ccount syntax reference and users manual

Users manual version 02.20.06

0. Preface – About this manual

0.1 Contents and format

This **ccount** manual and reference guide is designed to give the novice user a basis for writing specs and performing standard data manipulation and table generation. This manual includes descriptions of the major features of **ccount**, with many examples. These examples, as well as notes, warnings, programming hints, design suggestions, and **ccount** syntax information are contained in both single-line and double-line bordered sections in italics.

While this document is by no means a complete manual to every nuance of **ccount**, it should be a fairly comprehensive reference for all of the main and most often used features available. This will be a work in progress, but if you should notice any errors or omissions, or have any questions, comments, or concerns, please do no hesitate to contact the author at the email address: **mattski@pan-data.org**. Great effort will be made to reply to your questions/suggestions in a timely manner.

0.2 Examples, Illustrations, and Notes - how to read them

Throughout this document, examples will be used to display and illustrate the generic construction of syntax. Below is an example of a generic command, which illustrates the generic form of the syntax, followed by options. Any text contained within brackets is optional, and may or may not be an actual part of the command syntax. Brackets may also illustrate additional options or syntax.

command [optional] option1, [option2],...

Figure 0.2a Basic syntax example.

Attribute, environment and/or feature examples will be contained in single-line bordered boxes, and should further illustrate the abilities of **ccount**, or give specific syntax examples

feature example
feature1
feature2

Figure 0.2b Basic feature example.

Specific syntax examples will be offset from the main text using a separate font (`Courier New`) which will emulate a text editing environment. Tips, tricks, hints and warnings will be offset from the normal reference text using offset bars as shown below. These tips give valuable information, and can be useful in writing easy-to-read and error-free code. The tricks and shortcuts mentioned can help to save valuable time in programming, as well as provide warnings which can help the programmer avoid the most common pitfalls. An example of this is shown below:

*Compiled files are overwritten by **ccount** during each new run sequence. For this reason, it is imperative that the file extensions ".c", ".obj", ".h", and ".inf" are **NEVER** utilized for any user-defined files. Similarly, "out1", "out2", "lst_", and "hct_" should NEVER be used. The probability that these files will be overwritten by **ccount** is high, which could result in the loss of specs.*

1. Overview - basic ccount features

1.1 What is ccount?

ccount is a free package for market research / marketing research data editing, weighting, cleaning, manipulation, output, cross tabulation and data analysis. It is similar to, and uses the same syntax and data format as the SPSS-MR "Quantum" software package, which is a well known commercial package for processing market research data.

- Data Editing:
  - Recoding and manipulation of existing data.
  - Creating new data based on existing data.
- Data Cleaning:
  - Checking and validating data.
- Data Weighting:
  - Applying weights to data in order to balance the sample to reflect a particular population.
  - Adjusts the value (or count) of each respondent as accumulated on tables.
- Producing output:
  - Holecount distribution(s).
  - Frequency distribution(s).
  - New/Revised data files.
  - Report-type files (abbreviated data files and/or data calculations).
- Creating tables:
  - Create standard marketing research tables.
  - Create HTML style tables.
- Converting data.
  - Create csv (comma separated value) or other delimited-type files.

*Figure 1.1a Abilities of **ccount**.*

ccount is a non-GUI, console (DOS) based application and is distributed under the terms of the **ccount** Freeware License. **ccount** is an ongoing work in-progress. It should not be considered a finished product in any sense, and will undergo frequent upgrades. In order to download the latest version of **ccount**, or to view the latest documentation and change logs, please visit the **ccount** homepage at:

http://www.ccount.org

The **ccount** Data Cleaning and Analysis package for Market Research Data is distributed as freeware. Your use of this software indicates your acceptance of the license agreement shown in figure 1.1b, or available on the world-wide web at:

http://www.ccount.org/licenses/licensef.html

You should carefully read the following terms and conditions before using this software. Your use of this software indicates your acceptance of this license agreement.

This version of **CCOUNT** is Freeware Copyright © Volker Hoffmann 1992-2005,
CCOUNT weighting engine Copyright © René Donner 2004-2005,
CCOUNT documentation Copyright © Matthew Skawinski 2004-2005.

YOU MAY:
1. Install and use an unlimited number of copies of **CCOUNT** Freeware.

YOU MAY NOT:
1. Decompile, disassemble, reverse engineer or modify the executables or any portion of it.
2. Distribute **CCOUNT** or any portion of it in any form except under license by the copyright holders.
3. No part of the **CCOUNT** software may be disclosed or transmitted to others in any form or by any means except under license by the copyright holders.

Disclaimer of Warranty

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

*Figure 1.1b **ccount** freeware license agreement.*

1.2 Running ccount

In order to activate the **ccount** program, simply specify the run file and data file with which you will be working.

```
ccount run_file data_file
```

*Figure 1.2a Basic **ccount** execution syntax.*

This invokes the **ccount** program, compiles the **ccount** statements contained within your run file, and executes the statements specified in the run file upon the specified data file.

There are several stages contained within a standard **ccount** run. Each time **ccount** is executed the program file (run_file) is compiled and translated to a more machine-language friendly format. The computer uses this compiled program file for use with the specified data file. The compiled program is then run against the specified data file. During the run, the data file is read and passed through two sections of the run file – the edit section and the tabulation section. The edit section can validate data, recode existing data, create new data and/or variables, and output various file formats. The tabulation section produces tables, in a variety of formats, including standard format, semicolon separated format, and html format.

There are also a number of command line options available in **ccount**, which extend the capabilities of the standard execution syntax above. These include, but are not limited to, compiling the program file only and re-running the program file without re-compiling. These options are shown in Figure 1.2b below.

ccount command line options

```
/* compile program file (runfile) only (check for syntax errors)
ccount -c run_file


/* run specfile (program file) against data (datafile)
ccount run_file data_file


/* run existing compiled program against datafile without re-compile
ccount -r data_file


/*create executable program (ccprog_) only
ccount -o run_file
```

*Figure 1.2b Additional **ccount** command-line syntax.*

There are also a number of options for use with the **ccount** table generation engine, and are specific to table generation. The table generation is usually referred to as the ccout portion of the run. These ccout options and their specific syntax are listed in Figure 1.2c below. Table generation is discussed in detail beginning with chapter 10.

"**ccout**" table output engine command line options

```
/* generate semicolon separated tables
ccout -g


/*generate semicolon separated tables, no table identifier
ccout -gi


/* generate html tables (all pages are in one file)
ccout -h


/* generate html tables (each page is a new file)
ccout -hs


/* generate html tables with index page and navigation links
ccout –hn
```

```
/*generate html tables with index page and navigation links
/* (suppress flt text column)
ccout -hnf


/*generate html tables with index page and navigation links
/* (suppress table identifier column)
ccout -hni


/*generate html tables with index page and navigation links
/* (suppress flt text and table identifier columns)
ccout -hnfi
```

Figure 1.2b Additional ccout table output engine command-line syntax

1.2.1 Cleaning up after a ccount run

In order to delete all temporary files that are compiled during the **ccount** run, use the command:

```
cclean [-y/n]
```

Figure 1.2.1a Basic cclean syntax.

This command will prompt you to delete the standard (user-created) output files that are created during your **ccount** run, but will silently delete all of the program generated compiled files that are created. An example of the output produced by cclean is shown below in Figure 1.2.1b. As seen in the example, user confirmation is needed to delete (or not delete) any of the specified files. For a list of output files created by **ccount**, please refer to section 1.3.

```
C:\JOBS\963>cclean
C:\JOBS\963\out1, Delete (Y/N)? Y
C:\JOBS\963\out2, Delete (Y/N)? n
C:\JOBS\963\sum_, Delete (Y/N)? y
C:\JOBS\963\lst_, Delete (Y/N)? N
C:\JOBS\963\hct_, Delete (Y/N)? n

C:\JOBS\963>
```

Figure 1.2.1b cclean command output example.

As can be seen, cclean is case-insensitive. This means that confirmation for deletion can be either uppercase or lowercase. If any other letters are used than the expected **Y** (or **y**) or **N** (or **n**), the cclean routine will continue to prompt for the current file until an acceptable response is given by the user.

In order to suppress the prompts, simply use the **–y** option after the cclean command. This option will silently delete all program generated compiled files, including the standard user-created output files **out1**, **out2**, **sum_**, **lst_**, and **hct_**. An example of this command is shown below in Figure 12.1c.

```
C:\JOBS\963>cclean -y
C:\JOBS\963>
```

Figure 1.2.1c cclean –y command output example.

In order to silently delete all of the compiled files, but keep all of the user-generated files (**out1**, **out2**, **sum_**, **lst_**, and **hct_**), use the **–n** option after the command. Figure 12.1d shows an example of this option.

```
C:\JOBS\1143>cclean -n
C:\JOBS\1143>
```

Figure 1.2.1d cclean –n command output example.

1.3 Output files

During a typical **ccount** run, there are a number of compiled and user-defined files that are created. Compiled files will begin with "cc" or "ccc", and may end with any of a select number of file extensions. These extensions include, but are not limited to ".inf", "obj", ".c" and ".h". These compiled files are machine-language code that **ccount** creates while compiling the run file(s) of a standard **ccount** run, and are recreated and overwritten during subsequent **ccount** executions. There are also several output files which are created by **ccount** that are of specific value to the user. These files contain information about the compilation process, and may contain the results of reading the data, as well as tables and other data output. Figure 1.4 contains a listing and brief description of select output files.

ccount output files:

out1 = expanded program listing compiled from all spec files
- includes error messages (errors that are found in program files)

out2 = summary of data reads
- includes data of respondents that were rejected or failed any data validation conditions, as well as a summary of exceptions listing

out3 = table output summary

tab_ = tables file

hct_ = holecounts (marginals)
- raw or manipulated counts of the current c array

lst_ = frequency distributions with mean calculations (lists/arrays)
- raw or manipulated frequencies of the current c array

sum_ = ranked summary of data error messages
- includes defined variable information

weightrp = summary report of weighting results

*Figure 1.3a **ccount** compiled and user-defined output files.*

*Compiled files are overwritten by **ccount** during each new run sequence. For this reason, it is imperative that the file extensions ".c", ".obj", ".h", and ".inf" are **NEVER** utilized for any user-defined files. Similarly, "out1", "out2", "lst_", and "hct_" should NEVER be used. The probability that these files will be overwritten by **ccount** is high, which could result in the loss of specs.*

1.4 Input files

ccount run files can be created in any software package, but notepad or Vim is the suggested environment for use. These programs make it easier to edit files while in the DOS environment. Notepad is included in all windows environment packages, and is a fairly straightforward text editor, with minimal bells and whistles that may detract the user. For those users who desire a more powerful text editing tool, Vim is available, which is a DOS-based GUI version of the vi editor that is found on most Unix environments. For more information on Vim, as well as downloads, please visit

http://www.vim.org

For ease of execution, all spec input files must reside in the same directory as the data file. **ccount** will create the output files described in Figure 1.3a in the same directory as it is executed.

2. Getting started - workings of a standard ccount run

One basic requirement of a standard **ccount** run is a **data** file which **ccount** must read and act upon. **ccount** has the ability to read many different types of data files, depending on the specifications written in the run file. Another basic requirement is the **run** file, which contains the instructions and syntax for reading and processing the data file.

2.1 Data file(s)

Data can be defined as factual (sometimes numerical) information (as measurements or statistics) used as a basis for reasoning, discussion, or calculation. There are two (2) main formats of data: *flat ASCII* and *card* data. **ccount** can handle both of these formats easily, with little or no change in syntax referencing.

2.1.1 Flat ASCII data file(s)

ASCII stands for The American Standard Code for Information Interchange. This file format is a standard seven-bit code that was proposed by the American National Standards Institute in 1963, and finalized in 1968. ASCII was mainly established to achieve compatibility between various types of data processing equipment.

Flat ASCII data usually refers to a data file format where each respondent's data appears on only ONE line of the data file, and is referenced by a unique ID number. The data are also only single-punched in nature. That is, each column contains only one code.

Flat ASCII data files:

- One record per respondent (each line is a new record/respondent).
- Each record is identified by a unique respondent ID, or serial number.
- Columns are referenced by position (first position is column 1, referenced as c1).

Figure 2.1,1a Characteristics of flat ASCII data (single-card data).

2.1.2 Card-formatted data file(s)

Card data usually refers to the data format that can trace its roots back to the 1890 census. Originally, punch cards were a way for machinery to receive instruction. Machines in many industries would use the holes punches in cardstock (or metal sheets) to control program flow and execution. While technology has moved on since then, the idea of the punch card remains.

In 1928, one of the standard punch cards produced by IBM was 80 columns wide with 12 available "punches" per column. This maximized the number of punches available on any given card, and allowed the user to specify more than one punch value per column of instruction.

This 80-column card-format carries on to the modern day data processor. Card-formatted data usually refers to a data format which includes a unique respondent ID and a card number on each line. This way, respondents' data can be separated not only by the respondent number, but by the card number as well. Respondents can have more than one card present in the data file, and not all respondents are required to have all cards.

Lines in the card-formatted data file are limited to 80 columns, and each column can be multi-punched (each column can contain more than one code). Figure 2.1.2a further highlights the characteristics of card data.

Card-formatted data files:

- One record can span many lines of data.
- Each record is identified by a serial number and card number.
- Columns are referenced first by card number, followed by column number (card 1, column 1 is referenced as c101).

Figure 2.1.2a Characteristics of card-format data (multi-card data).

The main differences in these 2 types of data is that *flat ASCII* data contains ONE record per respondent, while *card* data contains MULTIPLE records per respondent and is usually limited to 80 columns per card. **ccount**, however, can handle card lengths of up to 1000 columns.

2.2 How ccount reads data

A standard **ccount** run can be comprised of as little as two files: the **data** file, from which **ccount** will read factual information collected during the interviewing process, and the **run** file, which will tell **ccount** exactly how to manipulate and output the data that is being read

The **run** file should contain all the information **ccount** needs in order to read, manipulate, and output the **data** in the expected manner and format.

ccount reads data one record at a time using the specs detailed in the **edit** section of the **run** file. Although data can be manipulated "on the fly", the original data file is NEVER changed or altered in any way.

ccount reads data records into what is commonly referred to as the "c array". Each data record is read into this c array, and processed consecutively through all the commands stored in the run file. Once a record is passed through the edit section of the run file (where most, if not all, data manipulation will occur), that respondent will also be processed through the tabulation (or tab) section of the run file (if one exists). Once a record has been processed completely, **ccount** will clear the contents of the c array, and continue at the top of the run file once again with the next record in the data file that has been loaded into the c array. This process continues until all records of the data file have been processed.

3. Ingredients – basic elements of a standard ccount run file

3.1 Sections of a standard run file

A standard **ccount** run file is comprised of two main sections: the **edit** section and the **tabulation (tab)** section.

The **edit** section can check and validate data, generate holecounts, lists and reports, correct (force clean) data, produce new data files, and recode data and create new variables. The various commands and syntax for the edit section is explained in detail beginning with this chapter.

The **tabulation** section produces tables and performs statistical calculations *(NOTE: statistical calculations are currently under development).* Table generation is discussed in detail beginning with chapter 10.

Not all run files will contain a tabulation, or tab, section but all run files should contain an edit section.

3.2 Comments and line continuation

Displaying comments within spec files is a very useful tool, and good programming practice. Comments can help to document and/or describe a particular section of specs more clearly. This is particularly helpful when many people share spec files. Comments can help to make code more readable.

Continuing a long line of specs is also helpful in making files more readable. Rather than allowing the editor to wrap the text to the next line, an artificial continuation within a long line of specs can help to break the spec up into more readable and manageable parts without sacrificing functionality of the program.

3.2.1 Comments

To leave "notes" or comments within any section of the run file that will be ignored by **ccount**, begin each line of the comment with a capital C, or a forward slash and a star (asterisk).

```
/* comment_text
```

Figure 3.2.1a Generic construction of comments using standard C syntax.

```
C comment _text
```

Figure 3.2.1b Generic construction of comments using “C”.

It is always a good idea to leave comment statements in your program files for documentation purposes.

Standard programming practices dictate that starting each run file with comment statements containing information about the file increases program clarity. Comments also help to separate sections within the run file, and increase program readability for the user or users.

While both applications of displaying comments are valid and acceptable, it is good programming practice to remain consistent within each file. That is, once a style of comment is used, that same style should be used throughout the file(s) for consistency.

3.2.2 Line continuation

In order to continue a long line, or to split up commands in such a way as to increase readability, plus signs may be used. Use a single plus sign, **+**, when continuing a line between keywords or a logical statement break. Use a double plus sign, **++**, when continuing a line between expressions.

```
statement
 +[+]continuation
```

Figure 3.2.2a Example of continuation.

Trying to split a line with continuation statements can result in invalid syntax, thus causing syntax errors. Always be sure to break lines in appropriate places, and do not split keywords, column or variable references that result in meaningless code.

3.3 The struct statement

The **data structure** (**struct**) statement is used to define the type of records and the main attributes of those records present in the data file that will be read. Aside from comments, the struct statement MUST appear FIRST in any given run file.

```
struct;option1;option2;option3;[option4];[option5];...
```

Figure 3.3a Basic construction of the struct statement.

Each **struct** statement begins with the word **struct** followed by several options which define the type and format of the data file. All struct options appear on the same line, separated by semicolons (**;**). The basic construction of a struct statement appears in Figure 3.3a.

The **ccount** default command separator is a semicolon (**;**). When **ccount** "finds" a semicolon, it assumes that the current command has ended, and will treat subsequent text as a new command until it reaches the end of a line, or the next semicolon.

*All run files MUST contain a **struct** statement in order to ensure that **ccount** will read the data file correctly. Failure to define a **struct** statement can result in syntax errors.*

Following are the most common options for the **struct** statement, and a brief description of their purpose:

- **read** is the keyword that defines what type of data file is being used, with **0** identifying a single record (flat ASCII) data file, **2** being used to identify a multicard data file, and **3** for multicard records that do not contain a card number. In the case of **read=3**, card number is internally calculated by the number of consecutive records with the same respondent ID.

- **ser** defines where in the data file the **ser**ial number is located

- **reclen** keyword that defines the maximum length of any given record (default is ***reclen=100***)
  *NOTE: multicard records have a maximum limit of **reclen=1000***

If the reclen value is less than the maximum number of characters in any given record in the data file, that record will be truncated to the specified length and a warning message will be printed stating that there are truncated records.

---

*If the reclen value is greater than the maximum number of characters in any given record in the data file, **ccount** appends the correct number of blank columns needed (keep in mind that this may increase processing time depending on the availability of memory to allocate said number of columns).*

---

*NOTE: the following options are struct line options relating to **multicard records only**:*

- **crd** defines where in the data record the **c**a**rd** number is located

- **req** defines the card number(s) which are **req**uired to be present in each record

- **rep** is used for trailer card processing

- **max** defines the **max**imum number of cards that appear in any given record

Examples of common struct statements follow in Figure 3.3b and 3.3c:

```
struct;read=2;ser=c(1,5);crd=c(6,7);max=10;req=1
```

Figure 3.3b Example struct statement for card data.

The struct statement in Figure 3.3b above contains the following information:

- the data structure is card data (multicard records)
- serial number can be found in columns 1 through 5
- card number is located in columns 6 and 7
- maximum number of cards per record is 10
- card number 1 MUST exist for all records

```
struct;read=0;ser=c(1,5);reclen=2000
```

Figure 3.3c Example struct statement for flat ASCII data.

Figure 3.3c exemplifies a struct statement which defines:

- the data file as being flat ASCII data (one record per respondent)
- the respondent (serial) number in columns 1 through 5
- the maximum length of any given record as 2000 columns

3.4 The edit section

The **edit** section in the run file is where most or all of your data manipulation statements will occur. One must define the beginning and end of any **edit** section within the **ccount** run file. The keyword **ed** defines the beginning of the edit section, and the keyword **end** delineates the end of the **edit** section.

```
ed

edit_statements

end
```

Figure 3.4a Standard construction of the edit section.

Data can be manipulated in the **edit** section by adding, copying or deleting punches, and/or adding, copying or deleting entire columns or blocks of data. We can also output the data in several different forms for ease of checking. The main forms of output include generating holecounts (frequencies), lists (ordered array listings), reports (text-type output files), and of course, writing out new data files.

Edit statements use the standard **ccount** syntax, which is explained in detail in the following chapters.

3.4.1 The return statement

As mentioned earlier, **ccount** reads data records into the “c array”, and continues until the processing for the current record has ended. If there is no tab section to the run file, then the record finishes processing when it encounters the “end” statement of the edit section.

However, there may be some instances where it becomes necessary to artificially end the processing of a respondent record. The keyword **return** can be used to terminate processing the current record. If a **return** statement is encountered, **ccount** will clear the contents of the c array, and continue at the top of the run file once again with the next record in the data file that has been loaded into the c array. This is identical to the results when the record reaches the “end” statement of the edit section (when there is no tab section) or the end of the tab section (when there are no more tables to be tabbed for the current record).

An example of the return statement is shown in Figure 3.4.1a. In this example, the record being processed will be discarded after the first series of editing statements if the condition evaluates true. The editing statements contained after the conditional return statement are ONLY processed for those respondents who evaluate the return condition as false. Thus, all respondents will be evaluated on the first series of edit statements, but it is possible that a smaller number of respondents (or none at all) will be processed through the second series of editing statements because of the return statement. Conditional statements are further explained in section 6.1.

```
ed

edit_statements1

if (condition) return

edit_statements2

end
```

Figure 3.4.1a Standard construction of a return statement.

3.5 The tab section

The **tab** section of the run file is where all table formatting options and questions (axes) are defined. This section begins immediately after the edit section, and is opened with a standard "**a-line**" or "**a-card**" statement. This "**a-card**" includes global run conditions for the tables to be produced.

```
a; option1; option2; option3;[option4];[option5]; ...
```

Figure 3.5a Generic construction of the "a-line" or "a-card" .

The options found on the a-card are keywords that define the characteristics of the current run and output options for table formatting. Any number of keywords can be used, but all keywords must be separated by semicolons and must appear on the same a-line.

`/*example run file`
`struct;read=0;ser=c(1,10);reclen=400`
	opening comments
	struct statement

`ed`
 	
`end`
	edit section

`a;dsp;flush;nopc;notype;op=12;side=30;printz;`	a-line/a-card
`ttcstudy_title`	
`*include tabs;ban=ban1`	tab section

`*include axes`
`*include banner`
	axis and banner statements

Figure 3.5b Generic illustration of a run file.

