1,493	45
Cameron	1,057	1,012	45
Maricopa	993	949	44
Bannock	1,235	1,192	43
Chippewa	938	899	39
Harrison	901	862	39
Blackfoot	1,047	1,013	34
Mohawk	943	912	31

Top 30 counties ranked by number of plural births for total births and each sex[1]

	Total	Single Births	Plural Births
Total			
Lincoln	12,296	11,860	436
Taylor	10,924	10,501	423
Eisenhower	5,831	5,638	193
Comanche	4,517	4,341	176
Cheyenne	5,433	5,270	163
Clinton	3,900	3,762	138
Coral	2,586	2,494	92
Bannock	2,564	2,474	90
Massachusett	3,114	3,025	89
Maricopa	2,039	1,956	83
Harrison	1,809	1,732	77
Cameron	2,079	2,004	75
Blackfoot	2,190	2,117	73
Mohawk	1,987	1,917	70
Chippewa	1,903	1,833	70
Norfolk	2,032	1,970	62
Vermillion	1,659	1,602	57
Adams	1,814	1,759	55
Holland	2,116	2,062	54
McKinley	1,356	1,303	53
Male			
Lincoln	6,404	6,177	227
Taylor	5,623	5,413	210
Eisenhower	2,983	2,888	95
Cheyenne	2,778	2,690	88
Comanche	2,285	2,202	83
Clinton	1,959	1,887	72
Bannock	1,329	1,282	47
Coral	1,309	1,263	46
Massachusett	1,576	1,532	44
Mohawk	1,044	1,005	39
Blackfoot	1,143	1,104	39
Norfolk	1,031	992	39
Maricopa	1,046	1,007	39
Harrison	908	870	38
McKinley	711	674	37
Holland	1,145	1,111	34
Vermillion	861	828	33
Chippewa	965	934	31
Cameron	1,022	992	30
Manhattan	616	588	28
Female			
Taylor	5,301	5,088	213
Lincoln	5,892	5,683	209
Eisenhower	2,848	2,750	98
Comanche	2,232	2,139	93
Cheyenne	2,655	2,580	75
Clinton	1,941	1,875	66
Coral	1,277	1,231	46
Massachusett	1,538	1,493	45
Cameron	1,057	1,012	45
Maricopa	993	949	44
Bannock	1,235	1,192	43
Chippewa	938	899	39
Harrison	901	862	39
Blackfoot	1,047	1,013	34
Mohawk	943	912	31
Adams	891	861	30
Sky	1,145	1,116	29
Cherokee	764	737	27
Vermillion	798	774	24
Gramblin	521	497	24

See footnotes at end of table.

[1] County residents only

SET FOOTNOTE

Note The SET FOOTNOTE statement can be used in the codebook, table request or profile (profile.tpl) in addition to the format request. For a complete description of footnote options and uses, see the "Footnotes" chapter.

Format

```
SET FOOTNOTE (name) TEXT label SYMBOL string;
```

where **name** is the name of the footnote. The parentheses around the footnote name are optional. The footnote TEXT can be any valid TPL TABLES label except that it cannot itself contain a reference to another footnote. The footnote SYMBOL is a character string enclosed in quotes. The TEXT and SYMBOL specifications are optional and can be in any order but there must be at least one of the two in the statement.

If SET FOOTNOTE is used in a codebook, the '**;**' at the end of the statement is optional. In the table request, format request or profile, the '**;**' is required.

If you wish, you can use **IS** or **=** following the words TEXT and SYMBOL. For example,

```
SET FOOTNOTE CONFIDENTIAL
        SYMBOL IS '(C)' TEXT IS 'Confidential Data';
```

Meaning The SET FOOTNOTE statement determines the footnote symbol and text for the named footnote. The footnote can be referenced in labels or masks in the codebook, table request, format request or profile. In any table where the label or mask is used, the footnote symbol will print where the footnote is referenced and the footnote text will print at the end of the table.

If a footnote is not referenced, it is ignored unless the FORMAT statement KEEP FOOTNOTE is used. Then it will be printed without a footnote symbol unless you have explicitly assigned a symbol. See also the statement SET NOTE as an alternate way of printing footnotes without symbols.

Level A particular footnote applies to all tables in a job. If the same footnote name is used in more than one SET FOOTNOTE statement, the text and/or symbol information from a later statement will replace those from an earlier statement.

The FOR clause has no effect on a SET FOOTNOTE statement. In the following sequence of statements, the footnote text from the second definition of footnote A will be used wherever footnote A is referenced in any table.

```
FOR TABLE 1:  SET FOOTNOTE A  TEXT 'Preliminary data.';
FOR TABLE 2:  SET FOOTNOTE A  TEXT 'Final data.';
```

Default

TPL TABLES has several built-in footnotes. All but two are automatically included in your tables, regardless of whether they are explicitly referenced in labels or masks. You can change the default symbols and text for these footnotes with SET FOOTNOTE statements. You can also delete any or all of them with the DELETE FOOTNOTE statement. The built-in footnotes are:

Footnote name	Symbol	Text
SEE_END	''	'See footnotes at end of table.'
EMPTY	'-'	'Data not available.'
ERROR	'**'	'Computation error.'
NO_FIT	'nf'	'Data does not fit.'
SMALL	'>0'	'Value is too small to display.'
SMALL_NEG	'<0'	'Negative value too near zero to display.'
NORANK	blank	*Reserved for blank rank numbers;* no text
ZERO		*Reserved for exact 0 value*

The ZERO footnote has no built-in symbol or text and is not used unless you assign a text and, optionally, a symbol. When these are assigned, cells which evaluate to precisely 0 will be displayed with this footnote.

For more information about NORANK, see the the RANK DISPLAY section of the "Ranking" chapter.

Example

```
SET FOOTNOTE DATE_NOTE  TEXT 'This table contains '
          'data on all employees hired before August 15.';

FOR TABLE 1: REPLACE TITLE WITH
        'Salary information by sex and job type'
        FOOTNOTE DATE_NOTE;
```

Effect

The title for the first table will be replaced with a title containing a reference to the footnote called DATE_FOOT. When the table is printed, a footnote symbol will be attached to the end of the title and the footnote text will appear at the end of the table. Since no symbol was specified for the footnote, a footnote number will be generated and used as the symbol.

Restrictions

You can have as many as 30,000 distinct footnotes and notes in one job. There is no limit on the number of footnote references.

Only one footnote can be displayed per table cell. If a user-specified footnote conflicts with a built-in footnote in a data cell, the following rules apply: for all of the built-in footnotes, except NO_FIT, if the mask applied to the table cell contains a footnote reference and no 9's, the footnote from the mask will override built-in footnotes. If the mask does contain 9's, a footnote in the mask will not override built-in footnotes.

If a footnote symbol is too wide to fit within the column width, it will be replaced with the NO_FIT built-in footnote. The characters 'nf' will be displayed in place of the footnote symbol.

SET NOTE

Note The SET NOTE statement can be used in the codebook, table request or profile (profile.tpl) in addition to the format request.

Format

```
SET NOTE  (name)  TEXT  label ;
```

where **name** is the name of the note. The parentheses around the note name are optional. The note TEXT can be any valid TPL TABLES label except that it cannot contain a reference to a footnote. The word TEXT is optional. Thus the simple format for the SET NOTE statement is:

```
SET NOTE  name  label ;
```

Meaning The SET NOTE statement lets you specify note text that can be displayed at the end of a table. When notes are displayed, they look like footnotes without symbols. Notes differ from footnotes in that unreferenced footnotes without symbols can only be displayed if you use an associated KEEP FOOTNOTE statement, whereas a note will always be displayed.

Unlike the SET FOOTNOTE statement, SET NOTE has no corresponding DELETE, RETAIN or KEEP statements. However, you can refer to a note with DELETE FOOTNOTE and KEEP FOOTNOTE. RETAIN FOOTNOTE has no effect on notes.

If you have several tables and want to apply a note to only particular tables, you can use a combination of DELETE FOOTNOTE and KEEP FOOTNOTE.

Level A particular note applies to all tables in a job. If the same name is used in more than one SET NOTE statement, the text from a later statement will replace that from an earlier statement.

Example

```
SET NOTE CD 'Confidential Data.';
```

Effect The note called CD will be displayed at the end of each table.

Example

```
SET NOTE PRELIM  'Preliminary data.';
SET NOTE FINAL  'Final data.';
FOR TABLES ALL:  DELETE FOOTNOTE PRELIM;
FOR TABLES ALL:  DELETE FOOTNOTE FINAL;
FOR TABLE 2:  KEEP FOOTNOTE PRELIM;
FOR TABLE 3:  KEEP FOOTNOTE FINAL;
```

The notes PRELIM and FINAL will be deleted from all tables except the second table where the PRELIM note will be kept and the third table where the FINAL note will be kept.

Restrictions You can have as many as 30,000 distinct notes and footnotes in one job.

SHADE

Shading is an excellent way to highlight selected parts of a table or to add color in a pleasing way. Even if you have only a monochrome printer, you can get some excellent effects by using GREY shading to emphasize selected parts of a table.

Format

```
SHADE table-element [COLOR] r g b;
SHADE table-element [COLOR] color-name;
SHADE table-element GREY n;
```

where

r, **g** and **b** are numbers between 0 and 100 (inclusive) which specify red, green, and blue components of color.

color-name is the name of a color defined in the color.tpl file.

GREY (or GRAY) is the color and n is a number between 0 and 100 (inclusive) that selects the degree of grey shading.

table-element can be any of the following:

TABLE	STUB
TITLE	DATA
WAFER LABEL	ROW
HEADNOTE	COLUMN
TOP	CELL
HEADING	FOOTNOTES
STUB HEAD	LABEL

See also the COLOR statements for color printing of labels, values.

Meaning

You can use SHADE statements to specify background shading for an entire table or for different sections of a table. You can choose the color for each element shaded. If you are using a monochrome printer, color will print in various shades of grey depending on the color.

GREY is a special built-in color that works equally well for shading tables on either a color or a monochrome printer. In most cases, GREY shading is the best choice for monochrome printing, because you can control the degree of darkness directly by specifying a number from 0 to 100 where 0 is no shading and 100 is black shading.

Colors can be referenced by name where the colors have been defined in a file called color.tpl. The color.tpl file is installed as part of the TPL TABLES system with several colors already defined. You can customize this file to add the colors of your choice. For complete details, see the section called "General Information about Color" in the "Color" chapter.

Example

```
FOR TABLE 2:  SHADE HEADING GREEN;
FOR TABLE 2, COLUMN 1:  SHADE CELLS RED;
```

Effect

All tables will be printed without shading except in the second table. The heading in the second table will be shaded in the color GREEN. The first column will be shaded with the color RED.

Example

```
SHADE TABLE PUMPKIN;
REPLACE TITLE WITH  COLOR RED 'Tableau 15B '
        'Statistiques sommaires selon le genre d'établissement.'
```

Effect

In this example, we combine background shading with color text in the table title. The entire table is shaded in pumpkin color (99 60 20) and the table title text is red. The other text, numbers and lines are printed in the default color black.

Tableau 15B Statistiques sommaires selon le genre d'établissement.

Dépenses de fonctionnement

	Salaires	Autres	Total
Musées			
Musées	42 357,34	282,36	42 639,70
Parcs naturels	71 628,05	76,25	71 704,30
Lieux d'intérêt historique	61 359,82	62,56	61 422,38
Archives	81 305,43	48,55	81 353,98
Centres d'expositions	511,48	10,53	522,01
Observatoires et planétariums	434,79	1,28	436,07
Zoos et aquariums	67,69	51,66	119,35
Jardins botaniques	74,15	6,98	81,13
Total	257 738,75	540,17	258 278,92

Example FOR ROWS 1 TO 21 BY 2: SHADE ROW GREY 10;

Effect Alternate data rows are shaded light grey.

Number of households by type of household and state.

	Type of Household		
	Married couple	Other family	Nonfamily household
Connecticut	57,980	18,666	37,626
Maine	22,535	1,937	11,952
Massachusetts	71,997	14,208	29,184
New Hampshire	11,541	744	3,451
Rhode Island	12,559	719	6,306
Vermont	1,456	360	–
New Jersey	134,095	37,399	70,480
New York	197,547	60,114	110,844
Pennsylvania	218,880	28,678	103,033
Illinois	25,297	1,869	13,212
Indiana	37,912	4,781	7,157
Michigan	2,964	1,437	–
Ohio	–	1,526	3,066
Wisconsin	22,528	–	12,172
Iowa	38,220	1,279	7,824
Kansas	11,169	1,250	1,276
Minnesota	10,192	2,598	7,973
Missouri	20,479	2,975	6,201
Nebraska	13,634	2,514	2,652
North Dakota	4,703	289	295
South Dakota	1,478	251	1,060

– Data not available.

Default The default is no shading.

Restrictions Shading can only be requested with the SHADE statements described here. If you insert a shading in an individual label or mask, you will get an error message.

Placing Tables in Other Documents

If you are placing Encapsulated PostScript(EPS) tables into documents created with other desktop publishing software, you will get different results, depending on whether the tables are unshaded or shaded.

Unshaded

In TPL TABLES, a table with no shading is transparent except for the table text and lines. If you place this table on top on something in a document produced with other desktop publishing software, anything below the table will show through when the page is printed. To place this unshaded table on a green background, you would create the background square and place the table on top. The green background would show through in the clear spaces.

Shaded

If you use shading in TPL TABLES, all tables will be opaque from the start of the first shading to the end of the last table. If you have multiple banks, wafers or tables on a page, even the “white space” between will be opaque. Thus, if you want to place a table on top of something and have nothing show through, you can use shading to make the table opaque. If you do not want color in the table, you can shade it white. For example:

```
SHADE TABLE 100 100 100;
```

If you have the color WHITE defined in your color.tpl file, you can use the statement:

```
SHADE TABLE WHITE;
```

How Shading Conflicts are Resolved

You can think of shading as being applied in layers with each layer being opaque. For example, if a RED layer is applied on top of a BLUE layer, only the RED shading will show. This "layering" explains how conflicts are resolved.

An example of a shading conflict is a shaded column that intersects with a shaded row. Whenever two shadings apply to the same place in a table, only one of the shadings will be used. The "winning" shading is the top layer.

The top layer is determined according to a shading order. If conflicting SHADE statements are at different levels, the shading order determines the shading at the overlapping points, regardless of the sequence of SHADE statements. If SHADE statements conflict on the same level in the shading order, they are applied according to their order in the format request.

The **shading order** follows. Each level overlays shading at any level above it. In general, the order allows finer levels of shading to overlay broader levels of shading.

```
table
heading, top, footnotes, stub
label, wafer-label, column, headnote, title, stubhead
row, data
cell
```

Example

```
SHADE TABLE GREEN;
FOR COLUMN 2:  SHADE COLUMN RED;
```

Shading of the entire table conflicts with shading for one of the columns. Since "table" is above "cell" in the shading order, the table shading will be applied first to shade the entire table GREEN. The cell shading will then be applied on top of the table shading and the color RED specified for column 2 will cover the GREEN shading of the table.

Example

```
FOR ROW 2:  SHADE ROW PURPLE;
FOR CONDITION SEX(1):  SHADE LABEL BLUE;
```

"Label" is above "row" in the shading order, so the label will be shaded first and the row shading will overlay it. If the label is a multi-line label, only the last line of the label will be shaded by the SHADE ROW statement. This means that you will get a label with two color shadings, one on the line with the data row and another on any label lines above. You could get a more pleasing result by using SHADE DATA instead of SHADE ROW. Since SHADE DATA shades the data row but does not extend into the stub, the row and label shadings would not overlap.

Using WHITE with Shading Conflicts

Sometimes, white shading can be used to limit the extent of shading in other colors. In the following example, the entire table is shaded RED, but, since "top" and "footnote" shading follow "table" in the shading order, the RED shading will be overlaid with WHITE for these parts of the table.

Example If you have defined the color WHITE in color.tpl, you can use it in your shade statements as follows:

```
SHADE TABLE RED;
SHADE TOP WHITE;
SHADE FOOTNOTES WHITE;
```

If you do not have the color WHITE in color.tpl, you can get the same result using:

```
SHADE TABLE RED;
SHADE TOP 100 100 100;
SHADE FOOTNOTES 100 100 100;
```

In either case, the table will be shaded in the color RED except for the area from the title down through the heading and the footnote area at the bottom.

SHADE OPTIONS

Following are descriptions of all SHADE options, illustrated to show the effect of shading different table elements. The basic illustrations, done with light grey shading, will print the same way on either a monochrome or color printer.

Shade Cell

Cell shading is generally useful only if there is a FOR clause to restrict the shading to particular cells. If there is no FOR clause, SHADE CELL will cause all data cells to be shaded. The result will be the same as with the SHADE DATA statement.

Level You can specify cell shading at the level of individual cells by a combination of table, wafer, row and column numbers.

Example SHADE CELLS GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Example FOR COLUMNS 1 TO 6 BY 2: SHADE CELLS GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Example

```
FOR ROW 1 COLUMN 1:
        COLUMN WIDTH = 12;
        REPLACE MASK WITH
                TEXT 'Total households surveyed:'/ VALUE;
        SHADE CELL GREY 10;
```

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	Total households surveyed: 1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

In this example, the shaded cell contains text that is broken into multiple lines. All lines of the cell are shaded.

Shade Column

Column shading extends from the bottom boundary of the heading to the bottom boundary of the data section of the table. It can be used to shade the entire section bounded by the heading and stub, or it can be used to shade selected columns.

Level Column shading can be specified at the column level.

Example

```
FOR COLUMN 1:  SHADE COLUMN GREY 10;
```

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Data

Rows that have data are shaded from the inside of the stub across to the opposite side of the table. If the data row has one or more TEXT masks long enough to wrap to multiple lines, the shading will extend to cover these lines.

Level Shading of data rows can be specified at the row level.

Example FOR ROWS 1 AND 3: SHADE DATA GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Note on Shade Data and Shade Row

As with rules and spanner labels that can span the entire row or just the data section of the row, SHADE DATA and SHADE ROW let you make a similar choice. In other words, they are the same except that SHADE ROW extends across the entire table and SHADE DATA spans across only the data section of the table, leaving the stub untouched.

Shade Footnotes

Footnote shading starts just below the boundary of the data section of the table and extends down to the bottom of the last footnote present. The shading spans the full width of the table. If a page has banks of unequal width footnotes for all banks on the page are shaded to the width of widest bank.

Level Footnote shading can be specified at the table level.

Example

```
SHADE FOOTNOTES GREY 10;
```

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Heading

The entire table heading is shaded. The shading extends to both sides of the table and includes the stub head.

Level Shading of the heading can be specified at the table level.

Example

```
SHADE HEADING GREY 10;
```

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Headnote

If a headnote has been entered with a REPLACE HEADNOTE statement, the headnote label rows will be shaded across the full width of the table.

Level Headnote shading can be specified at the table level.

Example SHADE HEADNOTE GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Label

Selected variable or condition labels can be shaded if the variables are in the heading or stub, including stub labels with the attribute SPANNER. For a multi-line label, the shading is applied to all lines of the label, including blank lines entered in the label with the **/** character.

SHADE LABEL cannot be used to shade wafer labels. Use the SHADE WAFER LABEL statement for these.

SHADE LABEL only works if it is preceded by a FOR VARIABLE or FOR CONDITION clause that indicates which labels should be shaded.

Format

```
FOR VARIABLE var-name: SHADE LABEL color;
FOR CONDITIONS con-var-name(c1, c2, ....):  SHADE LABEL color;
```

var-name can reference any type of variable used in the table.

con-var-name can be any control variable from the codebook or a DEFINE statement. **Note** that if a control variable is described in the codebook with the attribute DISPLAY AS SORTED, then the condition numbers will be determined by the sort order of the condition values.

Level

Label shading can be controlled at the table level. If the label referenced in the FOR clause occurs more than once in a table, all occurences will be shaded, but if it is used in more than one table, you can shade it in some tables and leave it unshaded in others.

Example

```
FOR VARIABLE AGE:  SHADE LABEL GREY 10;
FOR CONDITION SEX(1):
        REPLACE LABEL WITH 'Total, '/'All householders';
        SHADE LABEL GREY 10;
```

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
Total, All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Row

Rows that have data are shaded across the entire width of the table including the stub. If the data row has one or more TEXT masks long enough to wrap to multiple lines, the shading will extend to cover these lines. **Note** however that if the stub label for the row is more than one line long, the shading will only apply tothe last line of the label. To shade the entire stub label, you must use SHADE LABEL.

Level Row shading can be specified at the row level.

Example FOR ROWS 1 AND 3: SHADE ROWS GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Note on Shade Row and Shade Data

As with rules and spanner labels that can span the entire row or just the data section of the row, SHADE DATA and SHADE ROW let you make a similar choice. In other words, they are the same except that SHADE ROW extends across the entire table and SHADE DATA spans across only the data section of the table, leaving the stub untouched.

Shade Stub

The entire stub area is shaded from just below the stub head down to the bottom of the table. If row or label shading conflicts with stub shading, the row or label shading will overlay the stub shading for those rows.

Level Stub shading can be specified at the wafer level.

Example SHADE STUB GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Stub Head

The entire stub head area in the corner above the stub is shaded.

Level Stub head shading can be specified at the wafer level.

Example SHADE STUB HEAD GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Table

The entire table is shaded including the title and any footnotes displayed at the bottom of the table. Page markers in the top or bottom page margin are not shaded. If there are multiple tables, banks, or wafers on the same page, the space between them is not shaded. If a page has banks of unequal width, all banks on the page are shaded to the width of widest bank.

Level Table shading can be specified at the table level.

Example SHADE TABLE GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Title

The title lines are shaded across the full width of the table.

Level Title shading can be specified at the table level.

Example SHADE TITLE GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Top

The space above the heading is shaded from the top of the table title to the top boundary of the heading.

Level Shading of the table "top" can be specified at the table level.

Example SHADE TOP GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

Shade Wafer Label

If the wafer label is at the top of the table, the wafer label lines are shaded across the full width of the table. If the wafer labels have been converted to spanners with one of the statements WAFER LABEL = ROW SPAN or WAFER LABEL = DATA SPAN, the shading will be within the boundaries of the spanner sections of the table.

Level Wafer label shading can be specified at the wafer level.

Example

SHADE WAFER LABEL GREY 10;

Title: Number of Households Surveyed this Year by Sex of Householder and Age of Householder[1]

Wafer label: Numbers in thousands

Headnote: Weighted Data

Stub head: Characteristics	Age of Householder					
	Total, All ages	17 to 29	30 to 39	40 to 54	55 to 64	65 and over
Sex of Householder						
All householders	1,537	57	311	304	267	348
Male householder	1,095	34	237	250	207	173
Female householder	441	23	74	55	60	175

[1] Footnote referenced in title.
Data Source: Department of Commerce, Bureau of the Census

SKIP AFTER BANKS

Format SKIP n LINES AFTER BANK;
where **n** is a number.

Meaning If a table is too wide to fit on a page, it is automatically broken into sections called banks. Banking can also be requested explicitly with the **BANK AFTER COLUMN;** statement. By default, each bank begins on a new page. SKIP n LINES AFTER BANKS can be used to request a different spacing between banks so that more than one bank can be printed on a page.

If SKIP n LINES AFTER BANKS is specified by itself, TPL TABLES assumes that there should be two banks per page. The related statement BANKS PER PAGE can be used to request more than two banks per page.

With the statement **SKIP 0 LINES AFTER BANKS;** the banks are joined with no space between banks, and the stub head is deleted for banks after the first.

Table alignment is determined for each individual page of a banked table and depends on the width of the widest bank on the page. Narrower banks are aligned with the stub of the widest bank.

Table titles, wafer labels, headnotes and footnotes will appear only once on a page regardless of the number of banks. They will be aligned with the widest bank on the page.

Note For multiple banks per page, we recommend that you use equal width banks whenever possible. In general, unequal width banks on the same page do not look good. This is especially true if you have specified **SKIP 0 LINES AFTER BANKS;**.

Note All banks on a page will take up the same amount of vertical space. If you have different width banks on the same page, you may find that the page ends sooner than you expect, especially if you have a very narrow bank that requires the table title to be broken into several lines. In addition, if one bank has a very long heading label that must be broken into several lines, the space requirement for that label will apply to all of the banks. Each bank may then take more vertical space than you expect.

Level SKIP n LINES AFTER BANK can be specified for individual wafers.

Default Each bank begins on a new page. If the BANKS PER PAGE statement is used without SKIP AFTER BANKS, the default is **SKIP 1 LINE AFTER BANKS;**

Example

```
FOR COLUMN 2: BANK AFTER COLUMN;
SKIP 0 LINES AFTER BANKS;
```

Effect Each page of the table will contain 2 banks with no space between the banks.

U.S. Waterborne Exports

By Country

Country of Ultimate Destination	Short Tons of 2000 Lbs.	
	Liner	Tanker
Total	248,745	4,146,065
MEXICO, CENTRAL AMER. & CARIB.		
Total	3,410	527,313
0 BULK	1,932	526,500
1 GENERAL	1,478	813
SOUTH AMERICA		
Total	104,772	1,348,266
0 BULK	59,425	1,319,128
1 GENERAL	45,347	29,138

	Short Tons of 2000 Lbs.	
	Tramp	Total
Total	15,334,746	19,729,556
MEXICO, CENTRAL AMER. & CARIB.		
Total	584,649	1,115,372
0 BULK	576,483	1,104,915
1 GENERAL	8,166	10,457
SOUTH AMERICA		
Total	2,664,773	4,117,811
0 BULK	2,587,432	3,965,985
1 GENERAL	77,341	151,826

Example

```
BANKS PER PAGE = 3;
SKIP 2 LINES AFTER BANK;
```

Effect Each page of the table will contain 3 banks with 2 blank lines between the banks. If the number of table banks is not a multiple of 3, then the last page will contain fewer than 3 banks.