The tab section defines table formats and content and allows table creation and manipulation. The tab section MUST be ordered in a set fashion. That is, the a-line must appear first, followed by the tab statements, which are used in the making of tables. The tab statements specify which axes are to be tabbed against each other in an ordered fashion. Lastly, the actual axes (axis definitions) must appear, which define the questions that are to be used in the tables (for both rows and columns).

All of these statements combined make up a standard **ccount** tab section, which is further explored and defined in detail in the following chapters.

4. Basic ccount syntax - variables and column arrays

The main syntax of **ccount**, referencing column arrays and variables, is fairly intuitive. While there are some differences depending on the length of the array and the type of variables used, the standard syntax is strikingly consistent.

4.1 Single column punch notation

In order to reference the c array (or any array for that matter), one only needs to specify the array name, followed by the element (or position number) in that array. Figure 4.1a shows the generic construction of the column/code references used in **ccount**.

c(column_number)'punch_value(s)'

Figure 4.1a Standard construction of column and code references.

For instance, column 350 in the c array would be referred to as:

```
c350
```

To refer to column 127 for flat ASCII data, the syntax is similar to the above:

```
c127
```

For muticard data, the syntax may look the same, although the meaning is different. Here, we refer to card 1, column 57:

```
c157
```

4.1.1 single column data values

Data are normally represented by numerical values, usually referred to as “punches”. These punches are contained within each column of data, and may be single-punches (one value per column), or multi-punches (multiple codes per column).

Data punches:

- Represent one column or data position.
- Can be a singly-coded or multi-coded (one punch per column, or many punches per column, respectively).
- Are referenced in **ccount** by using single quotes (ex: '1', '2', '45').
- In a range can be specified with a slash (/) (ex: '1/5', '3/90').
- A blank, or null, punch is denoted by a space within single quotes or an empty set of quotes (' ', '').

Figure 4.1.1a Single punch characteristics.

Always use valid numerical codes (punches) within quotes as some alphabetical and special characters can be interpreted as multi-punch data.

ccount recognizes twelve (12) standard punches per column as well as blanks (or nulls). These twelve (or thirteen, including the null) standard punches are the values from zero to 9 and the codes **–** and **&** (sometimes called the X and Y punch, respectively).

To specify an either/or punch, enclose both in quote marks. The following example means that column 1123 is either a punch 1 OR a punch 2.

```
c1123'12'
```

For a range of punches, use a slash (***/***) between the first and last values in the range. This is read as column 57, punches 1 through 9:

```
c57'1/9'
```

If a multipunch reference is desired, then the punches enclosed in quote marks must be preceded by an equals sign:

```
c1123='12'
```

The above example translates as “column 1123 is BOTH a punch 1 AND a punch 2”.

In order to specify that a column does NOT contain a particular punch, simply place a “n” after the column reference, and before the punch notation.

```
c507n'7'
```

The above statement simply specifies that column 507 is not a punch 7.

To specify a blank punch, or a column which has no punches, simply use a blank space between the quote marks, or just two consecutive quote marks:

```
c299=' '
```

```
c299=''
```

Both examples above imply that column 299 is blank, or contains no punches.

Note that with card data, valid column references begin with c101 (card 1, column1), while flat ASCII data begins column referencing at c1 (column 1)

4.2 Column and value notation

To specify a range or series of columns, enclose the first column and last column in parentheses.

c(start_column,end_column)

Figure 4.2a Standard construction of column integer value range specification.

Thus, a value stored in more than one column (multi-column value) is referenced by specifying only the first and last column in the series. The syntax to reference a value contained within columns 101 through 103 would be written as the following:

```
c(101,103)
```

A multi-column value stored in columns 345 through 350 would be written

```
c(345,350)
```

Values stored within a series of columns are no longer referred to as punches, but must be considered as actual values. In order to reference values within a range of columns, logic connectors must be used, since punch notation does not apply. Logic connectors are explained in detail in chapter 4.5.

Attempting to use punch notation in the context of a multi-column value reference will result in syntax errors.

Sometimes it may become necessary to read real numbers from columns. In that case, we must use the column code reference shown in Figure 4.4b below.

cx(start_column,end_column)

Figure 4.2b Standard construction of column real value specification.

Using the notation above informs **ccount** that the values contained within the columns specified should be evaluated as a real number. In this case, **ccount** will interpret any periods, or dots, it may find in these columns as a proper decimal place rather than a multipunch data code. This type of notation is important when dealing with monetary values contained within the data, or when dealing with weights. Both of these topics are discussed in greater detail later in this manual.

4.2.1 Multi-column data values

When data values greater than nine are needed, multiple columns must be used to store these values. **ccount** recognizes an unlimited amount of data values stored in multiple columns. These values can be numerical (including decimal values), or can be of a type text, or string, nature.

Numerical or Alphanumerical Values:

- Represent two or more columns or data positions.
- Are also referred to as “strings”, “literals”, “literal strings” or “string literals”.
- Numerical values can be referenced individually, or as inclusive values using a **colon** operator (ex: (1,2,3,4) , (1:4)).
- String values must be enclosed within dollar signs (ex: $**ccount**$).
- Blank, or null, values are denoted by spaces within a set of dollar signs ($ $).
- Blank values can be part of the multi-column value when they are leading or trailing spaces.
- The number of characters and/or blank spaces within dollar signs must be equal to the total number of columns specified.

Figure 4.2.1a Multi-column value characteristics.

String constants, similar to punches, must be referenced with a delimiter. However, this delimiter is a dollar sign ($) rather than a single quote mark. Using the dollar sign delimiter references a literal string value. That is, the value contained within the columns must match EXACTLY to the string specified. This includes spaces and any punctuation, if applicable.

4.2.2 Evaluating values

When evaluating values, **ccount** looks at the entire column field before making the evaluation. There are some cases which may alter the perceived result. **ccount** will ignore blank spaces within column fields. This can cause codes separated by spaces to be evaluated as a single value. For instance, if column 811 contains a punch 2, and column 815 contains a punch 3, then **ccount** will evaluate the statement (assuming columns 812 through 814 are blank):

```
c(811,815)
```

as having a value of 23. since blanks are ignored, this is a perfectly acceptable result to the compiler. This result will be duplicated regardless of where the blanks occur within the range of columns, and whether or not they are together or if they straddle the punch values.

Similarly, any multi-column value will be evaluated as a negative value if a negative sign appears *anywhere* within that field. A negative sign negates the value contained within the multi-column field, regardless of where it appears in that field: at the beginning, in the middle, or at the end.

Lastly, there is a special case with regards to columns which contain null values, or are all blank. When evaluating single or multiple columns as a numerical value, **ccount** will say that they are equal to zero. Since there is no value contained within blank columns, they have no value, and are considered by **ccount** to be equivalent to a zero value. Columns which contain actual zeros or a zero punch are also considered to have no value, and also evaluate to zero. Since evaluating blanks as values can lead to incorrect results, it is important to always specify the difference between blank columns and columns which actually contain zeroes.

It is imperative that the user understand the format and type of data which is to be processed. Understanding the values contained within the data can help to eliminate many multi-column value data referencing errors.

Always reference zeroes contained within columns as string values, to eliminate value processing errors and to reduce the occurrence of incorrectly evaluating blank columns.

Evaluating multi-column values are explained in greater detail in section 4.5.

4.3 Assignment statements

Referencing values within a column or series of columns is an absolute necessity when dealing with data of any kind. Assigning values is also a requirement. By assigning new values to columns, it is possible to create data for use in additional analysis.

4.3.1 Assigning punches with the set statement

The assignment, or set, statement is very similar to standard punch specification as described above. However, to assign a punch, reference the punch value and surround it by single quotes preceded by an equals sign (**=**).

[emit] c(column_number)[=]’punch_value(s)’

Figure 4.3.1a Standard construction of column and code assignment statements.

Thus, assigning a punch 8 to column 482 would look like this:

```
c482='8'
```

To specify (and assign) a column which contains multipunches, enclose the punches in quote marks, but add an equal sign (see Figure 4.2):

```
c2005='14'
```

The above can be read as: column 2005 equals punches 1 AND 4.

---

*Assignment statements are normally destructive procedures. That is, assigning a punch to a column will overwrite any existing punches in that column. To assign punches that are NOT destructive, use the **emit** keyword.*

For single punch assignment, the equals sign (=) is optional. Thus, assigning a punch 8 to column 482 could also be written as:

```
c482'8'
```

To assign a value to a column **without** eliminating other punches that may be present, use the **emit** keyword. In order to add a punch 2 to column 413 without overwriting any other punches that may already be present in that column is as follows:

```
emit c413'2'
```

Trying to assign punch values with the emit statement before clearing out the destination column can result in erroneous data. Be sure that the destination column contains ONLY the data that is needed.

4.3.2 Assigning punches with the and, or and xor operators

Sometimes it may be necessary to check for specific values in a group of columns or for codes that are present in a series of columns. **ccount** provides several operators that can accumulate in different ways the punches that are present in a series of columns. these operators are known as the **and**, **or** and **xor** operators.

c(column_number)= operator(cX1, cX2, cX3, ...)

Figure 4.3.2a Generic construction of column assignment operators.

In order to use these operators, a column must be first selected that will be assigned (hold) the compacted columns’ codes. For instance:

```
c319 = or (c301,c302,c303)
```

assigns punches that are present in any of the columns 301, 302, or 303 to column 319.

Assignment Operators:

- and – assigns codes that are present in ALL specified columns.
- or – assigns codes that are present in ANY of the specified columns.
- xor – assigns codes that are present in ONLY ONE of the specified columns (collapses unique punches).

Figure 4.3.2b Assignment operators and their functionality.

Syntactically, the **and** and **xor** operators would be used in the same way as the **or** operator. The results, of course, would differ.

```
c998 = xor (c321,c322,c323)
```

Assuming column 321 contains a punch 1, column 322 contains a punch 2, and column 323 contains a punch 3, the xor operation would result in column 998 being multipunched as a ‘1’, ‘2’, AND ‘3’.

```
c999 = or (c475,c562,c705)
```

*Assignment operators in many cases can result in multi-punched columns. Be sure that you a**ccount** for this possibility when writing specs.*

4.4 Deleting punches/values and clearing columns

Not only does **ccount** allow the user the ability to create data, but gives the user the power to modify and/or delete data as well. This ability must be used with great care, and must be respected. Destroying data is risky business, and must be approached with care and concern to ensure that the integrity of the original data is maintained.

4.4.1 Deleting punches or values

In order to delete unwanted punches from a column, use the **delete** keyword. This is a destructive operation, but will only delete the punches specified in the statement.

*The **delete** keyword is only destructive to the punches specified, and will leave any multi-punched codes in the same column intact.*

delete c(column_number)’codes’

Figure 4.4.1a Standard construction of column and code delete statements.

In order to delete punch 2 from column 413 and **preserve** any other punches that may already be present in that column, use the following notation:

```
delete c413'2'
```

This will delete only the punch specified (punch 2) from column 413, leaving any other data punches that already exist intact.

4.4.2 Clearing punches or values

In order to clear an entire column, use the **clear** keyword along with the column reference. This will blank out the column specified, as in the example below:

```
clear c427
```

To clear out a series of columns, use the **clear** keyword in conjunction with a multi-column reference:

```
clear c(401,410)
```

Alternatively, specify the column reference followed by a blank, or empty punch notation. The following are all acceptable methods of clearing out a single column or a series of columns:

```
c350''

c350' '

c(121,124)' '
```

*Using the keyword **clear** or referencing a column followed by the empty punch notation results in a completely destructive event. Be very careful when clearing out columns or blocks of columns, which can result in loss of data.*

The **clear** keyword also works with defined variables of any type. Variables are discussed in detail in chapter 5.

4.5 Logic connectors

Connecting column and punch references together and/or referencing values in column ranges into a cohesive statement requires logic connectors. Logic connectors in **ccount** are surrounded by periods (**.**). Logic connectors allow for the joining of separate logical statements into larger logical expressions. These logical expressions become the backbone of **ccount** spec writing, and serve a multitude of used for checking data, adding/deleting data, and producing output files, including (but not limited to) tables.

Examples of some commonly used logic connectors and brief descriptions follow in the Tables below.

Specifying literal strings (also called “literals” or “column fields”) in **ccount** are similar to specifying punches, except that the string *MUST* be enclosed by the string delimiter (dollar) signs ($). For instance,

```
c(1201,1202)$TX$
```

refers to columns 1201 through 1202 containing the literal string TX.

In addition to the above connectors, there are also logic connectors for use with and specific to text strings. These are the string comparison equal operator (=) and the unequal string comparison operator (u).

Logic connectors:		
.and.	=	logic connector specifying mutually inclusive statements (logical and)
.or.	=	mutually exclusive logical or connector
.eq.	=	specifies an “equal to” value (logical equality)
.ne.	=	specifies a “not equal to” value (logical inequality)
.ge.	=	greater than or equal to
.gt.	=	greater than
.le.	=	less than or equal to
.lt.	=	less than
.in.	=	specifies a range value
.not.	=	negates any **ccount** logical statement

Figure 4.5a Logic connectors.

String comparison operators:		
=	=	string comparison operator (exact literal string match or assignment)
u	=	unequal string comparison (specific to string comparisons)

Figure 4.5b Logic connectors specific to string comparisons.

Following are several examples of referencing values in columns with logic connectors.

In order to specify that the value contained in columns 100 through 101 equals 10, write the following:

```
c(100,101).eq.10
```

To specify a range of values within a series of columns, simply use the.in. operator, and list the valid values contained in the column(s), as follows:

```
c(1305,1309).in.(1,4,5,12:99))
```

The above is read as “the value contained in columns 1305 through 1309 are in the range of 1, 4, 5, or 12 through 99.”

```
c(2501,2509).ne.5
```

The above is read as: “the value contained in 2501 through 2509 is not equal to 5.”

In order to negate an entire expression, place the .not. operator in front of the entire expression. The following shows how to negate the previous expression using the .not. operator. This statement reads: “the value contained in 2501 through 2509 is not equal to 5”

```
.not.(c(2501,2509).eq.5)
```

The next examples show how to link two separate logic statements with the .or. and .and. connectors, respectively. These are read as: “the value in columns 51 through 55 equals 10, or column 138 is a punch 1” and “columns 101 through 102 contain the literal text NY, and column 100 is a punch 1”, respectively.

```
c(51,55).eq.10.or.c138'1'
```

```
c(101,102)=$NY$.and.c100'1'
```

Using the string comparison operator “u” is very similar to using logic connectors. The statement below states that the string value that is contained within columns 1201 and 1202 is unequal to the literal string TX.

```
c(1201,1202)u$TX$
```

It is also important to note that using logic connectors and/or string comparison operators when comparing columns (or sets of columns) can have different results. When using logic connectors, the values contained within the columns will be evaluated differently by **ccount** depending on the syntax used. For instance, if columns 501 through 503 contain the literal string `$041$`, and columns 511 through 503 contain the literal string `$ 41$`, then the following statement would evaluate as false:

```
c(501,503).ne.c(511,513)
```

since both sides of the statement evaluate the values as 41. However, assuming the same case, the following statement would evaluate to true:

```
c(501,503)uc(511,513)
```

since the literal strings `$041$` and `$ 41$` are not identical.

It is important to realize that the equals string comparison operator, `=`, works in a similar way. Using the above example once again, the statement:

```
c(501,503)=c(511,513)
```

would also evaluate false because the items contained within the columns are not identical. Again, using logic operators (connectors) and evaluating these columns as values, rather then strings would lead to a true result:

```
c(501,503).eq.c(511,513)
```

When referencing fields that are all blank as a literal string, it is not necessary to specify a number of blank spaces that is consistent with the field width. The following references are equal:

```
c(320,324)$    $
```

```
c(320,324)$ $
```

Both of the above statements imply that all columns within the range specified contain nulls, or blanks. While the first example may seem more syntactically correct, the result is the same. Keep in mind that this “shortened” string notation only applies to column fields that contain ALL blanks. Blank columns are a special case and should always be treated as such.

Note that the “dot” logic operators cannot be used in conjunction with literal strings. To compare literal strings, use the string comparison operators, = and u.

The ***.not.*** *operator negates the ENTIRE expression which it precedes. Using this type of “not logic” can sometimes lead to unexpected results. When negating an entire expression, be sure that the outcome is consistent with expectations.*

5. ccount syntax – arithmetic operations and operators

While being able to reference single and multi-column values are usually enough for simple data extraction, it may become necessary to compute values “on the fly” during a **ccount** run. This chapter focuses solely on arithmetic operators, operations, and some additional variables that may be useful in computation.

5.1 Column-based arithmetic

As in most other languages, **ccount** allows you to form expressions using several types of variables and the standard arithmetic operators. standard arithmetic operations are valid syntax in **ccount**. The data contained in the c array may be referenced as numerical values, and arithmetic operations may be performed using the values contained in those columns.

Figure 5.1a shows the standard valid arithmetic operators.

+	-	addition
-	-	subtraction
*	-	multiplication
/	-	division

*Figure 5.1a Arithmetic operators in **ccount**.*

Valid arithmetic expressions may use one or more of these operators in any order within a given valid **ccount** syntax statement. Some examples follow:

```
c(101,105) * 2.5 / 3
```

```
c(397,398) + 1 * 4 – c(200,201)
```

In addition to calculation, **ccount** has the ability to fully parenthesize an expression. **ccount**, like other languages, will assume parenthesis based on standard arithmetic rules of precedence. For example:

```
t1 + 2 * c(100,101)
```

is evaluated by **ccount** as follows:

```
t1 + ( 2 * c(100,101) )
```

This makes sense, since standard arithmetic rules of precedence state that multiplication is evaluated before addition. Multiplication (*) takes precedence over all other operations, followed by division (/), addition (+), and subtraction (-). Figure 5.1b shows the operation precedence order as well.

Greatest precedence

*	-	multiplication
/	-	division
+	-	addition
-	-	subtraction

Least precedence

Figure 5.1b Arithmetic operators in order of precedence.

However, it sometimes becomes necessary within an arithmetic expression to specify which expressions are combined with each operator. Explicit use of parenthesis within an expression will override the implicit parenthesis determined by precedence. This is useful if the requested result requires a deviation from the standard precedence rules.

Again, standard arithmetic order of operation rules apply. **ccount** will first evaluate any expression found within parentheses, followed by the order of operations stated above in Figure 5.1b. Addition and subtraction operations are always evaluated last. If several operators are present which have equal precedence, they are evaluated left to right. In order to change precedence of any operation, enclose it in parenthesis.

ccount *will obey all standard order of operation rules within an arithmetic statement.*

If all values within an expression are whole integers, then the result will be an integer. If any of the values contained within the expression are real numbers, the result will be a real number. Of course, this result is also determined by the variable in which the result is stored, and by the order which **ccount** evaluates the expression.

ccount *always evaluates the right-hand side of an arithmetic expression before assigning values to the left-hand side.*

5.2 Integer and Real variables

Variables are useful in storing numeric values for use in data manipulation or for tabbing purposes. **ccount** offers many different types of variable options, including integer, real, and data-type variables. Each type of variable is useful in its own way, and all are explained in detail in the following sections.

Variables may be defined in several places in any given run file, with differing results. Figure 5.2a shows details on several places where variables may be defined and how each affects the variables characteristics and performance.

Named variables may be declared:

1. At the start of your program **before** the edit section *(before the ed statement)*. These variables are available in the edit section, tab section, and subroutines of the given program/run file. These variables may be modified within the edit section, or within a subroutine. These variables are referred to as “global” variables.

2. In the edit section **after** the *ed* statement. These variables are “local” variables only, and are not recognized by the section or subroutines. The edit section is the only place where these variables may be accessed.

Figure 5.2a Declaring named variables.

5.2.1 Integer variables

Integer values are also commonly referred to as “counting” numbers, or “whole” numbers. These values CANNOT be fractions or contain decimal values. These values may only be integer constants, but can originate from the c-array or may be calculated “on the fly” during the **ccount** run.

There are several ways to initialize and use integer variables in **ccount**. Figure 5.2.1a illustrates the standard notation for declaring user-defined integer variables. These variables can have any user-defined name and width. The “s” that follows the width of the variable array refers to the variables “size”.

While these variables can be defined before the edit section, they must be assigned values within the edit section of the run file. In order to reference these defined integer variables, simply append the cell (or array position) to the variable name, as seen in Figure 5.2.1b.

```
int [variable_name] [width][s]
```

Figure 5.2.1a Defining integer variables.

Variables should always be named meaningfully to avoid confusion within the run file, and to increase program clarity. Naming variables according to their use or purpose can help to avoid confusion.

```
int newvar 100s

newvar1=12
newvar(10)=4
```

Figure 5.2.1b Defining integer variables and assigning values.

If variables are defined with the trailing “s”, as in the example above (`int newvar 100s`), variables of this type may be referenced with or without parenthesis to denote their location in the variable array. For instance, newvar10 and newvar(10) are both acceptable references to the same newvar variable. However, if the trailing “s” is omitted, variables MUST be referenced using the standard parenthetic syntax (`newvar(10)`).

ccount creates 200 integer variables automatically available at the beginning of each **ccount** run. These variables do not have to be defined by the user, and are referenced using the lowercase letter “t” as the variable name. These 200 integer variables are initialized (given a value of zero) at the beginning of each **ccount** run, but are not incremented or reset for each data records processed.

For instance, the 5th element of the t-array would be referenced as: `t5`

```
t5=t5+1
```

would increment the t5 variable by one.

Remember that **ccount** will always evaluate the RHS (right-hand side) of an equation before assigning any value to the LHS of the expression. Thus, the above statement would add one to the current value of the **t5** variable, and then assign that value back to **t5**. This type of expression is very commonly used within looping structures.

Variable names must begin with a letter of the alphabet (no digits), and can be no more than 13 characters long.

Using an integer variable to store a real result will cause that result to be truncated. This could lead to incorrect reporting of data, or worse, loss of data. Be sure to always use correct variable types for the kind of arithmetic expression that is being evaluated.

A summary of integer variable characteristics follows.

Integer variables:

- Store whole, or counting numbers.
  - Minimum value is -2,147,483,648.
  - Maximum value is 2,147,483,648.
- Reference variables with variable name and cell (array position) number.
- Integer Variables store only ONE value per variable cell number/position.
- **ccount** default integer variables are named t.
  - 200 pre-defined t-variables are provided.
  - Initialized to 0 at the beginning of each **ccount** run.
  - Are not reset or artificially incremented with each consecutive data record.