Restrictions There must be enough vertical space on the page for each bank to contain at least one line of data.

SKIP AFTER ROW

Format SKIP amount AFTER ROW;

where **amount** can be an integer, an integer followed by the word **LINES**, or a measurement. Measurements can be expressed as a number of **inches**, **cm** or **points**. If only an integer is specified, not followed by the word LINES or a unit of measure, LINES will be assumed.

Meaning SKIP AFTER ROW can be used to insert extra space after data rows. If used with a FOR clause, it will insert space after selected rows. If used without a FOR clause, it will insert the extra space after every row of every table.

If you specify both SKIP AFTER ROW and RULE AFTER ROW for the same row, the extra space will be inserted below the rule. Likewise, if you specify both SKIP AFTER ROW and UNDERLINE ROW for the same row, the extra space will be inserted below the underline.

Note SKIP AFTER ROW applies only to ***data*** rows. If you want to skip space between lines of a multi-line label or between a variable label and a condition label, use slash characters in the labels as described in the "Labels" chapter.

Note If any rows of a table do not appear in the table because they are **empty** (do not have any data) or because the rows are **ranked**, you cannot determine row numbers by counting data rows in the printed table. You can find the row numbers for **PRINTED ROWS** in the OUTPUT file. If you reference an empty row, the SKIP AFTER ROW statement will have no effect.

Level SKIP AFTER ROW can be specified for individual rows.

Default Standard single spacing is used for all rows.

Example SKIP 1 LINE AFTER ROWS;

Effect One blank line of space will be inserted after all data rows in all tables.

Example FOR TABLE 2, ROWS 6, 13, 21: SKIP 1.2 CM AFTER ROWS;

Effect In the second table, 1.2 centimeters of space will be inserted after rows 6, 13 and 21.

Example SKIP AFTER ROW can be useful for lining up rows between row banks. In the next table, there are 3 banks on the page, where types of fires are shown by type of location. The first bank contains one less row than the other two, because there are no residential fires in vehicles. To line up the rows across the table so that fires of the same type are side by side across the banks, we can add a blank row to the first bank.

```
ROW BANKS PER PAGE = 3;
FOR ROWS 6 AND 12: BANK AFTER ROW;
FOR ROW 2: SKIP 1 LINE AFTER ROW;
```

Fires by Type

	Total		Total		Total
Residential		Special Property		Other Business Property	
Total Fires	818	Total Fires	1,450	Total Fires	397
Structure Fires	608	Structure Fires	20	Structure Fires	190
		Vehicle Fires	875	Vehicle Fires	11
Trees, Brush Fires	47	Trees, Brush Fires	255	Trees, Brush Fires	52
Refuse Fires	95	Refuse Fires	256	Refuse Fires	105
Other Fires	68	Other Fires	44	Other Fires	39

Restrictions If the SKIP AFTER ROW is specified for the last row on a page, the SKIP statement is ignored.

SKIP AFTER TABLE

Format SKIP n LINES AFTER TABLES;

where **n** is a number.

Meaning The **SKIP AFTER TABLES** statement can be used to control the grouping of tables on a page. It tells TPL TABLES to skip the specified number of lines after a table and continue printing without going to a new page.

When some tables are to be grouped together and others are to start on a new page, the statement **EJECT AFTER TABLES;** can be used to cause particular tables to start on a new page.

If the value of **n** is greater than 0, the tables are grouped but still look like separate tables.

If the value of **n** is 0, there will be no space between the tables. Footnotes from the joined tables are merged and are put at the end of the last table. If the same footnote appears in more than one of the tables, its text will be printed only once, and any automatic footnote numbering is adjusted so that the numbering does not restart at the beginning of each table.

You may wish to join the tables so that they look like a single table. This technique is especially useful if you cannot request all of the rows you want in a single TABLE statement because you need to put different observation variables in the heading for different rows. To make the tables look like one, you should delete the table title (and the headnote and wafer labels if present) for all tables except the first. If you want the same heading labels for the entire group of tables, you can also delete the heading for all tables except the first. However, if a table with a deleted heading extends to a second page, deleting the heading looks strange.

If you want a line to show where the tables are joined, use the statement RULE AFTER ROW; for the last printed line of each table that has another table joined to it.

SKIP 0 LINES AFTER TABLES; works best for short tables that can be grouped to fit on one page. If the tables have multiple banks or wafers, the result will probably not be what you want. Likewise, you will get the best results with tables that have the same number of columns, widths and alignment. Otherwise, the columns will not line up. When the columns do

not line up, the tables usually look bad. The results are especially strange-looking if you have deleted headings for tables other than the first.

Level Both SKIP AFTER TABLES and EJECT AFTER TABLES can be specified at the individual table level.

Default

```
EJECT AFTER TABLE;
```

Example

```
FOR TABLE 1:  SKIP 2 LINES AFTER TABLE;
```

Effect At the end of the first table, two lines will be skipped and the second table will begin without going to a new page. If there are other tables after the second, they will each begin on a new page.

Example

```
SKIP 3 LINES AFTER TABLES;
FOR TABLE 2:  EJECT AFTER TABLE;
```

Effect At the end of each table except the second, three lines will be skipped and the next table will begin. The third table will begin on a new page.

Example Assume that there are ten tables with one row each. We can join them as follows:

```
SKIP 0 LINES AFTER TABLES;
DELETE TITLE;
DELETE HEADING;
FOR TABLE 1:  RETAIN TITLE;
              RETAIN HEADING;
```

Effect The ten tables are joined so that they look like one table with ten rows. The table titles and headings are deleted for all tables except the first.

Example The following two tables can be joined on one page so that they look like a single table.

Table 44. Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms

Cost containment feature	All participants	Professional and administrative participants	Technical and clerical participants	Production participants
Covered by at least one of the listed cost containment features[1]	68	70	70	65
Incentive to seek second surgical opinion	35	40	40	28
Higher coinsurance, or lower or no deductible for outpatient surgery	28	31	33	23
Higher payment for generic prescription drugs	7	7	7	6
No or limited reimbursement for nonemergency weekend admissions to hospital	10	13	13	8
Separate deductible for hospital admission	9	10	9	7

[1] The total is less than the sum of the individual items because many workers participate in plans with more than one feature.

NOTE: This table was prepared by TPL TABLES, a product of QQQ Software, Inc.

Table 45. Additional information on health care benefits

Cost containment feature	All participants	Professional and administrative participants	Technical and clerical participants	Production participants
Urging prehospitalization testing	47	51	52	43
Preadmission certification requirement	16	18	15	16
Higher payment for delivery at birthing center	12	15	12	11
Incentive to audit hospital statement	2	3	2	1
Not covered by any of the listed cost containment features	32	29	29	34
Dental plan only[1]	1	1	1	([2])

[1] Participants who elected dental coverage only were not included in this tabulation.
[2] Less than 0.5 percent.

NOTE: This table was prepared by TPL TABLES, a product of QQQ Software, Inc.

To join the two tables, use the folloing statements:

```
SKIP 0 LINES AFTER TABLES;
FOR TABLE 2:  DELETE HEADING;
              DELETE TITLE;
```

Effect The two tables are joined to look like one. Footnotes are merged so that numbering and printing of footnote text is correct.

Table 44. Health care benefits: Percent of full-time participants by coverage with selected cost containment features, medium and large firms

Cost containment feature	All participants	Professional and administrative participants	Technical and clerical participants	Production participants
Covered by at least one of the listed cost containment features[1]	68	70	70	65
Incentive to seek second surgical opinion	35	40	40	28
Higher coinsurance, or lower or no deductible for outpatient surgery	28	31	33	23
Higher payment for generic prescription drugs	7	7	7	6
No or limited reimbursement for nonemergency weekend admissions to hospital	10	13	13	8
Separate deductible for hospital admission	9	10	9	7
Urging prehospitalization testing	47	51	52	43
Preadmission certification requirement	16	18	15	16
Higher payment for delivery at birthing center	12	15	12	11
Incentive to audit hospital statement	2	3	2	1
Not covered by any of the listed cost containment features	32	29	29	34
Dental plan only[2]	1	1	1	([3])

[1] The total is less than the sum of the individual items because many workers participate in plans with more than one feature.
[2] Participants who elected dental coverage only were not included in this tabulation.
[3] Less than 0.5 percent.

NOTE: This table was prepared by TPL TABLES, a product of QQQ Software, Inc.

Restrictions The last table to begin on a page must have enough space for at least one line of data. Otherwise, TPL TABLES will start the table on a new page.

SKIP AFTER WAFER

Format

SKIP n LINES AFTER WAFERS;

where **n** is a number.

Meaning

The SKIP AFTER WAFERS statement can be used to control the grouping of wafers on a page. It tells TPL TABLES to skip the specified number of lines after a wafer and continue printing without going to a new page. The table title is shown only for the first wafer on a page. If a headnote has been specified, the headnote will be shown only for the first wafer on a page. Footnotes are shown only for the last wafer on a page. In all other ways the wafer formats are unchanged.

When some tables are to be printed without going to a new page for each wafer and some are to be printed with each wafer starting on a new page, the EJECT AFTER WAFERS statement can be used where wafers are to start on a new page.

If you specify SKIP 0 LINES AFTER WAFERS; and the table is unbanked (all columns fit on the page), then the table heading is removed for all wafers except the first on each page. If the table is banked, the headings will not be removed.

SKIP 0 generally works best with spanning wafer labels between wafers. See the FORMAT statement WAFER LABEL (Spanner) for an example of this use.

Level

Both SKIP AFTER WAFERS and EJECT AFTER WAFERS can be specified at the individual wafer level. NOTE that only one of the two statements can apply to a particular wafer.

Default

EJECT AFTER WAFER;

Example

FOR TABLE 1: SKIP 3 LINES AFTER WAFERS;

Effect

At the end of a wafer in the first table, three lines will be skipped and the next wafer will begin without going to a new page.

Example

```
SKIP 2 LINES AFTER WAFERS;
FOR TABLE 2: EJECT AFTER WAFERS;
```

Effect At the end of each wafer, other than in the second table, three lines will be skipped and the next wafer will begin. In the second table, each wafer will begin on a new page.

Restrictions The last wafer to begin on a page must have enough space for at least one line of data. Otherwise, TPL TABLES will start the wafer on a new page.

If you have a banked table and have also specified FOOTNOTES EACH PAGE; each wafer will begin on a new page regardless of any other specification. In other words, the SKIP statement will be ignored.

SPANNER HEADING

Format SPANNER HEADING;

The word HEAD can be used in place of the word HEADING. The word SPAN can be used in place of the word SPANNER.

Meaning Whenever possible, the table heading will be formatted so that if the same label occurs in heading boxes in adjacent columns and the label has the SPANNER attribute, the label will be placed in one box that spans the columns.

Level SPANNER HEADING can be specified for individual tables.

Default The heading is built from the top down, and heading boxes are aligned with adjacent boxes, except at the bottom level where the box heights may need to vary to fill the space above. Labels are not consolidated into spanning boxes unless they are in adjacent boxes of the same height.

Example In the following table, all of the columns except the first have the label "Average Income" at the lowest level, but the label does not span across all of its columns because they are not all the same height.

Average income by household type and region.

	Number of Households	Married couple	Other family	Nonfamily household	Sex of Householder	
		Average Income			Male	Female
					Average Income	
Region						
Northeast	872	45,672	28,429	19,722	41,881	23,048
Midwest	180	26,798	19,476	15,772	25,008	19,313

If we use the following FORMAT statements to assign the SPANNER attribute to the "Average Income" label and request a SPANNER HEADING, the heading will be formatted so that the "Average Income" label can be consolidated into one spanning box.

```
FOR VARIABLE AVG_INCOME:
        REPLACE LABEL WITH SPANNER 'Average Income';
SPANNER HEADING;
```

Average income by household type and region.
The label for Average Income has been converted to a spanner.

	Number of House-holds	Married couple	Other family	Nonfamily household	Sex of Householder	
					Male	Female
		Average Income				
Region						
Northeast	872	45,672	28,429	19,722	41,881	23,048
Midwest	180	26,798	19,476	15,772	25,008	19,313

Note The above example demonstrates how the SPANNER attribute can be assigned to a label in the format request. You can also assign the SPANNER attribute to a label when you enter it in a codebook or table request. If a SPANNER label occurs in the table stub, it will be formatted to span the data section of the table; if it is used in the heading the SPANNER attribute will be ignored if you do not also use the SPANNER HEADING statement.

Example In the next example, two tables are created by one table request and displayed on a single page. The table at the top of the page shows the default treatment for a table heading with many labels that you would like to join as spanning labels. The table at the bottom of the page shows how you can combine the SPANNER HEADING statement with others to **join the labels in the heading** and, in addition, **remove extra space** from the heading, **delete the rules from the spanners** within a table and **delete the rule at the bottom** of the table.

Table Request

```
use survey codebook;

label label1 'label 1';
label label2 'label 2';
label label3 'label 3';
label label4 'label 4';
label label5 'label 5';
label label6 'label 6';
label label7 'label 7';
label label8 'label 8';
label label9 'label 9';
label label10 'label 10';
label label11 'label 11';
label label12 'label 12';
label label13 'label 13';
label label14 'label 14';
label label15 'label 15';
label label16 'label 16';
label label17 'label 17';
label label18 'label 18';
label label19 'label 19';
label label20 'label 20';
label label21 'label 21';
label label22 'label 22';

table one  center 'Table ONE':

    heading  (label1 by (label2 then label3) then label4
     then (label5 then label6 then label7) by label8) by label9
     then (label10 then (label11 by (label12 then label13 then label14)
     then label15 then (label16 then label17 then label18) by label19)
     by label20) by label21 then label22;

    stub       count;
```

```
define new_sex on sex;
''  if 'm';     /* Note: The first wafer has a null ('') label. */
    if 'f';
copy if 'm';
copy if 'f';

table two
center 'Table ONE with SPANNER HEADING and '
          'HEAD SPACE = .2;'/
          'Heading boxes with labels that have the SPANNER '
          'attribute are joined whenever possible.':

      heading  (label1 by (label2 then label3) then label4
       then (label5 then label6 then label7) by label8) by label9
       then (label10 then (label11 by (label12 then label13 then label14)
       then label15 then (label16 then label17 then label18) by label19)
       by label20) by label21 then label22;

      stub       age;

      wafer   new_sex;
```

Format Request

```
rotate;
column width = .45 in;
stub width = .7 in;
skip 5 lines after tables;   /* Puts both tables on the same page */

set footnote on_statements
        symbol '' text font hi 'Other new statements:'/
                        'head space = .2;'/
                        'delete end rules;'/
                        'delete spanner rules;';

/* Table two combines the new features. */

for table 2:
        keep footnote on_statements;   /* for documentation only */
        heading space = .2;     /* Remove space above and below
                                   heading labels. */
        wafer label = row spanner;
        skip 0 lines after wafers;
        delete down rules;
        delete spanner rules;
        delete last rules;      /* Required to remove rules between
                                   wafers that have spanner labels. */
        delete end rule;
        spanner heading;        /* Join the heading labels with the
                                   SPANNER attribute if possible. */

/* The following statements assign the SPANNER attribute to heading
labels that should be combined to span all of their columns.  Note that
since the statement "spanner heading;" has only been specified for
table 2, it only affects the heading of table 2, even though the same
labels are also used in table 1. */

for variable label8: replace label with spanner 'label 8';
for variabe label9: replace label with spanner 'label 9';
for variable label19: replace label with spanner 'label 19';
for variable label20: replace label with spanner 'label 20';
for variable label21: replace label with spanner 'label 21';
```

Table ONE

	label 1		label 4	label 5	label 6	label 7	label 10	label 11			label 15	label 16	label 17	label 18	label 22
	label 2	label 3	label 9	label 8			label 21	label 12	label 13	label 14	label 20	label 19			
	label 9			label 9				label 20			label 21	label 20			
								label 21				label 21			
Count	160	160	160	160	160	160	160	160	160	160	160	160	160	160	160

Table ONE with SPANNER HEADING and HEAD SPACE = .2; Heading boxes with labels that have the SPANNER attribute are joined whenever possible.

	label 1		label 4	label 5	label 6	label 7	label 10	label 11			label 15	label 16	label 17	label 18	label 22
	label 2	label 3		label 8				label 12	label 13	label 14		label 19			
	label 9							label 20							
							label 21								
Age															
14	72	72	72	72	72	72	72	72	72	72	72	72	72	72	72
15	69	69	69	69	69	69	69	69	69	69	69	69	69	69	69
16	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
Male															
Age															
14	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
15	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36
16	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Female															
Age															
14	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45
15	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
16	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10

Other new statements:
head space = .2;
delete end rules;
delete spanner rules;

STUB CONTINUATION

Format STUB CONTINUATION = amount [unit];

where **amount** is a number and **unit** is optional. If no unit is specified, **characters** are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as i**nches**, **cm** or **points**.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning If a stub label is too long for the available space, it will automatically be segmented over two or more lines. All continued segments will be indented n character positions from the first line segment for that label.

Level Stub continuation can be controlled at the individual table level. Stub continuation cannot change within a table.

Default STUB CONTINUATION = 3;

Example STUB CONTINUATION = 2;

Effect If a stub label is too long for the available space, the continued segments will be indented 2 character positions from the first line segment for that stub. Thus, if the first line of a label starts in position 1, continuation lines will start in position 3.

Restrictions If indention of a continuation segment of a label would cause the segment to start beyond the position specified by STUB STOP, the segment will not be indented. It will be aligned with the first line of the label. The default position for stopping indentation is at the middle of the stub.

In a text table, continuation will be rounded to the nearest character.

STUB INCREMENT

Format STUB INCREMENT = amount [unit];

where **amount** is a number and **unit** is optional. If no unit is specified, **characters** are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as **inches**, **cm** or **points**.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning For each level of nest in the stub, indent the stub labels n character positions.

Level Stub increment can be controlled at the individual table level. Stub increment cannot change within a table.

Default STUB INCREMENT = 2;

Example STUB INCREMENT = .4 inches;

Effect For each level of nest in the stub, indent the stub labels .4 inches. Thus, if the labels for the first level of nest starts 1 inch from the left margin, labels for the second level of nest will start 1.4 inches from the margin, and so on.

Restrictions If there are enough levels of nest in the stub to cause the indentation to go beyond the position specified by STUB STOP, indentation will stop. The default position for stopping indentation is at the middle of the stub.

In a text table, indentations will be rounded to the nearest character.

STUB LEFT

The default stub location is STUB LEFT which means that tables are normally formatted with the stub on the left side. Alternate stub locations are described under the STUB RIGHT statement.

STUB RIGHT

Format STUB RIGHT;

Meaning With STUB RIGHT, tables are formatted with the stub on the right side of the table instead of the left. This is most often used to prepare tables on facing pages. It is particularly useful if you need to do a table in two languages on facing pages where the left page has the stub on the left in one language and the right page has the stub on the right in another language.

Note To divide a table into left and right facing pages with left and right stubs, you need to prepare two table statements with half of the heading in each table. To align the two tables, you may need to make the right STUB WIDTH somewhat larger. Otherwise, the stub labels on the right side are likely to "break" sooner and require more lines than the stub labels on the left.

Note If you need different languages in the two stubs, you must have a different set of stub labels for each of the two tables. You can do this with REDEFINE in the codebook or with DEFINE and COMPUTE statements in the table request or with format label statements.

For right-hand table stubs, the default start position for stub labels is indented five positions (**STUB START = 5;**); for left-hand stubs, the default start position is the left edge of the stub (**STUB START = 0;**).

See also the section on LEFT, RIGHT and CENTER in the "Labels" chapter for an example of the effect of alignment specifications when used in labels for a stub on the right.

Level Stub position can be specified at the individual table level.

Default STUB LEFT;

Example FOR TABLE 2: STUB RIGHT;

Effect The second table will be formatted with the stub on the right.

Note If you have specified STUB RIGHT for a text table, and you are using a STUB START value expressed in characters, one of the characters will be taken up by the vertical bar between the data part of the table and the stub. Thus, if you are using the default **STUB START = 5;** for right-hand stubs, you will get only 4 dots preceding the beginning of the left-most stub labels.

Restrictions If you have specified STUB RIGHT; and you want to set **STUB START = 0;** the STUB START statement must follow the STUB RIGHT statement. In any other situation, the order of the statements is irrelevant. For esthetic reasons, we do not recommend the combination of STUB RIGHT and **STUB START = 0;** unless you delete down rules or replace the divide character with blank. If the rule is present, the stub labels may look like they are touching the vertical rule at the edge of the data section of the table.

STUB START

Format STUB START = amount [unit];

where **amount** is a number and **unit** is optional. If no unit is specified, **characters** are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as **inches**, **cm** or **points**.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning The left-most position for stub labels is indented by the specified amount.

Level Stub start amount can be specified at the individual table level.

Default

```
STUB START = 0;      for left-hand table stubs
STUB START = 5;      for right-hand table stubs.
```

Example

```
FOR TABLES 3 AND 4:  STUB START = 2 CM;
```

Effect For tables 3 and 4, the left-most stub label position will be indented 2 centimeters from the left edge of the stub.

Restrictions If you have specified STUB RIGHT; and you want to set **STUB START = 0;** the STUB START statement must follow the STUB RIGHT statement. In any other situation, the order of the statements is irrelevant. For esthetic reasons, we do not recommend the combination of STUB RIGHT and **STUB START = 0;** unless you delete down rules or replace the divide character with blank. If the rule is present, the stub labels may look like they are touching the vertical rule at the edge of the data section of the table.

STUB STOP

Format STUB STOP = amount [unit];

where amount is a number and unit is optional. If no unit is specified, characters are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as inches, cm or points.

The word IS can be used in place of **=**. Both are optional and can be left out altogether.

Meaning The last position within the stub at which a stub label line can begin printing is indicated by **n**.

Level Stub stop can be controlled at the individual table level. Stub stop cannot change within a table.

Default The default STUB STOP is the middle of the stub. For example, if the stub width is 20 and no STUB STOP is specified, no label or label segment can begin beyond position 10.

Example

```
STUB STOP = 15;
```

Effect Stub labels (including continuation segments) that would begin at a position beyond position 15 according to other rules of stub label placement will not be indented.

Restrictions STUB STOP cannot be greater than the stub width. If STUB STOP is 0 or 1, there will be no stub indentation.

STUB WIDTH

Format STUB WIDTH = amount [unit];

where amount is a number and unit is optional. If no unit is specified, characters are assumed. If a unit is specified, the amount can be a decimal number and unit can be expressed as inches, cm or points.

The word **IS** can be used in place of **=**. Both are optional and can be left out altogether.

Meaning Make the table stub **n** characters wide.

See also AUTOMATIC STUB AND COLUMN WIDTHS in this chapter for automatic adjustment of stub widths to fill the available space.

Level Stub width can be controlled at the individual table level. Stub width cannot change within a table.

Default STUB WIDTH = 20;

Example FOR TABLE 3: STUB WIDTH = 30;

Effect The stub will be 30 characters wide in table 3.

Note **STUB WIDTH = 0;** has the same effect as the **DELETE STUB** statement. All stub entries, including SPANNER labels, are deleted.

Restrictions The minimum stub width is 3 characters. The page must be wide enough to hold the stub + the margins + the widest column.

TABLE SPACE

Format

TABLE SPACE = n;

where **n** is a decimal number that is a multiplier of the font size.

Meaning

TABLE SPACE can be used to increase or decrease the vertical space between tables elements. For example, space is changed between title and top of heading, between bottom of heading and first data row, and between last data row on a page and the horizontal line below it.. The most common use is to decrease the space so that the table will take less vertical space on the page. TABLE SPACE is ignored in the export of text tables.

Important Note

If you simply want to scale down the size of your table, ***see the SCALE statement***. With the SCALE statement, you can reduce the overall size of everything in a table to a percentage of its original size and fit more of your table on a page or other smaller space. TABLE SPACE does not work with all types of table, but SCALE will work with all tables.

In general, the minimum recommended table space is .2.

The space *between* lines of data or labels is not affected by the TABLE SPACE statement. See the statement EXTRA LEADING to change the space between lines.

See also HEADING SPACE to change the vertical space in the headig only.

Level

TABLE SPACE can be specified for individual tables.

Default

TABLE SPACE = 1.15;

Example

Following are two tables, the first with the default table space of 1.15 and the second with table space of .3:

FOR TABLE 2: TABLE SPACE = .3;

Table with default table space.

	Total	Sex		
		Female	Male	No response
Total	160	88	70	2
Age				
14	72	45	27	–
15	69	33	36	–
16	19	10	7	2

– Data not available.

Table with table space reduced to .3.

	Total	Sex		
		Female	Male	No response
Total	160	88	70	2
Age				
14	72	45	27	–
15	69	33	36	–
16	19	10	7	2

– Data not available.

Restrictions TABLE SPACE should not be used with statements that join tables, wafers and banks, for example:

```
SKIP 0 LINES AFTER TABLES;
SKIP 0 LINES AFTER WAFERS;
SKIP 0 LINES AFTER BANKS;
```

The combination of these statements with TABLE SPACE will produce undesirable results such as extra horizontal rules or overlapping lines of text.

For a table that has stub or wafer labels that have the SPANNER attribute, TABLE SPACE may not give the desired results. For example, vertical rules may run into spanner labels.

If you have a table with one of the listed restrictions or another type of table for which TABLE SPACE does not give you the results you expect, we recommend the SCALE statement.

TEXT TABLE OUTPUT (UNIX only)

Format TEXT TABLE OUTPUT = YES or NO or PROMPT;

Normally, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for **TEXT TABLE**, and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**.

UNDERLINE ROW

This command has been replaced by
RETAIN RULE AFTER ROW UNDERLINE;

Format UNDERLINE ROW;

Note The UNDERLINE ROW has no effect when applied to exported text tables.

Meaning Underline the data section of the specified rows. A row is defined as a data row in the table. If a row has a stub label or cell contents that take more than one line, it still counts as one row, so the blank line will follow the bottom line for the row.

The statement can be used with a FOR clause to underline only selected rows. If used without a FOR clause, it will insert a line after every row of every table.

Note If any rows of a table do not appear in the table because they are **empty** (do not have any data) or because the rows are **ranked**, you cannot determine row numbers by counting data rows in the printed table. If you are using TED, you can just select the rows you wish to underline. If you do not have access to TED, you can find the row numbers for **PRINTED ROWS** in the OUTPUT file. If you reference an empty row, the UNDERLINE ROW statement will have no effect.

See also the statement RULE AFTER ROW for another way of adding horizontal lines to the body of a table.

Level UNDERLINE ROW can be specified for individual rows.

Default Tables are formatted without underlined rows.

Example

```
FOR TABLE ONE:  UNDERLINE ROWS;
```

Effect For Table One, the data portion of the row is underlined for all data rows.

Table F10: Amount of training and average age by sex.

	Total	*Sex*		
		Female	*Male*	*No response*
Total				
Average Age	*44*	*44*	*44*	*48*
Employer Training with multi-line label	*High*	*High*	*High*	*Low*
Manufacturing				
Average Age	*46*	*([1])*	*Multi-line text for mask*	*48*
Employer Training with multi-line label	*High*	*Medium*	*Medium*	*Low*
Other				
Average Age	*43*	*43*	*43*	*–*
Employer Training with multi-line label	*High*	*High*	*Medium*	*–*

[1] Confidential
– Data not available.

Example

```
FOR TABLE ONE, ROWS 2 TO 100 BY 2:  UNDERLINE ROWS;
```

Effect For Table One, the data portion of the row is underlined for alternate data rows in the range of row 2 to row 100.

WAFER LABEL SPANNER

Format

```
WAFER LABEL = DATA SPANNER;
WAFER LABEL = ROW SPANNER;
WAFER LABEL = HEADNOTE;
```

Any one of the three options can be chosen. The word IS can be used in place of **=**. Both are optional and can be left out altogether.

Meaning

Normally wafer labels are displayed at the top left of each wafer between the table title and the heading. This is the HEADNOTE position. If you choose WAFER LABEL = DATA SPANNER, the wafer label will be displayed as a spanner label, directly beneath the table heading, spanning the data columns. If you choose WAFER LABEL = ROW SPANNER, the wafer label will be displayed as a spanner label, directly beneath the table heading, spanning the entire width of the table. In either case, the spanning label will have a horizontal rule (line) above and below.

Spanning wafer labels are centered by default. You can override these defaults with the statement ALIGN WAFER LABELS. An alignment specification that is part of a wafer label will take precedence over any other alignment specifications. If you have nested variable labels in a wafer with different alignment specifications for each and you do not like the spanner label that results, you can replace it with the statement REPLACE WAFER LABEL.

Note

Wafer labels can only be turned into spanners by using a WAFER LABEL statement. A SPANNER specification entered directly into a label will only produce a spanner if the label is used in the stub.

For unbanked tables, WAFER LABEL = DATA or ROW SPANNER; works well with the statement SKIP 0 LINES AFTER WAFERS; Normally, each wafer begins on a new page. If you combine the WAFER LABEL and SKIP 0 statements, you can join wafers with spanning wafer labels.

Level

WAFER LABEL can be controlled at the individual table level.

Default

```
WAFER LABEL = HEADNOTE;
```

Example

```
WAFER LABEL = ROW SPANNER;
```

Effect

Wafer labels are displayed as spanners across the full width of the table under the table heading. Each wafer begins on a new page.

Example

```
WAFER LABEL = DATA SPANNER;
SKIP 0 LINES AFTER WAFERS;
```

Effect Wafer labels are displayed as spanners across the data portion of the table.

Does age and sex of siblings affect nervousness with members of the opposite sex?

	Nervous with opposite sex?		
	Total	Nervous	Not nervous
	Female		
Get along with Mom?			
Yes	60	37	23
No	3	1	2
So-So	23	12	11
N/A	1	1	–
Total	87	51	36
	Male		
Get along with Mom?			
Yes	54	22	32
No	2	–	2
So-So	11	4	7
N/A	2	2	–
Total	69	28	41
	Both sexes		
Get along with Mom?			
Yes	114	59	55
No	5	1	4
So-So	34	16	18
N/A	3	3	–
Total	156	79	77

– Data not available.

If the table is not banked, i.e. all columns fit across the page as shown in the illustration for three wafers, the wafers are joined as follows. For the first wafer on a page, the spanner is directly beneath the heading. For subsequent wafers on the same page, all title, headnote and heading lines are removed leaving only the spanning wafer labels between wafers. Footnotes are displayed only at the end of a page.

If the table is banked and footnotes are to be displayed at the end of the table, the wafers will be joined whenever possible, but the table heading will not be deleted.

If the table is banked and you have specified FOOTNOTES EACH PAGE; each wafer will begin on a new page regardless of other specifications. The SKIP 0 LINES statement will be ignored.

Restrictions The selection of spanner type, ROW, DATA or HEADNOTE, is unaffected by the DATA SPAN or ROW SPAN statements. These statements apply only to SPANNER labels entered in the table stub and horizontal rules specified with RULE AFTER ROW.

XLS OUTPUT (UNIX only)

Format XLS OUTPUT = YES or NO or PROMPT;

Normally, TPL TABLES will prompt you at the end of a job to find out if you would like to export the tables to other formats. To prevent the prompt for **XLS**, and the other export statements, you can use this statement and each of the other export statements with **YES** or **NO**.

APPENDIX A

Installation (Windows)

HOW TO INSTALL TPL TABLES UNDER WINDOWS

Installing from the CD

If you are replacing an earlier version of TPL TABLES, please review the next section before installing the new version.

To install, insert the CD in the CD drive. After a pause, the installation process may begin automatically. If it does not start automatically, go to **Start** then **Run**. Select the file **setup.exe** on the CD and click on "OK".

Respond to the prompts.

Installing from Download

If you are replacing an earlier version of TPL Tables, please review the next section before installing the new version.

To install, download the self-extracting file and execute it.

Respond to the prompts.

If You Have an Earlier Version of TPL TABLES

.tpl Files

During installation, new **profile.tpl**, **color.tpl** and **country.tpl** files will be placed in the TPL system directory. If you are installing a new version on top of a previous one and have previously edited these files to establish your own set of system defaults, you will probably want to save them in another place and copy them into the system directory after doing the new installation. If you are installing TPL Tables in a new location, you may wish to wish to update the new **.tpl** files with lines from the older **.tpl** files.

Replacing a Previous Version of TPL TABLES

If you do not want to retain your previous version and you want to install the new version in the same place as the old, we recommend that you uninstall the old version before installing the new one.

Using More than One Version of TPL TABLES

A new version of TPL TABLES can be installed without removing earlier versions. The different versions will not interfere with each other provided that they are installed in different directories and run against codebooks processed by the correct version.

tpl.ini

For Version 6, a file named **tpl.ini** was installed in the Windows system directory (e.g. c:\winnt or c:\windows). **tpl.ini** is a text file that contains information about your TPL TABLES preferences and also path information for TPL modules. Each time you use TPL TABLES, the current preference settings will be saved in **tpl.ini**. You should not directly modify this file. Instead use the various preferences menus in the TPL system to set these values. For Version 7 and following, the **tpl.ini** file has been moved to a location specified by the environment variable **TPL_INI.** This environment variable is set during installation. Version 6 will continue to use the copy of **tpl.ini** in the Windows system directory. So if you have both Version 6 and Version 7 installed, they will not necessarily have the same preferences. From version 7 forward, a common **tpl.ini** file is used.

Network Installation

In a network installation, it is desirable to have the TPL TABLES programs in a common location on a server but make preferences user specific. Version 7.1 supports this goal in such a way that it does not compromise restricted directories. TPL Version 7.1 uses two environment variables, **TPLPATH7.1** and **TPL_INI.** On each machine using TPL TABLES, set TPLPATH7.1 to the common server location where the modules are installed. Set TPL_INI to a location on the user's machine that he has read and write access to.

It is also desirable to add a directory structure to the user's start menu with entries for each of the programs, documentation, and help files included in a standard install. Adding a desktop shortcut which points to this directory structure in the start menu completes a full network install.

Compatibility

"Source" Files

"Source" files, including codebook sources, table requests, and format requests, that run with earlier versions of TPL TABLES should run without change. The only exception is if you have a name that has been added to the list of Keywords. In the unlikely event that this happens, you will get a message and will need to change the name.

Codebooks, table requests and format requests created interactively and saved by Codebook Builder, Table Builder or interactive TED may contain statements that will not work with earlier versions of TPL TABLES.

Codebooks and TPL Subdirectories

Codebook objects (processed codebooks) created by earlier versions are not compatible with Version 7.1. Before running tables jobs, you will need to reprocess the codebooks.

TPL subdirectories created by earlier versions of TPL Tables are not compatible with Version 7.1. You cannot do a rerun process using an old TPL subdirectory with this version of the system.

Default Settings in Profile.tpl

After you have installed TPL TABLES, there will be a file called **profile.tpl** in the directory where you installed the system. It contains a set of text statements that determine defaults for basic activities. A sample profile after installation is:

```
Postscript = yes ;
Default font = H 8;
Footnote text font = T 8;
Footnote symbol font = H 8;
Title font = HB 10;
paper = LETTER;
```

All statements entered in the profile during installation are described in detail in the FORMAT Language chapter of the manual.

If you wish, you can change the values in the profile after installation and also have different profiles for different sets of jobs.

Networks

Licensing Note

If you are accessing a copy of TPL TABLES installed on a network server, you must have a license to use TPL TABLES on your PC.

APPENDIX B

Run Instructions (Windows)

INSTRUCTIONS FOR RUNNING TPL TABLES UNDER WINDOWS

Introduction

TPL TABLES can be run using interactive menus, or it can be run as a batch process using scripts. Scripts are described in a separate appendix as well as in TPL Help. This appendix describes the basic information needed to run jobs from menus and the various input and output files for different types of jobs. Similar information can also be found in TPL Help, along with additional details about the options available in the interactive menus for running jobs.

TED and Other Editors

TPL TABLES is designed to allow you to use the editor (word processing program) of your choice to create codebooks, table requests and format requests. Any editor that creates stand-alone ASCII text files is acceptable.

You can also use TED, the TPL Editor. TED lets you edit, view, and print character files. It also allows you to view, print, and interactively edit tables produced by TPL TABLES. You can use it to print tables on any Windows compatible printer. Finally TED allows you to export tables in formats usable by other software.

If you are running a job that stops because of errors, TPL TABLES will transfer to TED to allow you to view the error messages and make corrections. When you are finished, you can return to TPL TABLES to resume processing.

TED is an integral part of TPL TABLES, but you can also use it without starting TPL TABLES. See TED Help for complete details.

Description of Jobs and Files

Getting Started

You can start TPL Tables by clicking on the TPL icon or by going to **Start** then **Programs** then **QQQ Software** then **TPL**.

You can run as many jobs as you wish without leaving TPL.

Selecting the Job Directory

The **Job Directory** is the directory in which your TPL jobs will run. You will probably find it most convenient to set the Job Directory to the directory where your codebook, data and request files are stored, but you can choose a different directory if you wish.

Outputs are stored in the Job Directory. These outputs include the processed codebooks and TPL subdirectories described in this appendix. For this reason, it is important to know what Job Directory you are in. If an output is not found in the expected place, the likely reason is that the Job Directory was set to a place other than the intended one.

To see what the current Job Directory is or to change to a different Job Directory, go to **File** then **Job Directory**.

Creating and Processing Codebooks

Codebooks can be created either with an editor or interactively with Codebook Builder. To create a codebook interactively, go to **File** then **Build Codebook** in the main TPL screen. Instructions for creating codebooks interactively can be found in ***Codebook Builder Help***.

After you have created the codebook, you can save it into a file with a name of your choice. Usually the codebook is saved into a file with the same name used at the beginning of the codebook. For example, if you name the codebook **Survey** with the codebook entry **Begin Survey Codebook**, save it with the name **Survey.cbk**. The codebook file you have created is called the **codebook source**.

Run the codebook processor, giving it the name of the codebook source. In the main TPL screen, go to **Run** then **Codebook**. When

prompted, enter the name of your codebook source file, for example **Survey.CBK**.

If any errors are found in your codebook, you will be transferred to TED where you will see two windows open, one containing your codebook source and another showing the source with error messages. Edit the codebook source to correct the errors. When you are finished, Go to **Return to TPL** then **Save changes and try again**. Your corrected codebook source will automatically be saved before processing continues.

Note In some cases, you may not wish to resume processing. For example, if you have accidentally entered in the menu the name of a file that is not a codebook source file, you will need to go back to the menu to correct it. In this case, go to **Return to TPL** then **Cancel**.

As the codebook is being processed, the source and any error messages are saved in a file with the same name as the codebook and a **.O** extension. For example, if your codebook is named Survey, the file is called **Survey.O**.

When your codebook has been processed successfully with no errors, the **.O** file will be deleted and the **.L** codebook abstract file will take its place. You can view and/or print the abstract in TED by clicking on the **Review/ Print** button when your job is completed. When you are finished, you can close TED or go to **Return to TPL** then **Resume**.

Codebook Abstract

The abstract includes the name of the codebook source file, the date and time of processing, and the TPL version number. In addition, it contains a list of the codebook variables in alphabetical order along with each variable's size and location within a record. This information is particularly useful if you have an alignment problem between your codebook and your data file. You may also find the abstract useful as a quick reference when preparing your table specifications. If you are creating a codebook describing a **CSV** or other type of delimited file or a database, the information in the abstract will differ slightly.

For each control variable, there is a count of the number of condition values.

Note If your data file is an ASCII file, it will have carriage return/line feed characters at the end of each record. These will not be included in the record sizes listed in the abstract. ASCII is the default file type.

Codebook Object

The processed codebook is called the **codebook object**. When you have successfully run the codebook processor, your codebook object will be stored with **.K** appended to the name. Thus, for a codebook named Survey, the codebook object will be given the name **Survey.K.**

Once your codebook is successfully processed, you can run any number of table jobs using the same codebook object.

Note The name of your codebook object will always be derived from the name you have used in the BEGIN codebookname .... entry in the codebook, regardless of the name you give to your codebook source file.

Database Codebook Source

The following applies if you have the TPL-SQL database interface. When a database codebook is processed, there is another file generated in addition to the **.K** and **.L** files. This is the **.S** file. When a you create a codebook source for a database, you omit some items such as field widths and control variable condition values. These are filled in by gathering data from the database. The **.S** file is a new codebook source with the additional data filled in. See the TPL-SQL chapter and/or TPL Help for more details.

Producing Tables

Tables requests can be created either with an editor or interactively with Table Builder. To create a table request interactively, go to **File** then **Build Table** in the main TPL screen. Instructions for creating table requests interactively can be found in ***Table Builder Help***.

In the USE statement at the beginning of a table request, you can refer to the codebook using the same name you used in the **BEGIN codebookname CODEBOOK** statement. Using the name **Survey** shown in the example above, you would say **USE Survey CODEBOOK;** at the beginning of your table request. TPL TABLES will know to look for a codebook object file called **Survey.K** for descriptive information about your data file.

If your codebook is in a subdirectory other than the one in which you are running your table job, you can give the complete name for the codebook in the USE statement.

Save your table request with any valid Windows file name, for example, **Survey.REQ.**

You may also have an optional format request giving detailed specifications for formatting your tables. The format request can have any valid Windows file name, for example, **Survey.FMT**.

To produce tables, you need to enter the names of the table request file, the data file and (optionally) the format request. In the main TPL screen, go to **Run** then **Table Request**. When prompted, enter the name of your table request, your data file, and (optionally) the format request. If your data is contained in more than one file, see the Data chapter for instructions on multi-file input.

If any errors are found in your table or format requests, you will be transferred to TED where you will see two windows open, one containing your table request or format request and another showing the output with error messages. Edit the request to correct the errors. When you are finished, go to **Return to TPL** then **Save changes and try again**. Your corrected request will automatically be saved before processing continues.

Note

In some cases, you may not wish to resume processing. For example, if you have entered an incorrect data file name in the menu, you will need to go back to the menu to correct it. In this case, go to **Return to TPL** then **Cancel**.

When TPL TABLES has finished processing your data and calculating the values for your tables, you can review the tables and other output on the screen, print them, or export the tables into files of different types such as Encapsulated PostScript or HTML. To do this, transfer to TED by clicking on the **Edit/Print** button. Your outputs will be opened in TED. When you are finished, you can close TED or go to **Return to TPL** then **Resume**.

You can also edit PostScript tables interactively in TED. Double-click on any part of a table and you will be presented with editing options. Complete instructions for interactive editing can be found in ***TED Help***.

The TPL Subdirectory

Each time you run a tables job, a subdirectory is created to contain the files needed to create your tables. The subdirectory has the name **TPLnnnnn** where **nnnnn** is a randomly generated number with 1 to 5 digits. You can override this number by entering the number of your choice in the Table Request screen where you enter the file names to be used for your tables job.

See also the Script arguments -O and -N for selecting subdirectory numbers in TPL scripts.

Completed tables are stored in a TPL subdirectory file called **TABLES.PS.** If Postcript = no; is used, the tables are stored in the file **TABLES**.

The file called **OUTPUT** contains your request files, the names of your data and request files, the date and time of execution for each part of the job, the TPL version number, and, at the end, the name of the TPL subdirectory in which it was created. It also shows how many pages of tables were created, and which lines and columns will be printed on each page.

The other files in the subdirectory are not intended to be read but are used by TPL TABLES and saved in case you want to reformat the tables without reading the data again.

Subdirectory Maintenance

If you go to Run then Remove Directories in the main TPL screen, you can get a list of TPL subdirectories. In addition to removing all or selected directories, you can add notes to the directories. If you click on a subdirectory on the list, you will get the date and time that the subdirectory was created and a display of any notes that have been added.

The subdirectories on the list are those contained in the current Job Directory. The Change button lets you change to a different Job Directory.

Rerunning the Format Step to Make Modifications

After running a table job, you may see that the appearance of your tables could be improved by changing certain formatting characteristics of the tables. For example, if the numbers in the tables are very large, the default column width may be too small, or you may want to change a table title or label. You can quickly change the format of your tables by rerunning only the table formatting part of a job.

You specify the changes that you want using the FORMAT language. The FORMAT statements go into a file called a **format request** that you create using TED or some other editor. The format request file can have any valid Windows name, for example, **Survey.FMT**.

You can also create or modify a format request interactively in TED. Complete instructions for interactive editing can be found in ***TED Help***.

To rerun the format step, TPL TABLES needs to know the number of the TPL TABLES subdirectory containing the existing tables and the name of the format request file. In the main TPL screen, go to Run then Rerun. When prompted, enter the name of your format request and the TPL Subdirectory for the original run. When the tables are reformatted, the new version of the tables will replace the originals in the TABLES.PS or TABLES file.

If you rerun the formatting step and do not like the result of your format changes, you can always get back to the original tables by rerunning this step again without entering the name of a format request file.

You can reformat a set of tables any number of times without reprocessing the data.

Interactive Edit and Export of Tables

As already described, you can transfer to TED after a run or rerun using the **Edit/Print** button and have the table output displayed for further activities. If you start TED later and simply open a **tables.ps** file, the full range of activities is not available. In particular, you cannot do most types of exports or edit the table interactively.

To make the full range of activities available, go to **File** then **Edit Table** in the main TPL screen. You will be prompted for the TPL subdirectory that contains the **tables.ps** file and will then be transferred to TED as you would be with the **Edit/Print** button.

Customizing with PROFILE.TPL

The TPL TABLES installation process creates a file called **PROFILE.TPL** and puts it in the TPL TABLES system directory. PROFILE.TPL is a text file that you can edit. It contains FORMAT statements that become defaults for such things as fonts and paper size. You can change these defaults and also add other FORMAT statements to set additional defaults. For example, if you always want your tables left-adjusted on the page, you can make it the default by including the FORMAT statement ALIGN TABLE LEFT; in PROFILE.TPL.

If you want to leave the system profile unchanged, but use a different profile for a particular set of jobs, you can make a copy of PROFILE.TPL in the directory where you are working and change that copy to fit the tables

you are preparing. The profile in the directory where you are working will override the one in the TPL TABLES system directory.

Encapsulated PostScript (EPS)

Most desktop publishing software that allows you to add PostScript files to a document requires that these files follow certain conventions. Files that follow these conventions are called Encapsulated PostScript (EPS) files. The **EPS** files created by TPL TABLES can be incorporated into your document according to your desktop publishing software's rules for bringing in Encapsulated PostScript files. EPS files have the file extension **.eps**.

There are three ways of converting PostScript tables to EPS.

1. If a **tables.ps** file is open in TED, you can export EPS interactively. See TED Help for details.

2. EPS files can be exported using TED arguments in scripts as described both in TPL Help and in the Scripts appendix.

3. The ENCAPS program provides a third way.

ENCAPS

ENCAPS is a stand-alone command line program that is installed in the TPL TABLES system directory. To run it, change into the TPL subdirectory that contains the tables.ps file you wish to convert. Assuming that TPL TABLES in installed in C:\QQQ\TABLE, give the command:

```
C:\QQQ\TABLE\WTPL\ENCAPS  .  0  TABLES.PS  <Enter>
```

Your tables will then be divided into pages and an **EPS** file will be created for each page. These files will be saved in the TPL subdirectory and will be named according to page and table number. For example, if you have a two page table followed by a one page table, the table output will be divided into three files with the following names:

```
P1T1.EPS
P2T1.EPS
P3T2.EPS
```

ENCAPS will report the names of the **EPS** files as they are created.

Other options are available, such as naming the directory for the output instead of specifying '.' for the current directory and running silently with

no reporting (1 instead of 0). The current options will be displayed on the screen if you type:

```
C:\QQQ\TABLE\WTPL\ENCAPS  <Enter>
```

Note If you have more than one table on a page, they will all be contained in the same **EPS** file.

Other Export Formats

When TPL Tables is run, it supports many different export formats. One of the new ones is a text table format very similar to the format produced when PostScript = NO; is specified.

There are two ways to produce exported tables.

1. You can export the files interactively from TED after doing a table run or a rerun using the Run menus. See TED Help for details.

2. You can export the files using TED arguments in scripts as described both in TPL Help and in the Scripts appendix.

Common Error and Warning Messages

Error and warning messages are intended to be self-explanatory. However, a few common messages deserve special note.

Syntax error message

```
*** ERROR: A syntax error was discovered while processing  'element'.
    Look for the error at or before that point.
```

This message appears whenever there is a syntax error in a codebook, table request, format request or profile. Examples of syntax errors are misspelled keywords or punctuation errors such as a missing colon (**:**) or semicolon (**;**). The point at which TPL TABLES discovered the error is indicated by the element in quotes.

Example The following sequence in a table request will produce the message shown below.

```
TABLE ONE 'Average Income by Region'
        HEADING REGION,

*** ERROR: A syntax error was discovered while processing 'HEADING'.
    Look for the error at or before that point.
```

Since the error was found when the word HEADING was encountered, we can assume that there is something wrong with the word HEADING, or that an error preceded the word HEADING so that it appears to be in the wrong place. In this example, a colon (:) is missing following the table title. TPL TABLES is looking for the colon when it finds the word HEADING.

Undefined variable error message

```
*** ERROR: The variable 'variable name' is undefined.
```

A frequent cause of this error is a misspelled name. Another cause is a reference to a variable that has not yet been defined. For example, if a variable is computed in a COMPUTE statement and used in a TABLE statement that precedes the COMPUTE statement, the computed variable is unknown to TPL TABLES when it finds it in the TABLE statement.

Example Misspelling of the variable name INCOME as INCOM produces the message shown below.

```
POST COMPUTE AVG_INCOME = INCOM / PERSONS;

*** ERROR: The variable 'INCOM' is undefined.
```

Narrow column warning message

```
*** WARNING:  Some columns in your tables are too narrow to hold
your table cells.  See the output file for details.
```

When TPL TABLES is formatting a table, if a data value is too wide to fit in the column, it will be replaced with the built-in **NO_FIT** footnote, making it obvious that the value does not fit. However, TPL TABLES first attempts to display the value by removing mask items such as commas, percent signs and footnote symbols and displays the value without these

items. This warning message will alert you to the fact that one or more values are missing some mask items.

If you get this message, it will be at the end of the file called **output**. You can then search for other instances of *** WARNING in the layout section of the **output** to get more detailed information about where values had items removed. For example,

> *** WARNING: For table 1, page 1, column 1 is too narrow to hold some data cells.

Specifying Extra Memory

For certain types of large jobs, you may be able to improve performance by increasing cell memory space with the CELL MEMORY statement. The statement is in PROFILE.TPL and is described in the Format chapter.