Figure 5.2.1c Characteristics of integer variables.

5.2.2 Real variables

Similar to integer variables, real variables can be used to store values that can be used during a **ccount** run. Real variables are defined in a similar manner to integer variables, as seen in Figure 5.2.2a.

real [variable_name] [width]s

Figure 5.2.2a Defining real variables.

Real variables (or decimal values) can be stored in **ccount** with precision up to 6 significant digits. Variables assigned with more than 6 significant digits will be rounded accordingly. Referencing real variables is the same as referencing integer variables. An example of defining an assigning values to a real variable array called “money” can be found in Example 5.2.2b.

```
real money 100s
money5=12.99
money12=1042.27
money82=7.9995
```

Figure 5.2.2b Defining integer variables and assigning values.

For convenience, **ccount** also provides 100 pre-defined real variables for use. These variables are denoted by the lowercase letter “x” and are referenced in exactly the same way as the integer t variables. These real variables are also initialized (set to a value of 0.0) at the beginning of a **ccount** run, and are not reset or artificially incremented in any way for each data record. The following assigns a value of 1.53204 to the real variable x3.

```
x3= 1.53204
```

Real variables:

- Store whole, or counting numbers.
  - Accuracy is limited to 6 decimal places.
- Reference variables with variable name and cell (array position) number.
- Real Variables store only ONE real value per variable cell number/position.
- **ccount** default real variables are named x
  - 100 pre-defined x-array variables are provided.
  - Initialized to 0.0 at the beginning of each **ccount** run.
  - Are not reset or artificially incremented with each consecutive data record.

Figure 5.2.2c Characteristics of integer variables.

Integer values may be stored in real variables. In these cases, the zero following the decimal place would be an implied value, and is automatically appended to the defined variable.

5.3 Data variables

ccount provides the ability to create separate data arrays. Similar to the c-array that is the standard data array that is used in a **ccount** run, these arrays can be used to store data or parts of data, or to maintain separate data variables for use in calculation. Defining data variables are similar to defining other variables, as seen in Figure 5.2.3a.

Data variables provide powerful data manipulation abilities. Data variables can be used to store values or results of complex calculations without having to rely on sequential blank columns (available space) within the c-array.

data [array_name] [variable_width]s

Figure 5.3a Defining data variables.

As with all variables, they must be created before they are used. Referencing this new data array is identical to any c-array referencing, and must include the data array name, as well as the specific column reference. Figure 5.2.3b illustrates the specifics of defining a data variable called “water” that is 142 columns wide. This figure also illustrates valid examples of using this data variable.

```
data water 142s

water(4)=7
water4=7

water4'7'

water(2,3)=$NY$
```

Figure 5.3b Defining data variables and assigning values.

5.4 Counting punches

A **numb** statement is useful in counting the **numb**er of punches in a specified column or columns. The standard numb statement will return an integer value that is the total number of punches in the given column list, regardless of which punches are present. The result of a numb statement can then be used in any standard syntax for comparison purposes, but if the result is to be saved, it must be assigned to an integer variable or column.

numb(column1,[column2],[column3],...)

Figure 5.4a Generic construction of a numb statement.

In order to count the number of a particular set(s) of punches, add the specific codes to check to the column list.

numb(column1'codes',[column2'codes'],[column3'codes'],...)

Figure 5.4b Generic construction of a numb statement using punch specification.

For instance, checking to see how many codes are punched in columns 101 through 104 are written as follows:

```
numb(c101,c102,c103,c104)
```

While checking specific instances of punches in the same columns is written:

```
numb(c101'1',c102'2',c103'3',c104'4')
```

in order to save the result, use a column or variables to store the result:

```
c105 = numb(c101,c102,c103,c104)
```

Unlike column value references, the columns in which to count the number of codes MUST be explicitly listed. Defining only the start and end columns of a range will cause an incorrect integer result.

Also note that EACH column specified in the numb statement MUST have the preceding c to designate that data column as being from the c-array.

When storing the result of a numb statement into column variables, be sure that the number of columns allocated are sufficient to store the highest possible result.

6. ccount syntax – conditional statements, loops and routing

Program flow and the flow of control within a program is a major component of any programming language. **ccount** provides many different flow control constructs which help to further direct data through the program which is being processed. The ability to define the specific path each data record will take allows for program flexibility.

Based on the characteristics of the data file being processed, it may be necessary to conditionally select some records for specific programming statements, or route certain records around these statements so that they are not evaluated. These control structures are not limited to logical decision-making, but expand and include structures which allow for the execution of repetitive code quickly and easily.

6.1 Conditional (selection) statements

Conditional statements, or selection structures, are used when the path of execution needs to be redirected. With selection structures, the execution of a command can be "filtered" to execute only if certain conditions apply. The path that is chosen depends on whether the selection structure evaluates to truth or falsity.

These structures or conditions can be added in order to evaluate **ccount** expressions for a select group of people. The standard **if** statement is one of the most commonly used conditional statements. This statement is sometimes referred to as a conditional statement, a filter, or a "SIFT". The word SIFT is actually an acronym for "Select If True". This acronym accurately describes the functionality of the conditional statement in that respondents are selected for the remainder of the statement if the condition evaluates to true.

The standard if statement has two generic constructs. They are displayed in Figures 6.1a and 6.1b.

```
if (condition) expr1;[expr2]...;[else];[expr3];...
```

Figure 6.1a if statement selection structure.

```
if (condition) then
   expression1
  [expression2]
...
[else]
  [expression3]
...
endif
```

Figure 6.1b Alternate if statement selection structure.

The specified expression(s) can be any operation performed on a column or series of columns or any number of statements, including routing statements.

Conditional statements can also be used on other statements such as counts, lists, assignment and most validation statements. Figure 6.1c shows several examples of the usage of conditional statements.

```
if (c(2065,2066).eq.1) c(2067,2068)=$ 0$

if (c549'2'.and.c550'2') r sp c2402'3'

if (c2399'2') count c53 $number of cell 2 females$
```

Figure 6.1c conditional statement code examples.

6.2 Routing statements

Routing statements are used whenever it is necessary to "skip" a respondent or number of respondents over a logic section within the edit section. Quite simply, the routing statement begins with a "*goto"* command which needs only a label number. **ccount** will look for the next instance of that label number followed by the word "continue" and will bring the respondent to that point in the edit section. While goto statements can be used by themselves, they are most commonly used in conjunction with conditional statements.

```
if (condition) goto label_number

   editing statements

label_number continue
```

Figure 6.2a Construction of routing statement.

```
if (c(65,67).eq.0) r (c(71,88)' '); goto 88

/*q2b
t2=0
do 71 t1=71,77,3
  r (c(t1,t1+2).in.($  0$,1:100)) $q2b$
  t2=t2+c(t1,t1+2)
71 continue
r (t2.eq.c(65,67)) $q2b total$

/*q2c
t2=0
do 80 t1=80,86,3
  r (c(t1,t1+2).in.($  0$,1:100)) $q2c$
  t2=t2+c(t1,t1+2)
80 continue
r (t2.eq.c(65,67)) $q2c total$

88 continue
```

Figure 6.2b example of goto statement within standard code.

As is visible in Figure 6.2b above, the goto statement can appear within a line of standard **ccount** syntax. The conditional statement (if it evaluates to true) will route respondents over the standard data check/requirement statements of Q2b and Q2c, and continue at the “88 continue” statement. Otherwise (if the condition is false), respondents will process through the Q2b and Q2c sections of the edit section.

*There **CANNOT** be duplicate label numbers present within any given edit section of a run file.*

6.3 Looping constructs

Looping constructs in **ccount** are useful for performing a series of repetitive edits or function calls. These loops require a variable whose lower and upper limits are defined in order to execute a series of statements a specified number of times.

6.3.1 Value-based loops

The generic construction for **do** loops follows:

```
do label_number integer_variable=value_list

   editing statements

label_number continue
```

*Figure 6.3.1a Generic construction of a **do** loop.*

The *label_number* defines the boundaries of the loop, while the *integer_variable* is the variable which is used in the substitution during iteration of the loop. In the *value_list*, we can specify numeric values, a single range of values, multiple ranges of values, column values, and even punch values. Most commonly, though, the *value list* will be a series of values that coincide with columns from the c array that we are going to be referencing in the editing statements present in the **do** loop.

```
do 100 t1=(101,103,107,131,230)
   editing statements
100 continue
```

*Figure 6.3.1b Example of a **do** loop with non-consecutive column values.*

```
do 100 t1=101,105,1
   editing statements
100 continue
```

*Figure 6.3.1c Example of a **do** loop with numeric (or column) values.*

Figure 6.3.1b displays the standard generic syntax for a looping construct which uses non-sequential column values in the value list. In this case, the values assigned to the *integer_variable* loop driver will follow the order of values placed in parenthesis.

Most commonly, the *value_list* contains values that are in the order of “start, stop, step”. This means that the first value present in the list will be starting number (or column location), the second value is the ending value, and the last is the incremental (step) value. An example of this type of loop can be seen in Figure 6.3.1c.

*Remember, label numbers **MUST** be unique throughout the run file. Labels for do loops must not repeat any existing label numbers defined for goto statements.*

Loops can also be adapted and used in conjunction with a series of ranges. An example of looping with numerical ranges can be seen in Figure 6.3.1d.

```
do 100 t1=101,105,1 / 210,215,1 / 315,321,1
   editing statements
100 continue
```

*Figure 6.3.1d Example of a **do** loop with numeric (or column) ranges.*

6.3.2 The “punch loop” (looping with punch codes)

It may sometimes be handy to loop through a series of punch codes (punches) rather than actual numerical values (or column values). **ccount** provides a “punch loop” specification that allows just such a process. The construction of a punch loop is almost identical to the construction of a value-based loop, with one major exception. The integer variable loop driver is replaced by a punch variable. This punch variable uses the standard **ccount** punch notation (surrounded by single quotes), and is symbolized by any letter of the alphabet. By convention, *p* is the letter variable most widely used to represent the punch loop driver.

There are two main types of punch loops in **ccount**. The first type is similar to looping through a non-sequential value list. This type of punch loop uses specified codes (using standard punch notation) enclosed in parenthesis. Figure 6.3.2a shows an example of this type of loop.

```
do 100 ‘p’=(’1’,’4’,’5’,’7’,’9’)
   editing_statements
100 continue
```

Figure 6.3.2a Example of a punch loop using specified codes.

The next example will loop through a series of codes, similar to the “start, stop, step” notation of looping through a value list. However, with this type of punch loop, the “step” notation is not needed. Since punch values are contained in only one column and are sequential, a step value of 1 is implied (see Figure 6.3.2b).

Routing statements can be used to route around looping constructs or to route from the INSIDE to the OUTSIDE of a loop.

```
do 100 ‘p’=’1’,’9’
   editing_statements
100 continue
```

Figure 6.3.2b Example of a ***do*** *loop using consecutive codes, commonly referred to as a punch loop.*

NEVER jump into the body of a loop. Attempting to route from the OUTSIDE to the INSIDE of a loop will cause compiler errors.

Loops can be nested within loops, but MUST NOT OVERLAP.

An example of a syntactically correct **ccount** loop which demonstrates different types of nested loops, and includes routing appears below:

```
do 10 t1=1035,1039,1
           r sp c(t1)'1/5'o
     do 20 ’p’=('1','2','3')
            if (c(t1)'p') goto 50
              c1999'9'
     20 continue
10 continue

50 continue
```

Figure 6.3.2c Nested Loop with routing.

Indenting code within loops increases readability and program clarity.

The above looping example (Figure 6.3.2c) replaces the code shown below (Figure 6.3.2d). After understanding this example, the power of loops should become abundantly clear.

```
r sp c1035'1/5'o
if (c1035'1') goto 50
if (c1035'2') goto 50
if (c1035'3') goto 50
c1999'9'

r sp c1036'1/5'o
if (c1036'1') goto 50
if (c1036'2') goto 50
if (c1036'3') goto 50
c1999'9'

r sp c1037'1/5'o
if (c1037'1') goto 50
if (c1037'2') goto 50
if (c1037'3') goto 50
c1999'9'

r sp c1038'1/5'o
if (c1038'1') goto 50
if (c1038'2') goto 50
if (c1038'3') goto 50
c1999'9'

r sp c1039'1/5'o
if (c1039'1') goto 50
if (c1039'2') goto 50
if (c1039'3') goto 50
c1999'9'

50 continue
```

Figure 6.3.2d Example of longhand (spaghetti) code.

7. ccount syntax – data cleaning and validation

ccount can check to make sure that records contain certain characteristics, verify skip logic and validate punches in a record, and “force clean” records if they do not meet certain defined criteria. **ccount** provides a number of statements which can be used to **require** that columns and records contain certain characteristics.

7.1 Cleaning/validating data

The basic require command is denoted by the letter **r**:

r – the basic require statement is usually used to test the validity of columns or logical expressions, and the generic syntax is displayed in Figure 7.1a below.

```
r condition column’codes’[o]  [$text$]
```

Figure 7.1a Generic syntax for standard require statement.

The standard require statement can validate any given column, whether it be a single-punch (sp) column, a multi-punch column, blank (b), or not blank (nb). These conditions are defined in figure 7.1b.

For instance, in order to verify that column 1108 contains ONLY punches 1 through 5:

```
r sp c1108'1/5'o
```

To require that a column contain no value, or contain blanks:

```
r b c140
```

Require statement options:		
sp	**=**	single punch
b	**=**	blank
spb	**=**	single punch OR blank
nb	**=**	NOT blank

Figure 7.1b Conditions for require statements.

The condition of the column is usually stated before the column is specified, and only the named codes within the statement will be considered valid codes for that column. Adding the optional “**o**” (only) keyword further specifies that any unnamed codes MUST be blank. Optional text at the end of the require statement provides a unique, user-defined error message. If left unspecified, **ccount** will print the failed condition in its entirety to the out2 print file.

If requiring that a series of columns contain only certain valid punches, the punch value notation occurs before the column notation, and the columns are referenced using standard multi-column value notation, as seen below:

```
r sp '1/5'o c(1101,1108)
```

The above statement requires that columns 1101 through 1108 each contain a single valid punch of 1 through 5, and that these punches are the only valid punches allowed.

Basic uses of the require statement:

- Column/code validation.
- Validate logical expressions.
- Test equivalence of logical expressions.

Figure 7.1c Basic uses of the require statement.

Requirements can also be made through logical expressions. Any logical expressions that are to be included in a require statement **MUST** be enclosed in parentheses.

r (logical_expression)

Figure 7.1d Generic syntax for using a logical expression in conjunction with a require statement.

To require that columns 182 through 184 contain a valid range of values between 18 and 65:

```
r (c(182,184).in.(18:65))
```

Another way to require a column be blank, or contain no punches is as follows:

```
r (c255' ')
```

When requiring the *equivalence* of logical expressions, each statement **MUST** be enclosed in parentheses, and all statements **MUST** have the same value (true or false).

r = (condition1)(condition2)

Figure 7.1e Syntax for requiring equivalence of logical expressions.

For instance, suppose that for a particular set of data, columns 457 through 459 can each contain a punch 1 or that column 460 must contain a punch 4. If column 460 contains a punch 4, then the other columns (457 through 459) cannot have a punch 1. Conversely, 460 cannot have a punch of 4 if the columns 457 through 459 contain a punch 1. It is possible to check this data condition using the equivalence requirement:

```
r =(c457'1'.or.c458'1'.or.c459'1')(c459n'4')
```

Since both logic expressions should evaluate to true based on the above assumptions of the data characteristics, the entire require statement would evaluate to true. Rewriting the above equivalence so that both expressions evaluate to false can be done as follows:

```
r =(c457n'1'.and.c458n'1'.and.c459n'1')(c459'4')
```

Again, this require statement should evaluate to true (again based on our data assumptions) since both logical expressions evaluate to the same value (in this case, false).

*To be truly effective, any edits written using the require statement **MUST** be **BOTH** necessary **AND** sufficient to cover all possibilities/occurrences in a given column or number of columns.*

If any given respondent does not meet the “requirements” or criteria set for a particular column, that respondents’ data is printed as part of the out2 file for further review.

These respondents can also be identified by the system variable ***rejected_*** , which will be assigned a value of “true” when that respondent fails any given requirement present in the edit section. By default, any **rejected_** respondents are rejected from the table generation and are sent to the out2 file for inspection.

*Respondents marked as **rejected_** will still continue through the rest of the edit section for further processing.*

Use the return statement to end processing the current record and begin processing the next record at the beginning of the edit section.

ccount keeps track of **ALL** of the failed requirements, and prints them into the **out2** file, below that particular respondents’ data record. A summary of exceptions is also printed into the out2 file, as well as into the **sum_** file, which gives an outline of all the required statements, and how many occurrences of failures are present for each condition.

If a record is marked as ***rejected_*** and **ccount** encounters the end of the edit section, it will continue with the next respondent at the *beginning* of the edit section. If a respondent is **NOT** ***rejected_*** they will continue through to the tab section (if there is one), and when they are fully processed, **ccount** will then continue with the next respondent at the beginning of the edit section

7.1.1 Action codes

The basic require statement has several optional “action codes” that can be used which further detail the exit status of rejected_ respondents (respondents who fail required data conditions). As mentioned earlier, the default action is to reject respondents from the tables and print their data to the out 2 file for further inspection.

```
r /action_code/ condition column’codes’[o]  [$text$]
```

Figure 7.1.1a Generic syntax for standard require statement using action codes.

Action codes must appear immediately after the “r” in the require statement, and before any of the column conditions. These action codes must be surrounded by forward-slash marks (/). These action codes can be helpful when it becomes necessary to highlight some questionable issues within the data that might be present but may not be serious enough to warrant rejecting the respondent record altogether.

Figure 7.1.1b includes a list of other **ccount** action codes which are available.

To illustrate, consider the following:

```
r /2/ sp c1108'1/5'o
```

The statement above states that if a record fails the specified condition, that record will be printed to the out2 file. Any record that fails this condition will ONLY be printed to the out2, and will NOT be flagged as rejected_.

Require statement action codes:		
0	**=**	print summary of errors only
1	**=**	reject records from tables
2	**=**	print entire record output to out2 file
3	**=**	reject respondents from tables and print entire record output to out2 file. this is the **ccount** default action.

Figure 7.1.1b Action codes for use with require statements.

7.1.2 Default editing

When using require statements, it may become commonplace to "force edit", or "clean" respondents' data based on some default. That is, if a respondent's data fails a specific require statement, a default code can be inserted into the offending column or columns. This default code can be any valid data column punch. The standard syntax is shown below.

r /action_code/ condition column’codes’[o]:[new_code]

Figure 7.1.2a Generic syntax for standard require statement using default editing.

It is fairly common to use action codes in conjunction with this feature, namely the /2/ action code. Since the data error will be automatically corrected using this syntax, there is no need to actually reject the respondent. However, it is usually a good idea to print the record to the out2 file in order to validate the results.

For illustrative purposes, consider the following code snippet:

```
r /2/ sp c1120'1/5'o:'9'
```

Any data record which fails this require statement, will have column 1120 overwritten with punch code 9. The result is identical to the following lines of code:

```
r /2/ sp c1120'1/5'o
if (c1120n'1/5'.or.c1120'6/&') c1120='9'
```

As is immediately visible, using the automatic editing option is much more efficient, both in terms of typing time and amount of code generated. Using a more complex example, the forced editing could be shortened to look like this:

```
if (numb(c1120'1/5').ne.1.or.c1120'6/&') c1120'9'
```

This statement most closely duplicates the actions and results of the "forced editing" statement above. This statement will destructively write a punch 9 to column 1120 whenever any of the following are true:

1) column 1120 is blank
2) no punch codes between 1 and 5 are found in column 1120
3) more than one punch code in the range of 1/5 are found in c1120
4) any other codes than 1/5 are found in c1120
5) codes between 1 and 5 are found in conjunction with other codes (6 through &)

This option is also useful when using require statements in conjunction with a field of columns. Any columns within the given field which fail the require statement will be overwritten with the specified default code.

```
r /2/ sp '12'o c(101,105):'0'
```

In this example, assume that columns 102 and 104 both have invalid column punches, while columns 101, 103 and 105 all pass the require statement. Only columns 102 and 104 will be overwritten with the default code '0'. These other columns, since they pass the validation test, will remain intact and will not be modified in any way.

Keep in mind that specifying a default punch for require statements causes a destructive overwrite of any existing data in the specified column.

As with all data editing, do not forget to use a separate filedef and write statement to save any data which has been automatically edited.

7.1.3 Editing with 'priority'

In order to "force clean" a multi-punched (multi-coded) column into a single-punch column, the **priority** keyword can be used. The **priority** keyword checks the listed column against the listed codes in the order specified. When a listed priority code is found, **ccount** keeps that code within the specified column and deletes any additional codes which may be found within that column. The standard construct is as follows:

priority column’code1’, ’code2’, ’code3’, [additional codes]

Figure 7.1.3a Generic syntax for standard priority statement.

In this generic example, when this statement is encountered, **ccount** will scan the given column for the first code listed. If the column contains the first code listed, then all other codes within this column will be deleted (if there are any). If this code does not exist in the given column, then the column is scanned for the second code specified. If this code is found and the column is multipunched, then all other punches will be deleted and ignored. If none of the specified columns are found, the specified column remains unaltered.

For instance, suppose that column 300 has been multipunched with punch codes 1, 3 and 5. The following priority statement

```
priority c300’4’,’3’,’5’,’2’,’1’
```

will result in column 300 only having a punch 4 once the statement is executed. If column 300 contains punches 1, 3, and 7, the same statement above will result in column 300 having punches 3 and 7. Since the 7 punch was not listed in the priority statement, it is completely ignored.

Priority will check only the listed punch values contained within the statement. Other values found within the column are ignored.

Priority statements may also be used in conjunction with multiple columns as a field. The statement works in exactly the same fashion, but can look across multiple columns to determine which punch values should be kept and which ones should be discarded. If a listed code is found in the first specified column, then any additional codes listed will be deleted, as will any that appear in the second listed columns. Any codes NOT listed in any of the specified columns will again be ignored. The generic construction of this type of priority statement is given in Figure 7.1.3b, where ‘p1’, ‘p2’, ‘p3’ and ‘p4’ are all punch codes, and col1 and col2 are specified columns. Note that column punch codes can be repeated for each column within a multiple column specification, but should not be used more than once for any given column.

priority col1’p1’, ’p2’, col2’p3’, ’p4’, [additional columns/codes]

Figure 7.1.3b Generic syntax for priority statement specifying multiple columns.

Using an example, consider the following statement:

```
priority c1972’4’,’3’,’2’, c1973’2’,’1’,’0’
```

Assume that column 1972 is multipunched a code 4 and code 2, and column 1973 is also multipunched codes 2 and 0. When this priority statement is executed, punch 4 within column 1972 will be given the priority and will remain intact. Punch 2 in column 1972 will be deleted, as will punches 2 and 0 within column 1973.

As with single-column priority statements, any unlisted codes present in any of the columns are completely ignored. Using the same example above, if column 1972 contained a punch 5, and column 1973 contained punches 1 and 6, then both columns would remain unchanged. Since punch 5 in column 1972 was unlisted, it is ignored. Similarly, punch 6 in column 1973 is also unlisted and is subsequently ignored. Since punch 1 in column 1973 was a listed code, and punch 2 was nonexistent, punch 1 was given the priority and was kept intact.

This example also works for non-consecutive columns and punches as well, as shown below:

```
priority c1972’4’,’3’,’2’, c111’5’,’7’,’9’,'8','6'
```

This statement will be evaluated using the same process as the previous example. The priority statement can be used as a powerful tool as long as it is used with care, since the result is essentially a destructive data edit.

8. External files and subroutines

Include files and subroutines are useful when splitting the run file into sections, or when it becomes necessary to reference a set of repetitive commands for ease of file management. Although fundamentally different, both include files and subroutines perform similar tasks. Include files allow a user to use external files as part of the main processing, while subroutines are user-defined functions stored within the main program file. Both include files and subroutines can serve to reduce programming repetitiveness, which can lead to greater modularity (re-use of code) and fewer errors.

Include files greatly increase the efficiency of code, and also help to increase code readability by separating repetitive portions of the code into separate modules.

Modular code helps to reduce error by reducing the amount of code written, and can be used to split complex code fragments into more manageable snippets.

8.1 Include files

Include files are external files that are referenced from within the run file. They are referenced directly by the file name preceded by a "`*include`" statement. The standard syntax is shown in Figure 8.1a.

include filename

Figure 8.1a Standard include file syntax.

Include files allow for more flexibility within a run file, and also serves to help separate groups of similar statements to increase program readability. Include files also have the ability to use symbolic parameters for substitution each time the file is included. This increases the efficiency greatly, as one generic file can be used many times with substitution to result in seemingly different files, each with specific and unique variable parameters.

When **ccount** encounters an include statement, processing moves directly from the main file to the included file. After all processing has completed within that included file, control resumes at the point in which it left off within the main run file and continues processing the current record in the normal fashion, until the end of the edit section or tabulation section has been reached.

To illustrate this example, consider the following code snippet from a standard edit section of a run file:

```
r sp c1011'1/5'o

*include icheck

r (c(1012,1020)' ')
r sp '1/5'o c(1021,1025)
```

Figure 8.1a include file example.

As a data record is processed through this section of a run file, the include statement would be executed immediately after the first require statement. When the include statement is encountered, **ccount** will jump to the first line of the file called icheck and begin processing the current record through that file. When the end of the icheck file is reached, processing resumes in the original run file at the second require statement (`r (c(1012,1020)' ')`).

The contents of the icheck file are not currently important. Suffice it to say that the icheck file is processed as if the contents of the file were directly typed into the edit section. This include functionality helps to differentiate parts of the file, improves readability, and can help minimize the size of the run file by reusing repetitive code.

Be sure that all include files reside in the same directory as the current run file. Using references to files that are not present or do not exist will result in syntax errors.

*Include file syntax must begin in column 1 of the run file in which it appears. The * must be in the first column of the line or a syntax error will occur.*

Include statements cannot appear within a conditional statement. If a file is to be included conditionally, a goto statement must be used in order to skip unwanted respondents over the include statement.

Include files can be nested up to 6 levels deep.

8.1.1 Include files using text substitution

Include files also have the flexibility of using text or column substitution. This allows for even greater flexibility when and if a similar, but not identical, snippet of code is repeated many times over within the run file. The substitution is accomplished through the use of symbolic parameters within the include file. These parameters can be considered as “place holders” for the item to be substituted. When the include file is called by the main program, this item is specified, and **ccount** automatically substitutes the specified item for its symbolic parameter. This kind of inline substitution creates a myriad of opportunities to re-use code and increase the modularity of programs.

In the case of text substitution, the symbolic parameter is signified with an ampersand character (&) followed by a variable name which will be substituted. These variable names should be meaningful so that they are easy to remember when using the substitution. For instance, the include file (called the “tabs” file) that follows in Figure 8.1.1a uses substitution for the banner name.

```
/*tabs file

tab q1 &ban
tab q2 &ban
tab q3 &ban
```

Figure 8.1.1a tabs include file before substitution.

In the above include file, the & (ampersand) character alerts **ccount** that a substitution is to take place. When including this file, the text to be substituted must be specified as well as the text that will replace it.

Using the example above, the include statement within the run file might look something like the include statement shown in Figure 8.1.1b below.

```
/*example run file

struct;read=0;ser=c(1,10);reclen=400

ed
/**edit section
end

a;dsp;flush;nopc;notype;op=12;side=30;printz;

*include tabs;ban=banner1

*include axes
*include banner
```

Figure 8.1.1b Example run file highlighting include file with text substitution.

Thus, when the tabs file is included into the original run file (at compile time), the text "banner1" is substituted for the text "&ban" within that include file. When **ccount** compiles the run file, and includes the tabs file, it will interpret the contents as follows:

```
/*tabs file

tab q1 banner1
tab q2 banner2
tab q3 banner3
```

Figure 8.1.1c tabs include file after substitution.

8.1.2 Include files using column substitution

The flexibility of include files do not stop with text substitution. Include files also have the ability to substitute columns and punches. Now the real power of the include file can be imagined. By writing code only once, and substituting different columns and punches, it is possible to create one template that can be used time and time again in certain situations.

This occurs frequently when **ccount** code requires a set of statements to be similar, but not necessarily identical. By using an include file and substituting symbolic parameters for the parts which differ, a significant reduction in programming and development time can be realized.

```
*include filename;[col(Þ)=col_num];[col(ß)=col_num];...
```

Figure 8.1.2a Standard include file syntax using column substitution.

In order to substitute a column reference within an include file, a symbolic parameter must be used. This parameter must be defined in the include statement so that **ccount** will know what value to substitute for the parameter. The symbolic parameter is denoted with the **col** keyword. The parameter is normally defined within the include file using a letter of the alphabet followed by a number.

When used, the **col** keyword must be followed by the symbolic parameter used, as well as the column value which is being substituted. As seen in Figure 8.1.2a above, the symbolic parameters used within the include file must be defined (denoted by *Þ and ß*), followed by the actual column number which they are replacing within the included file. The symbolic parameter must be followed by a number denoting how many columns to add to the original defined value. Normally, the symbolic parameter used is simply a letter of the alphabet, followed by a 2-digit number.

For column substitutions, the letters c, t, n, u and x are reserved, and cannot be used as symbolic parameters. However, any other letter of the alphabet is valid. Using these reserved letters will result in syntax errors.

For instance, consider the following code:

```
r sp c200'1/5'o
r sp c201'1/7'o
if (c201'1') r b c202
r sp c203'12'o

r sp c300'1/5'o
r sp c301'1/7'o
if (c301'1') r b c302
r sp c303'12'o

r sp c400'1/5'o
r sp c401'1/7'o
if (c401'1') r b c402
r sp c403'12'o
```

Figure 8.1.2b Example of repetitive code.

It is clear from the example that there is a pattern. Aside from the column numbers which are changing, the code seems to be fairly repetitive. Using an include file with column substitution can help to minimize this code and reduce errors.

By substituting the letter 'a' for the column (symbolic parameter), and following that with a 2-digit number which denotes how many columns too add to the original, it is possible to simplify the repetitive code into a simpler component which can be reused as many times as needed. The simplified code can be placed inside of an include file, and can be referenced (called) within the main run file numerous times. Each time this include file is called, the actual value of the columns to be substituted are redefined. Figure 8.1.2c illustrates the simplified code from Figure 8.1.2b.

```
r sp c(a00)'1/5'o
r sp c(a01)'1/7'o
if (c(a01)'1') r b c(a02)
r sp c(a03)'12'o
```

Figure 8.1.2c Example of simplified code placed into an icheck include file.

This simplified code can then be used within an include file to be referenced from the run file, as shown in Figure 8.1.2d.

```
*include icheck;col(a)=200

*include icheck;col(a)=300

*include icheck;col(a)=400
```

Figure 8.1.2d Example of include file statements using column substitution.

When the run file is compiled by **ccount** and the icheck file called the first time, the value 200 will be substituted for the symbolic parameter letter a. Thus, the statement

```
r sp c(a00)'1/5'o
```

will be translated into the following:

```
r sp c(200)'1/5'o
```

Similarly, the values 201, 202, and 203 will be substituted for the parameters (a01), (a02), and (a03), respectively. The result is that the 3 include statements (Figure 8.1.2d) and the include file (Figure 8.1.2c) above, replace the repetitive code shown in Figure 8.1.2b. The savings are immediately clear, just considering the number of lines needed for the code to be typed. The savings in efficiency and error reduction are slightly more ambiguous, but they are significant.

The substitution of multi-column values is similar to the format of single-column value substitution. In order to substitute a multi-column value, place the symbolic parameter within parenthesis as in the single-column substitution shown above. Figure 8.1.2e gives examples of the format of multi-column substitution.

To substitute...	Format for include file...
2-digit value	c((a00),(a01))
3 digit value	c((a00),(a02))
4-digit value	c((a00),(a03))
...	...
...	...
...	...

Figure 8.1.2e Multi-column substitution examples.

When substituting columns, be sure to enclose the entire symbolic parameter used in the include file within parenthesis. Failure to do so could result in syntax errors or incorrect substitution.

It is also possible to use several different symbolic parameters within the same include file. The format for the include file is the same aside from the usage of different letters to denote different parameters. For instance, the code

```
*include icheck2;col(a)=200;col(b)=300;col(d)=400
```

uses 3 different symbolic parameters for substitution purposes. The associated include file would include references to c(aß), c(bß) and c(dß), where ß is any number which results in a valid column substitution within the include file.

Column substitution used within include files can be comprised of any valid logical expression.

8.1.3 Include files using punch substitution

Include files also have the ability to substitute punch values. This allows for even more flexibility with include files. This additional flexibility allows for template files to be even more generic, and to be substituted in a myriad of circumstances. This feature is most commonly used in conjunction with column substitution, but is often also used on its own.

```
*include filename;[punch(p)=’ ß’];
```

Figure 8.1.3a Standard include file syntax using punch substitution.

As seen above in Figure 8.1.3a, the punch is again substituted using a symbolic parameter. In this case, the parameter is again a letter of the alphabet. The codes to be substituted (denoted by ß, can be any valid punch value, whether a single code or group of codes.

Consider the repetitive code shown in Figure 8.1.3b.

```
r sp c500'1'o
r sp c501'1'o
r sp c502'1'o

r sp c600'2'o
r sp c601'2'o
r sp c602'2'o
```

Figure 8.1.3b Example of repetitive code.

In this example, the code is again slightly repetitive. However, the column numbers as well as the punches are variable. It is possible to again use an include file which will use substitution in order to simplify this code. Figure 8.1.3c shows the contents of the include file. Note the symbolic parameter used for the punch value.

When included into the main run file, these parameters will have to be explicitly defined, as well as their actual value.

```
r sp c(a00)'p'o
r sp c(a01)'p'o
r sp c(a02)'p'o
```

Figure 8.1.3b Example of repetitive code.

Figure 8.1.3d illustrates a portion of the contents of the run file which calls the include file, and defines the substitution.

```
*include icheck3;col(a)=500;punch(p)='1'

*include icheck3;col(a)=600;punch(p)='2'
```

Figure 8.1.2d Example of include file statements using column substitution.

The resulting file after these specs are compiled would be identical to the code shown in Figure 8.1.3b. Again, the immediate savings in sheer volume of code is apparent.

There is a limit of 30 symbolic punch parameters (substitutions) which can be used within any given run file.

8.1.4 Data files as include files

Data files may be included in a similar manner. Data files may be included within an existing data file, or may be included as part of a file which includes many data files. Consider the following data file (assuming the record ID, or respondent number, is contained in columns 1 through 5):

```
18736 35021498 110000002 1100111
00601 0606050806060606060606091
10901 0107111110111006111009111
20909 0909090909090908100908090
*include data2.dat
```

Figure 8.1.4a example data file using include.

In the above example, the contents of data2.dat would be read immediately after record 20909 is processed. Consider now a new data file, consisting of the records below:

```
*include data1.dat
*include data2.dat
*include data3.dat
```

Figure 8.1.4b data file of include statements.

ccount would read this data file as a normal data file. Initially, all of the records within the file data1.dat would be processed. Once completed, the contents of data2.dat would be processed, immediately followed by all the records contained within data3.dat.

Included data files must not contain any blank lines or extraneous spaces. Unreliable results could occur when the data file is processed.

Please note that all entries in a data file are treated as data records. Therefore, any blank lines between an include statement within a data file would be treated as a separate record, possibly resulting **ccount** translating that record as a null record. This could lead to unexpected results within the data processing edit or within the table processing. Always be sure that there are no unintentional blank lines within the data file.

8.2 The corrections file (corrfile)

The corrections file is a user-defined file which specifies adjustments to be made to the data while it is being processed. The corrections file must be defined within the same project directory as the current spec files (run file, data file, etc), and must be named **corrfile**.

While the data is being processed, **ccount** will read the contents of the **corrfile**, and if the current record number (ID number) matches a record in the **corrfile**, any editing statements contained within the corrfile will be executed. Statements within the **corrfile** will be executed *before* any statements contained within the edit section of the run file.

The standard notation of a **corrfile** is shown below in Figure 8.2a. Note that the corrfile MUST begin with the respondent ID (the serial number defined in the struct statement of the run file) and MUST be sorted in order by this serial number. Note also that the data file must also be sorted in the same order. The optional "/n" is the read number, which is only applicable to data files which contain trailer cards.

serial_ID[/n]; corrections

Figure 8.2a Standard corrfile file syntax.

Corrections which can be made to the data file include overwriting (or setting) a specific code to a column, adding (or emitting) a specific code to a column, and deleting specific codes from specified columns. An example of the correction syntax follows:

`s c(n)'p'` – Set (overwrite) column n equal to punch p.
`e c(n)'p'` – Emit (add) punch p to column n.
`d c(n)'p'` – Delete punch p from column n.

Figure 8.2b shows an example of a standard **corrfile**. Note that the file is sorted in serial order.

```
2040334748; s c14'3'
2040334749; s c14'3'
2040354944; s c14'2'; d c12'7'
2040376567; d c27'9'; e c101'8'
```

Figure 8.2b Example of a standard corrfile.

The above corrections file sets column 14 to a punch value of 3 for respondents 2040334748 and 2040334749. For respondent 2040354944, column 14 is set with a value of 2, while punch value 7 is deleted from column 12. Lastly, the data record for respondent 2040376567 is updated by deleting punch 9 from column 27 and emitting a punch value of 8 to column 101.

*Multiple corrections can be made to a single data record using the standard **ccount** command delimiter, **;** (semicolon).*

*A syntax error will result if the **corrfile** and data file are not sorted in the same order by the same serial number.*

8.3 subroutines

In many programming languages, a subroutine is a sequence of code which performs a specific task (as part of a larger program) and is grouped as one or more statement blocks. Subroutines can be "called" by the main program, thus allowing programs to access all the instructions within the subroutine repeatedly. This eliminates the need for repetitive instructions within the run file, since the subroutine code is written only once even though these same instructions are accessed any number of times.

While subroutines are similar in theory to include files, they are actually vastly different. Be sure to understand the similarities as well as the differences and choose based on your particular programming needs.

ccount also allows for the use of subroutines, which can be used to simplify any repetitive action found in the edit section of the run file. Figure 8.3a illustrates the benefits of using subroutine functions within a standard **ccount** run file.

ccount subroutines:

- can be used to reduce redundancy, or repetitive code actions
- helps to split complex problems into simpler component parts
- enables code reuse among multiple program files
- improves ability to maintain larger programs
- improves readability of a program
- reduces errors

Figure 8.3a Benefits of using subroutines.

All subroutines contained within a run file MUST have a unique name by which they will be called. Using the same name to define multiple subroutines will result in syntax errors.

Most subroutines require that one or more parameters (variables, or values) be “passed in”. These parameters can be of different types, and will differ with each subroutine. Parameters which are passed from the main calling program to the subroutine are substituted and used within the subroutine to perform the tasks within the subroutine.

Almost all subroutines will be written using standard **ccount** integer or real variables, which will then be substituted for the actual values being passed in. Figure 8.3b shows an example of a **ccount** subroutine, as well as the syntax used to call that subroutine.

In the example, the subroutine “outfld” is accessed when a call statement is encountered. Here, the subroutine is called a total of 4 times. Each time the subroutine is called, control passes from the current line into the subroutine. Any parameters that are passed in are transferred to the subroutine as well, and these parameters take the place of the variables that are defined within the body of the subroutine.

The first time the subroutine is called, the value 15 is substituted for the variable a within the subroutine, and the value 16 is substituted for the variable b. These variable values are then used in the subroutine as part of a loop, and these values are assigned to a t1 variable to be written out to a report file.

```
ed

filedef v2atv02.txt report
/*call subroutine
call outfld(15,16)
call outfld(101,107)
call outfld(201,207)
call outfld(315,340)

reportn v2atv02.txt $ $

return

/**subroutine for report statements
subroutine outfld(a,b)
int a
int b
do 1 t1=a,b
   report v2atv02.txt c(t1)
1 continue
   report v2atv02.txt $|$
return
/**end subroutine

end
```

Figure 8.3b Example of a ***ccount*** *subroutine.*

As you can see from the example, the subroutine is defined within the confines of the edit section. It appears just after the end statement, and after a “return” statement. This return statement causes the current record to stop processing, and **ccount** begins again at the top of the edit section with a new record placed in the c array. A subroutine is never formally accessed by the main run file, but rather is only accessed when a call statement is encountered.

A subroutine should ONLY be accessed by the main program when specifically called. Syntax errors will occur if subroutines are accessed by the main program during normal processing flow.

Normal processing flow should end before any subroutine is reached. Be sure to include a return statement before defining any subroutines. This will help to avoid syntax errors.

9. Output files

As mentioned previously, **ccount** has the ability to create external (output) files. These output files can be useful for examining data, exporting partial data records, printing specific reports, or creating data tables.

Similar to compiled files, any output files created by ***ccount*** *will be overwritten during any subsequent runs of the same program file.*

9.1 Holecounts

Holecounts are used to get an overall picture of the data, and give much information in a compact form. Some of the information included in a standard holecount:

- distribution of punches in a column or series of columns.
- punches are shown in actual frequencies and percentages of total.
- “density” of coding for specified columns.

Holecounts are created using a simple **count** command followed by the column or columns you wish to have displayed in the holecount output file, which is called **hct_**

```
count c(start_column,end_column) $text$
```

Figure 9.1a Standard count statement.

Figure 9.1a shows the standard construction of a count statement. The text contained within dollar signs at the end of the count command is the title that will appear on the top of each page for that particular portion of your counts. If no title is specified, the generic default text *“Hole Count”* will be displayed. Counts can be labeled individually, or as a series. Counts can also be filtered using conditional statements.

Figure 9.1b shows an example of a holecounts file created with the statement

```
count c(177,188) $text$
```

The data column locations (columns 177 through 188) are clearly labeled along the left hand side, while the punch values are listed across the top. The punches are displayed in numerical value order, with the Y and X punches (& and –, respectively) first. Punch values then proceed to the right beginning with zero and ending with 9. The “Blank” column displays the number of respondents who have no value for that column. This blank punch is sometimes referred to as the null, or “zed” punch.

The columns “Den1”, “Den2”, and “Den3+” display the density of punches for that column. Any respondents who fall under these columns are said to be “multi-punched”, or have multi-punched data contained within the columns. The number of respondents who have only 1 punch in the column will be displayed under the “Den1” column. Those respondents who have 2 punches are displayed under the “Den2” column, while those with more than 2 will be tallied under the “Den3+” column.

Finally, a “Total” column is provided, which tallies the respondents who have at least 1 punch present in the given column.

```
text

Total = 3267

   Col      &      -      0      1      2      3      4      5      6      7      8      9  Blank        Den1   Den2   Den3+ Total
-------------------------------------------------------------------------------------------------------------------------------------
        |                                                                                           |
   177  |   0      0      0    164      0      0      0      0      0      0      0      0   3103  |    164      0      0    164
        |                     5.0%                                                            95.0% |   5.0%                 0.05
        |                                                                                           |
   178  |   0      0   2072      0      0    164      0      0      0      0      0      0   1031  |   2236      0      0   2236
        |             63.4%                5.0%                                               31.6% |  68.4%                 0.68
        |                                                                                           |
   179  |   0      0      0    358      0      0      0      0      0      0      0      0   2909  |    358      0      0    358
        |                    11.0%                                                            89.0% |  11.0%                 0.11
        |                                                                                           |
   180  |   0      0   2257      0      0      0    358      0      0      0      0      0    652  |   2615      0      0   2615
        |             69.1%                      11.0%                                        20.0% |  80.0%                 0.80
        |                                                                                           |
   181  |   0      0      0    233      0      0      0      0      0      0      0      0   3034  |    233      0      0    233
        |                     7.1%                                                            92.9% |   7.1%                 0.07
        |                                                                                           |
   182  |   0      0   1854      0      0      0      0    233      0      0      0      0   1180  |   2087      0      0   2087
        |             56.7%                                    7.1%                           36.1% |  63.9%                 0.64
        |                                                                                           |
   183  |   0      0      0    243      0      0      0      0      0      0      0      0   3024  |    243      0      0    243
        |                     7.4%                                                            92.6% |   7.4%                 0.07
        |                                                                                           |
   184  |   0      0   2335      0      0      0      0      0    243      0      0      0    689  |   2578      0      0   2578
        |             71.5%                                           7.4%                    21.1% |  78.9%                 0.79
        |                                                                                           |
   185  |   0      0      0     67      0      0      0      0      0      0      0      0   3200  |     67      0      0     67
        |                     2.1%                                                            97.9% |   2.1%                 0.02
        |                                                                                           |
   186  |   0      0   1853      0      0      0      0      0      0     67      0      0   1347  |   1920      0      0   1920
        |             56.7%                                                  2.1%             41.2% |  58.8%                 0.59
        |                                                                                           |
   187  |   0      0      0    229      0      0      0      0      0      0      0      0   3038  |    229      0      0    229
        |                     7.0%                                                            93.0% |   7.0%                 0.07
        |                                                                                           |
   188  |   0      0    623      0      0      0      0      0      0      0    229      0   2415  |    852      0      0    852
        |             19.1%                                                         7.0%      73.9% |  26.1%                 0.26
```

Figure 9.1b Example of a hct_ file.