In earlier versions of TPL Tables, the LABEL MEMORY statement could be used to control the amount of work space available for certain kinds of labels. This statement in no longer needed or used by TPL Tables due to improvements in memory management. The statement is ignored if present.

Networks

Licensing Note

If you are accessing TPL TABLES on a PC connected to a network, you must have a license to use TPL TABLES on your PC.

APPENDIX C

Scripts (Windows)

RUNNING BATCH JOBS WITH TPL SCRIPTS

Introduction

TPL jobs can be run from the character mode command line, the **Run** command found in **Start**, or a batch file. It is also possible to create a script which allows multiple TPL and non-TPL jobs to run without your intervention. To start TPL in one of these ways, run the program **WTPL**.

To run an individual job, you just type the command with all required command-line arguments. If a required argument is missing, menus will open prompting you for the missing data. At the end of the job, all menus will close and the job will terminate. For example, assuming that TPL TABLES is installed in c:\qqq\table, suppose you type in the Run menu:

```
c:\qqq\table\wtpl codebook -p c:\qqq\table\examples -c cps.cbk
```

WTPL will process the **cps.cbk** source found in the examples directory. If you omitted the **-c cps.cbk**, a menu would prompt you for the name of the codebook to be processed.

If you wish to run a collection of jobs, you can start WTPL with a file name containing a script consisting of a list of the commands you wish to execute. The scripts may include substitution arguments. The values of these are placed on the command line. Again if you omit required arguments and the job is run in foreground, the system will prompt you. When one command has completed, the next in the script will execute without calling TED or prompting for TED. This will continue until the script is exhausted.

Notes

The command arguments are case-sensitive.

Exactly one command and its arguments can appear on each line of a script. There is no way to continue a line and you cannot put multiple commands on a line.

A line that is completely blank will terminate the script so that nothing following the blank line will be executed.

There is no facility for conditional execution.

If a job is run in foreground, an error in a request will put you in TED. When you have corrected the error the script processing will resume. An error in the script itself will usually result in that portion of the script being skipped. If you just omit an argument or enter an incorrect one, you will usually be put in a menu to fill in the missing information.

Paths and arguments including blanks are supported but such items must be in quotes. ***They must be double, not single, quotes.*** For example using the Call command described below you might have:

```
CALL "my programs\program.exe" 1 xxx "new arg"
```

WTPL has a startup directory which may be changed from within the program using the **Job Directory** option under **File** and saved using the **Save Job Directory** option under **Preferences**. However, if you do not run all of your TPL jobs from the same directory, it is easier to include a **CHDIR** command as the first entry in each of your scripts. This will make it unnecessary to include full path names for all files referenced in your scripts.

Files and directories may use absolute (full) paths or relative paths. Paths are relative to the most recent CHDIR command. For example:

```
CHDIR c:\qqq\table\examples
TED -Pp tpl1\tables.ps
```

will print the same file as

```
TED -Pp c:\qqq\table\examples\tpl1\tables.ps
```

Job Script Example

The following sample script is contained in a file called **sample.lst** in the **examples** subdirectory of the TPL system directory. It runs several jobs, re-using the **TPL1** subdirectory after copying the PostScript **tables.ps** to another location. Note that all of the job files are in the same directory. Starting the script with a **chdir** to that location means that full path names are not required for job files. The TPL system is assumed to be located in **c:\qqq\table**. If you want to try running this script and your TPL system is located in a different directory, you will need to edit the **chdir** line.

Start the script from the command line or using the **Run** option of **Start** by entering:

```
c:\qqq\table\wtpl -A c:\qqq\table\examples\sample.lst
```

The **sample.lst** file is:

```
chdir c:\qqq\table\examples
mkdir mytables
codebook -c cps.cbk
table -r cps1.req -d cps.dat -f cps1.fmt -O 1
copy tpl1\tables.ps mytables\cps1.ps
table -r cps2.req -d cps.dat -f cps2.fmt -O 1
copy tpl1\tables.ps mytables\cps2.ps
table -r cps3.req -d cps.dat -f cps3.fmt -O 1
copy tpl1\tables.ps mytables\cps3.ps
codebook -c police.cbk
table -r police1.req -d police.dat -f police1.fmt -O 1
copy tpl1\tables.ps mytables\police1.ps
```

Wild Cards (* and ?) in TED, COPY, and DELETE Commands

File name arguments in TED, COPY, and DELETE can include the * and ? wild cards.

The * wild card can take the place of 0 or more characters. For example, if PostScript tables have been collected in a single directory with different names followed by the extension **.ps**, they can all be printed by the following TED command.

```
TED -pP *.ps
```

The ? wild card can take the place of exactly one character. For example, if a table has been converted to multiple EPS table files, one for each page, the command:

```
TED -pP P?T1.eps
```

will print the files P1T1.eps, P2T1.eps, ... P9T1.eps. It will not however print files with names such as P10T1.eps or P25T1.eps, because a match would require more than one character.

The same wild cards can be used to copy files to another directory and to delete files. In the following example, all .eps files are copied from the current directory to the directory e:\my_eps_files. Then the .eps files are deleted from the current directory.

```
COPY *.eps e:\my_eps_files
DELETE *.eps
```

Running a Script in Foreground or Background

If the line invoking a script begins with **-A** the script is run in foreground. If the line begins with **-B**, the script is run in background.

A script run in foreground shows the progress of the steps of the script. For example, as the data is read, the hourglass shows its progress. If a required argument is omitted or is incorrect, the system displays a prompt for the argument and processing is stopped until the argument is provided. If a request error is detected, you are put into TED for editing just as if your were running the job interactively.

A script run in background behaves quite differently. There is no activity shown on the screen except for an icon at the bottom. The icon changes to reflect the approximate percent of the script completed. If any error occurs, the script is terminated. It is recommended that you use a Script Log for background scripts.

Script Log

A Script log is a brief listing of the results of the steps of a script. It is created when it is specified on the command line for the script. The script log specification is an optional parameter **-G** followed by the name you choose for the Script log file. If it appears, it must follow the **-B script-name** or **-A script-name** and must precede any substitution arguments.

The resulting log has 1 line for each Table, Codebook, TED, Report, or Rerun step in a script. The first word in each line is one of the following:

```
SUCCESS:
FAIL:
WARNING:
ERROR:
```

The line will also contain the name of the program being run and the script line number. If the line of the script fails, the log will contain either an error message or the comment to look in the output file for more details.

The script log is useful for debugging scripts. It is also useful for programs which include TPL scripts. You can check the success of a job by reading the first character of each line of the script. If all lines begin with **S** or **W**, then the TPL jobs in the script executed successfully.

A **WARNING** line can appear when a format request has warnings. These can usually be ignored. The **output** file messages associated with these format request warnings are preceded by ***** NOTE**.

Some other things that can cause a **WARNING** are: a table request references condition values that do not exist in the codebook; the layout step of a job removes part of one or more data values because they were too wide for the column; data errors are found when the data is being read; or the job runs successfully to the layout step, but there is no data for the table(s), for example because a Select or Define statement caused no data to be selected for the table(s).

Example

```
c:\qqq\table\wtpl.exe -B c:\myfiles\myscript.lst -G c:\myfiles\mylogfile
```

Substitutions in Scripts

A list of one or more substitution arguments may be added to the **-A** or **-B** command line following the script name. These replace the items in the script referenced by **%1**, **%2**, etc. For example, if the command line is:

```
wtpl -A sample.lst T1 F1
```

and the first line of **sample.lst** is:

```
table -r %1.req -d cps.dat -f %2.fmt -O 1
```

The result is:

```
table -r T1.req -d cps.dat -f F1.fmt -O 1
```

Example Using Substitution Arguments

The script in the earlier example can be modified to use substitution arguments. The new command line is:

```
c:\qqq\table\wtpl -A c:\qqq\table\examples\sample.lst cps1 cps2 cps3
```

The substitution arguments follow the script name and are referenced in order as %1, %2, etc.

The new script is:

```
chdir c:\qqq\table\examples
mkdir mytables
codebook -c cps.cbk
table -r %1.req -d cps.dat -f %1.fmt -O 1
copy tpl1\tables.ps mytables\%1.ps
table -r %2.req -d cps.dat -f %2.fmt -O 1
copy tpl1\tables.ps mytables\%2.ps
table -r %3.req -d cps.dat -f %3.fmt -O 1
copy tpl1\tables.ps mytables\%3.ps
codebook -c police.cbk
table -r police1.req -d police.dat -f police1.fmt -O 1
copy tpl1\tables.ps mytables\police1.ps
```

Commands and Arguments

WTPL Arguments for Starting Scripts

```
-A script-name [run script in foreground]
-A script-name substitution-argument-1 substitution-argument-2...
-A script-name -G log-file substitution-argument-1
      substitution-argument-2...
-B script-name  [run script in background (as icon)]
-B script-name substitution-argument-1 substitution-argument-2 ...
-B script-name -G logfile substitution-argument-1
      substitution-argument-2 ...
```

Script Commands and Arguments

TABLE (or table, tables)

-p working-directory
working-directory is either the path to the directory where you want the job to run or the word DEFAULT to indicate the current job directory. A period (.) can be used in place of the word DEFAULT. **-p** is most useful for submitting a request for a single job on the command line. For a command in a script, chdir is more convenient.

-r request [REQUIRED]

-d data-file [REQUIRED except as noted below]
Instead of using the **-d** argument to specify a data file, you may use **-l** with the name of a file whose contents is a list of data files or you may use ODBC Database arguments.

-l file-list
Note: File lists are described in the "Data" chapter.

-f format-request

-O old-run-directory
Use subdirectory ***nnnnn*** and overlay its contents if it already exists. You can use **TPL*nnnn*** if you wish.

-N new-run-directory
Create a new sudirectory with the number ***nnnnn*** only if there is not already a subdirectory in your current directory with the name **TPL*nnnn***. If such a subdirectory already exists, TPL TABLES will not use it but will instead create a new subdirectory with a random number. You can use ***nnnn*** or **TPL*nnnn.***

-C codebook-name
Codebook-name must include **.k** The table request must have a **USE** statement but the codebook name on the use statement is ignored. This feature is useful when you are creating a table from multiple data files with different formats.

ODBC Database Arguments

If you have the TPL-SQL interface for ODBC, you can use the following arguments with the TPL commands: TABLE, CODEBOOK, RERUN, and RMTPL. Normally they would only be used with TABLE or CODEBOOK. For more information, see the section on Arguments for ODBC.

-q [Use **-q** or **-Q** instead of **-d** when using an ODBC Data Source. If -q is used, TPL TABLES may prompt for the ODBC Data Source.]

-Q ODBC-datasource-name

-U database-user

-P database-password

CODEBOOK (or codebook)

-p working-directory

working-directory is either the path to the directory where you want the job to run or the word DEFAULT to indicate the current job directory. A period (.) can be used in place of the word DEFAULT. **-p** is most useful for submitting a request for a single job on the command line. For a command in a script, CHDIR is more convenient.

-c codebook-source [REQUIRED]

codebook-source is the name of the codebook source file.

CBUILDER (or cbuilder) - ODBC Databases only

This command lets you call Codebook Builder from a script to update condition value lists if your database has changed since you last created the codebook. New values are added at the ends of the condition value lists.

-u [REQUIRED]

-K codebook-object.K [REQUIRED]

Provide the complete name of the codebook object ("old" processed codebook) to be used as input. The **.K** extension must be included in the name.

-c updated-codebook-source [REQUIRED]

-Q ODBC-datasource-name [REQUIRED]

If you need a user name and password to access your database, you can add the following arguments to avoid being prompted for the information.

-U database-user

-P database-password

CBUILDER does not have an argument for working-directory. Thus, if you are using this command as a stand-alone, you may need to include full path information for the codebook names. If you are using the command in a script, you can precede it with a CHDIR command to get to the directory where you want to update the codebook.

Example

```
CHDIR f:\myjobs
CBUILDER -u -K survey.K -c survey_new.cbk -Q "Survey Data"
```

RERUN (or rerun)

-p working-directory

working-directory is either the path to the directory where you want the job to run or the word DEFAULT to indicate the current job directory. A period (.) can be used in place of the word DEFAULT. **-p** is most useful for submitting a request for a single job

on the command line. For a command in a script, chdir is more convenient.

-w rerun-tplnnnn-directory [nnnn or TPLnnnn] [REQUIRED]

-f format-request

RMTPL (or rmtpl)

-p working-directory

-X jobs-to-delete [nnnn or TPLnnnn or full path] [REQUIRED]

-X ALL (may be used instead of the above)

CALL command-and-args

CALL can take any executable including "built-ins" such as dir and copy. It supports redirections > | and <.

CHDIR path

CHDIR supports changing to any existing path including ones on different drives. If most of your job files are in the same directory, you will probably want to include a **CHDIR** command as the first entry in your script so that you do not have to provide full path names for all files referenced in the script.

MKDIR path

In TPL scripts MKDIR can make a path more than 1 segment at a time.

MOVE old-name new-name

MOVE allows you to move a file from one directory to another.

REM any-text [no action performed]

REM can be used to add comment lines to a script.

COPY

COPY from-file to-file

COPY from-file(s) to-directory

In COPY file(s) to directory, the from-file(s) argument can include the * and ? wild cards. Wild cards are explained elsewhere in this appendix.

DELETE file(s)

DELETE arguments can include the * and ? wild cards. Wild cards are explained elsewhere in this appendix.

TPLDIR reference-name

TPLDIR is described in the TPLDIR section of this appendix.

ETED

ETED is identical to TED below except that the script stops at the Ted step so you can do custom modifications to your tables. When you close TED, your script continues.

TED

TED arguments can include the * and ? wild cards. Wild cards are explained elsewhere in this appendix.

-e text-file-to-review
-p Postscript-file-to-review
-eP text-file-to-print
-pP Postscript-file(s)-to-print
-pE Postscript-file-to-convert-to-eps
Note that if you are converting tables from multiple runs into eps files, you must also use **-N *Export-core-name*** to avoid overlaying eps files with duplicate named files.
-pF Postscript-file-to-convert-to-PDF
The entire file is converted and placed in the same directory as the source with the same name except **ps** is changed to **pdf**.
-pC Postscript-file-to-convert-to-CSV
-pC can be followed by a divider character to be used in place of comma to separate the values in the exported file(s). If you want to use a blank, enclose the entire argument in quotes: "-pC ". Note: Tab cannot be specified in a script. If you are exporting interactively from TED, you can select Tab as the divider. You can also use the CSV DIVIDE format statement to specify the divide character that will be used for Unix or Windows.
-pA Postscript-file-to-convert-to-PC-AXIS
-pH Postscript-file-to-convert-to-HTML
-pO Postscript-file-to-convert-to-ODS
-pX Postscript-file-to-convert-to-XLS
-pT Postscript-file-to-convert-to-Text-table
-pD Postscript-file-to-convert-to-data-table
-D Export-directory (See below for details)
-N Export-core-name (See below for details)
-C PC-Axis-Contents-name (See below for details);
-F PDF-properties (See below for details)

Notes on Exporting

When TED converts a PostScript file to other formats, the PostScript file must be in the TPLnnnn subdirectory where it was created. TED uses other files in the subdirectory to do the conversion and will not be able to find them if the PostScript file has been moved to a different location.

Also, the TED command should always reference a PostScript (.ps) file, not an Encapsulated PostScript (**.eps**) file. If you reference a **.eps** file,

there will only be one page of converted output created and it will always be the first page of the output no matter which page of **.eps** is referenced.

The **-e** and **-p** arguments will stop the processing stream to enable you to review the tables and output file. You may run several TPL TABLES jobs and then review all of the tables and output at once using TED with multiple **-e** and **-p** arguments. If you use **-eP** or **-pP**, TED will be invoked and the files will be printed and TED will close without any human action. **-pH**, **-pE** and **-pN** will also do their task without stopping the processing.

Notes on HTML Export

For HTML export, you can place additional options markers after the **H**. There should not be any spaces either between the **H** and the additional options or between the options. You can add as many options as you wish. If the options conflict, later ones will override earlier ones.

- n Include a navigation bar if the request produces multiple pages of output.
- s Place all output into a single file. This turns off the **n** option.
- a Automatically size the "page" to allow the entire table to fit in a single html file (no automatic banking or skipping to a new page because the table is too long).

Autosized and Single File HTML

It is reasonable to use both the **a** (autosize) and **s** (single file) options for the same HTML export. Autosize causes the "paper" to expand so that you do not get page breaks because of too many columns or rows in the table. Page breaks will still occur between wafers and tables or if there are explicit ejects. The single file option does not affect what gets put on each page of the table. It just puts all of the pages together into a single file rather than splitting them across files.

Notes on Data Table Export

For data table output, you can place additional options markers after the D. There should not be any spaces either between the D and the additional options or between the options. You can add as many options as you wish. If the options conflict, later ones will override earlier ones.

- s all of the data from all tables should be place in a single file.
- b The table stub should be retained.
- z leading and trailing blanks in fields should be replaced by zero.

Notes on PDF Properties

In the **Properties** menu of a pdf there is information about the pdf. This command allows you to specify this information. Options are

a author
t title
s subject
k keywords

Each option must have a separate **-F** followed by a space and the value. If the value contains a blank, it must be in quotes. All of the -F options must precede the **-pF.**

Example

```
TED -Fa "Jules Verne" - Ft "20 Thousand Leagues" -Fs submarine
-Fk "Sea, Adventure" -pF
```

Notes on Export to PC-Axis

The script command for the TPL run that creates the PostScript file must immediately precede the TED command that converts the table to PC-Axis format.

PC-Axis Contents Name in Scripts

By default, the Contents name is the label of the observation variable used in the table request or "Count" if no observation variable is used. When you export interactively in Ted, you can change this name with the PC Axis export Options.

To specify a Contents name in a script, place *-C PC-Axis-Contents-name* before the **-pA** export argument. The name must be inside double quotes. The new contents name remains in effect until the end of the script or until there is another -C argument.

Example

```
CHDIR c:\qqq\myjobs
TABLE -r PC_Axis.req -d cps.dat -f PC_Axis.fmt -O 1
TED -C "Households" -pA tpl1\tables.ps
```

Setting the TED Export Directory in Scripts

By default, exported files are placed in the same directory as the source **.ps** file. When you export interactively in TED, you can change this destination. The **-D** argument allows you to change the export directory in a TPL script.

To specify an export directory, place **-D** *export-directory* on a TED line before the **-p** export argument. The new export directory remains in effect

until the end of the script or until there is another **-D** argument. To return to the default behavior specify **-D DEFAULT**.

Export Core Name in Scripts

When PostScript files are exported, they are divided into a number of files equal to the number of table pages. The file names for the exported files consist of three parts: an **export directory**, a **core name**, and an **extension**. For export to HTML, the extension is **.htm**. For Encapsulated PostScript, it is **.eps** and for bit mapped graphics it is **.bmp**.

When files are exported from TED interactively, the default core name is always **Table***n* where *n* is the table page number.

Encapsulated PostScript and HTML can also be exported by TED script commands. When files are exported with a script command, the default core name varies depending on the export type. For HTML, the default core name is **Table***n* where *n* is the table page number. For Encapsulated PostScript, the default core name is **P***n***T***m* where *n* is the page number and *m* is the table number.

To specify a different export core name, place **-N** *core-name* on a **TED** line before the **-p** export argument. The new export core name remains in effect until the end of the script or until there is another **-N** argument. To return to the default behavior specify **-N DEFAULT**.

Example For a table request with 5 table pages and a **tables.ps** file in **TPL2**, the following script will export both .eps and .htm files with a core name of **salary**. The .eps files will be named **salary1.eps, salary2.eps, ..., salary5.eps**. The .htm files will be named **salary1.htm, salary2.htm, ..., salary5.htm**.

```
CHDIR TPL2
TED -N salary -pE tables.ps
TED -pH tables.ps
```

Example For a table request with 5 table pages and a **tables.ps** file in **TPL2**, the following script will export both .eps and .htm files with a core name of **Table**. The .eps files will be named **Table1.eps, Table2.eps, ..., Table5.eps**. The .htm files will be named **Table1.htm, Table2.htm, ..., Table5.htm**.

```
CHDIR TPL2
TED -N Table -pE tables.ps
TED -pH tables.ps
```

Note If you are exporting to a single HTML file, the default core name is **Table.** There is no number appended, so the default HTML file is named **Table.htm**.

Note You can also create Encapsulated Postscript using the ENCAPS command line program described elsewhere in this manual. The ENCAPS program uses the core name **P***n***T***m*.

Note If you have more than one table on a page, they will all be contained in the same **.eps** file.

TPLDIR Script Command

When a table job is run, a TPLnnnnn directory is created. This directory contains the finished tables, the output file and other information needed to modify the tables using TED or Rerun. When operating interactively, you may select a specific TPLnnnnn directory or allow the system to select a unique name. In a script, if you use a specific name, you run the risk that some other job might have used that directory name. If you let the system select the name, you have no way of doing additional things with the directory. For example you can't use TED to print the tables or convert them into HTML since you don't know what the directory name is.

The TPLDIR command solves this problem. TPLDIR creates a unique TPLnnnnn subdirectory in the currently active directory and associates it with a user-selected *reference-name*. The script can then reference the directory by using *%reference-name*.

Example

```
CHDIR c:\test
TPLDIR cpsjob
TPLDIR dispatchjob
TABLE -r cps.req -d cps.dat -f  cps.fmt -O %cpsjob
TABLE -r dispatch.req -d dispatch.dat -f dispatch.fmt -O %dispatchjob
TED -pP c:\test\%cpsjob\tables.ps
TED -pH c:\test\%dispatchjob\tables.ps
```

This script will run the cps and dispatch table requests and will print the tables produced by the cps job and convert the dispatch tables into HTML.

Note that in the table jobs we used **-O** for old directory rather than **-N** for new directory since the directories were actually created by the TPLDIR command. Also note that the CHDIR command occurs before the TPLDIR commands. Otherwise the directories created by TPLDIR might be in the wrong place.

Arguments for ODBC

If you have the TPL-SQL interface for ODBC, you can use the following arguments to access ODBC Data Sources from scripts.

```
-q    [If -q is used, TPL TABLES prompts for the ODBC Data Source.]
-Q    ODBC-datasource-name
-U    database-user
-P    database-password
```

Arguments which have blanks or special characters must be put in quotes. They must be double, not single, quotes.

Depending on your environment, you may or may not be required to provide a user name and password to access the Data Source. If you do not wish to include a database user name and password in your script, you may use substitution arguments for these parameters and then provide the user name and password when you run the script.

If you provide all required arguments, you can run your request without being prompted for any information about your ODBC Data Source.

The -q argument can be used if you wish to continue with the same database. You can enter a new -Q and other arguments if you wish to change databases.

Example

In the following sample script, a codebook will be processed for the ODBC Data Source named "My datasource", a table request will be run using the data from the same Data Source, and a second table request will be run using data from a different Data Source. No prompts will be needed for Data Source.

```
CODEBOOK  -c my_db.cbk -Q "My datasource"  -P xxx  -U "John Doe"
TABLE -q  -r sample.req
TABLE -Q "my other datasource"  -P yyy  -U sew  -r another.req
```

Notes

For codebook processing, ODBC Data Source arguments are only required if the codebook needs information from the database. For example, if an ODBC codebook is created interactively in Codebook Builder, all required database information will already be included in the codebook source.

APPENDIX D

Installation (UNIX/Linux)

HOW TO INSTALL TPL TABLES UNDER UNIX

How to Stop

You can stop the **setup** procedure by entering **<Ctrl>C.**

Before You Start

The TPL TABLES installation process copies TPL TABLES to your hard disk. It also asks you about certain characteristics of your operating environment, such as printer, so that it can set defaults for system operation. We recommend that you scan through the following instructions before you start, so that you will know in advance how you want to answer the installation questions.

If you wish to move the TPL TABLES system to another location in your file system after it is installed, you must remove it from the original location and reinstall it. Merely copying the files will not work correctly. If you have customized your TPL TABLES **profile.tpl**, **color.tpl** or **country.tpl** files, you may wish to save them for use in the new location before removing TPL TABLES from the previous location.

Installation Steps

The exact installation procedure depends upon the platform on which you are installing TPL. Specific directions can be found in the README file in the directory where you found the software.

So that users do not have to start tpl using full paths or modify their **.profile** (or equivalent) PATH statements, you may wish to use the **ln** command to link some TPL programs into directories that are already in their paths; e.g. **/usr/bin**. You can link just **tpl**. If you do this the users will need to prepend **tpl** to each of their programs; e.g. **tpl rerun;** Alternately you can link each of the following which will make it unnecessary to prepend **tpl**.

The programs which can be usefully linked are:

tpl
rerun
codebook
conditions
rmtpl
encaps
psp
report (if you have installed TPL Report)

Detailed Description of Setup Prompts

When you begin the setup program, **setup.tpl**, it displays some introductory information on your screen and begins asking questions about the installation:

> This program sets up the TPL TABLES system after it has already been copied to your selected directory. It customizes your copy of TPL TABLES by allowing you to specify some default parameters. If you wish to move your copy of TPL TABLES you must rerun this program. UNIX mv, cp, or mvdir commands can be used to move the modules, but setup must be rerun for the system to work after it is moved.
>
> Respond to each question prompt "**==>**" with the appropriate file name or value then press the **<ENTER>** key.
>
> You must have write permission for the path which is to receive the TPL TABLES system. If you do not, you must terminate this session and install the system using an appropriate id.
>
> The following questions are used to set system default values for the **profile.tpl** file. If there are multiple users on your system or you wish to use different defaults for different data files, you may make modified copies of the **profile.tpl** file for different directories.

Prompt:

Do you wish to install TPL TABLES?

Response:

y for yes or **n** for no.

Prompt:

Do you wish to install TPL REPORT?

Response:

y for yes or **n** for no.

Where Do You Want the System Installed?

Prompt:

Please specify the path of the directory which is to RECEIVE the TPL TABLES system. Relative path specifications may be used.

Response:

Enter the path as directed in the prompt. If the directory does not exist, you will be asked if you want the system to create it.

Prompt:

Please specify the path for the directory containing the modules which are to be installed.

Response:

Enter the location of the TPL software modules you wish to install as directed in the prompt. Note that relative paths including "." are supported.

Table Viewer

TPL TABLE output is valid PostScript. Most versions of Linux and Unix have utilities for viewing PostScript output. By specifying one of these programs, you can view your tables before printing them or exporting them.

Prompt:

Many versions of Unix and Linux have Postscript page viewers such as Solaris **pageview** and Linux (KDE) **kghostview** or okular or Linux (GNOME) evince. If you have such a program, please specify its name (and path if necessary);

Response:

Enter the name of your PostScript viewer program (and path if necessary).
For Solaris a recommended program is **pageview.**
For Linux using the KDE environment, **kghostscript** is recommeded.
For Linus using the GNOME environment, **evince** is recommended.

Paper Size

TPL TABLES will automatically format your tables according to the paper size you specify in answer to the next prompts. Your answers will depend on the type of printer, size of paper and the type style you wish to use.

Prompt:
You may specify your standard paper size either by picking one of the following or by picking **NONE** and then specifying a length and width when prompted.

Select one of:
LETTER (8.5 inches by 11 inches)
LEGAL (8.5 inches by 14 inches)
A3 (42.0 cm by 29.7 cm)
A4 (21.0 cm by 29.7 cm)
B5 (18.2 cm by 25.7 cm)
NONE

Response:
You can choose one of the standard page types by entering its name, for example **letter**.

If you enter **none** and are prompted for length and width, you can specify them in inches, centimeters, points or characters. Fractions should be expressed as decimal numbers. For example, a page width of 8 1/2 inches should be entered as 8.5 inches.

It is best to express page size in something other than characters. This is because you can choose different character sizes. If page size is expressed in characters, the size of the page will vary as the character size changes. This result is usually undesirable.

Editor

TPL TABLES has been designed so that you can use the text editor of your choice to create codebooks, table requests, and format requests. Any editor that creates standalone text files is acceptable.

Prompt:

If a TPL TABLES job fails because of a request error, the job will be put into the selected editor. When editing is completed and the editor terminated, TPL TABLES processing will resume. The default editor is the UNIX editor, **vi**.

The editor selected must be such that it can be invoked by:

editor-name dataset

where *editor-name* is the name of the editor and dataset is the name of the file to be edited.

Please type:

<ENTER> if you wish to use the currently selected or default editor,
none <ENTER> if you do not wish to use an editor
editor-name <ENTER> if you wish to select an editor.

Response:

If you have an editor other than **vi** on your system, you may wish to enter its name. However do not use a word processor which inserts formatting information into your file unless there is an option to save the file in "text only" mode.

If You Change Your Mind

The installation process will now give you the option to change any of your answers to the questions you have been asked. First it displays the options you have already chosen. For example:

Prompt:

The current values which you have set are:
Editor = 'vi'
Postscript printer
Paper type = LETTER

```
Do you wish to change any of these values?
```

Response:

If you are satisfied with your choices, enter **y**. If not enter **n** for no. Even if you respond with no to this prompt you will still be able to change the effect of your responses by editing **profile.tpl** after installation is complete.

Completion of Installation

Installation will continue. If you have indicated at the beginning that you also wish to install TPL REPORT, there will be additional prompts similar to those already described. The installation process will tell you when it is finished.

If You Have Multiple Printers Connected to Your Computer

TPL TABLES will direct its output to the default printer for your computer. If you wish to change this, you may modify the profile statement

```
Print  Command = 'lp';
```

For example you might replace the command with

```
Print Command = "lp -dpost";
```

where ***post*** is the name of your PostScript printer. Note that if different people wish to use different printers they should create local profiles with different print commands.

APPENDIX E

Run Instructions (UNIX/Linux)

INSTRUCTIONS FOR RUNNING TPL TABLES UNDER UNIX

General Information

Editor

TPL TABLES is designed to allow you to use **vi** or another editor of your choice to create codebooks, table requests and format requests. Any editor that creates standalone UNIX files is acceptable.

If you have installed TPL TABLES so that it can access your editor and you are running a job that stops because of errors, TPL TABLES will prompt you to find out if you want to transfer to the editor. If you are transferred to the editor, TPL TABLES will automatically resume processing when you are finished with your editing.

Where to Run Jobs: Paths and Files

We do not recommend that you mix your request files with system files by putting your own TPL-related files in the TPL system directory. Instead, put your own files in one or more other directories and run your jobs from those directories. If you cannot invoke **tpl** from your command line without providing a path, you may wish to add the path to tpl tp the path command in **.profile** or a different unix/linux startup script file. It is a good idea to run your TPL TABLES jobs in the directory where your TPL-related files are stored, because then you can simply provide the file names without including path information. For any of your files that are not in the directory where you are running a job, you may include the path information.

How to Stop

The easiest way to stop a TPL job in the middle of processing is to type:

<Ctrl>C

If this doesn't work, open a new window and type:

ps -A

Then type:

kill -9 *pid*

where ***pid*** is the process id associated with the TPL process.

Note on Running in Background

All processes can be run in background, with the exception of **rmtpl.** The prompt for background processing and the **-b** argument are described under **How to Run a Table request**.

Codebook Processing

Prepare your codebook (data description) file using your editor. We recommend that you save it with the same name you use at the beginning of the codebook. For example, if you name the codebook **survey** with the codebook statement **begin survey codebook**, save your codebook file as **survey.cbk**. The codebook file you have prepared will be referred to as the codebook source.

Note

If you have a partial codebook source that needs to be completed with information from the data or if your data has changed such that new condition values need to be added for control variables, run TPL **conditions** first to create a complete or updated codebook source.

How to Run *codebook*

To run the codebook processor, type:

codebook <Enter> (or tpl codebook <Enter>)

The codebook processor will display the prompt:

```
Please type the name of your codebook request  and <Enter>

==>
```

If you have a codebook source named **survey.cbk**, as in the example above, you would type:

```
survey.cbk  <Enter>
```

Codebook Command Line Arguments

You can bypass the prompt for the codebook source name by entering your codebook command as:

```
tpl  codebook  -c  cbsource  <Enter>
```

where ***cbsource*** is the name of your codebook source.

Error Handling

As the codebook processor runs, it will display your codebook on the screen along with messages about any errors it finds. All information displayed on the screen during processing will be stored with the same name as the codebook except that it will be capitalized and **.O** will be appended to the name. If your codebook is named **survey**, the processing information will be stored in a file called **SURVEY.O.**

If the codebook processor finds errors in your codebook, you will need to correct them with your editor and process the codebook again. If any syntax errors are found in the codebook, processing will stop. For most other types of errors, processing will continue to the end of the codebook. In that case, you will probably want to look for the error messages in the file containing processing information (e.g. **SURVEY.O**).

When your codebook has been processed successfully with no errors, the **.O** file will be deleted and the **.L** codebook abstract file will take its place.

Codebook Abstract

The codebook abstract name ends with **.L** (e.g. **SURVEY.L**). The abstract includes the name of the codebook source file, the date and time of processing, and the TPL version number. In addition, it contains a list of the codebook variables in alphabetical order along with each variable's size and location within a record. This information is particularly useful if you have an alignment problem between your codebook and your data file. You may also find the abstract useful as a quick reference when preparing your table specifications. If you are creating a codebook describing a **CSV** or other type of delimited file or a database, the information in the abstract will differ slightly.

Codebook Object

We will refer to the processed codebook as the codebook object. When you have successfully run the codebook processor, your codebook object will be stored with the same name as the codebook except that it will be capitalized and **.K** will be appended to the name. Thus, for a codebook named **survey**, the codebook object will be produced with the name **SURVEY.K**.

Once your codebook is successfully processed, you can run any number of table jobs using the same codebook object.

Producing a Codebook Source with the *conditions* Procedure

If you do not already have a codebook source, TPL **conditions** can be used to create a full codebook source from a partial one. It can also be used to update a codebook source if the data has changed such that new condition values need to be added for control variables.

Prepare your partial codebook with your editor as described in the Appendix called "TPL Conditions".

How to Run a *conditions* Request

To run a **conditions** job, type:

```
conditions <enter> (or tpl conditions <enter>)
```

The program will prompt you for your partial codebook source (with missing conditions).

It will then prompt you for your data file or database name. If the program cannot find the name it will ask whether the name is a SQL database name. Answer **y** or **n** as appropriate. If you answer **n**, you will be re-prompted for the data file name.

Finally you will be asked for the name of the completed codebook source you wish to create. You can use the same file name for your original source and your completed source. If you do, the completed source will be placed on top of the original source. The original source will be saved, along with a few extra statements, in the **.O** output file until the completed source has been created successfully. Thus, if there are any errors or problems that interrupt the creation of the completed source, you do not risk losing your original source. It will still be available in the **.O** file.

You will then be asked if you want to run the job in background.

If your codebook describes a database, you will be prompted for database user name, password, database server, etc.

The resulting complete codebook source file can be passed directly into a TPL codebook run or it can be edited to improve condition labels before codebook processing.

Command Line arguments for *conditions*

-c	*incomplete-codebook-source*
-s	*complete-codebook-source*
-d	*data-file* (if your data is fixed format or delimited, e.g. csv)
-q	*database* (if your data is in *database*)
-Q	(denotes data is in database - Oracle only)
-U	*user* (SQL only) or *user@connect-identifier* (Oracle only)
-S	*database-server* (Sybase only)
-P	*database-password* (SQL only - password may need quotes)
-b	to run job in background

Error Handling

During the first part of TPL **conditions**, error handling is identical to codebook error handling as described above. After the original, incomplete codebook has been found to be valid, the program moves to the data reading step to get the information it needs to complete the codebook. Data errors such as incorrect characters in observation fields are added to the **.O** file. Data errors will not stop processing and will not put you into an editor.

Producing Tables with the *tables* Procedure

Prepare your TPL TABLES table request with your editor. In the **USE** statement at the beginning of a table request, you can refer to the codebook using the same name you used in the **begin** codebookname **codebook** statement. Using the name **survey** shown in the example above, you would say **use survey codebook;** at the beginning of your TPL TABLES request. TPL TABLES will know to look for a codebook object file called **SURVEY.K** for descriptive information about your data file. Path names are allowed in the USE statement.

Store your table request with any valid UNIX file name, for example, **survey.req**. You may also have an optional format request giving detailed specifications for formatting your tables. The format request can have any valid UNIX file name, for example, **survey.fmt**.

How to Run a Table Request

To run TPL TABLES, type

```
tpl  tables  <Enter>
```

TPL TABLES will display the prompt

```
Please type the name of your request and <Enter>:

==>
```

Using the name from the example above, you would type:

```
survey.req   <Enter>
```

TPL TABLES will display the prompt

```
Please type the name of your data file and <Enter>

==>
```

Your data file can have any valid UNIX file name. To continue the "survey" example, we will assume that your data is called **survey.dat**. You would type:

```
survey.dat <Enter>
```

If you are running against a database rather than a file, you should enter the database name. If you have entered a database name or an incorrect file name you will be asked if the name is a SQL database name. If it is, answer **y** and processing will continue with questions about your database user name, password, and server. If you answer **n,** you will be re-prompted for your data file.

TPL TABLES will display the prompt

```
Please type the name of your format request and <Enter>
or just type <Enter> if you do not wish to provide a format
request file:

==>
```

Often you will not have a format request. In this case, simply press the **<Enter>** key to continue. Otherwise, type the name of your format request. For example:

```
survey.fmt   <Enter>
```

Next you will be asked:

```
Do you wish to run this request in background?

y or n ==>
```

Answering **y** to this prompt is the proper way to run TPL TABLES as a background process. Don't just use **&**. When the job is put in background, all output except the tables goes to the **output** file. Nothing is displayed on the screen and you are not put into your editor when errors are found in your request.

```
Do you wish to be notified when the request completes?

y or n =>
```

If you answer **y** to this prompt, when the job completes a message will appear on the screen telling whether the job has completed successfully or whether errors were detected in the request. In any case you should examine the **output** file in the TPL subdirectory. The TPL subdirectory is explained later.

Tables Command Line Arguments

If you wish, you can bypass some or all of the prompts by entering your tables command with any of the following parameters. Note that many of these options will be explained more fully later.

-r *request-file*	where r*equest-file* is the name of your table request file
-f *format-file*	where *format-file* is the name of your format request file
-d *data-file*	where *data-file* is the name of your data file
-b	to run job in background
-n	to notify when job has completed
-E	to request only a partial display of output on the screen when running in foreground. For details, see the section on controlling screen display.
-e	convert tables into Encapsulated PostScript
-h	convert tables into HTML. For additional -h options and details, see the section on HTML table arguments.
-V	convert tables to CSV (delimited) format.
-D	produce a data table. For additional -D options and details see the section on Data Table arguments.
-B	produce pdf output. ghostscript must be installed for this command to work.
-a	produce (ASCII) text table.
-o	produce spreadsheet output (ods - the current spreadsheet standard)
-X	produce spreadsheet output (xls - a format used by older versions of Excel)
-N *nnnnn*	use TPLnnnnn as TPL subdirectory where nnnnn is a user selected number of one to five digits. If there is already a directory of that name, create a new number.
-O *nnnnn*	use TPLnnnnn as a new TPL TABLES subdirectory overwriting any existing subdirectory of that name.
-i *includepath*	where *includepath* is the path to the directory where %include files are located. Use if you have include files in a directory other than the run directory. For details, see the section "Path for INCLUDE files".

-U *user* (SQL only) or *user@connect-identifier* (Oracle only)
-P *database-password* (SQL only - password may need quotes)
-S *database-server* (Sybase only)
-Q (denotes database with no database-name required - Oracle only)
-q *database-name* (Data in SQL database *database-name*)

Example tpl tables -r survey.req -d survey.dat <Enter>

Table Request Processing

As TPL TABLES processes your request, it will display the request on the screen along with messages about any errors and other information to show you the status of the job. If there are any errors in the table or format requests, you will be asked:

```
If you wish to edit your request and continue
respond with 'y' to the prompt. You will then be
put in your editor. Upon termination of your editing
session you will be returned to TPL and processing
will continue. A response of 'n' will terminate the
TPL session
```

If you answer **y**, you will be allowed to correct your errors and processing will continue. If you can't figure out your errors from what is displayed on the screen, you should answer **n** to the prompt and examine the error messages in your **output** file (described later). When you have fixed your errors you should start your table request again. Processing will stop immediately if a syntax error is encountered. For most other errors, processing will continue to the end of the request.

For some operating systems. if no request or format errors are found, TPL TABLES will draw an hour glass on the screen as it begins to read your data. You will be able to tell how much of your data has been processed by the amount of sand that has fallen to the bottom of the hour glass. For other operating systems you will get a changing line to report how much of the data has been processed. If your codebook does not match your data or if there are errors in the data, messages will be displayed at the bottom of the screen.

When TPL TABLES has finished processing your data and calculating the values for your tables, it will report whether the job has completed successfully.

You may examine the **output** file in the TPL subdirectory to review any data errors and determine whether you should print your tables. The **output** and **tables** files are described in the next section.

Example

```
tpl tables -r survey.req -d survey.dat -b -n <Enter>
```

Since **-b** and **-n** have been specified, no output will be displayed on the screen except for the final status of the job. You will not be given the opportunity of correcting errors and continuing processing. Instead you must examine the output file in your TPL subdirectory and resubmit your job if an error is found. If the job has run correctly, you may print your tables.

Controlling the Amount of Screen Display in Foreground

You can use the statements **display output = no;** and/or **display tables = no;** in your profile to reduce the amount of screen display when running in foreground.

You can also use the **-E** command line option with both codebook and table runs. It provides a convenient way of running jobs in foreground, because it lets you see what is happening but reduces the volume of screen display. Display of codebooks or requests is suppressed. If an error is encountered in your codebook or request, the output ends with the error message and the preceding line of your codebook or request. This way, you can often see where the error is without looking at the entire output file. Note that this option will not work if you have the statement **display output = no;** in your profile.

The TPL Subdirectory

Each time you run TPL TABLES, it creates a subdirectory to hold the files it needs to create your tables. The subdirectory always has the name **TPLnnnnn** where **nnnnn** is a number with 1 to 5 digits. The process id is used for the **nnnnn** part of the subdirectory name, unless there is already a subdirectory using that number. You can find these subdirectories with the UNIX command **ls TPL***. If you do not wish to let TPL TABLES select your TPL subdirectory number, you can specify one yourself by using **-O nnnnn** or **-N nnnnn** on your command line. If you use **-N**, TPL TABLES will use **nnnnn** only if there is not already a TPL subdirectory with that number in your current directory. If such a subdirectory already exists, the **-N** argument will be ignored and a new numbered subdirectory will be generated. If **-O nnnnn** is chosen, the new directory will be **TPLnnnnn** regardless of whether there was already one by that name. The old one will just be overwritten.

Most of the files that go into a subdirectory are not intended to be read by you. However, there are two files in the subdirectory that you will want to see. One is called **output** and contains all of the information that was displayed on the screen while your job was running — all except the tables, that is. The completed tables are stored in a file called **tables**. If **Postscript = yes**; was specified, there is no **tables** file but instead there is a file **tables.ps**.

If the messages go by on the screen too fast for you to read while your job is running, you can find them in the **output** file. If you run your job in the background or leave your computer while the job is running, you can find all the information that was displayed on the screen in the **output** file.

To help you keep track of your jobs, the **output** file contains the names of your data and request files, the date and time of execution for each part of the job, the TPL version number, and, at the end, the name of the TPL subdirectory in which it was created.

Printing and Exporting

Note If no tables are created, for example because of data errors or because no data is selected, there will be no prompt for printing or exporting tables.

When a TPL TABLES job ends you will be presented with the following printing options:

```
Please specify the numbers for all of the print and export options you
wish to use:
  1. Print Tables
  2. Print Output

==>
```

If you have run your job in Postscript mode, you will have a larger set of printing and output options. If you have specified a value for **DISPLAY NAME** in your profile, you will first be asked if you wish to display your tables in the PostScript displayer you have specified. If you answer **yes**, the displayer will open your tables in a separate process. TPL TABLES will then continue with the following prompt:

```
Please specify the numbers for all of the print and export options you
wish to use:
  1. Print Tables
  2. Print Output
  3. Export Encapsulated Postscript (eps) files
  4. Export delimited (csv) files
  5. Export internet (HTML) files
  6. Export Spreadsheet (ods) files
  7. Export Spreadsheet(xls - old Excel format) files
  8. Export Text Table (txt)
  9. Export Data Table (dat)
 10. Export PDF (ghostscript must be installed)
==>
```

Select the numbers of the options you wish to use and place them on the prompt line separated by blanks. For example

```
==> 1 3 5
```

will result in your tables being printed and also exported as **EPS** files and **HTML** files.

Export files will be placed in the **TPLnnnnn** subdirectory where the job is run.

If you have selected item 5, export to **HTML**, you will be presented with additional options for specifying how you want your html to look.

```
Please specify the numbers of the html options you wish to use:
  1. HTML with navigation (Can't be used with Single)
  2. Single file HTML
  3. Autosized HTML - page size limits removed
==>
```

Normally, TPL puts each page of your tables into a separate **HTML** file. If option 1 is selected and your table has multiple pages, each **HTML** page will have a navigation bar at the top. Someone who is viewing the table in a web browser can click on the arrows in the navigation bar to move among the pages. If option 2 is selected, the pages of the table are still broken into separate **HTML** tables but they are all placed in the same file. If option 3 is selected, the "paper" size is expanded so that there is no banking of the table caused by it being too wide and there are no page breaks caused by it being too long. Multiple files may still result if there are wafers or multiple TPL tables in the request and there are no format statements to keep the wafers and tables on the same page.

If you have selected item 9, Export Data Table, you will be presented with additional options for how the data should be formatted.

```
Please specify the numbers of the data table options you wish to use:
  1. Retain Stub
  2. Combine all tables into a single file
  3. Zero fill (replace all blanks with zeros)
==>
```

Normally, TPL creates a data table from the cells of a table and creates a separate file for each table. If option 1, Retain Stub, is selected, the stub is prepended to the file. This provides identification for the rows of the file. The meaning of options 2 and 3 is obvious.

Preventing Prompts for Printing and Exporting

The above prompts can be avoided by putting the appropriate options in your **profile.tpl** file or format request and by using some command line options when you submit your job.

If you run your job in background and do not include any print options or export options in your profile, format request, or command line, then you will not be prompted and no printing or exporting will occur. To get printed output, add **print output = yes;** and/or **print tables = yes;** to your **profile.tpl** file or format request. To invoke the various export options, use the command line options discussed above.

To avoid the prompts when running the job in foreground, you must put all of the following in your **profile.tpl** or format request. For each, the *value* should be **yes**, **no**, or **prompt**. If **prompt** is set for any of them or you omit any, you will get the standard prompts. Responses to the prompts will override the profile values.

print output = *value*;
print tables = *value*;

If the job is being run in postscript mode, you must also include:

eps output = *value*;
csv output = *value*;
html output = *value*;
ods output = *value;*
xls output = *value;*
pdf output = *value;*
datatable output = *value;*
text table = *value;*

Final Disposition of Generated Files

When your job completes, the output, tables, and any exported files can be found in the **TPLnnnnn** subdirectory along with a few other files needed if you wish to modify your request with **tpl rerun**. If your job is run in non-PostScript mode, the tables can be found in **tables**. This file may be printed using the standard unix **lp** command. It can also be displayed in an editor or with **more** or **cat** but it may not look quite right since it is formatted for printing rather than display. If your job is run in PostScript mode the tables can be found in **tables.ps**. If you are using a Sun computer, this file can be viewed using the **pageview** program. Other PostScript display programs may also be used. **tables.ps** may be printed directly to a PostScript compatible printer without passing it through a PostScript filter.

Path for INCLUDE files

For tables or codebook runs, if you have %INCLUDE files that are in a directory other than the run directory, you can use the **-i** argument to enter the path to the directory where the %INCLUDE files are located.

For example, if you have an include file called **stubs.txt** that is located in the directory called **/usr3/tplwkgrp/ALB.FILES**, you can use the **-i** argument on the command line as follows:

```
-i  /usr3/tplwkgrp/ALB.FILES
```

Then in your %include statement, use the file name:

```
%include stubs.txt
```

You may only have one include path.

Another way to access include files in another directory is to use the UNIX **ln** command to make the include files appear to be in the local directory.

Encapsulated PostScript (eps)

Many desktop and professional publishing systems allow importation of PostScript files provided they are in **EPS** format. Presentation programs such as Microsoft PowerPoint can also display **EPS** files created with TPL. If you have run your job in PostScript mode and have specified **-e** on your command line or selcted **Export Encapsulated Postscript** at the prompt, then you will have created EPS files in your **TPLnnnnn** directory. If no tables were created, for example because of data errors or because no data was selected, then no EPS files will be created.

If you have not specified creation of EPS files and later decide you need them, the **encaps** program may be used to create them. Change into the **TPLnnnnn** subdirectory and type:

```
encaps  .  0  tables.ps  <Enter>
```

encaps will report the names of the **EPS** files as they are created.

Other options are available, such as naming the directory for the output instead of specifying '.' for the current directory and running silently with no reporting (1 instead of 0). The current options will be displayed on the screen if you type:

```
encaps  <Enter>
```

The **encaps** program works only with **.ps** files created by TPL software. It cannot be used with PostScript files created by other programs.

Tables can be most conveniently imported into another system if each page is in a separate file. Consequently TPL TABLES creates one file for each page of table output. The files are named by page and table number. For example, if you have a two page table followed by a one page table, the table output will be divided into three files with the following names:

```
P1T1.eps
P2T1.eps
P3T2.eps
```

The **tables.ps** file containing the complete table output will still be available for printing or you can print individual pages of your tables by printing the EPS files.

If you have more than one table on a page, they will all be contained in the same **EPS** file.

CSV

Comma Separated Variable (CSV) format is a common data interchange format. TPL can read **CSV** and other types of delimited files as data and can output tables in CSV format for use by other programs. If you have run your job in PostScript mode and have specified **-V** on your command line or selected **Export Delimited** at the prompt, then you will have created **CSV** files in your **TPLnnnnn** directory. If no tables were created, for example because of data errors or because no data was selected, then no **CSV** files will be created.

Each table with data produces a separate **CSV** file in your **TPLnnnnn** directory. The files are labeled **Table1.csv**, **Table2.csv**, etc.

HTML

HTML is the standard language interpreted by Internet browsers to create web pages. If you run your job in PostScript mode and include **-h** on your command line or select **Export Internet** at the prompt, then you will create **HTML** files in your **TPLnnnnn** directory. If no tables were created, for example because of data errors or because no data was selected, then no **HTML** files will be created.

By default, one **HTML** file is created for each page of a table. These are labeled **Table1.htm**, **Table2.htm**, etc. There are several options which affect how many files are created and how they are formatted. These can be specified on the command line as described below or they can be specified though the prompts at the end of a tables run.