9.1.1 Holecounts with weights [1]

Holecounts are normally a standard count of the number of punches within the specified data columns. However, holecounts can be created by including a calculation of a weight, or "multiplier". Both integer and real variables are valid to use as multipliers.

In the case of weighted holecounts, each column field value will be incremented by the weighted multiplier, instead of just by the standard increment of one. Figure 9.1.1 shows the standard syntax of weighted holecounts.

count c(start_col,end_col) $text$ [cx(startweight,endweight)]

Figure 9.1.1 Standard count statement using weights.

An example of a weighted holecount is as follows:

```
count c(101,280) $card 1 and 2$ cx(311,318)
```

where 101 through 280 are the columns to be counted, the text within dollar signs is the text to be printed at the top of the count, and `cx(311,318)` is the weight, or multiplier to be used in

[1] This feature is supported in **ccount** versions 1.99 and greater.

calculating the resulting count. This is useful in checking weighted tables, and verifying that the weighted numbers shown on the tables are, in fact, correct.

9.2 Frequency distributions

Frequency distributions of numeric values or alphanumeric strings (string literals) in **ccount** can be created just as easily as holecounts. To create an alphabetic frequency distribution of a column or columns that you wish to have displayed in the output file called **lst_**, simply use the generic form shown in Figure 9.2a.

```
list c(start_column,end_column) $text$
```

Figure 9.2a Generic construction of a list statement.

The standard list statement will provide frequency distributions in absolutes, percentages, and cumulative percentages. Again, the text title is optional. If included, it will display above the frequency distribution. If no title is given the generic text *"Frequency Distribution"* will be used. Similar to holecounts, lists can be based on filtered conditions as well.

The standard list statement has many options to display the data in an ordered format. Options are shown in Figure 9.2b, and an example of a frequency distribution is shown in Figure 9.2c.

List statement options:

lista = creates an alphabetical frequency distribution

listc = creates a range frequency distribution

listr = creates a ranked frequency distribution

Figure 9.2b List statement options.

```
Q11_36
                                                              Alphabetical sort

Total = 3267

String                                                 Frequency      Cumulative
------                                                 ---------      ----------

 1                                                    79    2.4%      79    2.4%
 2                                                    33    1.0%     112    3.4%
 3                                                    90    2.8%     202    6.2%
 4                                                   123    3.8%     325    9.9%
 5                                                   265    8.1%     590   18.1%
 6                                                   409   12.5%     999   30.6%
 7                                                   415   12.7%    1414   43.3%
 8                                                   563   17.2%    1977   60.5%
 9                                                   512   15.7%    2489   76.2%
10                                                   778   23.8%    3267  100.0%


Number of categories = 10

Number of numeric items = 3267
Sum of factors =  24483.00
Mean value     =      7.49
Std deviation  =      2.25
```

Figure 9.2c Example of lst_ file.

The title of the frequency distribution is printed at the top, with the Total number of respondents qualifying directly beneath the title on the left-hand side. Continuing down the left-hand side is a listing of the String contained in the specified column or columns. One the right-hand side is the individual and cumulative frequencies (in actuals and percents) which each string value represents. At the bottom of the distribution is a summary of the number of categories, the number of numeric items, and statistical calculations (sum of factors, mean and standard deviation).

9.2.1 Frequency distributions with weights [2]

Similar to weighted holecounts, frequency distributions can be processed using weights to multiply the counts of the frequency according to some pre-defined segment definition (or segment of population). Weighted frequency distributions are written in a very similar fashion as weighted holecounts. Again, each field frequency will be incremented by the weighted multiplier, instead of just by the standard increment of one. Figure 9.2.1 shows the standard syntax of weighted frequencies.

```
list c(start_col,end_col) $text$  [cx(startweight,endweight)]
```

Figure 9.2.1a Generic construction of a list statement using weights.

For more detailed information about weights please read chapter 14 of this manual.

9.3 Data files

Since **ccount** edits happen "on the fly" and the original data file that is being read is **NEVER** altered in any way, there may be a need to create a data file that incorporates all the edits that are performed upon the c array during the **ccount** run. It also may be necessary to output a data file that contains only the non rejected_ (or a separate subset of) respondents from the main data file.

To **def**ine a **file** type for output, use the **filedef** statement. The generic construction of a **filedef** statement is as follows:

```
filedef filename file_type
```

Figure 9.3a filedef statement.

where the output file_type can be either `data` or `report`. In the example above, the *filedef* statement defines and opens a file named *filename* of the specified type *file_type*.

Although files can be defined anywhere in the edit section, it is a good practice to always define files at the beginning of the edit to avoid any confusion with filenames and/or file types.

Once the new data file has been defined, and edits have been performed on the c array, the data can be written out by using a simple **write** or **print** command shown in figures 9.3b and 9.3c.

[2] Functionality will be supported beginning with release version 2.0.

```
write filename [c(column_start,column_end)] [$text$]
```

Figure 9.3b write statement.

```
print filename [c(column_start,column_end)] [$text$]
```

Figure 9.3c print statement.

If a write statement is used with no specifications, the entire contents of the c array will be written to the new record within the file. If columns are specified as part of the write statement, only those columns which are specified will be written to the file.

Remember that the filename to be written must match EXACTLY to the defined filename, and that the file must be defined BEFORE it is written out.

Figure 9.3d illustrates an example of defining a data file within a standard edit section. Note that the data file is defined at the very beginning of the edit section, and is written out at the end. Also note that the data file that is written out only includes records who have NOT been flagged as rejected_. Thus, the **write** and **print** keywords can be used in conjunction with conditional statements to produce filtered output.

```
struct;read=0;ser=c(1,7);reclen=200

ed
filedef data.clean data

/**static variables
if (c(1,7)' ') reject $Missing Resp ID$ ; return
r (c(1,7).in.(1:9999999))

/**count and write out data
*include icounts
c200'1'

if (rejected_) print$Record Rejected!$
+else; write data.clean

end
```

Figure 9.3d edit section example using filedef and write.

9.4 Report files

Another type of file we can output is called a **report** file. **Report** files are useful for writing out *parts* of your data file, for outputting a data file in some other format, or for outputting your data variables in some other order. Again, we have to **def**ine our **file** type using a **filedef** statement, but the difference here will be that our *file_type* will be a **report.**

```
filedef filename report
```

Figure 9.4a write statement using report option.

*Defined filenames within a **ccount** run MUST be unique.*

Writing out **report** files is slightly different than writing data files. Report files allow the flexibility to output records, fields and/or variables in the format of your choice. The generic construction of writing out **report** files is as follows:

```
report[n] filename options
```

Figure 9.4b reportn example.

Options specify exactly which columns will be output, and any formatting which is to occur when that data is written. Following is a brief description of the options available for writing out report files in **ccount**.

```
report filename c(1,11)
```

In the above statement, report will try to convert the data into a numerical value. This is the easiest and most straightforward method to output numerical fields, but be aware that if the field is blank, or contains alphanumeric characters, the result could be unexpected.

In order to output literal data (with no conversion to numerical values), or to output alphanumeric text, the preferred method is to output one column at a time. Usually, this is accomplished through the use of looping constructs, as shown below:

```
do 10 t1=1,11,1
    report filename c(t1)
10 continue
```

A **reportn** statement which will start a *new line* every time the **report** statement is executed. For instance, to **report** a file called *respids* which contains only the respondent numbers (assuming it occurs in columns 1 through 10 of each record), the following lines would be needed in the edit section of the run file:

```
filedef respids report
reportn respids c(1,10)
```

Figure 9.4a Example of report and reportn statements.

Note that only one reportn statement is needed, since only one data field is being reported to the output file.

In the following example (Figure 9.4b), one reportn statement is used to report several items to the report file. This file, called repfile, writes out several variables to the output including an integer variable, a real variable, and a real number stored within a column.

```
reportn repfile t1,$ $,x5:2.3,$ $,cx(13,17):4,$Text$
```

Figure 9.4b Example of report and reportn statements.

Without a reportn, the report file will output as one consecutive string of data. Use reportn whenever a new record or new line is needed.

To report more than one (nonconsecutive) field(s) to the file, you would need several **report** statements, with the last statement being a **reportn**, so that a new line would be created for the next processed respondent. Figure 9.4c displays a series of report statements, terminating with a reportn statement. This output will be identical to the output from Figure 9.4b.

```
report repfile t1,$ $
report repfile x5:2.3,$ $
reportn repfile cx(13,17):4,$Text$
```

Figure 9.4c Example of report file output.

In order to report a new line only, simply use the reportn statement in conjunction with the filename – No column or text specification is necessary.

9.5 out2 file

Even though the out2 file is, for the most part, system-defined, there may be a need to "redefine" the out2 based on certain characteristics of the data file that is being examined.

Since the out2 file is technically referred to as a print file, the filedef statement can be used to redefine the out2 file using the file type option **print**.

Valid options for filedef statement when redefining out2 file:

mpd = multi punch data. displays multi-punch information in the out2 file vertically (down) in the file

mpa = displays multipunch data horizontally (across) in the out2 file.

mpe = display multipunch information as an asterisk (*), but lists the multicodes beneath the record

norule = suppresses the ruler guide normally printed along with the respondents data in the out2 file

noser = suppresses printing of the serial number of records in the out2 file

len = specification of length of output record if different than original reclen value given in struct statement

$TEXT$ = heading text to be printed at the top of each page

Figure 9.5a filedef options used in conjunction with the out2 file.

10. Table Generation

In addition to data manipulation, editing and output, **ccount** has the ability to produce output in the form of data tables. In order to produce data table output, the tab section must be defined in detail. Refer to Figure 3.5b for an example run file. The a-line (or a-card) opens the tab section, the tab statements define which axes are to be displayed on the tables, and the axis and banner sections define these axes to be tabbed and displayed as data tables.

Data tables, also called cross tabs, are a powerful tool for examining and applying meaning to data. Data tables are a representation of the data in a standard format. They are basically a grid layout of the data that includes rows and columns. Each row and column is labeled so that the data are easily deciphered. Tables display a large amount of information in a very compact space, and are fairly easy to manipulate. This makes for a very flexible way to visually interpret the data contained within the data file.

Once a data record has been processed completely through the edit section, it is sent and processed through the tabulation section. Any non-rejected respondents will be processed through this tabulation section one at a time, in the same fashion as they were processed through the edit section. All respondents processed through the tabulation section will be tallied into the **tab_** output file.

10.0 Introduction to tables

An example of a standard table, or cross tab, is shown in figure 10.0a below. How to create this table will be explained in great detail in the following chapters. This section will give an overview of the contents of the table and their meaning.

```
  Page 1
                                             Out Of This World
                                             Study Number s05-018

                                             ** FEMALES ONLY **
Age of Female Respondents
Q.1)


                                        Gender          Age                  Ethnicity
                                     ----------- ----------------- -----------------------------
                               Total  Male Fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
                               ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

Base: Total Respondents           50     -    50     2    30    18     7     3    36     2     2

20-21 (20.5)                       2     -     2     2     -     -     -     -     2     -     -
                                 4.0     -   4.0 100.0     -     -     -     -   5.6     -     -

22-28 (25.0)                      30     -    30     -    30     -     6     2    19     1     2
                                60.0     -  60.0     - 100.0     -  85.7  66.7  52.8  50.0 100.0

29-34 (31.5)                      18     -    18     -     -    18     1     1    15     1     -
                                36.0     -  36.0     -     - 100.0  14.3  33.3  41.7  50.0     -

Mean                           27.16  0.00 27.16 20.50 25.00 31.50 25.93 27.17 27.46 28.25 25.00
Std. Dev.                       3.40  0.00  3.40  0.00  0.00  0.00  2.46  3.75  3.61  4.60  0.00
Std. Err.                       0.48  0.00  0.48  0.00  0.00  0.00  0.93  2.17  0.60  3.25  0.00

Total                             50     -    50     2    30    18     7     3    36     2     2
                               100.0     - 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0
```

Figure 10.0a Example of a standard tab_ output file.

Left-justified at the very top of the table is the page number, followed by the axis table title. This table is labeled “Age of Female Respondents”, and is useful in describing the type of data that will be included in the table.

Continuing down the left-hand side, there is the “base line”, which describes the population of the table as “Total Respondents”. However, looking at the gender column shows that this particular table only includes data from the population of female respondents. In most cases, tables will be based to the total population. Tables that are based to a smaller portion of the total data population are said to be filtered. The table in Figure 10.0a is said to be filtered.

Additionally, the data are mapped out in a grid-type format, with rows and columns. The rows (still left-justified), or stubs, are clearly labeled with the age ranges of the respondent, and the values used when calculating the mean. The stub “Total” at the bottom of the table shows that the data included in all the previous stubs adds back to the total base. All of the stubs for this table are defined by a specific, unique axis definition. Other tables (not shown) are defined by separate, unique axes.

The columns in the table are determined by the banner, which can be seen at the top of the page, under the global table title. This banner shows the column breaks (banner points) which will be used to further analyze the data, and is a specially defined axis.

The intersection of the banner points and the stubs are the cells of the table. Each cell represents a combined filter, or condition. The resulting number that is contained within these cells are called “actuals”. These actuals represent the number of respondents from the base who qualify for this combined filter.

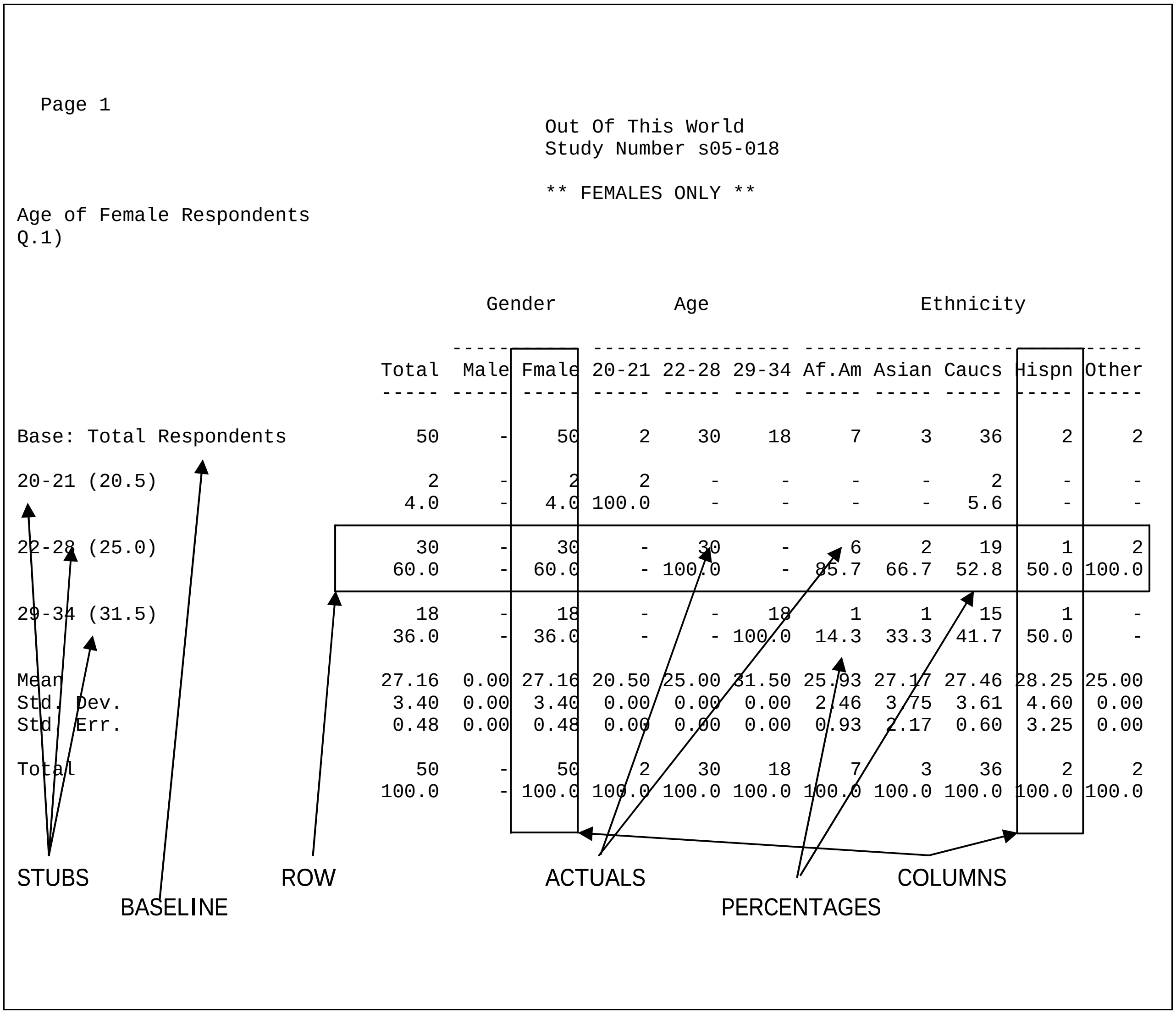
For instance, of the 50 total females present in this particular data file, 18 are between the ages of 29 and 34. In this particular table, the intersection of ages (from the stubs and the banner) happen to be the same as the age stub in the total column, and the same as the base stub for that column in the banner. This is an expected result, since the intersection of 2 identical filters should be the same as one filter alone.

The intersection of the “Caucs” Ethnicity banner point and the “29-34” age stub shows that there are 15 females who are Caucasian who fall into this age category. In other words, of the 36 total Caucasian females (from the banner baseline) in the data, and 18 total females (from the total column in the age stub) who are 29-34, there are 15 who satisfy both conditions (the intersection of these 2 points on the table).

Directly underneath the actuals are percentages. These percentages reflect the percentage of the population in each cell as compared to the base line. Using the second example above, it is easy to see that Caucasian Females who are between the ages of 29 and 34 make up 41.7 percent of the total Caucasian Female population contained within the data file.

Note that all the qualifying conditions for the table cells explained thus far are most likely extracted from data which is already found in the data file. However, table stubs can be based off of mathematical operations as well. Means, Medians and other statistical calculations can be programmed to be displayed as stubs on the tables as well. The table in Figure 10.0a shows the statistically calculated stubs Mean, Standard Deviation, and Standard Error.

Figure 10.0b illustrates and highlights some of the different parts of the table described above.



```
  Page 1
                                        Out Of This World
                                        Study Number s05-018

                                        ** FEMALES ONLY **
Age of Female Respondents
Q.1)


                                   Gender          Age                Ethnicity

                               ----------- ----------------- -----------------------------------
                         Total  Male Fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
                         ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

Base: Total Respondents     50     -    50     2    30    18     7     3    36     2     2

20-21 (20.5)                 2     -     2     2     -     -     -     -     2     -     -
                           4.0     -   4.0 100.0     -     -     -     -   5.6     -     -

22-28 (25.0)                30     -    30     -    30     -     6     2    19     1     2
                          60.0     -  60.0     - 100.0     -  85.7  66.7  52.8  50.0 100.0

29-34 (31.5)                18     -    18     -     -    18     1     1    15     1     -
                          36.0     -  36.0     -     - 100.0  14.3  33.3  41.7  50.0     -

Mean                     27.16  0.00 27.16 20.50 25.00 31.50 25.93 27.17 27.46 28.25 25.00
Std. Dev.                 3.40  0.00  3.40  0.00  0.00  0.00  2.46  3.75  3.61  4.60  0.00
Std. Err.                 0.48  0.00  0.48  0.00  0.00  0.00  0.93  2.17  0.60  3.25  0.00

Total                       50     -    50     2    30    18     7     3    36     2     2
                         100.0     - 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0
```

Figure 10.0b Example of a standard table highlighting various items.

10.1 The a-line (or a-card)

As mentioned in chapter 3, The **tab** section of the run file is where all table formatting options and questions (axes) are defined and is opened with a standard "**a-line**" or "**a-card**" statement. These options define the global default format for the tables to be processed. In order to "open" the tab section, the a-card must appear immediately after the edit section, following the end statement. Figure 10.1a again shows the standard construction of the a-card.

a; option1; option2; option3;[option4];[option5]; ...

Figure 10.1a Generic construction of the "a-line" or "a-card" .

Also mentioned in chapter 3, the options on the a-card are keywords that define the characteristics of and output options for table formatting. The a-card defines the format and overall appearance of tables, as well as the data display options which determine how cell counts and percentages will appear. Figure 10.1b contains a fairly comprehensive list of a-card options, and their meaning.

summary of a-line (global tabulation) options:		
a	**=**	defines the a-line (a-card), and opens the tab section
dsp	**=**	double-spaces all rows
nodsp	**=**	suppresses the printing of double-spaced rows
op (op=)	**=**	declares the type of output to display on tables. The 3 most commonly used output options are: **0** = row percentages **1** = absolute figures (default) **2** = column percentages **5** = prints the text 100% under each base element **&** = total percentages (to the upper-left baseline number)
dec	**=**	number of decimals for absolute figures (*maximum:* dec=7)
decp	**=**	number of decimal places for percentages (*maximum:* decp=7)
flush	**=**	right aligns both absolutes and percentages in cells on tables
indent	**=**	auto-indent wrapped axis element text (*example:* **indent=3**)
spechar	**=**	prints specified characters instead of zeros in cells containing (weighted or unweighted) zeros (*most common:* **spechar=-***)
printz	**=**	prints blank tables (default is to suppress printing of blank tables)
noprintz	**=**	suppresses the printz keyword
type	**=**	prints output type in top right hand corner of the table
notype	**=**	suppresses the printing of the output type
side	**=**	can be used to alter the text width (default=24, maximum=120) (*example:* **side=30**)
paglen	**=**	defines the page length(in number of lines) (*example:* **paglen=60**)
pagwid	**=**	defines the character width of each page (*example:* **pagwid=140**)
nz	**=**	drops blank lines (stubs) in a table (default is to print all elements in a table)
nonz	**=**	suppresses the nz keyword
page	**=**	activates automatic page numbering (default)
nopage	**=**	suppresses page numbering
pc	**=**	prints percent signs after percentage figures on tables (default)
nopc	**=**	suppresses the printing of percent signs on tables
colwid	**=**	column width to use when using banners without g-cards and p-cards
pagtxt	**=**	Replaces standard page text (Page nnn) with specified text for international page numbering (*example*: pagtxt=Seite)

Figure 10.1b Most commonly used a-card options.