HTML Table Arguments

For a **tables** run or **rerun** job, the command line arguments for requesting HTML output are:

-h produce html
-hn produce html with a navigation bar if there are multiple pages of output
-ha produce autosized html; automatically sizes the "page" to allow the entire table to fit in a single html file (no automatic banking or skipping to a new page because the table is too long)
-hs produce html in a single file (this turns off the **-hn** option)

You can request multiple HTML options at the same time, either by entering multiple **-h** arguments or by combining options in one **-h** argument. If you combine options in a single **-h** argument, do not put any spaces between options.

Example To run a tables job and request HTML with navigation and also with built-in footnotes in the same column as the data, you could enter either of the following:

```
tpl tables -r survey.req -d survey.dat -hn -ha <Enter>

tpl tables -r survey.req -d survey.dat -hna <Enter>
```

Note on Autosized and Single File HTML

It is reasonable to use both the **-ha** (autosize) and **-hs** (single file) options for the same HTML export. Autosize causes the "paper" to expand so that you do not get page breaks because of too many columns or rows in the table. Page breaks will still occur between wafers and tables or if there are explicit ejects. The single file option does not affect what gets put on each page of the table. It just puts all of the pages together into a single file rather than splitting them across files.

ODS and XLS

ODS and XLS are both spreadsheet formats which can be read by most spreadsheet programs. ODS is the current standard and is the preferred format. It can be read by most spreadsheet programs including versions of Excel available since 2007. The version of XLS produced by TPL can be read by nearly any spreadsheet program. When it is brought into a recent version of Excel, a warning message appears but if you select **yes**, the file opens and displays correctly.

PDF

PDF is a widely used format for accurately displaying tables and other output. The option currently works in Linux or Unix only if you have installed the free program **ghostscript**. If you wish to use a different **distiller** program (not pdf reader), contact us and we will try to accomodate your request.

TXT

This option produces the same output you would get if you ran your job with Postscript = no; in your profile or format statements. It is really just a convenience to enable you to get both text and other export formats from the same table run.

DAT

DAT, like TXT, is provided as a convenience for people who wish to produce data tables as well as other table formats in the same request. It is equivalent to using the format command **Data Tables;** in your format request or profile.

DAT Table Arguments

For a **tables** run or **rerun** job, the command line arguments for DAT table output are:

-D produce data table
-Ds produce single table
-Dz produce data table with zeros replacing leading and trailing blanks on each data value.
-Db produce data table with the stub retained.

Note s, z, and b may be used together; e.g. -Dzb will produce a data table with with the stub retained and zero-filled data values.

Removing Subdirectories with the *rmtpl* Command

The **rmtpl** command makes it easy for you to erase TPL subdirectories that you no longer want to keep.

How to Run *rmtpl*

To erase a subdirectory, first be sure that you are in the directory that contains the subdirectory. Then type the command

rmtpl *nnnnn* <Enter> (or TPL rmtpl *nnnn*)

where ***nnnnn*** is the number of the subdirectory you want to erase.

You can delete multiple TPL subdirectories by including multiple numbers on the command line. For example,

rmtpl 123 456 78345 <Enter>

To delete all TPL subdirectories contained in the current directory, type:

rmtpl all <Enter>

Note If you also have TPLR subdirectories created by TPL REPORT in the same directory, the command **rmtpl all** will remove these subdirectories as well.

Modifying Tables with the *rerun* Procedure

After running a TPL job, you may see that the appearance of your tables could be improved by changing certain formatting characteristics of the tables. For example, if the numbers in the tables are very large, the default column width may be too small, or maybe you want to change a table title or label. The **rerun** procedure allows you to quickly change the format of your tables by rerunning only the table formatting part of a job.

You specify the changes that you want using the FORMAT language. The FORMAT statements go into a file that you create using your editor. The format file can have any valid UNIX name, for example, **survey.fmt.**

How to Run *rerun*

To use the **rerun** procedure, you will need to know the number of the TPL TABLES subdirectory containing the existing tables. To rerun, type

rerun <Enter> (or tpl rerun <Enter>)

TPL TABLES will display the prompt

Please type the name of your format request and <Enter>
or just type <Enter> if you do not wish to provide a format
request file:

==>

Using the name from the example above, you would type:

survey.fmt <Enter>

TPL TABLES will display the prompt

Please type the name of the TPL working directory and
<Enter>. 'TPL' may be omitted if the path is not included.

==>

Assuming that the tables you want to reformat are in the TPL TABLES subdirectory **TPL9467**, you would type:

9467 <Enter>

Rerun Command Line Arguments

If you wish, you can bypass the prompts by entering your **rerun** command with the following parameters:

-f *format-file*	where *format-file* is the name of your format request file
-w *nnnnn*	where *nnnnn* is the number of the TPL subdirectory you are working with
-e	if PostScript is set, convert tables into Encapsulated PostScript
-h	if PostScript is set, convert tables into HTML. For additional -h options and details, see the section on HTML table arguments.
-V	if PostScript is set, convert tables to CSV (delimited) format.
-D	if PostScript is set, produce a data table. For additional -D options and details see the section on Data Table arguments.
-B	if Postscript is set, produce pdf output. ghostscript must be installed for this command to work.
-a	if Postscript is set, produce (ASCII) text output.
-o	if Postscript is set, produce spreadsheet output (ods - the current spreadsheet standard)
-X	if Postscript is set, produce spreadsheet output (xls - a format used by older versions of Excel)

For example,

```
tpl  rerun  -f  survey.fmt  -w  9467
```

Rerun Processing

The **rerun** procedure will process your FORMAT statements and reformat your tables. The new version of your tables will replace the originals in the tables file of the TPL9467 subdirectory.

If you do a **rerun** and don't like the result of your format changes, you can always get back to the original tables by doing a **rerun** without a format request. Just type <Enter> when you are prompted for the format request file name.

You can reformat a set of tables any number of times without reprocessing the data.

Creating Your Own Environment with the profile.tpl File

The TPL TABLES installation process creates a file called **profile.tpl** and puts it in the TPL TABLES system directory. This file allows TPL TABLES to adjust to your operating environment.

The **profile.tpl** file contains statements that you can change with your editor after installation if something changes in your operating environment. For example, if you begin using a PostScript printer, you might want to edit **profile.tpl**.

You can also change table format defaults by including FORMAT statements in **profile.tpl**. For example, if you always want your tables left-adjusted on the page, you can make it a default by including the FORMAT statement **align table left;** in **profile.tpl**.

If you want to leave the system profile unchanged, but use a different profile for a particular set of jobs, you can make a copy of **profile.tpl** in the directory where you are working and change that copy to fit the tables you are preparing. The profile in the directory where you are working will override the one in the TPL TABLES system directory.

If your copy of TPL TABLES is being shared over a network, you may wish make a copy of **profile.tpl** that is appropriate for the way you want to use TPL TABLES.

Specifying Extra Memory

For certain types of large jobs, you may be able to improve performance by increasing cell memory space with the CELL MEMORY statement. The statement is in PROFILE.TPL and is described in the Format chapter.

In earlier versions of TPL Tables, the LABEL MEMORY statement could be used to control the amount of work space available for certain kinds of labels. This statement in no longer needed or used by TPL Tables due to improvements in memory management. The statement is ignored if present.

Piping Data to TPL TABLES

TPL TABLES supports standard piping of data into a request and also supports the more flexible named pipes.

Standard Piping

Standard piping is done by using just the standard 'I' symbol plus the TPL TABLES keyword **%pipe.**

An example is:

```
cat  datafile  I  tpl  tables  -r request  -d  %pipe
```

The piped input may of course come from the output of any program which writes to the standard output (console). The hourglass is not shown while data is being read.

With this type of piping, TPL TABLES reads the piped data as if it were coming from the standard input (the keyboard). Thus, the following rules apply:

1. Both the **-r** and **-d** arguments must be included and be correct or the job will fail to execute.
2. TPL TABLES will not prompt you for missing or incorrect arguments. Since the standard input (keyboard) is used for the pipe, there is no way to respond to prompts using the keyboard.
3. Jobs can only be run in foreground. You cannot use the **-b** argument to run TPL TABLES in background.

Named Pipes

Named pipes or FIFOs provide a more flexible method for connecting the output of one program to the input of another. TPL TABLES treats a named pipe just like a file except that the hourglass is not displayed when a named pipe is used.

Named pipes are usually preferable to the type of piping described above as "standard piping". Since the named pipe is not the standard input, but rather a separate entity with its own name, the keyboard is free for repond-

ing to prompts. In addition, you can use the **-b** argument to run jobs in background.

To use named pipes, first create a named pipe using the **mknod** command:

```
mknod /dev/your-name p
```

where ***your-name*** is whatever you want. The pipe need not be created in ***/dev*** though this is customary. The **p** is required to indicate that the node is to be a pipe. The pipe need only be created once as it will stay around between jobs.

Now you can direct the output from your data-generating program into the pipe. Start TPL TABLES with the pipe name as the input file. TPL TABLES detects that the input file is a pipe rather than a regular file and modifies the processing as appropriate.

Suppose your pipe name is **/dev/my_pipe**. You can pipe a data file called my_data into TPL TABLES with the following sequence:

```
cat my_data > /dev/my_pipe &
tpl tables -d /dev/my_pipe
```

Most UNIX programs which write output to a user-specified file can write their output to a named pipe and hence can pipe their output into TPL TABLES.

Silent Use of Pipes

Named pipes can be used to run jobs silently in background in such a way that there is no output on the screen. The following example shows how TPL TABLES can read data from a pipe and run without displaying even a process id on the screen.

```
cat datafile > named_pipe &
tpl tables -r request -d named_pipe -b > /dev/null
```

Although TPL TABLES will run silently in this example, we will get a process id displayed from the **cat** program. In a real case, we would not be using **cat** to fill the pipe so there would be no problem.

For example, we can replace the **cat** with a trivial program called **pipe_fill** as follows:

```
main()
 {
  system("cat  datafile  >  named_pipe  &");
 }
```

Then **pipe_fill** will not display the process id so the following sequence will be completely silent:

```
pipe_fill
tpl  tables  -r  request  -d  named_pipe  -b  >  /dev/null
```

Common Error and Warning Messages

Error and warning messages are intended to be self-explanatory. However, a few common messages deserve special note.

Syntax error message

```
*** ERROR: A syntax error was discovered while processing
        'element'.  Look for the error at or before that point.
```

This message appears whenever there is a syntax error in a codebook, table request, format request or profile. Examples of syntax errors are misspelled keywords or punctuation errors such as a missing colon (:) or semicolon (;). The point at which TPL TABLES discovered the error is indicated by the *element* in quotes.

Example Following is an example showing the beginning of a TABLE statement and the error message that would result:

```
TABLE ONE 'Average Income by Region'
        HEADING REGION,

*** ERROR: A syntax error was discovered while processing
'HEADING'.  Look for the error at or before that point.
```

Since the error was found when the word HEADING was encountered, we can assume that there is something wrong with the word HEADING, or that an error preceded the word HEADING so that it appears to be in the wrong place. In this example, a colon (**:**) is missing following the table title. TPL TABLES is looking for the colon when it finds the word HEADING.

Undefined variable error message

*** ERROR: The variable '*variable-name*' is undefined.

A frequent cause of this error is a misspelled name. Another cause is a reference to a variable that has not yet been defined. For example, if a variable is computed in a COMPUTE statement and used in a TABLE statement that precedes the COMPUTE statement, the computed variable is unknown to TPL TABLES when it finds it in the TABLE statement.

Example Misspelling of the variable name INCOME as INCOM produces the message shown below.

```
POST COMPUTE AVG_INCOME = INCOM / PERSONS;

*** ERROR: The variable 'INCOM' is undefined.
```

Narrow column warning message

*** WARNING: Some columns in your tables are too narrow to hold your table cells. See the output file for details.

When TPL TABLES is formatting a table, if a data value is too wide to fit in the column, it will be replaced with the built-in **NO_FIT** footnote, making it obvious that the value does not fit. However, TPL TABLES first attempts to display the value by removing mask items such as commas, percent signs and footnote symbols and displays the value without these items. This warning message will alert you to the fact that one or more values are missing some mask items.

If you get this message it will be at the end of the file called **output**. You can then search for other instances of ***** WARNING** in the layout section of the **output** to get more detailed information about where values had items removed. For example,

*** WARNING: For table 1, page 1, column 1 is too narrow to hold some data cells.

APPENDIX F

TPL Conditions (UNIX/Linux)

WHAT IS *TPL CONDITIONS*?

The **tpl conditions** program converts partial codebook sources into complete codebook sources. In doing so, it saves you work in creating codebooks and also assures that the codebook source accurately describes the data. It works with codebooks for databases, fixed format files and delimited files such as CSV. The program can also be used to update a codebook source when the data file or database has changed in such a way that additional condition values are needed. **tpl conditions** fills in condition values and labels for all types of codebooks. For delimited (CSV) and database codebooks it also fills in field sizes. For database codebooks, **tpl conditions** fills in data types of observation variables such as **float**. See *Producing A Codebook Source with the conditions Procedure* in **Run Instructions (UNIX)** for details on how to run a **tpl conditions** job.

Control Variable Conditions

As the name of the procedure implies, the biggest use of **tpl conditions** is to fill in condition values for control variables. If the codebook is new, the condition value lists are presumably empty. In this case **tpl conditions** inserts all of the conditions found in the data for each control variable. The conditions are assigned default labels.

If the codebook is old and is merely being updated, all existing conditions and their labels are retained. **tpl conditions** just adds the new conditions found in the data. Where the new conditions are added depends upon the **display as** clause. If there is no **display as** clause or **display as sorted** is specified, the old and new conditions for a variable are intermixed and placed in sort order based on the value. If **display as listed** is specified, the old conditions are retained at the start of the condition list and the new

conditions are placed at the end of the condition list in the order they are encountered.

When **tpl conditions** has finished, you may edit the new codebook source to provide better labels for the new conditions and to rearrange them if desired.

Note **tpl conditions** cannot update codebooks that contain groups.

Fixed Format Sequential File Example

The following is an incomplete fixed format sequential file codebook before **tpl conditions** has been run. Note that all fields must have a width since this is the only way TPL can identify the boundaries of a field. **aip** has no conditions but it must have parentheses. **Complainant**, **shift_** and **squad** all have some condition values.

```
Begin dispatch codebook ascii

dispatches 'Dis'-'patches' record level 0
  filler 6
  A_I_P 'A-I-P' control 1
   (
   )
  filler 5
  STREET1 '1STREET' char 4
  STREET2 '2STREET' char 4
  COMPLAINANT 'Complainant' control 30
   (
    'Alarm Panel'  = 'ALARM PANEL'
    'Blairs Florists/John'  = 'BLAIRS FLORISTS/JOHN'
    'Cowden,George'  = 'COWDEN,GEORGE'
    'Marion High School'  = 'MARION HIGH SCHOOL'
   )
  td obs 4
  tr obs 4
  ta obs 4
  tc obs 4
  unit 'Unit' char 4
  filler 36
  full_date 'Date' char 6
  shift_ 'Shift' control 1 display as sorted
   (
    'first shift'  = '1'
    'third shift'  = '3'
   )
```

```
  filler  1
  squad 'Squad' control 1 Display as listed
   (
    'unknown'  = ' '
    'squad 9'  = '9'
   )
End dispatch codebook
```

The following is the completed codebook source after **tpl conditions** has been run. Note that **filler** has been removed. Instead, the field following the filler has a start position. **Complainant** does not have a **display as** clause so the old conditions are sorted into value order along with the new values. **shift_** has a **display as sorted** clause so the old conditions are also sorted into value order with the new conditions. **squad** uses **display as listed** so the old conditions retain their order and the new conditions are added after them.

```
Begin DISPATCH codebook ascii
DISPATCHES "Dis" - "patches" Record Level 0
A_I_P "A-I-P" start 6 Con 1
 (
   = "A"
   = "I"
   = "P"
 )
STREET1 "1STREET" start 12 Char 4
STREET2 "2STREET" Char 4
COMPLAINANT "Complainant" Con 30
 (
   = " "
   = "7 AV STD"
   "Alarm Panel" = "ALARM PANEL"
   = "ALEXANDER,DICK"
   = "ARP,MICHAEL"
   = "BEETS,GENEVA"
   = "BEHNKE,MRS"
   "Blairs Florists/John" = "BLAIRS FLORISTS/JOHN"
   = "COOK,TOM"
   = "COOPER,DEB"
   "Cowden,George" = "COWDEN,GEORGE"
   = "CR 727"
   = "MARION 76/RANDI"
   = "MARION FIRE"
   "Marion High School" = "MARION HIGH SCHOOL"
   = "MATTESON,KENNETH"
   = "WORTMAN,DAVID"
   = "YATES,DOUG"
```

```
  = "YEISLEY,BILL"
  = "YIRKOUSKY,DARREL"
  = "YOUNG,MARVIN"
 )
TD "TD" Obs 4
TR "TR" Obs 4
TA "TA" Obs 4
TC "TC" Obs 4
UNIT "Unit" Char 4
FULL_DATE "Date" start 106 Char 6
SHIFT_ "Shift" Con 1
Display as sorted
 (
  "first shift" = "1"
  = "2"
  "third shift" = "3"
 )
SQUAD "Squad" start 114 Con 1
Display as listed
 (
  "unknown" = " "
  "squad 9" = "9"
  = "1"
  = "R"
 )

End DISPATCH codebook
```

Delimited (CSV) Sequential File Example

The following is a small incomplete CSV codebook before **tpl conditions** has been run. Note that sizes are not specified but field number is. Some of the fields have been skipped. For the field **complaint** some of the condition values have been provided. **aip** has no fields provided but it does have the required parentheses.

```
Begin dispatch_csv Codebook CSV
        (Head = Yes Delimiter = COMMA)
dispatch_csv Record
ID "id" Field = 1 Char
AIP "aip" Field = 2 Control ()
COMPLAINANT "complainant" Field = 6 Control Right Blank Fill
 (
  "ALARM PANEL"  =  "ALARM PANEL"
  "BLAIRS FLORISTS/JOHN"  =  "BLAIRS FLORISTS/JOHN"
  "COWDEN,GEORGE"  =  "COWDEN,GEORGE"
  "MARION HIGH SCHOOL"  =  "MARION HIGH SCHOOL"
```

```
 )
SQUAD "squad" Field = 22 obs
End dispatch_csv
```

The following shows the complete codebook after **tpl conditions** has been run. Field widths have been filled in as have condition values for **aip**. Conditions have also been filled in for complainant, Since there is no **display as** clause, the old conditions are sorted in with the new conditions.

```
Begin DISPATCH_CSV codebook CSV
(Delimiter = Comma  Head = Yes  Quote = '"')
DISPATCH_CSV "DISPATCH CSV" Record Level 0
ID "id" Report Error = No
Field = 1 char 5
AIP "aip" Field = 2 Con 1
 (
   = "A"
   = "I"
   = "P"
 )
COMPLAINANT "complainant" Field = 6 Con Right Blank Fill 28
 (
   = " "
   = "7 AV STD"
   "ALARM PANEL" = "ALARM PANEL"
   = "ALEXANDER,DICK"
   = "ARP,MICHAEL"
   = "BEETS,GENEVA"
   = "BEHNKE,MRS"
   "BLAIRS FLORISTS/JOHN" = "BLAIRS FLORISTS/JOHN"
   = "COOK,TOM"
   = "COOPER,DEB"
   "COWDEN,GEORGE" = "COWDEN,GEORGE"
   = "CR 727"
   = "MARION 76/RANDI"
   = "MARION FIRE"
   "MARION HIGH SCHOOL" = "MARION HIGH SCHOOL"
   = "MATTESON,KENNETH"
   = "WORTMAN,DAVID"
   = "YATES,DOUG"
   = "YEISLEY,BILL"
   = "YIRKOUSKY,DARREL"
   = "YOUNG,MARVIN"
 )
SQUAD "squad" Field = 22 Obs 1

End DISPATCH_CSV codebook
```

Error Detection

In addition to producing a new codebook source, **tpl conditions** detects errors. For this example, the error messages were placed in **DISPATCH_CSV.O**. The field **squad** is described as **obs** but it has some letters in it. The following is the last part of the file **DISPATCH_CSV.O** where the errors are reported.

```
For record 255 Variable SQUAD:  'R' cannot appear in an ascii observation value.
For record 255 Variable SQUAD:  An observation value must contain a digit.
For record 256 Variable SQUAD:  'R' cannot appear in an ascii observation value.
For record 256 Variable SQUAD:  An observation value must contain a digit.
For record 267 Variable SQUAD:  'K' cannot appear in an ascii observation value.
For record 267 Variable SQUAD:  An observation value must contain a digit.
For record 268 Variable SQUAD:  'K' cannot appear in an ascii observation value.
For record 268 Variable SQUAD:  An observation value must contain a digit.
For record 280 Variable SQUAD:  'R' cannot appear in an ascii observation value.
For record 280 Variable SQUAD:  An observation value must contain a digit.

285 records read.
60 data errors were found.

End CODEBOOK CONDITIONS processing
```

SQL Database Example

The following is a small incomplete SQL codebook before **tpl conditions** has been run. Note that field sizes are not specified. Data types, such as **float**, have not been filled in for observation variables and no conditions are provided for the control variables. Instead, **get conditions from data** or **get conditions from *table*(*label,code*)** are used. Since this codebook describes a Sybase database with lowercase field names, each variable must have a **defines** clause.

```
begin sample codebook sql

employee defines "employee" table
company_id defines "company_id" obs
last_name defines "name" control from data
salary defines "salary" obs

company defines "company" table
company_name defines "name" control get conditions from data
company_id defines "company_id" obs
location defines "location" control
  get conditions from "locations"("location_name","location_id")
gross defines "gross" obs
```

```
company is parent of employee where company_id = company_id
```

After the incomplete codebook has been processed by **tpl conditions** the result is as listed below. **Last_name** and **Company_name** have conditions obtained from the data. **Location** has obtained its conditions from the **location_name** and **location_id** of the **locations** table. Field widths are filled in. Since the program was run against a Sybase data base, the **begin** statement references **Sybase** instead of **SQL**.The fields **Company_id** and **Gross** are now **obs float** instead of just **obs** and **salary** is now **obs money** and has a mask rather than just being **obs.**

```
Begin SAMPLE codebook Sybase
EMPLOYEE "EMPLOYEE" Defines "employee" table
COMPANY_ID "COMPANY ID" Defines "company_id" obs float 8
LAST_NAME "" Defines "name" control 9
 (
   "Balmer" = "Balmer"
   "Einstein" = "Einstein"
   "Gates" = "Gates"
   "Newton" = "Newton"
   "Watson" = "Watson"
   "Weeks" = "Weeks"
   "Weiss" = "Weiss"
 )
SALARY "SALARY" Mask Center $ 999.99
          Defines "salary" obs money
COMPANY "COMPANY" Defines "company" table
COMPANY_ID "COMPANY ID" Defines "company_id" obs float 8
COMPANY_NAME "" Defines "name" control 12
 (
   "IBM" = "IBM"
   "Microsoft" = "Microsoft"
   "QQQ Software" = "QQQ Software"
 )
GROSS "GROSS" Defines "gross" obs float 8
LOCATION "" Defines "location" control 2
from "locations" ("location_name", "location_id")
 (
   "Arlington" = "01"
   "Everywhere" = "02"
   "Redmond" = "03"
   "New Carrollton" = "04"
   "Nowhere" = "05"
   "hometown" = "06"
   "Atlantis" = "07"
 )
```

```
COMPANY is parent of EMPLOYEE where
        COMPANY_ID = COMPANY_ID

End SAMPLE codebook
```

Comments

tpl conditions preserves comments in your codebook source. To assure accurate placement of your comments in the output, the comments should be put in one or more of the following places:

At the start of your codebook
Before the end codebook statement
Before a variable or record entry
Before a condition entry
Before an association statement

APPENDIX G

International

FORMATS, SYMBOLS AND LANGUAGES

Important

The CODEPAGE and COUNTRY statements described in this appendix are special statements that can be used in the profile for your jobs. If you add a CODEPAGE or COUNTRY statement to your profile, change a CODEPAGE or COUNTRY statement in your profile, or make changes to **country.tpl**, ***you need to restart TPL*** to activate the changes.

Your codebook must be processed with the same CODEPAGE and COUNTRY statements that you use when running your table requests. Otherwise, you will have conflicting standards. In particular, conflicts in CODEPAGE will cause the sort order to be scrambled.

Alphabets and Sort Order: The CODEPAGE Statement

The CODEPAGE determines the character set and sort order for your requests and tables. The default CODEPAGE will work with many languages. If you need additional characters for your alphabet, you can select a different CODEPAGE from those shown in the Appendix called "Character Sets". See also the CODEPAGE statement in the FORMAT chapter.

Entering characters, using them in labels and printing them. The most desirable way of entering characters is with a keyboard that is appropriate for the alphabet of the language you are using and an editor that supports it.

Any character that can be entered on the keyboard, either directly or by using **Alt** and the numeric keypad, can be used in TPL TABLES labels and other character strings such as condition values.

Characters not on your keyboard can also be entered by typing in their numeric code or by entering a character name.

Character Name. A character name is the name of a character preceded by **&** and terminated with **;.** For example **É** refers to the letter **E** with an acute accent above it. Character names are case sensitive. **é** is the letter **e** with an accute accent.The acceptable names are the names for the codepage you have selected. See *Special Character names* in the "Character Sets" Appendix. Use of character names instead of character codes has the advantage of being more portable. If you switch codepages, the table will look the same provided the character name is in both code pages. Also, if you are using a table for multiple purposes -- creating a pdf, creating a web page, and printing the table -- then use of a character name will in general result in a constant display of the character. Finally, table requests written using character names area easier to read than requests using character codes.