An example of a standard a-card using some of the options above can be seen in Figure 10.1c.

```
a;dsp;op=12;flush;spechar='-*';printz;nopc;side=30;pagwid=158;paglen=56
```

Figure 10.1c standard a-card statement using options.

Any number of keywords can be used on an a-line, but all keywords must be separated by semicolons and must appear on the same a-line.

Defining more than one a-card in any given run file will result in compile-time errors.

The a-card MUST appear first (immediately after the edit section) in order to "open" the tab section.

10.2 Titles and global definitions

10.2.1 Table titles

Following the a-card in the run file and before the tab section is where global table titles appear. These global table titles will be displayed at the top of each page of the tables. In order to display global table titles use a **tt**, or title statement. An example of a title statement is shown in Figure 10.2a below.

tt[Y]table_title_text

Figure 10.2.1a Standard syntax for table titles.

Table titles can appear on the page left-justified, centered or right-justified. In order to specify that a table title be centered, use the **ttc** statement. Figure 10.2b shows the different table title options and their usage.

ccount table title options:

ttl	**=**	table title, left justified
ttc	**=**	table title, centered
ttr	**=**	table title, right-justified

Figure 10.2.1b Table title options.

Table titles will appear exactly as typed. Spaces following the tt statement before the actual title text will be displayed on the table. Trailing spaces will be ignored.

Table titles may appear in different places on tables depending on where they are defined. There are many different types of table titles including global table titles, tab-level table titles and axis-level table titles. All of these table titles are defined using the standard tt statements.

10.2.2 Global definitions

Global definitions for tables work similarly to symbolic parameter substitution used in conjunction with include files. For tables, symbolic parameters may be numbers, specific punch codes, standard text, or any combination. This is useful for **ccount** tables which contain many standard elements, such as a company or project name within the tables. In order to define global symbolic parameters for use with tables, the ***def (**or **#def**) keyword must be used. An example of the ***def** keyword and its syntax can be found in Figure 10.2.2a.

****def [col(a)=column];[punch(p)='code'];[txt=substituted_text]***

Figure 10.2.2a Standard syntax for table titles.

Note that in the example above, the letters a and p are used as examples only. When defining symbolic parameters, any letter can be used for the col substitution and any other letter for the punch substitution. When using the defined text parameter within the axes or tabs files, be sure to use the preceding "&" character to signify that a substitution is to be made. When using the symbolic column and punch substitution within the axes or tabs files, be sure to also use the same substitution format used within include files (c(a00) and 'p', respectively). For clarification on include file and substitution specifics, please refer to section 8.1. An example of an actual *def statement is shown in Figure 10.2.2b below.

```
/*example run file using global definitions

struct;read=0;ser=c(1,10);reclen=400

ed
/**edit section
end

a;dsp;flush;nopc;notype;op=12;side=30;printz;
*def col(a)=133; punch(p)='123'; btr=Base: Total Respondents


*include tabs;ban=banner1

*include axes
*include banner
```

*Figure 10.2.2b Example run file highlighting *def global substitution.*

*While the ***def (#def**) statement can normally appear anywhere within a **ccount** run file, it is most commonly found after the a-card and before the tab section.*

*Similar to include file syntax, the *def statement must begin in the first column of the line in which it appears. Otherwise, syntax errors will result.*

*As with table titles, any text defined within a *def (or #def) statement will appear exactly as typed. Trailing spaces will be ignored.*

10.3 Tab section

The tab statements specify which axes are to be tabbed against each other in an ordered fashion. Tab statements determine which axes will be displayed and what orientation on the table they will have. Figure 10.3a shows the generic construction of a tab statement.

```
tab [axis_row] [axis_column] [axis_filter]...
```

Figure 10.3a Generic construction of tab statement.

The tab statement is invoked simply with the keyword **tab**, and is followed by the axes which are to be tabbed. The axes which are included in the tab statement represent the rows and columns of the table to be displayed.

The first axis describes the rows, or stubs, of the table, and includes table-level table titles, as well as base definitions and any statistical calculations, such as means or medians. The second axis defined is the axis which will be used as the banner, or column(s) of the table. Any axis can serve as a banner, but there are axes which can be specially defined as banners to display the columns in a more readable and ordered fashion. These specially defined banners allow for the data to be displayed in formatted columns, which increase the readability of the data on the table.

Below, in Figure 10.3b is an example of a standard tab section in a run file.

```
/**tab statements

tab q1 qbanner
tab q2 qbanner
tab q3 qbanner
```

Figure 10.3b Sample tab section.

Questions listed in the tab section will be processed in the order specified, with each table beginning on a new page (default). In the above example, axis q1 will be tabbed against the banner axis qbanner. This table will be followed by axes q2 and q3, both of which will be tabbed against the same banner axis, qbanner.

10.3.1 Tab statements as include files

As mentioned in chapter 8, include files can be very useful in certain situations. Notice how repetitive tab statements can be in Figure 10.3b above. Using an include file in this case can help to save time and effort, as well as creating a much more readable (cleaner) run file. Figure 10.3.1a displays an example run file that uses an include statement for the tab section.

Similar to the example shown in chapter 8, this include file (called "tabs") uses substitution, and the & character is the alert that a substitution is to take place. When the tabs file is included into the original run file, the text "banner1" will be substituted for the text "&ban" during compile time (and/or run time) within that include file. When **ccount** compiles the run file above, and includes the tabs file, it will interpret the contents of the tabs file, shown in Figure 10.3.1b, so that it will become identical to the tab section displayed in Figure 10.3b.

```
/** run1
/* AnyStudy
struct;read=0;ser=c(1,7);reclen=1000

ed
filedef respids.txt report
reportn respids.txt c(1,7)

*include icounts

end
a;dsp;flush;nopc;notype;op=12;side=30;spechar=-*
++;printz;pagwid=150;paglen=70

ttcOut Of This World
ttcStudy Number s05-018

*include tabs;ban=qbanner

*include axes
*include banner
```

*Figure 10.3.1a Example run file using *include statement (and substitution) for tab section.*

```
/*tab statements
tab q1 &ban
tab q2 &ban
tab q3 &ban
```

Figure 10.3.1b tabs include file before substitution.

```
/** run1
/* Out Of This World Study
struct;read=0;ser=c(1,7);reclen=1000

ed
filedef respids.txt report
reportn respids.txt c(1,7)

*include icounts

end
a;dsp;flush;nopc;notype;op=12;side=30;spechar=-*;printz;pagwid=150;
++paglen=70

ttcOut Of This World
ttcStudy Number s05-018

*include tabs;ban=qbanner
*include tabs;ban=qban2

*include axes
*include banner
```

*Figure 10.3.1c Example run file using 2 *include statements (with substitution) within the tab section.*

If these axes (q1, q2, and q3) need to be tabbed again using a second, separate banner simply copy the line and change the banner name. This avoids having to retype the entire contents of the tab file only to change the banner name. Figure 10.3.1c shows the revised run file which utilizes 2 include files, one for each banner. By moving the tab statements to an external file and using substitution, additional tables can be created by creating another `*include` statement, and only modifying the banner name which is being substituted for the symbolic parameter ban. This type of banner name substitution is very common and frequently used in run files.

10.3.2 Filters within tab statements

Sometimes it is necessary to run a group of tables based on an external filter which cannot be incorporated into the axis base filter. When this becomes necessary, a **flt** (filter) statement can be used. This is a special type of filter that can be used within the tab section of the run file. In order to declare a filter statement, simply use the **flt** keyword followed by a “condition equals” statement. Figure 10.3.2a displays the generic construction of a **flt** statement.

```
flt;[c=condition]
```

Figure 10.3.2a Declaring a filter condition within tab section.

The filter condition appears before any grouping of tab statements, and is declared (or opened) with a “condition equals” statement. This condition statement begins with a “c=” and is followed by any logical expression with which to filter the group of tables. To end (or close) the filter section, simply clear the filter with an unfiltered **flt** statement. An example of this is shown in Figure 10.3.2.b. See Chapters 4, 5 and 6 for additional information about logical expressions.

```
flt;
```

Figure 10.3.2b Suppressing a filter condition within tab section.

For instance, suppose that column `c127` contains the data punch for gender. In order to filter a series of tables on the gender punch, simply add a **flt** statement before the tab statement (or series of statements).

Figure 10.3.2c shows an example of a filter statement used in the tab section of a run file. In this example, the first 4 questions (q1, q2, q2a and q2b) will be filtered on column 127 having a punch of 2 (`'2')`. That is, if `c127` is again the column which contains data information for gender, then when tabbed, these first four questions would only include the data for the females contained within the data file.

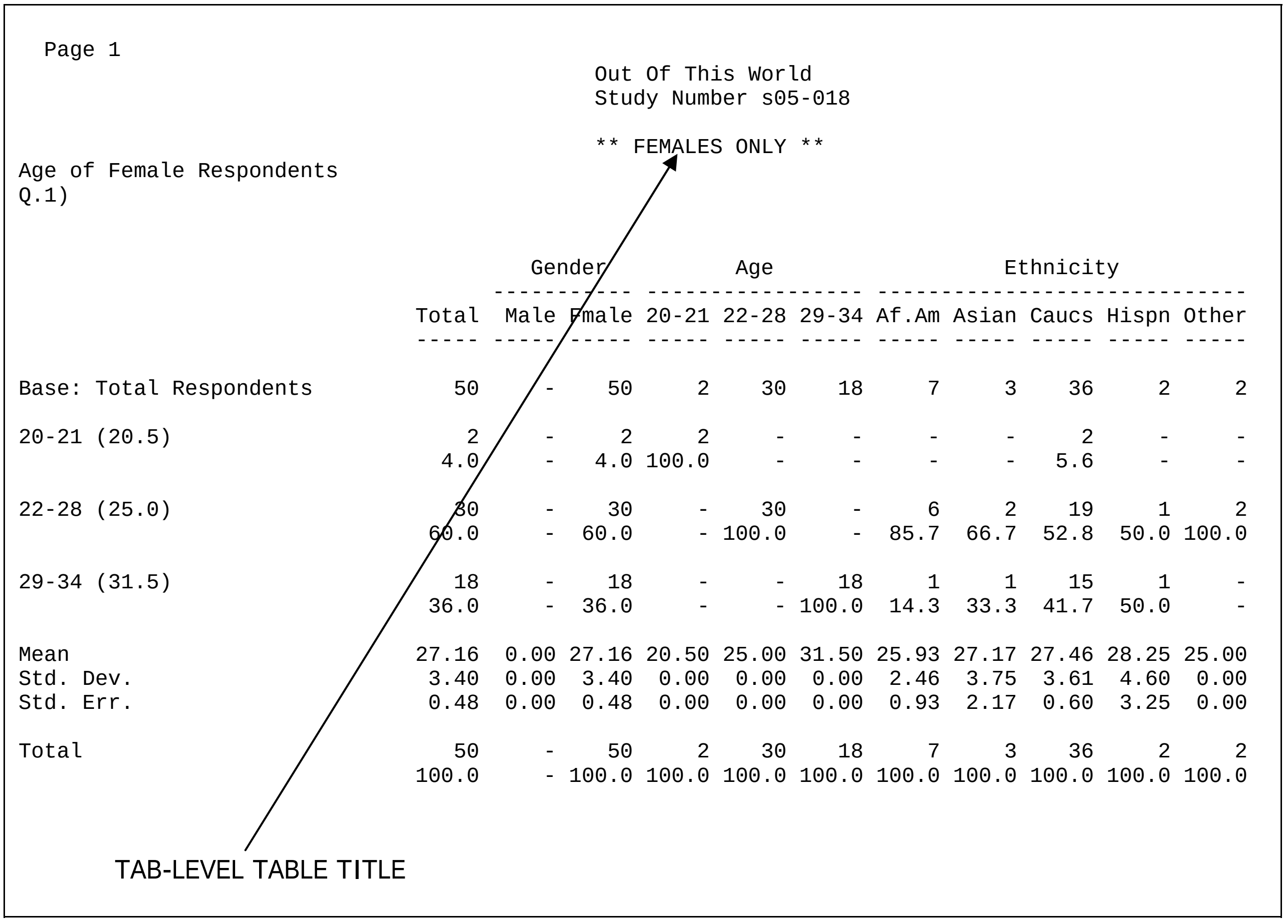
```
/*filtered tab statements

flt;c=c127'2'
ttc** FEMALES ONLY **
tab q1 qbanner
tab q2 qbanner
tab q2a qbanner
tab q2b qbanner
flt;
tab q3 qbanner
```

Figure 10.3.2c Example tab section using the flt statement.

Notice also the inclusion of a tab statement level table-title. This title specifies the filter that is being used on the group of tables, and will display beneath the global header, but above the axis title on the page (as seen in Figure 10.0a). This table title will appear on each table for which the filter statement (**flt**) is active. The printing of this tab statement level title will be suppressed when the filter statement is closed.

It is recommended to use tab-level tt statements to describe any filters placed within the tab section. This increases clarity of the tables, and reduces the chance for error.

Redisplaying the table seen in Figure 10.0a, the tab-level table title can be explicitly seen. Even though the baseline may describe the table as being based to "Total Respondents", the tab-level table title clearly shows that the table includes the population of "Females Only".



```
  Page 1
                                             Out Of This World
                                             Study Number s05-018

                                             ** FEMALES ONLY **
Age of Female Respondents
Q.1)


                                        Gender          Age                Ethnicity
                                     ----------- ----------------- -----------------------------
                               Total  Male Fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
                               ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

Base: Total Respondents           50     -    50     2    30    18     7     3    36     2     2

20-21 (20.5)                       2     -     2     2     -     -     -     -     2     -     -
                                 4.0     -   4.0 100.0     -     -     -     -   5.6     -     -

22-28 (25.0)                      30     -    30     -    30     -     6     2    19     1     2
                                60.0     -  60.0     - 100.0     -  85.7  66.7  52.8  50.0 100.0

29-34 (31.5)                      18     -    18     -     -    18     1     1    15     1     -
                                36.0     -  36.0     -     - 100.0  14.3  33.3  41.7  50.0     -

Mean                           27.16  0.00 27.16 20.50 25.00 31.50 25.93 27.17 27.46 28.25 25.00
Std. Dev.                       3.40  0.00  3.40  0.00  0.00  0.00  2.46  3.75  3.61  4.60  0.00
Std. Err.                       0.48  0.00  0.48  0.00  0.00  0.00  0.93  2.17  0.60  3.25  0.00

Total                             50     -    50     2    30    18     7     3    36     2     2
                               100.0     - 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0
```

Figure 10.3.2d Figure 10.0a highlighting tab-level table title.

11. Table-generating files

The last items in the run file are the axes and banners. The actual axes (axis) definitions must appear after the tab section, just before the end of the run file. These axes define the data and/or questions that are to be used in the tables (for both rows and columns). While these files are technically part of the tab section, they are usually referred to by name, and are normally built as separate files to be included within the run file (*see Figure 10.3.1c*) . These files (axes and banner statements) are usually the most time consuming and detailed part of any run file.

11.1 The axes file

As described in Figure 10.0a, an axis represents the rows of a cross tab, and is an integral part of all table generation. Normally, all axes are defined within the same file for ease of readability. This file is usually called the axes file. Since an axis generally represents one question within a questionnaire, the axes file usually looks similar to the questionnaire which it is based on, with all the questions appearing in the same order as the questionnaire and all having a uniform look and "feel".

In order to define an axis, simply define the name of the axis on an individual "**l-line**" (or l-card), as shown in Figure 11.1a. Axis names must be unique throughout any given **ccount** run. That is, any given axis name can appear once, and only once within any given run file.

```
l axis_name
```

Figure 11.1a Declaring an axis.

All the items within an axis definition are called "elements". Each element serves a specific purpose, and may display differently on the page. The first element that normally follows the axis name are the table titles. As mentioned previously, table titles will display on the page in different areas depending on where they are defined. For axis-level table titles, text will appear just before the banner on each page. In order to define axis-level table titles, use a tt statement immediately after the l-card. An example of this is shown in Figure 11.1b.

*Axis names MUST be unique within any given axes file, and must be unique within any given **ccount** run.*

```
l q1
ttlAge of Female Respondents
ttlQ.1)
```

Figure 11.1b Axis level table titles.

Once the table titles for an axis have been defined, the base and stubs must be declared. The base is the total population of respondents that qualify for that particular axis (or table). All subsequent percentages on the table are calculated to this base. The stubs are the counts (or population) of respondents who qualify for that particular answer sequence. Both the base and stubs are defined using n-statements. These n-statements are usually referred to as count-creating elements, since they are the cause of the actuals which appear on the tables. Figure 11.1c gives the general construction of a count-creating element.

```
n[n_num]stub_text;[condition];[options]
```

Figure 11.1c Construction of a count creating element.

There are a number of count-creating elements (n-statements) that can be used within an axis definition, each with a specific purpose. Figure 11.1d contains a listing of n-statements and their purpose.

summary of n statements:		
n00	**=**	filtering within an axis
n01	**=**	basic count-creating element
n03	**=**	display text only
n05	**=**	subtotal
n07	**=**	average of values contained within other elements (statistical element)
n09	**=**	page break
n10	**=**	base element
n11	**=**	non-printing base element
n12	**=**	mean element
n13	**=**	sum of factors
n15	**=**	non-printing basic counts
n17	**=**	standard deviation
n19	**=**	standard error of the mean
n23	**=**	subheading
n25	**=**	individual values for statistical elements (mean, standard deviation, etc.)
n33	**=**	text continuation

Figure 11.1d Summary of n-statement options.

The n10 statement, or base element, will generally be used immediately after the table title statements to define the number of valid respondents who have answered the question. Additionally, this base will be used for percentaging the remainder of the table.

For most tables, a series of n01 statements will follow the n10 statement. If an axis represents a question within any given questionnaire, then the n01 statement represents one of the answer choices for that particular question and a series of n01 statements represents the entire response list. The n01 statements within an axis will generally be filtered to the response list which it will represent upon the data table.

In order to filter a count-creating element, a “condition equals” statement must be used. While the “condition equals” statement is valid when filtering a series of tab statements (*section 10.3.2*), it is usually (and more commonly) used when filtering stubs within an axis file.

In order to duplicate the results of Figure 10.0a, a baseline and stubs need to be added to the current axis. Review the updated axis spec below in Figure 11.1e.

```
l q1
ttlAge of Female Respondents
ttlQ.1)
n10Base: Total Respondents
n0120-21 (20.5);c=c59'2'
n0122-28 (25.0);c=c59'3'
n0129-34 (31.5);c=c59'4'
```

Figure 11.1e Revised axis including count-creating elements.

The axis now begins to take some additional shape, and includes the basic count-creating elements. The baseline describes the population which qualifies for the axis and the n01 statements below the baseline include the data filters which describe each stub (or answer sequence). Since the count-creating baseline contains no conditional statement, the table will display a count of the total number of respondents who qualify for the table.

The n01 statements, however, DO include a conditional statement, so the counts created by these statements will only be a count of the respondents who have data which qualifies for that particular filter. From the table shown in Figure 10.0a, it can be seen that there are 30 records which qualify for the second n01 statement, and the data filter

```
c59'3'
```

in total (in addition to the other filters placed upon the table). That is, there are 30 respondents within the data file who have a punch 3 in column 59. Respondents who qualify for this conditional statement when processed will be tallied on the table.

Note that the text immediately following the n10 or n01 statement is displayed on the table exactly as typed. **ccount** displays the text immediately following the n-statement, and will not ignore spaces. Therefore, any additional spaces between the n-statement and the stub text will be translated onto the table.

Count-creating element text will display exactly as typed.

The last items seen on the table are the statistical calculations and the total. The total stub is simply a single n05 statement. This total stub is sometimes helpful in checking tables. This total can be compared against the base to make sure that the number of answers for any given question adds back to the total number of respondents who qualify for the table. Note that this is very helpful for single-response questions (questions where respondents are allowed to select only one answer), but may not be as helpful for multiple response questions (where respondents are allowed to select more than one answer).

The statistical calculations (mean, standard deviation, etc) shown in the table, while somewhat simple to use, require some explanation. Therefore, this topic is reserved for Section 11.1.5.

11.1.1 Filters within axes

Rather than using a **flt** statement to filter groups of tables within the tab section, it is possible to filter an individual axis. In order to filter an individual axis, simply use the “condition equals” statement immediately after the l-line axis declaration as shown below in Figure 11.1.2a. This filter will apply to the entire axis, but will not affect any other tables within the tab section.

```
l axis_name;[c=condition]
```

Figure 11.1.2a Filtering within an axis.

Once again, the condition must begin with a “c=” statement, and should be followed by any valid logical expression. Remember to separate the l-line axis name and the condition with a semicolon (**;**).

```
l q5b;c=c459'1'
ttlDo you remember any other ads for this product?
ttlQ.5a)
n10Base: Saw coffee ad on TV
n01Yes; c=c512'1'
n01Yes; c=c512'2'
n05Total;
```

Figure 11.1.2a Example axes file.

Within the filter in the above axis, the condition

```
c459'1'
```

is applied to the entire axis. That is, only respondents who qualify for this filter (where the condition is true) will be tallied on the table. Using filters in this way may be slightly less confusing than using filters that act on the tab statements. However, filters used this way cause the entire axis to be filtered on the specified condition each time the table is processed.

When filtering within an axis, remember to update the base line (n10) with a correct description of the qualified population for that table. This will reduce confusion when reading tables.

11.1.2 Using axes as include files using substitution

Advocating the power of include files carries over to axes files as well. Most run files will include the axes as a separate file, as shown in Figure 3.5b. This helps to differentiate the axes file from the run file, and helps with readability of the run file as well.

There may also be instances within the axes file where a particular set of answer sequences is repeated for multiple questions. It is advisable in this case to use an include file within the axes file. The following example will show yet another application of include files.

Figure 11.1.2a shows a portion of an axes file. The answer lists for the above questions (axes) are identical, which may signal to the user that an include file can be used to simplify the code. By using column substitution, the answer sequences can be simplified to look like the include file shown in Figure 11.1.2b.

```
l q5a;c=c100'1'
ttlHave you seen an ad on TV for coffee?
ttlQ.5a)
n10Base: Total Respondents
n01Yes; c=c459'1'
n01No; c=c459'2'
n05Total;

l q5b;c=c459'1'
ttlDo you remember any other ads for this product?
ttlQ.5b)
n10Base: Saw coffee ad on TV
n01Yes; c=c512'1'
n01No; c=c512'2'
n05Total;
```

Figure 11.1.2a Example axes file.

```
/*iyes file

n01Yes;c=c(a00)'1'
n01No;c=c(a00)'2'
n05Total
```

Figure 11.1.2b iyes include file.

Using the include file above, the axes in Figure 11.1.2a can be simplified to look like the axes in Figure 11.1.2c. Again, the use of include files simplifies the code and reduces the possibility of error. It also has the effect of “cleaning up” the appearance of the axes file to make it more readable. Lastly, this “iyes” file can be used as a module to be used over and over within the same axes file, or imported into other projects and included in other, unrelated axes files. The time savings and reduction of error possibilities become exponential when include files are used in an efficient manner.

```
l q5a;c=c100'1'
ttlHave you seen an ad on TV for coffee?
ttlQ.5a)
n10Base: Total Respondents
*include iyes;col(a)=459


l q5b;c=c459'1'
ttlDo you remember any other ads for this product?
ttlQ.5a)
n10Base: Saw coffee ad on TV
*include iyes;col(a)=512
```

Figure 11.1.2c Example axes file.

In fact, by becoming slightly more aggressive in regards to the substitution, the include file can be modified further so that it becomes even more generic. Aside from the original column numbers, the question number, the filter on the l-card, the question text and the baseline text can all be formatted to accept parameters for substitution.

```
/*iqyes file

l q&num;c=c(b00)'1'
ttl&text
ttlQ.&num)
n10Base: &base
n01Yes;c=c(a00)'1'
n01No;c=c(a00)'2'
n05Total
```

Figure 11.1.2d “Aggressive” axis include file.

The include file shown in Figure 11.1.2d uses substantial substitution to create a very generic template that can be used for any question which includes a “yes/no” answer choice. The axes file must be modified to fit these substitutions, and should look similar to Figure 11.1.2e.

```
*include iqyes;num=5a;col(b)=100;col(a)=459;
+base=Total Respondents;
+text=Have you seen an ad on TV for coffee?

*include iqyes;num=5b;col(b)=459;col(a)=512;
+base=Saw coffee ad on TV;
+text=Do you remember any other ads for this product?
```

Figure 11.1.2e Revised axis file using “aggressive” include file.

This aggressive substitution may not sit well with some users. However, it cannot be denied that it is a powerful application of the include file functionality.

It is recommended that include files be named meaningfully within the working directory, beginning all filenames with the letter “i”, which can help organize specs and reduce confusion.

11.1.3 Options on n-statements

There are many options that can be used to modify the functionality or change the formatting of the standard n-statement. Some of these options are listed in Figure 11.1.3a. These options are usually valid when used on any count-creating element, but are most commonly used on the n01 and n10 statements. Some of these same keywords are valid when used on the a-card as global options, but may also be repeated inside of the axis in order to activate a suppressed action or to suppress an otherwise global default option.

*All options which appear after an n-statement MUST be separated by the **ccount** default statement separator, which is a semicolon (;).*

dsp	**=**	Double-space keyword. This keyword causes a blank line to be printed immediately after the stub which it appears.
nodsp	**=**	Suppresses double-spacing.
nz	**=**	Activates the dropping of stubs which are all blank
nonz	**=**	Suppresses the nz keyword
unl(x)	**=**	Underline keyword, where (x) specifies the total number of lines to be used: **unl1** = single underline skipping blank strings **unl2** = single underline including all blanks **unl3** = single underline, skipping all blanks
inc=	**=**	Individual incremental amount to be displayed on accumulated tables (default=1); available on **n01**, **n15**, **n10**, **n11**, and **net** statements.
supp	**=**	Suppress percentages for the row in which this keyword appears (even if percentages have been requested for the remainder of the table elements)

Figure 11.1.3a Most commonly used n-statement options.

11.1.4 Netting responses

In order to display a count-creating element which nets a group of responses, use the **net** keyword and specify the net level to be used. This net statement will add up the number of respondents who are present in subsequent stubs on the table. This is useful when the number of responses outnumber the number of respondents available in a given set of stubs. In order to terminate the net level, use the **netend** keyword. An example of this is shown in Figure 11.1.4a

```
net[net_level]net_text;[options]
[n-statements to group within net]
netend[net_level]
```

Figure 11. 1.4a Summary of standard net statement.

Nets may be nested within other higher-level nets. These are called subnets. Nets may have up to 8 subnets. Each level must be specified on the net statement which begins the net group. The statement net1 is the highest level net and this level is not a subset of any other group. Statements net2, net3, and net4 through net9 are all subsets of the previous level net. A net2 statement may follow a net1 statement, and must be present for a net3 level to be applicable. A subnet may appear anywhere within the net grouping.

Nets may be nested up to nine levels deep.

It is recommended to indent statements which fall into a net grouping in order to increase readability of the table.

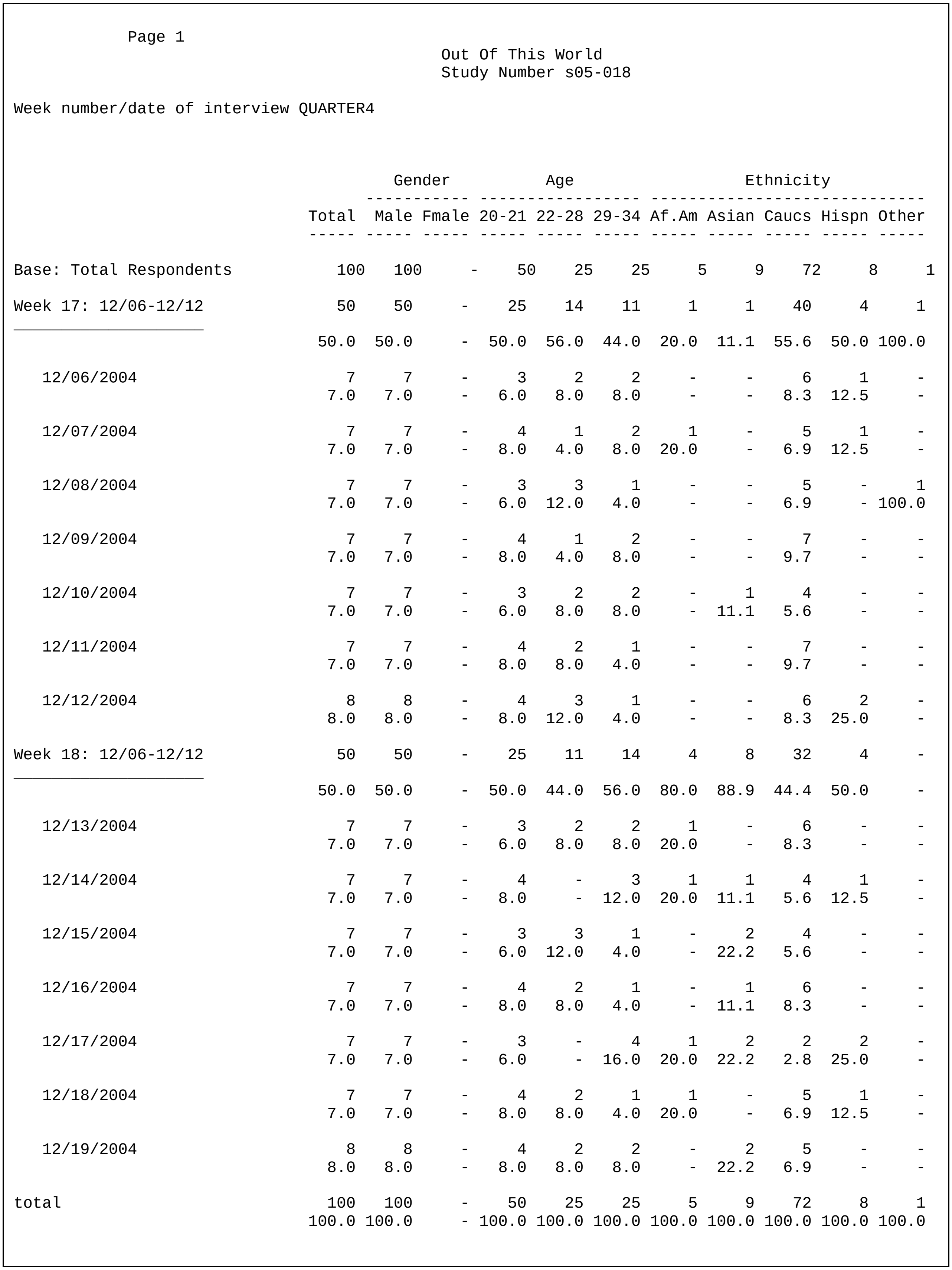
An example of net statements in use is shown in Figure 11.1.4b below. An example of the table output is shown in Figure 11.1.4c. Note that there is no netend statement after the first net1 grouping. As mentioned previously, a net grouping can be terminated by using the **netend** statement. This **netend** statement must also specify the net level which is being terminated. However, nets can also be terminated by using another net statement at the same level. Thus, a net1 statement can be terminated by using either a netend1 statement or another net1 statement. Likewise, a net2 statement may be terminated by using a netend2 or another net2 statement.

Nets can also be terminated by using a higher-level net statement. Thus, a net3 block can be terminated by a net2 statement and a net2 block can be terminated by a net1 statement.

```
l qdateq4;nz
ttlWeek number/date of interview
ttl
n10Base: Total Respondents;dsp

net1Week 17: 12/06-12/12 ;unl1
n01   12/06/2004;c=c(46,53)$20041206$
n01   12/07/2004;c=c(46,53)$20041207$
n01   12/08/2004;c=c(46,53)$20041208$
n01   12/09/2004;c=c(46,53)$20041209$
n01   12/10/2004;c=c(46,53)$20041210$
n01   12/11/2004;c=c(46,53)$20041211$
n01   12/12/2004;c=c(46,53)$20041212$

net1Week 18: 12/06-12/12 ;unl1
n01   12/13/2004;c=c(46,53)$20041213$
n01   12/14/2004;c=c(46,53)$20041214$
n01   12/15/2004;c=c(46,53)$20041215$
n01   12/16/2004;c=c(46,53)$20041216$
n01   12/17/2004;c=c(46,53)$20041217$
n01   12/18/2004;c=c(46,53)$20041218$
n01   12/19/2004;c=c(46,53)$20041219$

netend
n05total;nodsp
```

Figure 11.1.4b net statements in use within an axis.

While Figure 11.1.4b shows an example of using net1 statements, Figure 11.1.4d displays the use of subnets within nets. It quickly becomes apparent that the usage is very similar. The output, however, is slightly different as shown in Figure 11.1.4e.



```
            Page 1
                                             Out Of This World
                                             Study Number s05-018

Week number/date of interview QUARTER4



                                        Gender          Age                 Ethnicity
                                     ----------- ----------------- -----------------------------
                               Total  Male Fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
                               ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

Base: Total Respondents          100   100     -    50    25    25     5     9    72     8     1

Week 17: 12/06-12/12              50    50     -    25    14    11     1     1    40     4     1

                                50.0  50.0     -  50.0  56.0  44.0  20.0  11.1  55.6  50.0 100.0

   12/06/2004                      7     7     -     3     2     2     -     -     6     1     -
                                 7.0   7.0     -   6.0   8.0   8.0     -     -   8.3  12.5     -

   12/07/2004                      7     7     -     4     1     2     1     -     5     1     -
                                 7.0   7.0     -   8.0   4.0   8.0  20.0     -   6.9  12.5     -

   12/08/2004                      7     7     -     3     3     1     -     -     5     -     1
                                 7.0   7.0     -   6.0  12.0   4.0     -     -   6.9     - 100.0

   12/09/2004                      7     7     -     4     1     2     -     -     7     -     -
                                 7.0   7.0     -   8.0   4.0   8.0     -     -   9.7     -     -

   12/10/2004                      7     7     -     3     2     2     -     1     4     -     -
                                 7.0   7.0     -   6.0   8.0   8.0     -  11.1   5.6     -     -

   12/11/2004                      7     7     -     4     2     1     -     -     7     -     -
                                 7.0   7.0     -   8.0   8.0   4.0     -     -   9.7     -     -

   12/12/2004                      8     8     -     4     3     1     -     -     6     2     -
                                 8.0   8.0     -   8.0  12.0   4.0     -     -   8.3  25.0     -

Week 18: 12/06-12/12              50    50     -    25    11    14     4     8    32     4     -

                                50.0  50.0     -  50.0  44.0  56.0  80.0  88.9  44.4  50.0     -

   12/13/2004                      7     7     -     3     2     2     1     -     6     -     -
                                 7.0   7.0     -   6.0   8.0   8.0  20.0     -   8.3     -     -

   12/14/2004                      7     7     -     4     -     3     1     1     4     1     -
                                 7.0   7.0     -   8.0     -  12.0  20.0  11.1   5.6  12.5     -

   12/15/2004                      7     7     -     3     3     1     -     2     4     -     -
                                 7.0   7.0     -   6.0  12.0   4.0     -  22.2   5.6     -     -

   12/16/2004                      7     7     -     4     2     1     -     1     6     -     -
                                 7.0   7.0     -   8.0   8.0   4.0     -  11.1   8.3     -     -

   12/17/2004                      7     7     -     3     -     4     1     2     2     2     -
                                 7.0   7.0     -   6.0     -  16.0  20.0  22.2   2.8  25.0     -

   12/18/2004                      7     7     -     4     2     1     1     -     5     1     -
                                 7.0   7.0     -   8.0   8.0   4.0  20.0     -   6.9  12.5     -

   12/19/2004                      8     8     -     4     2     2     -     2     5     -     -
                                 8.0   8.0     -   8.0   8.0   8.0     -  22.2   6.9     -     -

total                            100   100     -    50    25    25     5     9    72     8     1
                               100.0 100.0     - 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0
```

Figure 11.1.4c Example of standard table using nets.

```
l qsatstore
ttlSatisfaction with Store
ttlq.bb
n10Base: Total Male Respondents;dsp

net1Store (net);unl1
net2  Convenience (subnet);unl1
n01     Convenience of location;c=c1949'1'
n01     Convenience of hours;c=c1950'1'
n01     Cleanliness;c=c1951'1'
netend2
n01   Selection;c=c1945'1'

net1Sales Staff (net);unl1
n01  Courteousness;c=c1947'1'
n01  Helpfulness;c=c1948'1'
net2  Explanation (subnet);unl1
n01     Explanation of features;c=c58'2'
n01     Explanation of return policy;c=c58'3'
n01     Explanation of warranty;c=c1876'2'
netend

n05total;nodsp
```

Figure 11.1.4d net and subnet statements within an axis.

Note the use of spaces within the axis on the n-statements. This serves to make the table more readable, and also helps to offset the net and subnet responses so that they are not confused with the other count-creating elements of the table. Other than the addition of the net2 statements, this axis differs from the previous (Figure 11.1.4b) in that it contains a netend2 statement.

It is recommended that the use of "(net)", "(subnet)" and "(sub-subnet)" text be specified on the appropriate net statements in order to avoid confusion when deciphering tables.

```
            Page 3
                                             Out Of This World
                                             Study Number s05-018

Satisfaction with Store
q.bb


                                        Gender          Age                   Ethnicity
                                     ----------- ----------------- -----------------------------
                               total  male fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
                               ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

Base: Total Male Respondents     100   100     -    50    25    25     5     9    72     8     1

Store (net)                       40    40     -    20    10    10     2     2    28     6     -

                                40.0  40.0     -  40.0  40.0  40.0  40.0  22.2  38.9  75.0     -

  Convenience (subnet)            38    38     -    20     8    10     2     2    27     5     -

                                38.0  38.0     -  40.0  32.0  40.0  40.0  22.2  37.5  62.5     -

     Convenience of location       9     9     -     5     2     2     -     -     6     2     -
                                 9.0   9.0     -  10.0   8.0   8.0     -     -   8.3  25.0     -

     Convenience of hours         24    24     -    11     4     9     2     1    16     3     -
                                24.0  24.0     -  22.0  16.0  36.0  40.0  11.1  22.2  37.5     -

     Cleanliness                  12    12     -     7     2     3     1     1     8     2     -
                                12.0  12.0     -  14.0   8.0  12.0  20.0  11.1  11.1  25.0     -

   Selection                       3     3     -     1     2     -     -     -     2     1     -
                                 3.0   3.0     -   2.0   8.0     -     -     -   2.8  12.5     -

Sales Staff (net)                 48    48     -    25    11    12     2     4    35     2     -

                                48.0  48.0     -  50.0  44.0  48.0  40.0  44.4  48.6  25.0     -

  Courteousness                    9     9     -     4     2     3     1     2     5     -     -
                                 9.0   9.0     -   8.0   8.0  12.0  20.0  22.2   6.9     -     -

  Helpfulness                     13    13     -     9     1     3     1     -    12     -     -
                                13.0  13.0     -  18.0   4.0  12.0  20.0     -  16.7     -     -

  Explanation (subnet)            35    35     -    17    10     8     -     3    26     2     -

                                35.0  35.0     -  34.0  40.0  32.0     -  33.3  36.1  25.0     -

     Explanation of features      23    23     -     9    10     4     -     2    17     2     -
                                23.0  23.0     -  18.0  40.0  16.0     -  22.2  23.6  25.0     -

     Explanation of return         8     8     -     6     -     2     -     1     5     -     -
policy                           8.0   8.0     -  12.0     -   8.0     -  11.1   6.9     -     -

     Explanation of warranty       5     5     -     3     -     2     -     -     5     -     -
                                 5.0   5.0     -   6.0     -   8.0     -     -   6.9     -     -

total                            106   106     -    55    23    28     5     7    76    10     -
                               106.0 106.0     - 110.0  92.0 112.0 100.0  77.8 105.6 125.0     -
```

Figure 11.1.4e Example of standard table using nets and subnets.

11.1.5 Statistical elements on axes

In certain situations, it may be necessary to apply statistical elements and calculations in order to extract additional information from the data. **ccount** provides the ability to apply statistical elements directly into tables without having to code complex formulas. These elements are displayed in Figure 11.1.5a below.

summary of statistical n statements:

n12	**=**	mean element
n17	**=**	standard deviation
n19	**=**	standard error of the mean
n25	**=**	individual values for means and other statistical elements

Figure 11.1.5a Summary of statistical n-statement options.

These statistical n-statements are most commonly applied to numeric response questions or rating scales, where an average score may be of interest. During processing and the accumulation of the tables within the tab section, **ccount** keeps a running total, or tally, of the responses called a “sum of factors”. This sum is then divided by the total number of records present in the data file. When specified by an n12 statement, this results in the mean, or average score being printed onto the table. Similarly, the n17 statement causes the standard deviation of the mean to be reported to the table, and n19 will display the standard error of the mean.

In normal processing flow, **ccount** requires an element value to be assigned with which to calculate the statistic. It is therefore necessary to “force” a particular value to be used as the factor in the calculation. This can be accomplished a number of different ways, but is most commonly applied through the use of the **fac=** keyword. Figure 11.1.5b shows the generic usage of the **fac=** keyword, which is normally applied to a standard count creating n01 statement. Note that the factor score must follow immediately after the **fac=** keyword.

```
n01element_text;c=condition[;fac=factor_score][;inc=increment]
```

Figure 11.1.5b Usage of the fac= keyword.

This **fac=** statement assigns a factor score for the stub, which is then used in any statistical calculations. If an element does not include the **fac=** statement, then it will not be factored into the calculation.

Figure 11.1.5c shows usage of the **fac=** and statistical n-statements upon the example axis shown in Figure 11.1e. Note that the **fac=** statement includes the factor score value. This value can be any valid numerical value, including real numbers (decimals). The n12, n17, and n19 statements follow the n01 statements in the axis, and calculate the mean, standard deviation, and standard error respectively based on the factor values supplied in the **fac=** statement.

There are a number of n-statement options which are valid to use on statistical elements. The most common of these are seen in the example below. The option **dec=2** specifies the number of decimal places to be used when printing the result on the tables. The **nodsp** keyword specifies that these elements are to be printed without double-spacing the rows. The **nocol** keyword specifies that the element is not a column element. This **nocol** keyword causes the stub to be suppressed if the axis in which this stub appears is used as a column axis. That is, if this axis is to be used as a column axis (banner), these statistical elements will not be displayed. Lastly, the **nonz** option forces the row to be printed on the table, regardless if the entire row is null.

```
l q1
ttlAge of Female Respondents
ttlQ.1)
n10Base: Total Respondents
n0120-21 (20.5);c=c59'2';fac=20.5
n0122-28 (25.0);c=c59'3';fac=25.0
n0129-34 (31.5);c=c59'4';fac=31.5
n12Mean;dec=2;nodsp;nocol;nonz
n17Std. Dev.;nodsp;nocol;nonz
n19Std. Err.;nodsp;nocol;nonz
n03
n05Total;nodsp
```

Figure 11.1.5c Revised axis including statistical count-creating elements.

While the **fac=** statement is used often when dealing with certain rating scales and some stubs which represent value ranges, there is another statistical element which uses the actual value contained within the column (or columns) as the factor score. This element, the **n25** statement (shown in Figure 11.1.5a) is useful in calculating the factor score when the value contained in the data column (or columns) is sufficient. This is most commonly used on rating scales where the highest punch value corresponds to the highest rating on the scale, and on questions where the exact age of the respondent has been captured in the data.

```
l q3
ttlProbability of Purchase
ttlQ.3)
n10Base: Total Respondents
net1Top 2 Box (net);unl1
n01Definitely would;c=c272'5'
n01Probably would;c=c272'4'
netend1
n01Might or might not;c=c272'3'
net1Bottom 2 box (net);unl1
n01Probably would not;c=c272'2'
n01Defintely would not;c=c272'1'
netend1
n25;inc=c272;c=c272'1/5'
n12Mean;dec=2;nodsp;nocol;nonz
n17Std. Dev.;nodsp;nocol;nonz
n19Std. Err.;nodsp;nocol;nonz
n03
n05Total;nodsp
```

Figure 11.1.5d Revised axis including n25 elements.