Character Code. A Character code is a \ followed by a 3 digit number which identifies the character. Three digits are always required. If the character can be represented by fewer than 3 digits, add leading zeros. For example, for a character represented by the code 65, enter **\065**.

The value **nnn** must be the ***decimal*** code for the character. Note that the character code tables in some software manuals show the octal or hexidecimal codes for the characters. If you are using this type of table, you must convert the code to its decimal equivalent. Character set tables showing decimal codes are included in the Appendix called "Character Sets".

These characters will print correctly if they are included in the character set for the selected CODEPAGE. In exported text tables, the characters will print correctly if they are available on the printer.

Alphabet for user-specified names. If an alphabetic character is included in the character set for the selected CODEPAGE and the character can be entered on the keyboard, either directly or by using **Alt** and the numeric keypad, it can be used in names for variables, tables, and other items.

The Sort Sequence. The proper order for sorting depends on the character set used. TPL will use the sequence that goes with the character set selected by the CODEPAGE statement. The sort sequences for all character sets are stored in a file called **sort.tpl** that is installed in the TPL system directory.

Note Your should insert CODEPAGE *at the beginning* of your profile. You cannot do this until after TPL TABLES is installed.

The COUNTRY Statement

The COUNTRY statement is fully described in the FORMAT chapter of the manual. It lets you select standards for the characters to be used as decimal and thousands separators, the currency symbols and format, and formats for date and time. These standards are set in a file called **country.tpl** that is installed with TPL TABLES. US is the default country.

Note You should insert COUNTRY *at the beginning* of your profile. You cannot do this until after TPL TABLES is installed. Before inserting the COUNTRY statement, you should check to see if there are any decimal numbers already used in the profile. For example, decimal numbers can be used in the page size specifications. If you have any such instances, you should edit your profile to match your country standard.

Specifying Right-hand Stubs with the FORMAT Statement STUB RIGHT

With the FORMAT statement STUB RIGHT, tables are formatted with the stub on the right side of the table instead of the left. This is most often used to prepare tables on facing pages. It is particularly useful if you need to do a table in two languages on facing pages where the left page has the stub on the left in one language and the right page has the stub on the right in another language.

See the STUB RIGHT statement in the FORMAT chapter of the user manual for details.

Replacing Default English Text

If you regularly use TPL TABLES to produce tables in a language other than English, you may wish to replace the default text for labels such as TOTAL (**"Total"**), title continuation (**" - Continued"**) or the built-in footnotes such as the SEE_END footnote **"See footnotes at end of table."**

We recommend that you replace these labels by entering the appropriate FORMAT statements in your **profile.tpl** file. The new labels will then automatically apply to all of your tables.

APPENDIX H

Keywords

TPL TABLES Keywords

The following words are TPL TABLES keywords. They should not be used as names for tables, variables, conditions, codebooks, or footnotes.

ABS
AFTER
ALIGN
ALL
ALTERNATE
AND
AS
ASCENDING
ASCII
AT
AUTO
AUTOMATIC
BANK
BANKS
BEGIN
BINARY
BIT
BLANK
BLANKS
BOLD
BOTH
BOTTOM
BY
CELL
CELLFILE
CELLS
CENTER
CENTRE
CHANGE
CHAR
CHARACTER
CM
CODEBOOK
CODEPAGE
COLOR
COLOUR
COLUMN
COLUMNS
COMMAND
COMPRESS
COMPUTE
CON
CONDITION
CONDITIONS
CONT
CONTINUATION
CONTINUE
CONTINUED
CONTROL
COPY
COUNT
CREATED
CSV
DATA
DATE
DECIMAL
DEFAULT
DEFINE
DEFINES
DELETE
DELIMITER
DESCENDING
DISPLAY
DIV
DIVIDE
DIVIDER
DO
DOUBLE
DOWN
EACH
EIA
EJECT
EMPTY
END
EOF
EPS
EQUAL
EQUALS
EVALUATED
EVERY
EXCEPT
EXTRA
FETCH
FIELD
FILE
FILL
FILLER
FLOAT
FMEDIAN
FONT
FOOTNOTE
FOOTNOTES
FOR
FQUANTILE
FROM
GET
GRAY
GREATER
GREY
GROUP
HEAD
HEADER
HEADERS
HEADING
HEADINGS
HEADNOTE
HEADS
HIERARCHIES
HTML
I
IF
IN

INCH
INCHES
INCOMPLETE
INCREMENT
INDENT
INPUT
INS
IS
ITALIC
JUSTIFIED
JUSTIFY
KEEP
KEY
LABEL
LABELS
LAST
LEADING
LEFT
LENGTH
LESS
LEVEL
LINE
LINES
LISTED
MARGIN
MARKER
MASK
MAX
MAXIMUM
MEAN
MEDIAN
MEMORY
MIN
MONEY
MONITOR
NAME
NAMES
NO
NORMAL
NOT
NOTE
NULL
NUMBER
NUMBERS
NUMERIC
OBS
OBSERVATION
ODBC
ODS
OF
ON
OR
ORACLE
OTHER
PAGE
PAPER
PARENT
PATH
PDF
PERCENT
PLAN
POINT
POINTS
POST
POSTCOMPUTE
POSTSCRIPT
PRIMARY*
PRINT
PT
PTS
QUANTILE
QUANTILES
QUOTE
RANK
RECORD
REDEFINES
REPEAT
REPEATS
REPLACE
REPORT
REPORTS
RETAIN
RIGHT
ROTATE
ROUND
ROW
ROWS
RULE
RULES
SCALE
SELECT
SEQUENCE
SET
SHADE
SHIFT
SIB
SIBLING
SIDE
SKIP
SORTED
SPACE
SPACES
SPAN
SPANNER
SPANNERS
SQL
SQRT
START
STARTS
STATCAN
STDERR
STDEV
STDEVP
STOP
STUB
STUBS
SUB
SUBSTR
SUBSTRING
SUP
SUPER
SYBASE
SYM
SYMBOL
TABLE
TABLES
TABULATE
TEXT
THAN
THEN
TITLE
TITLES
TO
TOP
TOTAL
U
UNDERLINE
UNJUSTIFIED
UNJUSTIFY
UNLESS
UNSIGNED
UP
USE
USING
VALUE
VALUES
VAR
VARIABLE
VARIABLES
VARP
VARYING
WAFER
WAFERS
WEIGHTED
WEIGHTING
WHERE
WIDTH
WITH
XLS
YES

* Codebook only. You can continue to use this word as a variable name, if you precede it with a : in the codebook. For example, :PRIMARY

APPENDIX I

Limits

SUMMARY OF FEATURES AND SYSTEM CONSTRAINTS

Platforms and Operating Systems

Windows 98, XP, 2000, VISTA.

UNIX platforms, including Sun and HP.

Can be ported to other UNIX platforms.

Contact QQQ Software for current list.

Minimum Hardware Configuration

Hard disk space: 30 megabytes

Printer: any

Optional Hardware

PostScript printer: On UNIX systems, a PostScript printer is required to print PostScript tables directly. You can however export the tables to pdf and print them on most printers. Alternately some PostScript display programs support printing of PostScript tables on printers which do not print Postscript. On Windows systems, PostScript tables can be printed from TED on any printer. When PostScript tables are inserted in documents with desktop publishing software, a PostScript printer may be required to correctly print the tables. If you convert PostScript tables, or documents containing PostScript tables, to Adobe Acrobat PDF format, they can be printed from Adobe Acrobat Reader.

Hard disk space: The installed system occupies about 30 megabytes of hard disk space. Additional space is needed for temporary work files and

for your data and output tables. Alternate drives can be substituted for anything other than the installed TPL TABLES system.

Features/Constraints

There are very few fixed limits in TPL TABLES. The available computer resources are allocated according to the unique requirements of each job so that space not needed for one feature can be used by another. Thus, it is highly unlikely that you will ever encounter a limitation on the size of your job. If you do, please contact Software Support for suggestions.

Maximum cells per request: no limit

Maximum number of tables per request: no limit

Maximum number of variable references: no limit

Maximum number of values for a single control variable (including variables created by DEFINE statements): no system limit, although performance may degrade with many hundreds of thousands of values, depending on the capacity of your computer and what you are doing with the variable.

Maximum columns per table: no system limit, although tables with thousands of columns may encounter memory limitations

Maximum print label length: no limit

Maximum record types and groups in codebook: 30

Input data file requirements:

Record formats: fixed length records with data fields in fixed columns; variable length CSV (comma separated) and other types of delimited files

Datafile type: sequential

Maximum record length: 32,764 bytes for fixed length records; 50,000 bytes for CSV and other delimited files

Datafile organizations: flat (single level) and hierarchical (multi-level)

Data field types: character (ASCII), binary and floating point (single or double precision)

SQL databases: The TPL-SQL Database Interface is optional. For

Window systems, databases can be accessed via ODBC. For UNIX systems, contact QQQ Software for the current list of supported database systems.

Accuracy of computed results: Computations are done in ANSI standard double precision floating point with special code to prevent comparison errors introduced by radix conversion.

Format for codebooks, table requests and format requests: free format

Variable name format: up to 30 characters long, starting with letter, # or underbar(_), and containing only letters, digits, # and _

Statistics: percentages, medians, quantiles (percentiles, quartiles, etc.), maxima, minima, means, variances, standard deviations, standard errors; others can be generated with COMPUTE and POST COMPUTE statements

Statement types: table, select, define, compute, post compute, conditional compute and post compute, median, quantile, percent, rank, weighting, footnote, note, label, and use

APPENDIX J

Utilities

STAND-ALONE UTILITY PROGRAMS

Several stand-alone utility programs are installed with TPL TABLES. You may find some of these programs useful in applications other than TPL TABLES.

FOR_WORD

Note FOR_WORD is a public domain program.

Location

Windows: Installed in the TPL TABLES system directory

UNIX: Installed with TPL TABLES in the tpldebug subdirectory

File Name

for_word.exe (if Windows)
for_word (if UNIX)

Purpose

To take a file that was prepared with a line editor and convert it to word processing format. By "line editor", we mean a program that puts one or more return characters at the end of each line. By "word processor", we mean a program that works with paragraphs rather than lines. The FOR_WORD program will convert a line editor file for use with a word processor by removing the return characters from lines within paragraphs.

Instructions

The program is self-documenting. On the command line, type

```
for_word <Enter>
```

Instructions will be displayed on the screen.

HEXLIST

Location

Windows: Installed in the TPL TABLES system directory.

UNIX: Installed with TPL TABLES in the tpldebug subdirectory

File Name

hexlist.exe (if Windows)
hexlist (if UNIX)

Purpose

The hexlist program displays the contents of a file as hexidecimal values and, when possible, as ascii characters. Where there is no ascii character equivalent for the hexidecimal value, a % symbol is displayed on the character line.

The hexlist program can be very useful in identifying problems in a data file when the file has errors or is not in the format that you expected.

Instructions

On the command line, type

```
hexlist arg1 arg2 arg3 <Enter>
```

where

arg1 is file name
arg2 (optional) is a line width <= 75. 75 is the default
arg3 (optional) indicates that the file should be opened in ascii rather than default binary mode. It must be a lower case letter “a” or the word “ascii” (not in quotes).

If you want the file opened in ascii mode, you must provide both arguments 2 and 3.

If the file is opened in binary mode (the default), all characters in the file, including any end-of-record or end-of-file indicators, will be displayed.

If the file is opened in ascii mode, carriage returns and end-of-file markers will not be displayed. Line feeds will be displayed.

On Windows systems, most ascii files have a carriage return and line feed at the end of each record and a control-Z at the end of file.

The hexidecimal codes for these end-of-record and end-of-file characters are:

```
0D    <CR>
0A    <LF>
1A    control-Z
```

UNIX Note If you are working with a UNIX system, the binary/ascii distinction is irrelevant since you will get the same result either way. Most UNIX ascii files have a line feed (hexidecimal 0A) at the end of each record.

How to Stop

If you have a large file, you may wish to stop the hexlist after displaying just a part of it. You can stop the hexlist by entering <Ctrl><Break> or <Ctrl>C.

UNIX Note In UNIX, you can stop the hexlist with the key or key combination that you normally use to cancel jobs.

Redirection

The screen output can also be redirected to a file. For example,

```
hexlist  mydata  >  hexout  <Enter>
```

will do a hexlist of the file mydata, displaying 75 characters per line and saving the output in the file called hexout.

PSP -- PostScript Print Program

Location

Installed in the TPL TABLES system directory.

File Name

psp.exe (if Windows)
psp (if UNIX)

Purpose

PSP is a powerful utility for selective formatting and printing regular ASCII character files on a PostScript compatible printer.

Instructions

The program is self-documenting. On the command line, type

```
psp <Enter>
```

Instructions will be displayed on the screen. Wild cards can be used to print multiple files that have a portion of the name in common. For example, to print all files that have the suffix **.txt**, type:

```
psp *.txt <Enter>
```

Note

For a line of text that ends with a return character and is longer that the width of the page, PSP will "wrap" the long line, then go to a new line for the following text. For example:

This is the first line of text. It is too long for the page width so it wraps when it is printed.

This is the second line of text. It starts on a new line.

If you want to print this type of text file with PSP, you can get a better result by using the FOR_WORD program to remove the return characters within paragraphs. When you use FOR_WORD, write the output to a temporary file. Then print the temporary file with PSP. For example,

```
for_word myfile tempfile <Enter>
psp tempfile <Enter>
```

TO_SHOW (Windows only)

This program is not needed if text tables come from export instead of from POSTSCIPT = NO;

Location

Windows: Installed in the TPL TABLES system directory.

UNIX: Not available

File Name

TO_SHOW.EXE

Purpose

When TPL TABLES formats tables with POSTSCRIPT = NO, it formats horizontal rules as extensions of other lines of the tables. This format will gives the best possible result on any type of printer. The tables can be conveniently reviewed on the screen with TED, the TPL Editor, because TED is custom-programmed to work correctly with the table format. If, instead, you try to edit the tables or display them with other software, the horizontal rules may not display the way you want.

You can use the program TO_SHOW to convert a tables file created using **POSTSCRIPT = NO**; to a format that will work and editors (word processors), and display correctly on the screen using any display software.

Instructions

On the command line, type

```
TO_SHOW tables-in tables-out  <Enter>
```

where **tables-in** is the original tables file and **tables-out** is the converted tables file.

Example

```
TO_SHOW  TABLES  TABLES.SHO  <Enter>
```

APPENDIX K

Character Sets

CHARACTERS AND CODEPAGES

The WIN character sets are recommended for the Windows version; the ISO character sets are recommended for the UNIX version.

The default for Windows is WIN88591. The default for UNIX is ISO88591. To select a different character set, use the CODEPAGE statement described in the Format chapter.

If you want your jobs to give identical results using both the Unix and Windows versions, you should use Windows and ISO codepages with all of the characters you need and use character names rather than character codes in your request.

EURO Symbol

TPL Tables provides full support for the euro symbol provided your printer and computer fonts support it. Windows 2000 may not support the euro symbol but Windows XP and Vista do. Sun Solaris 8 does not support the euro but later versions do.

The **country.tpl** has been changed so the currency symbol is a euro for those countries which have adopted it.

If you look in the codepage files such as win88591.cp, you will see 4 different euro entries, **display_euro**, **pdf_euro**, **eps_euro**, and **psprint_euro**. This is because in certain computer environments the the correct way to specify a euro for one purpose is different from the way to express it for a different purpose. If for example you find that a euro symbol is displayed correctly on the screen but does not convert to a pdf correctly, then you should change the **pdf_euro** code but not the **display_euro** code. If you then use **€** in your request, TPL Tables will select the correct euro code to use for the action you are performing. If you have problems with these, please give us a call.

Mapping of Decimal Values to Postscript Codes for Standard Fonts using CODEPAGE=WIN88591

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		053	5	105	i	157		209	Ñ
002		054	6	106	j	158		210	Ò
003		055	7	107	k	159	Ÿ	211	Ó
004		056	8	108	l	160		212	Ô
005		057	9	109	m	161	¡	213	Õ
006		058	:	110	n	162	¢	214	Ö
007		059	;	111	o	163	£	215	×
008		060	<	112	p	164	¤	216	Ø
009		061	=	113	q	165	¥	217	Ù
010		062	>	114	r	166	¦	218	Ú
011		063	?	115	s	167	§	219	Û
012		064	@	116	t	168	¨	220	Ü
013		065	A	117	u	169	©	221	Ý
014		066	B	118	v	170	ª	222	Þ
015		067	C	119	w	171	«	223	ß
016		068	D	120	x	172	¬	224	à
017		069	E	121	y	173	-	225	á
018		070	F	122	z	174	®	226	â
019		071	G	123	{	175	¯	227	ã
020		072	H	124	l	176	°	228	ä
021		073	I	125	}	177	±	229	å
022		074	J	126	~	178	²	230	æ
023		075	K	127		179	³	231	ç
024		076	L	128		180	´	232	è
025		077	M	129		181	µ	233	é
026		078	N	130	‚	182	¶	234	ê
027		079	O	131	*ƒ*	183	·	235	ë
028		080	P	132	„	184	¸	236	ì
029		081	Q	133	…	185	¹	237	í
030		082	R	134	†	186	º	238	î
031		083	S	135	‡	187	»	239	ï
032		084	T	136	ˆ	188	¼	240	ð
033	!	085	U	137	‰	189	½	241	ñ
034	"	086	V	138	Š	190	¾	242	ò
035	#	087	W	139	‹	191	¿	243	ó
036	$	088	X	140	Œ	192	À	244	ô
037	%	089	Y	141		193	Á	245	õ
038	&	090	Z	142		194	Â	246	ö
039	’	091	[	143		195	Ã	247	÷
040	(	092	\	144		196	Ä	248	ø
041	)	093	]	145	‘	197	Å	249	ù
042	*	094	^	146	’	198	Æ	250	ú
043	+	095	_	147	“	199	Ç	251	û
044	,	096	‘	148	”	200	È	252	ü
045	-	097	a	149	•	201	É	253	ý
046	.	098	b	150	–	202	Ê	254	þ
047	/	099	c	151	—	203	Ë	255	ÿ
048	0	100	d	152	˜	204	Ì		
049	1	101	e	153		205	Í		
050	2	102	f	154	š	206	Î		
051	3	103	g	155	›	207	Ï		
052	4	104	h	156	œ	208	Đ		

Mapping of Decimal Values to Postscript Codes for Standard Fonts using CODEPAGE=WIN88592

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		053	5	105	i	157	ť	209	Ń
002		054	6	106	j	158	ž	210	Ň
003		055	7	107	k	159	ź	211	Ó
004		056	8	108	l	160		212	Ô
005		057	9	109	m	161	ˇ	213	Ő
006		058	:	110	n	162	˘	214	Ö
007		059	;	111	o	163	Ł	215	×
008		060	<	112	p	164	¤	216	Ř
009		061	=	113	q	165	Ą	217	Ů
010		062	>	114	r	166	¦	218	Ú
011		063	?	115	s	167	§	219	Ű
012		064	@	116	t	168	¨	220	Ü
013		065	A	117	u	169	©	221	Ý
014		066	B	118	v	170	Ş	222	Ţ
015		067	C	119	w	171	«	223	ß
016		068	D	120	x	172	¬	224	ŕ
017		069	E	121	y	173	-	225	á
018		070	F	122	z	174	®	226	â
019		071	G	123	{	175	Ż	227	ă
020		072	H	124	|	176	°	228	ä
021		073	I	125	}	177	±	229	Í
022		074	J	126	~	178	˛	230	ć
023		075	K	127		179	ł	231	ç
024		076	L	128		180	´	232	č
025		077	M	129		181	µ	233	é
026		078	N	130	,	182	¶	234	ę
027		079	O	131		183	·	235	ë
028		080	P	132	„	184	¸	236	ě
029		081	Q	133	…	185	ą	237	í
030		082	R	134	†	186	ş	238	î
031		083	S	135	‡	187	»	239	ď
032		084	T	136		188	Ľ	240	đ
033	!	085	U	137	‰	189	˝	241	ń
034	"	086	V	138	Š	190	ľ	242	ň
035	#	087	W	139	‹	191	ż	243	ó
036	$	088	X	140	Ś	192	Ŕ	244	ô
037	%	089	Y	141	Ť	193	Á	245	ő
038	&	090	Z	142	Ž	194	Â	246	ö
039	’	091	[	143	Ź	195	Ă	247	÷
040	(	092	\	144		196	Ä	248	ř
041	)	093	]	145	‘	197	Ĺ	249	ů
042	*	094	^	146	’	198	Ć	250	ú
043	+	095	_	147	“	199	Ç	251	ű
044	,	096	‘	148	”	200	Č	252	ü
045	-	097	a	149	•	201	É	253	ý
046	.	098	b	150	–	202	Ę	254	ţ
047	/	099	c	151	—	203	Ë	255	
048	0	100	d	152		204	Ě		
049	1	101	e	153		205	Í		
050	2	102	f	154	š	206	Î		
051	3	103	g	155	›	207	Ď		
052	4	104	h	156	ś	208	Đ		

Mapping of Decimal Values to Postscript Codes for Standard Fonts using CODEPAGE=WIN88599

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		053	5	105	i	157		209	Ñ
002		054	6	106	j	158		210	Ò
003		055	7	107	k	159	Ÿ	211	Ó
004		056	8	108	l	160		212	Ô
005		057	9	109	m	161	¡	213	Õ
006		058	:	110	n	162	¢	214	Ö
007		059	;	111	o	163	£	215	×
008		060	<	112	p	164	¤	216	Ø
009		061	=	113	q	165	¥	217	Ù
010		062	>	114	r	166	¦	218	Ú
011		063	?	115	s	167	§	219	Û
012		064	@	116	t	168	¨	220	Ü
013		065	A	117	u	169	©	221	İ
014		066	B	118	v	170	ª	222	Ş
015		067	C	119	w	171	«	223	ß
016		068	D	120	x	172	¬	224	à
017		069	E	121	y	173	-	225	á
018		070	F	122	z	174	®	226	â
019		071	G	123	{	175	¯	227	ã
020		072	H	124	|	176	°	228	ä
021		073	I	125	}	177	±	229	å
022		074	J	126	~	178	²	230	æ
023		075	K	127		179	³	231	ç
024		076	L	128		180	´	232	è
025		077	M	129		181	µ	233	é
026		078	N	130	‚	182	¶	234	ê
027		079	O	131	*ƒ*	183	·	235	ë
028		080	P	132	„	184	¸	236	ì
029		081	Q	133	…	185	¹	237	í
030		082	R	134	†	186	º	238	î
031		083	S	135	‡	187	»	239	ï
032		084	T	136	ˆ	188	¼	240	ğ
033	!	085	U	137	‰	189	½	241	ñ
034	"	086	V	138	Š	190	¾	242	ò
035	#	087	W	139	‹	191	¿	243	ó
036	$	088	X	140	Œ	192	À	244	ô
037	%	089	Y	141		193	Á	245	õ
038	&	090	Z	142		194	Â	246	ö
039	’	091	[	143		195	Ã	247	÷
040	(	092	\	144		196	Ä	248	ø
041	)	093	]	145	‘	197	Å	249	ù
042	*	094	^	146	’	198	Æ	250	ú
043	+	095	_	147	“	199	Ç	251	û
044	,	096	‘	148	”	200	È	252	ü
045	-	097	a	149	•	201	É	253	ı
046	.	098	b	150	–	202	Ê	254	ş
047	/	099	c	151	—	203	Ë	255	ÿ
048	0	100	d	152	˜	204	Ì		
049	1	101	e	153		205	Í		
050	2	102	f	154	š	206	Î		
051	3	103	g	155	›	207	Ï		
052	4	104	h	156	œ	208	Ğ		

Mapping of Decimal Values to Postscript Codes for Standard Fonts using CODEPAGE=ISO88591

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		053	5	105	i	157	˝	209	Ñ
002		054	6	106	j	158	˛	210	Ò
003		055	7	107	k	159	ˇ	211	Ó
004		056	8	108	l	160		212	Ô
005		057	9	109	m	161	¡	213	Õ
006		058	:	110	n	162	¢	214	Ö
007		059	;	111	o	163	£	215	×
008		060	<	112	p	164		216	Ø
009		061	=	113	q	165	¥	217	Ù
010		062	>	114	r	166	¦	218	Ú
011		063	?	115	s	167	§	219	Û
012		064	@	116	t	168	¨	220	Ü
013		065	A	117	u	169	©	221	Ý
014		066	B	118	v	170	ª	222	Þ
015		067	C	119	w	171	«	223	ß
016		068	D	120	x	172	¬	224	à
017		069	E	121	y	173	-	225	á
018		070	F	122	z	174	®	226	â
019		071	G	123	{	175	¯	227	ã
020		072	H	124	|	176	°	228	ä
021		073	I	125	}	177	±	229	å
022		074	J	126	~	178	²	230	æ
023		075	K	127		179	³	231	ç
024		076	L	128	–	180	´	232	è
025		077	M	129	—	181	µ	233	é
026		078	N	130		182	¶	234	ê
027		079	O	131		183	·	235	ë
028		080	P	132		184	¸	236	ì
029		081	Q	133		185	¹	237	í
030		082	R	134		186	º	238	î
031		083	S	135		187	»	239	ï
032		084	T	136		188	¼	240	ð
033	!	085	U	137		189	½	241	ñ
034	"	086	V	138		190	¾	242	ò
035	#	087	W	139		191	¿	243	ó
036	$	088	X	140		192	À	244	ô
037	%	089	Y	141		193	Á	245	õ
038	&	090	Z	142		194	Â	246	ö
039	’	091	[	143		195	Ã	247	÷
040	(	092	\	144	ı	196	Ä	248	ø
041	)	093	]	145	`	197	Å	249	ù
042	*	094	^	146	´	198	Æ	250	ú
043	+	095	_	147	ˆ	199	Ç	251	û
044	,	096	‘	148	˜	200	È	252	ü
045	-	097	a	149		201	É	253	ý
046	.	098	b	150	˘	202	Ê	254	þ
047	/	099	c	151	˙	203	Ë	255	ÿ
048	0	100	d	152		204	Ì		
049	1	101	e	153		205	Í		
050	2	102	f	154	˚	206	Î		
051	3	103	g	155		207	Ï		
052	4	104	h	156		208	Đ		

Mapping of Decimal Values to Postscript Codes for Standard Fonts using CODEPAGE=ISO88592

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		053	5	105	i	157		209	Ń
002		054	6	106	j	158		210	Ň
003		055	7	107	k	159		211	Ó
004		056	8	108	l	160		212	Ô
005		057	9	109	m	161	Ą	213	Ő
006		058	:	110	n	162	¢	214	Ö
007		059	;	111	o	163	Ł	215	×
008		060	<	112	p	164		216	Ř
009		061	=	113	q	165	Ľ	217	Ů
010		062	>	114	r	166	Ś	218	Ú
011		063	?	115	s	167	§	219	Ű
012		064	@	116	t	168	¨	220	Ü
013		065	A	117	u	169	Š	221	Ý
014		066	B	118	v	170	Ş	222	Ţ
015		067	C	119	w	171	Ť	223	ß
016		068	D	120	x	172	Ź	224	ŕ
017		069	E	121	y	173	›	225	á
018		070	F	122	z	174	Ž	226	â
019		071	G	123	{	175	Ż	227	ă
020		072	H	124	|	176	°	228	ä
021		073	I	125	}	177	ą	229	Í
022		074	J	126	~	178	†	230	ć
023		075	K	127		179	ł	231	ç
024		076	L	128	–	180	·	232	č
025		077	M	129	—	181	l’	233	é
026		078	N	130		182	ś	234	ę
027		079	O	131		183	•	235	ë
028		080	P	132		184	,	236	ě
029		081	Q	133		185	š	237	í
030		082	R	134		186	ş	238	î
031		083	S	135		187	ť	239	d’
032		084	T	136		188	ź	240	đ
033	!	085	U	137		189	‰	241	ń
034	"	086	V	138		190	ž	242	ň
035	#	087	W	139		191	ż	243	ó
036	$	088	X	140		192	Ŕ	244	ô
037	%	089	Y	141		193	Á	245	ő
038	&	090	Z	142		194	Â	246	ö
039	’	091	[	143		195	Ă	247	÷
040	(	092	\	144		196	Ä	248	ř
041	)	093	]	145		197	Ĺ	249	ů
042	*	094	^	146		198	Ć	250	ú
043	+	095	_	147		199	Ç	251	ű
044	,	096	‘	148		200	Č	252	ü
045	-	097	a	149		201	É	253	ý
046	.	098	b	150		202	Ę	254	ţ
047	/	099	c	151		203	Ě	255	
048	0	100	d	152		204	Ë		
049	1	101	e	153		205	Í		
050	2	102	f	154		206	Î		
051	3	103	g	155		207	Ď		
052	4	104	h	156		208	Đ		

Mapping of Decimal Values to Postscript Codes for Standard Fonts using CODEPAGE=ISO88599

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		053	5	105	i	157		209	Ñ
002		054	6	106	j	158		210	Ò
003		055	7	107	k	159		211	Ó
004		056	8	108	l	160		212	Ô
005		057	9	109	m	161	¡	213	Õ
006		058	:	110	n	162	¢	214	Ö
007		059	;	111	o	163	£	215	×
008		060	<	112	p	164		216	Ø
009		061	=	113	q	165	¥	217	Ù
010		062	>	114	r	166	¦	218	Ú
011		063	?	115	s	167	§	219	Û
012		064	@	116	t	168	¨	220	Ü
013		065	A	117	u	169	©	221	İ
014		066	B	118	v	170	ª	222	Ş
015		067	C	119	w	171	«	223	ß
016		068	D	120	x	172	¬	224	à
017		069	E	121	y	173	-	225	á
018		070	F	122	z	174	®	226	â
019		071	G	123	{	175	¯	227	ã
020		072	H	124	l	176	°	228	ä
021		073	I	125	}	177	±	229	å
022		074	J	126	~	178	²	230	æ
023		075	K	127		179	³	231	ç
024		076	L	128	–	180	´	232	è
025		077	M	129	—	181	µ	233	é
026		078	N	130		182	¶	234	ê
027		079	O	131		183	·	235	ë
028		080	P	132		184	¸	236	ì
029		081	Q	133		185	¹	237	í
030		082	R	134		186	º	238	î
031		083	S	135		187	»	239	ï
032		084	T	136		188	¼	240	ğ
033	!	085	U	137		189	½	241	ñ
034	"	086	V	138		190	¾	242	ò
035	#	087	W	139		191	¿	243	ó
036	$	088	X	140		192	À	244	ô
037	%	089	Y	141		193	Á	245	õ
038	&	090	Z	142		194	Â	246	ö
039	’	091	[	143		195	Ã	247	÷
040	(	092	\	144		196	Ä	248	ø
041	)	093	]	145		197	Å	249	ù
042	*	094	^	146		198	Æ	250	ú
043	+	095	_	147		199	Ç	251	û
044	,	096	‘	148		200	È	252	ü
045	-	097	a	149		201	É	253	ı
046	.	098	b	150		202	Ê	254	ş
047	/	099	c	151		203	Ë	255	ÿ
048	0	100	d	152		204	Ì		
049	1	101	e	153		205	Í		
050	2	102	f	154		206	Î		
051	3	103	g	155		207	Ï		
052	4	104	h	156		208	Ğ		

Mapping of Decimal Values to Postscript Codes for Symbol(Y) Font

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		054	6	107	κ	160		213	∏
002		055	7	108	λ	161	ϒ	214	√
003		056	8	109	μ	162	′	215	·
004		057	9	110	ν	163	≤	216	¬
005		058	:	111	ο	164	⁄	217	∧
006		059	;	112	π	165	∞	218	∨
007		060	<	113	θ	166	*f*	219	⇔
008		061	=	114	ρ	167	♣	220	⇐
009		062	>	115	σ	168	♦	221	⇑
010		063	?	116	τ	169	♥	222	⇒
011		064	≅	117	υ	170	♠	223	⇓
012		065	A	118	ϖ	171	↔	224	◊
013		066	B	119	ω	172	←	225	〈
014		067	X	120	ξ	173	↑	226	®
015		068	Δ	121	ψ	174	→	227	©
016		069	E	122	ζ	175	↓	228	™
017		070	Φ	123	{	176	°	229	∑
018		071	Γ	124	|	177	±	230	⎛
019		072	H	125	}	178	″	231	⎜
020		073	I	126	~	179	≥	232	⎝
021		074	ϑ	127		180	×	233	⎡
022		075	K	128		181	∝	234	⎢
023		076	Λ	129		182	∂	235	⎣
024		077	M	130		183	•	236	⎧
025		078	N	131		184	÷	237	⎨
026		079	O	132		185	≠	238	⎩
027		080	Π	133		186	≡	239	⎪
028		081	Θ	134		187	≈	240	
029		082	P	135		188	…	241	〉
030		083	Σ	136		189	⏐	242	⌠
031		084	T	137		190	—	243	⌡
032		085	Y	138		191	↵	244	⎮
033	!	086	ς	139		192	ℵ	245	⌡
034	∀	087	Ω	140		193	ℑ	246	⎞
035	#	088	Ξ	141		194	ℜ	247	⎟
036	∃	089	Ψ	142		195	℘	248	⎠
037	%	090	Z	143		196	⊗	249	⎛
038	&	091	[	144		197	⊕	250	⎜
039	ə	092	∴	145		198	∅	251	⎝
040	(	093	]	146		199	∩	252	⎞
041	)	094	⊥	147		200	∪	253	⎬
042	*	095	_	148		201	⊃	254	⌡
043	+	096	‾	149		202	⊇	255	
044	,	097	α	150		203	⊄		
045	−	098	β	151		204	⊂		
046	.	099	χ	152		205	⊆		
047	/	100	δ	153		206	∈		
048	0	101	ε	154		207	∉		
049	1	102	ϕ	155		208	∠		
050	2	103	γ	156		209	∇		
051	3	104	η	157		210	®		
052	4	105	ι	158		211	©		
053	5	106	φ	159		212	™		

Mapping of Decimal Values to Postscript Codes for Dingbats(D) Font

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
001		054	✖	107	✳	160		213	→
002		055	✗	108	●	161	❡	214	↔
003		056	✘	109	❍	162	❢	215	↕
004		057	✙	110	■	163	❣	216	↘
005		058	✚	111	❑	164	❤	217	→
006		059	✛	112	❒	165	❥	218	↗
007		060	✜	113	❑	166	❦	219	→
008		061	†	114	❒	167	❧	220	➜
009		062	✞	115	▲	168	♣	221	→
010		063	✟	116	▼	169	♦	222	→
011		064	✠	117	◆	170	♥	223	➟
012		065	✡	118	❖	171	♠	224	➟
013		066	☩	119	◗	172	①	225	➡
014		067	✣	120	❙	173	②	226	➢
015		068	✤	121	❚	174	③	227	➢
016		069	✥	122	❯	175	④	228	➤
017		070	✦	123	❛	176	⑤	229	➡
018		071	✧	124	❜	177	⑥	230	➡
019		072	★	125	❝	178	⑦	231	➧
020		073	☆	126	❞	179	⑧	232	➡
021		074	✪	127		180	⑨	233	➪
022		075	✫	128		181	⑩	234	➪
023		076	✬	129		182	❶	235	➫
024		077	✭	130		183	❷	236	➬
025		078	✮	131		184	❸	237	➭
026		079	✯	132		185	❹	238	➮
027		080	✰	133		186	❺	239	➯
028		081	✱	134		187	❻	240	
029		082	✲	135		188	❼	241	➱
030		083	✳	136		189	❽	242	➲
031		084	✴	137		190	❾	243	➳
032		085	✵	138		191	❿	244	➴
033	✁	086	✶	139		192	①	245	➵
034	✂	087	✷	140		193	②	246	➶
035	✃	088	✸	141		194	③	247	➷
036	✄	089	✹	142		195	④	248	➸
037	☎	090	✺	143		196	⑤	249	➹
038	✆	091	✻	144		197	⑥	250	➺
039	✇	092	✼	145		198	⑦	251	➻
040	✈	093	✽	146		199	⑧	252	➼
041	✉	094	✾	147		200	⑨	253	➽
042	☛	095	✿	148		201	⑩	254	➾
043	☞	096	❀	149		202	❶	255	
044	✌	097	❁	150		203	❷		
045	✍	098	❂	151		204	❸		
046	✎	099	❃	152		205	❹		
047	✏	100	❄	153		206	❺		
048	✐	101	❅	154		207	❻		
049	✑	102	❆	155		208	❼		
050	✒	103	❇	156		209	❽		
051	✓	104	❈	157		210	❾		
052	✔	105	❉	158		211	❿		
053	×	106	❊	159		212	➔		

Special Character Names for WIN88591

Name	Symbol	Name	Symbol	Name	Symbol
Aacute	Á	endash	–	ordmasculine	º
aacute	á	Eth	Ð	oslash	ø
acircumflex	â	eth	ð	Oslash	Ø
Acircumflex	Â	euro		otilde	õ
acute	´	exclamdown	¡	Otilde	Õ
Adieresis	Ä	florin	ƒ	paragraph	¶
adieresis	ä	germandbls	ß	periodcentered	·
ae	æ	guillemotleft	«	perthousand	‰
AE	Æ	guillemotright	»	plusminus	±
Agrave	À	guilsinglleft	‹	questiondown	¿
agrave	à	guilsinglright	›	quotedblbase	„
Aring	Å	hyphen	-	quotedblleft	“
aring	å	Iacute	Í	quotedblright	”
atilde	ã	iacute	í	quoteleft	‘
Atilde	Ã	icircumflex	î	quoteright	’
brokenbar	¦	Icircumflex	Î	quotesinglbase	,
bullet	•	Idieresis	Ï	registered	®
caron	ˇ	idieresis	ï	Scaron	Š
ccedilla	ç	igrave	ì	scaron	š
Ccedilla	Ç	Igrave	Ì	section	§
cedilla	¸	logicalnot	¬	sterling	£
cent	¢	macron	¯	Thorn	Þ
circumflex	ˆ	mu	µ	thorn	þ
copyright	©	multiply	×	threequarters	¾
currency	¤	ntilde	ñ	threesuperior	³
dagger	†	Ntilde	Ñ	tilde	˜
daggerdbl	‡	Oacute	Ó	twosuperior	²
degree	°	oacute	ó	uacute	ú
dieresis	¨	ocircumflex	ô	Uacute	Ú
divide	÷	Ocircumflex	Ô	ucircumflex	û
Eacute	É	odieresis	ö	Ucircumflex	Û
eacute	é	Odieresis	Ö	Udieresis	Ü
Ecircumflex	Ê	oe	œ	udieresis	ü
ecircumflex	ê	OE	Œ	Ugrave	Ù
Edieresis	Ë	Ograve	Ò	ugrave	ù
edieresis	ë	ograve	ò	Yacute	Ý
egrave	è	onehalf	½	yacute	ý
Egrave	È	onequarter	¼	Ydieresis	Ÿ
ellipsis	…	onesuperior	¹	ydieresis	ÿ
emdash	—	ordfeminine	ª	yen	¥

Special Character Names for WIN88592

Name	Symbol	Name	Symbol	Name	Symbol
Aacute	Á	ellipsis	…	perthousand	‰
aacute	á	emdash	—	plusminus	±
Abreve	Ă	endash	–	quotedblbase	„
abreve	ă	eogonek	ę	quotedblleft	“
Acircumflex	Â	Eogonek	Ę	quotedblright	”
acircumflex	â	Eth	Đ	quoteleft	‘
acute	´	eth	ð	quoteright	’
acute	ˊ	euro		quoteright	’
Adieresis	Ä	germandbls	ß	quotesinglbase	,
adieresis	ä	guilsinglleft	‹	racute	ŕ
aogonek	ą	guilsinglright	›	Racute	Ŕ
Aogonek	Ą	hungarumlaut	˝	Rcaron	Ř
breve	˘	hungarumlaut	˝	rcaron	ř
breve	˘	hyphen	-	registered	®
brokenbar	¦	iacute	í	ring	˚
bullet	•	Iacute	Í	Sacute	Ś
cacute	ć	Icircumflex	Î	sacute	ś
Cacute	Ć	icircumflex	î	scaron	š
caron	ˇ	Lacute	Ĺ	Scaron	Š
caron	ˇ	lacute	ĺ	scedilla	ş
Ccaron	Č	logicalnot	¬	Scedilla	Ş
ccaron	č	lquoteright	ľ	Tcaron	Ť
ccedilla	ç	Lquoteright	Ľ	tcedilla	ţ
Ccedilla	Ç	lslash	ł	Tcedilla	Ţ
cedilla	¸	Lslash	Ł	tquoteright	ť
cedilla	¸	mu	µ	uacute	ú
copyright	©	multiply	×	Uacute	Ú
currency	¤	Nacute	Ń	Udieresis	Ü
dagger	†	nacute	ń	udieresis	ü
daggerdbl	‡	ncaron	ň	Uhungarumlaut	Ű
Dcaron	Ď	Ncaron	Ň	uhungarumlaut	ű
degree	°	Oacute	Ó	Uring	Ů
dieresis	¨	oacute	ó	uring	ů
divide	÷	ocircumflex	ô	yacute	ý
dotaccent	˙	Ocircumflex	Ô	Yacute	Ý
dquoteright	ď	Odieresis	Ö	Zacute	Ź
eacute	é	odieresis	ö	zacute	ź
Eacute	É	ogonek	˛	Zcaron	Ž
Ecaron	Ě	ogonek	˛	zcaron	ž
ecaron	ě	Ohungarumlaut	Ő	zdotaccent	ż
edieresis	ë	ohungarumlaut	ő	Zdotaccent	Ż
Edieresis	Ë	periodcentered	·		

Special Character Names for WIN88599

Name	Symbol	Name	Symbol	Name	Symbol
Aacute	Á	Egrave	È	onequarter	¼
aacute	á	ellipsis	…	onesuperior	¹
acircumflex	â	emdash	—	ordfeminine	ª
Acircumflex	Â	endash	–	ordmasculine	º
acute	´	euro		Oslash	Ø
adieresis	ä	exclamdown	¡	oslash	ø
Adieresis	Ä	florin	*ƒ*	Otilde	Õ
ae	æ	gbreve	ğ	otilde	õ
AE	Æ	Gbreve	Ğ	paragraph	¶
Agrave	À	germandbls	ß	periodcentered	·
agrave	à	guillemotleft	«	perthousand	‰
Aring	Å	guillemotright	»	plusminus	±
aring	å	guilsinglleft	‹	questiondown	¿
Atilde	Ã	guilsinglright	›	quotedblbase	„
atilde	ã	hyphen	-	quotedblleft	“
breve	˘	iacute	í	quotedblright	”
brokenbar	¦	Iacute	Í	quoteleft	‘
bullet	•	Icircumflex	Î	quoteright	’
caron	ˇ	icircumflex	î	quotesinglbase	,
ccedilla	ç	idieresis	ï	registered	®
Ccedilla	Ç	Idieresis	Ï	scaron	š
cedilla	¸	Idotaccent	İ	Scaron	Š
cedilla	¸	igrave	ì	scedilla	ş
cent	¢	Igrave	Ì	Scedilla	Ş
circumflex	ˆ	logicalnot	¬	section	§
copyright	©	macron	¯	sterling	£
currency	¤	mu	µ	threequarters	¾
dagger	†	multiply	×	threesuperior	³
daggerdbl	‡	ntilde	ñ	tilde	˜
degree	°	Ntilde	Ñ	twosuperior	²
dieresis	¨	Oacute	Ó	Uacute	Ú
divide	÷	oacute	ó	uacute	ú
dotaccent	˙	Ocircumflex	Ô	ucircumflex	û
dotlessi	ı	ocircumflex	ô	Ucircumflex	Û
Eacute	É	odieresis	ö	Udieresis	Ü
eacute	é	Odieresis	Ö	udieresis	ü
ecircumflex	ê	oe	œ	ugrave	ù
Ecircumflex	Ê	OE	Œ	Ugrave	Ù
Edieresis	Ë	ograve	ò	Ydieresis	Ÿ
edieresis	ë	Ograve	Ò	ydieresis	ÿ
egrave	è	onehalf	½	yen	¥

Special Character Names for ISO88591

Name	Symbol	Name	Symbol	Name	Symbol
Aacute	Á	Egrave	È	ograve	ò
aacute	á	egrave	è	onehalf	½
Acircumflex	Â	emdash	—	onequarter	¼
acircumflex	â	endash	–	onesuperior	¹
acute	´	eth	ð	ordfeminine	ª
acute	´	Eth	Ð	ordmasculine	º
Adieresis	Ä	euro		oslash	ø
adieresis	ä	exclamdown	¡	Oslash	Ø
ae	æ	germandbls	ß	Otilde	Õ
AE	Æ	grave	`	otilde	õ
agrave	à	guillemotleft	«	paragraph	¶
Agrave	À	guillemotright	»	periodcentered	·
aring	å	hungarumlaut	˝	plusminus	±
Aring	Å	hyphen	-	questiondown	¿
atilde	ã	iacute	í	registered	®
Atilde	Ã	Iacute	Í	ring	°
breve	˘	Icircumflex	Î	section	§
brokenbar	¦	icircumflex	î	sterling	£
caron	ˇ	Idieresis	Ï	thorn	þ
Ccedilla	Ç	idieresis	ï	Thorn	Þ
ccedilla	ç	igrave	ì	threequarters	¾
cedilla	¸	Igrave	Ì	threesuperior	³
cent	¢	logicalnot	¬	tilde	˜
circumflex	ˆ	macron	¯	twosuperior	²
copyright	©	mu	µ	Uacute	Ú
degree	°	multiply	×	uacute	ú
dieresis	¨	Ntilde	Ñ	Ucircumflex	Û
divide	÷	ntilde	ñ	ucircumflex	û
dotaccent	˙	Oacute	Ó	Udieresis	Ü
dotlessi	ı	oacute	ó	udieresis	ü
eacute	é	Ocircumflex	Ô	ugrave	ù
Eacute	É	ocircumflex	ô	Ugrave	Ù
ecircumflex	ê	odieresis	ö	Yacute	Ý
Ecircumflex	Ê	Odieresis	Ö	yacute	ý
edieresis	ë	ogonek	˛	ydieresis	ÿ
Edieresis	Ë	Ograve	Ò	yen	¥

Special Character Names for ISO88592

Name	Symbol	Name	Symbol	Name	Symbol
Aacute	Á	emdash	—	quoteright	’
aacute	á	endash	–	Racute	Ŕ
abreve	ă	eogonek	ę	racute	ŕ
Abreve	Ă	Eogonek	Ę	Rcaron	Ř
Acircumflex	Â	Eth	Đ	rcaron	ř
acircumflex	â	eth	ð	ring	˚
acute	´	euro		Sacute	Ś
Adieresis	Ä	germandbls	ß	sacute	ś
adieresis	ä	hungarumlaut	˝	Scaron	Š
Aogonek	Ą	Iacute	Í	scaron	š
aogonek	ą	iacute	í	scedilla	ş
breve	˘	icircumflex	î	Scedilla	Ş
Cacute	Ć	Icircumflex	Î	Tcaron	Ť
cacute	ć	Lacute	Ĺ	tcedilla	ţ
caron	ˇ	lacute	ĺ	Tcedilla	Ţ
Ccaron	Č	lquoteright	l’	tquoteright	t’
ccaron	č	Lquoteright	Ľ	uacute	ú
ccedilla	ç	lslash	ł	Uacute	Ú
Ccedilla	Ç	Lslash	Ł	udieresis	ü
cedilla	¸	multiply	×	Udieresis	Ü
Dcaron	Ď	Nacute	Ń	uhungarumlaut	ű
degree	°	nacute	ń	Uhungarumlaut	Ű
dieresis	¨	Ncaron	Ň	uring	ů
divide	÷	ncaron	ň	Uring	Ů
dotaccent	˙	Oacute	Ó	yacute	ý
dquoteright	d’	oacute	ó	Yacute	Ý
eacute	é	Ocircumflex	Ô	Zacute	Ź
Eacute	É	ocircumflex	ô	zacute	ź
Ecaron	Ě	odieresis	ö	Zcaron	Ž
ecaron	ě	Odieresis	Ö	zcaron	ž
edieresis	ë	ogonek	˛	zdotaccent	ż
Edieresis	Ë	ohungarumlaut	ő	Zdotaccent	Ż
		Ohungarumlaut	Ő		

Special Character Names for ISO88599

Name	Symbol	Name	Symbol	Name	Symbol
Aacute	Á	Edieresis	Ë	ograve	ò
aacute	á	egrave	è	Ograve	Ò
Acircumflex	Â	Egrave	È	onehalf	½
acircumflex	â	emdash	—	onequarter	¼
acute	´	endash	–	onesuperior	¹
acute	ˊ	euro		ordfeminine	ª
adieresis	ä	exclamdown	¡	ordmasculine	º
Adieresis	Ä	Gbreve	Ğ	Oslash	Ø
AE	Æ	gbreve	ğ	oslash	ø
ae	æ	germandbls	ß	otilde	õ
agrave	à	guillemotleft	«	Otilde	Õ
Agrave	À	guillemotright	»	paragraph	¶
aring	å	hungarumlaut	˝	periodcentered	·
Aring	Å	hyphen	-	plusminus	±
Atilde	Ã	iacute	í	questiondown	¿
atilde	ã	Iacute	Í	quoteright	’
breve	˘	icircumflex	î	registered	®
brokenbar	¦	Icircumflex	Î	ring	˚
caron	ˇ	idieresis	ï	scedilla	ş
ccedilla	ç	Idieresis	Ï	Scedilla	Ş
Ccedilla	Ç	Idotaccent	İ	section	§
cedilla	¸	igrave	ì	sterling	£
cedilla	¸	Igrave	Ì	threequarters	¾
cent	¢	logicalnot	¬	threesuperior	³
copyright	©	macron	¯	twosuperior	²
degree	°	mu	µ	Uacute	Ú
dieresis	¨	multiply	×	uacute	ú
divide	÷	Ntilde	Ñ	Ucircumflex	Û
dotaccent	˙	ntilde	ñ	ucircumflex	û
dotlessi	ı	oacute	ó	Udieresis	Ü
eacute	é	Oacute	Ó	udieresis	ü
Eacute	É	ocircumflex	ô	ugrave	ù
Ecircumflex	Ê	Ocircumflex	Ô	Ugrave	Ù
ecircumflex	ê	odieresis	ö	ydieresis	ÿ
edieresis	ë	Odieresis	Ö	yen	¥
		ogonek	˛		

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z