Figure 11.1.5d shows an example of a fairly common rating scale axis using the **n25** statement to assist in calculation of the mean. In this example, the factor scores are derived directly from the data values (punches) present. The n12, n17, and n19 elements then display the statistical elements on the table in the same way as in the previous example. In an **n25** statement, the incremental value, **inc=**, must be specified. The condition (or conditions) under which this incremental is to occur also needs to be specified. This **inc=** statement can be considered as taking the place of the **fac=** statement. Each statement has its own benefits and drawbacks, and it is up to the user to decide which format is most understandable and convenient.

Conditional (c=) statements may be applied to n25 statements in order to exclude certain data records from the calculations.

11.1.6 Special conditions within axes

There are many conditions which can be used that are completely unique to the tabulation section. Within the tab axis, there are also a number of special conditions which may be used in order to accumulate record counts in specific ways. These special conditions are used exclusively with the n statements in order to produce these counts.

summary of special n-statement conditions:

c=- = count of the number of records not included in any element (other than the base)

c=+ = count of the number of records already included within any element (excluding the base)

c=-*n* = counts number of records not included in any of the previous n elements

c=+*n* = counts number of records already included within the previous n elements

Figure 11.1.6a Summary of special n-statement conditions.

These special conditional statements are very useful in determining if the listed elements of an axis provide a comprehensive definition of the axis. For instance, consider the axis definition given in Figure 11.1.6b and the table generated by this axis, shown in Figure 11.1.6c.

```
l q27a
ttlWhat age bracket most accurately defines you?
ttlQ.27a)
n10Base: Total Respondents
n01under 18 years old;c=c111’1’
n0118-23 years old   ;c=c111’2’
n0124-29 years old   ;c=c111’3’
n0130-39 years old   ;c=c111’4’
n0140-49 years old   ;c=c111’5’
n0150-59 years old   ;c=c111’6’
n0160 years or older ;c=c111’7’
n05Total;nodsp
```

Figure 11.1.6b Example axes file.

```
  Page 1
                                            Out Of This World
                                            Study Number s05-018

What age bracket most accurately defines you?
Q.27a)

                              Total
                              -----

Base: Total Respondents         245

under 18 years old               21
                                8.6

18-23 years old                  15
                                6.1

24-29 years old                  53
                               21.6

30-39 years old                  35
                               14.3

40-49 years old                  18
                                7.3

50-59 years old                  29
                               11.8

60 years or older                29
                               11.8

Total                           200
                               81.6
```

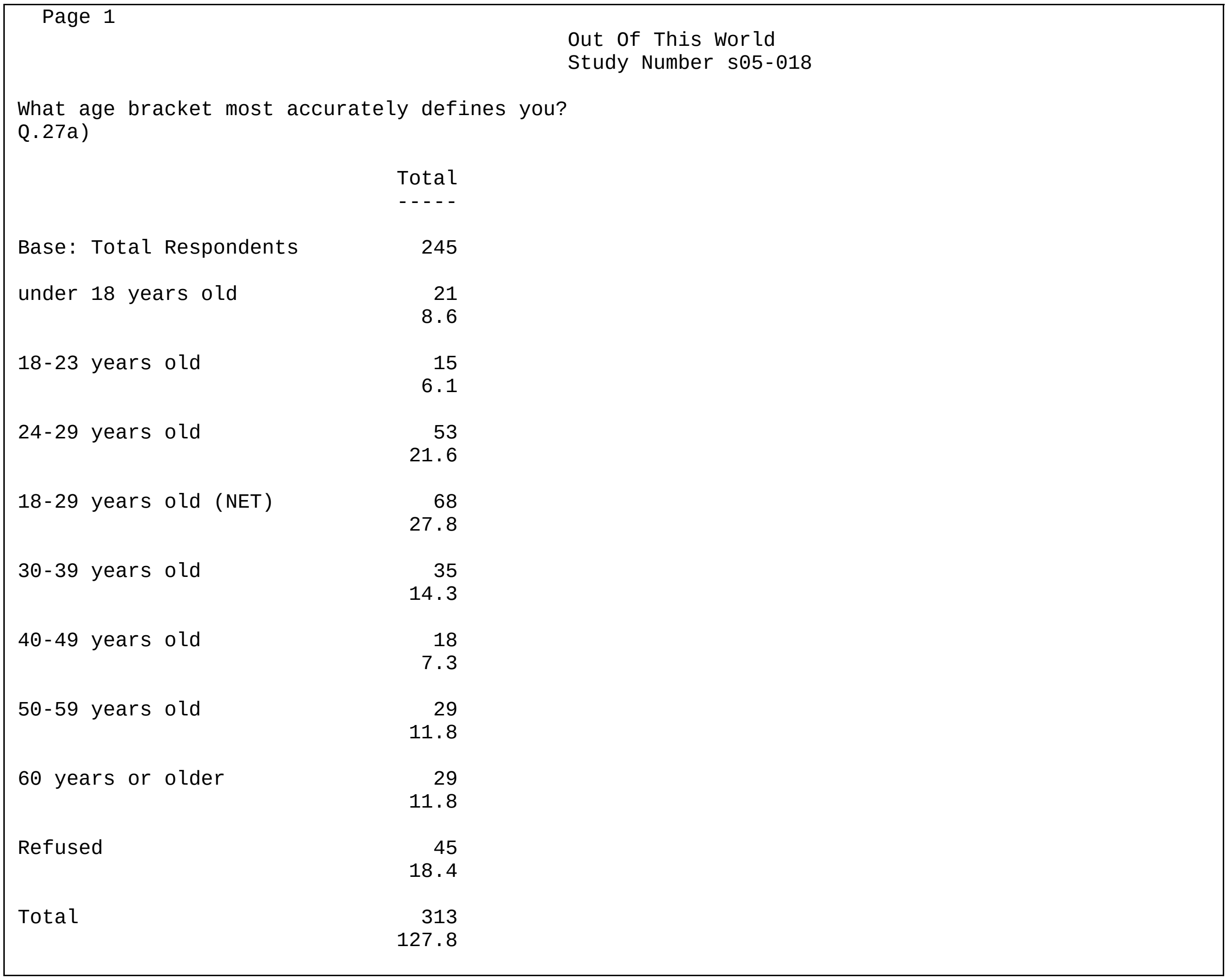
Figure 11.1.6c Example table.

As can be seen by the example, there seems to be some discrepancy between the number of respondents defined within the base, and the number of respondents answering the question. The ***c=-*** special condition can be used for this purpose, and can help identify respondents who fall within the base of the question, but have not yet been counted within any of preceding elements.

The **c=+** condition can be used in order to provide a "net" or total number of respondents who have been counted up to a certain point within the axis. Updating the axis definition as shown in Figure 11.1.6d gives the revised table output shown in Figure 11.1.6e.

```
l q27a
ttlWhat age bracket most accurately defines you?
ttlQ.27a)
n10Base: Total Respondents
n01under 18 years old;c=c111’1’
n0118-23 years old   ;c=c111’2’
n0124-29 years old   ;c=c111’3’
n0118-29 years old (NET);c=+2
n0130-39 years old   ;c=c111’4’
n0140-49 years old   ;c=c111’5’
n0150-59 years old   ;c=c111’6’
n0160 years or older ;c=c111’7’
n01Refused;          ;c=-
n05Total;nodsp
```

*Figure 11.1.6d Example axes file using special condition **c=-**.*



```
  Page 1
                                              Out Of This World
                                              Study Number s05-018

What age bracket most accurately defines you?
Q.27a)

                               Total
                               -----

Base: Total Respondents          245

under 18 years old                21
                                 8.6

18-23 years old                   15
                                 6.1

24-29 years old                   53
                                21.6

18-29 years old (NET)             68
                                27.8

30-39 years old                   35
                                14.3

40-49 years old                   18
                                 7.3

50-59 years old                   29
                                11.8

60 years or older                 29
                                11.8

Refused                           45
                                18.4

Total                            313
                               127.8
```

Figure 11.1.6e Revised example table.

The **c=+** condition is used to provide a net of responses for the 18-23 and 24-29 year old age brackets. Note that the number of elements to be included within the net has been specified as well. The **c=-** special condition is used to identify respondents who have not answered the question. This option is especially valuable when blank (or missing) data is to be considered akin to "Don't Know", "Refused", "Prefer Not To Answer" or "No Answer" responses commonly found within surveys.

*Be sure that all possible data elements are defined before resorting to the use of the **c=-** condition. This condition can be a "catch-all" condition, and may mask actual data results if all possible data values are not a**ccount**ed for within the axis definition.*

11.1.7 Counting numeric codes from an array

In order to create an axis which can count numeric codes across a series of columns, it is necessary to first define the field by setting up an array. This array must be defined before the edit section in the same manner as any generic variables are to be defined. These variables will also be assigned values within the edit section, but the values will not be specifically referenced until the tab section of the run file. Figure 11.1.7a shows an example of the defined variable, and how it is further defined as an array within the edit section.

```
int cars 99s
ed

field cars=c77,c127:2

end
```

Figure 11.1.7a Defining an integer variable for use as an array.

In order to specify an integer variable as an array, the **field** keyword must be used. This field statement defines the array as beginning at column 77 and ending at column 127 and specifies the field length of each value to be 2 columns wide. This specification is reminiscent of the looping construct and proves to be just as flexible. Figure 11.1.7b shows the many different generic ways that a field may be defined.

```
field array_name=start_col, end_col:field_width

field array_name=(column1,column2,column3,[additional_columns]):field_width

field array_name=start_col,end_col / start_col,end_col:field_width

field array_name=start_col / start_col [/ start_col] :field_width
```

Figure 11.1.7b Usage of the field statement.

As with looping constructs, the options for defining a field are varied. Consecutive columns can be part of a field, in which case only the start and end columns need to be specified. Consecutive columns may be defined individually, in which case they must be encased within parenthesis. Non-consecutive columns may be defined by separating the columns with a slash (**/**). Similarly, non-consecutive column groupings can also be separated by a slash character. In all of these examples, the field width must be defined, and must be separated from the column specifications with a colon (**:**).

Since a field array only increments its cells based on numeric codes, it is possible to “redefine” non-numeric items within the field statement so that they are treated as numeric codes. In order to redefine a non-numeric code, the code must be specified within the field statement followed by the equivalent cell within the array which is to be incremented when this code is encountered.

```
field array_name=start_col, end_col:field_width, $code$=cell_number
```

Figure 11.1.7c Field statement using numeric recoding of non-numeric values.

Once the field has been defined and all values have been a**ccount**ed for (and recoded, if necessary), this array can be used during the table generation portion of the **ccount** run to produce tables. Using the field array within an axis produces a total count of each numerical value found within the array, and can be used as filers or conditions within axes and/or banners.

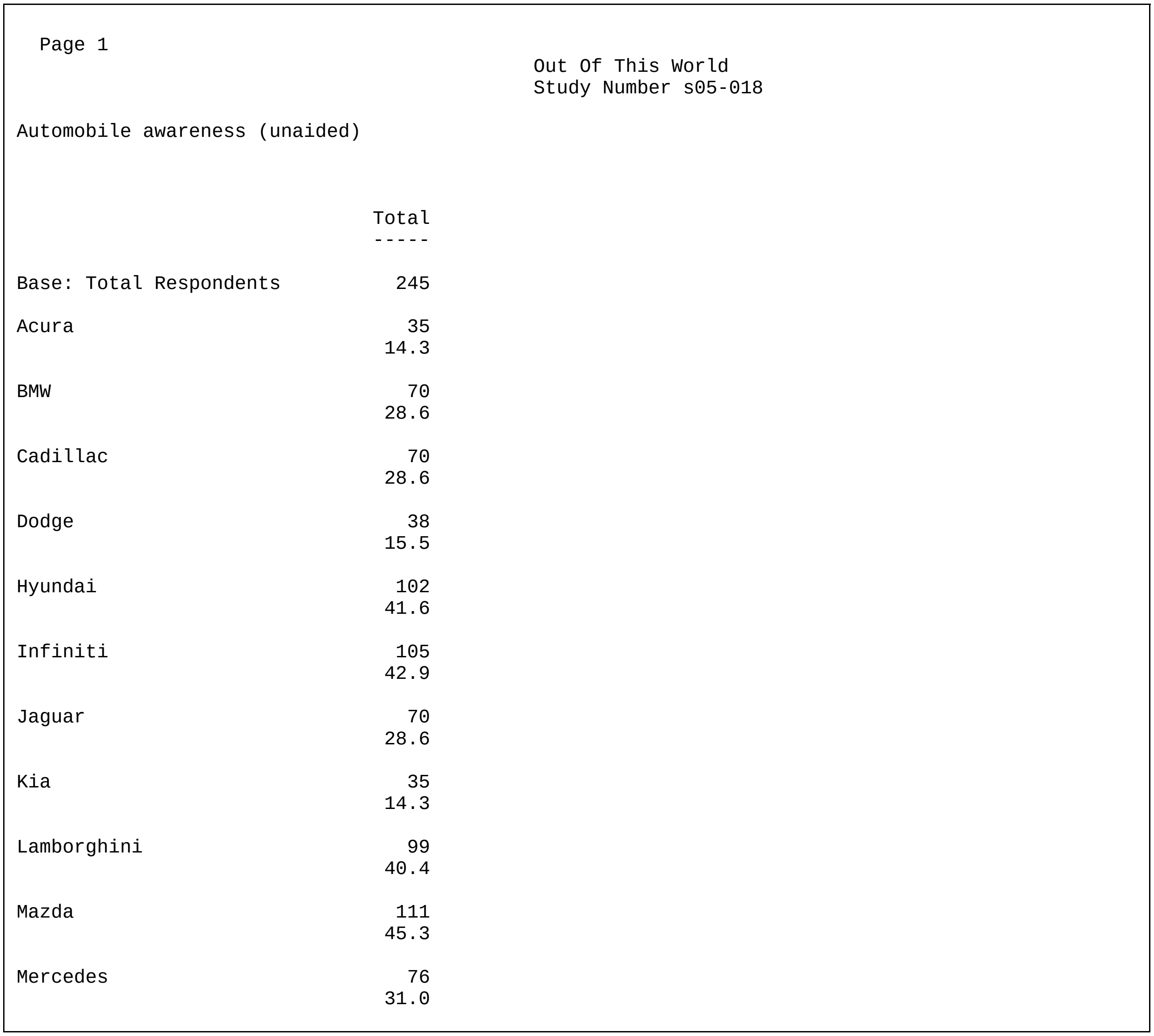
```
l brands1
ttlAutomobile awareness (unaided)
n10Base: Total Respondents
n01Acura      ;c=cars(01)
n01BMW        ;c=cars(02)
n01Cadillac   ;c=cars(03)
n01Dodge      ;c=cars(24)
n01Hyundai    ;c=cars(05)
n01Infiniti   ;c=cars(06)
n01Jaguar     ;c=cars(07)
n01Kia        ;c=cars(20)
n01Lamborghini;c=cars(18)
n01Mazda      ;c=cars(10)
n01Mercedes   ;c=cars(26)
```

Figure 11.1.7d Usage of field array within axis.

As is shown in Figure 11.1.7d, the usage of a defined field array within an axis follows the standard construction. To check whether a specific value exists within the defined array, simply refer to that value within parenthesis after naming the array to be searched. In the example above, cars(01) refers to the value “1”, or “01”, and cars(38) refers to the value “38”. If these values exist in the array cars, then the condition will evaluate to true.

It is for this reason that fields can be used within conditional statements as binary operators. That is, they can be used in terms of being “true” or “false”. In the example above, if code “5” or “05” appears in the original defined column locations (and subsequently in the field cars) then cars(05) will evaluate to true or false within the conditional statement. There is no need to use logic connectors to try and evaluate a value for cars(05). The value referred to by cars(05) either exists or not. Once a field array has been defined, the usage is very simple, and proves to be very helpful when values appear in a completely random fashion within a field in the data file.

Figure 11.1.7e shows the output table created by the axis in Figure 11.1.7d.



```
  Page 1
                                                             Out Of This World
                                                             Study Number s05-018

Automobile awareness (unaided)



                               Total
                               -----

Base: Total Respondents          245

Acura                             35
                                14.3

BMW                               70
                                28.6

Cadillac                          70
                                28.6

Dodge                             38
                                15.5

Hyundai                          102
                                41.6

Infiniti                         105
                                42.9

Jaguar                            70
                                28.6

Kia                               35
                                14.3

Lamborghini                       99
                                40.4

Mazda                            111
                                45.3

Mercedes                          76
                                31.0
```

Figure 11.1.7e Example table created using field arrays.

11.2 The banner file

Although any axis can be used to create the columns of a table (called the banner), this task is usually reserved for a specially-defined axis in which the columns are explicitly laid out for readability. This specially-defined axes is defined manually by using what are referred to as “g statements” (or “g-cards”) and “p statements” (or “p-cards”). Although this specially-defined banner contains some additional items that the average axis does not, a banner can be considered nothing more than a glorified axis.

Banners are defined in the same manner as most axes, beginning with an l-card. This l-card defines the banner axis name, and includes any conditions which the banner should be filtered by. This l-card definition is followed by the g-cards and p-cards, which specify the text format that the banner will have, while the p-cards define the position where the actuals (counts) in the table are to be aligned.

```
l ban_name;[c=condition]
[tstat statistical testing specifications]
g row text
[g row text]
[g row text]
[g row text]
p    *    [*]    [*]    [*]
n01;[condition for point 1]
[n01;condition for point 2]
[n01;condition for point 3]
...
...
...
```

Figure 11.2a Generic construction of a banner axis.

Filters placed on the l-card of a banner apply to ALL axes tabbed against that banner.

Immediately after the g-cards and p-cards, the definitions for each point must be given, identical to the set up of a normal axis using n-statements. The g-cards and p-cards simply take the place of the usual table titles and display at the top of each table, creating the columns. Text is displayed on the table exactly as typed (formatted) on the g-card, with a new line of text displayed on the table for each g-card. The p-card points can be defined with any non-blank character, but are generally defined by using asterisks (*) or any letter of the alphabet.

Figure 11.2a shows the generic construction of a banner axis, while Figure 11.2b gives an example of a specific banner axis.

Actuals (cell counts) are displayed on the table right-justified based on the location of the p-card points.

The number of p-card points MUST match the number of n-statement definitions given below the axis or syntax errors will result at compile-time.

Note that the text on the n01 statements is optional. **ccount** will ignore any text on the n-statements when g-cards are present in the axis. g-card text takes precedence over any text which is defined on n-statements. However, including text on the n-statements increases readability of the banner axis, and helps to avoid confusion when defining the banner points. This is especially true when abbreviations are used on the banner points, since banner points are usually limited in width in order to fit on the page.

---

g-card banner points display text exactly as typed.

```
l qbanner
g          Gender          Age                  Ethnicity
g       ----------- ----------------- -----------------------------
g total  Male Fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
g ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
p     a     b     c     d     e     f     g     h     j     k     l
n10total
/**gender
n01males  ;c=c56'1'
n01females;c=c56'2'
/**age
n0120-21 ;c=c59'2'
n0122-28 ;c=c59'3'
n0129-34 ;c=c59'4'
/**ethnicity
n01African American  ;c=c(2121,2122).eq.1
n01Asian             ;c=c(2121,2122).eq.2
n01Caucasian         ;c=c(2121,2122).eq.3
n01Hispanic          ;c=c(2121,2122).eq.4
n01Other             ;c=c(2121,2122).eq.7
```

Figure 11..2d Banner axis qbanner defined within the banner file.

The width of the defined banner should not exceed the width of the page which it will be printed on. Use judgment when defining banner points.

n-statement text will be ignored when g-cards are present in an axis.

There is technically no limit to the number of g-cards that any one banner may have. However, each banner should contain no more than is absolutely necessary, since table readability may be affected.

```
/** run1
struct;read=0;ser=c(1,7);reclen=1000

ed
filedef respids.txt report
reportn respids.txt c(1,7)

*include icounts

end
a;dsp;flush;nopc;notype;op=12;side=30;spechar=-*
++;printz;pagwid=150;paglen=70

ttcOut Of This World
ttcStudy Number s05-018

*include tabs;ban=qbanner

*include axes
*include banner
```

*Figure 11.1.2e Example run file using *include statement for banner file.*

As with the axes file, the banner is usually stored in a separate file, called banner. This file is then included in the main run file, immediately after the axes file. This is illustrated in Figure 11.1.2e below. This serves to improve readability in the main run file, as well as separate the banner axis (or banner axes) from the standard axes within the project. Portability of specs is also maintained by creating a separate banner file.

11.2.1 Options on banner l-cards (l-line)

Similar to filtering axes, banners may be filtered as well. Filtering a banner is accomplished in the same fashion as filtering an axis. Simply place a "condition equals" (c=) statement immediately after the l-card banner name definition. Keep in mind that this will cause the banner to be filtered by this condition each time it is tabbed. Figure 11.2.1a shows an example of a filtered banner.

```
 l ban2;c=c1058'1'
g                      Cells                        Segments                    Target
g       ----------------------------------- ----------------------- -----------------------------
g Total   1     2     3     4     5     6     A     B     C     D   Total  Male Fmale QF=1  QF=2
g ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
p     a     b     c     d     e     f     g     h     j     k     l     m     n     p     q     r
n10total
/**cells
n01;c=c24'1'
n01;c=c24'2'
n01;c=c24'3'
n01;c=c24'4'
n01;c=c24'5'
n01;c=c24'6'
/**segments
n01;c=(c22'345'.and.(c18+c19+c20).ge.7)
n01;c=(c22'345'.and.(c18+c19+c20).le.6)
n01;c=(c22'12'.and.(c18+c19+c20).ge.7)
n01;c=(c22'12'.and.(c18+c19+c20).le.6)
/**target
n00;c=c17'12'.and.((c18+c19+c20).le.6.or.c22'12')
n10total target
n01;c=c12'1'
n01;c=c12'2'
n01;c=c17'1'
n01;c=c17'2'
```

Figure 11.2.1a Example banner using l-card filter.

Banners may be filtered using any valid logical expression.

11.2.2 Statistical testing of banner points

This feature is in development, and is not currently supported.

11.2.3 Other options on banners [3]

Columns of figures may be separated by pipe characters (|). These vertical dividers must be defined on the p-card of the banner axis. These pipe characters will be displayed throughout the table, in the positions where they are defined on the p-card. Refer to the example banner shown below (p-card point n-statement definitions are omitted, but can be seen in Figure 11.2d).

[3] This feature is not currently supported.

```
l qbanner
g          Gender          Age                   Ethnicity
g       ----------- ----------------- -----------------------------
g total  Male Fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
g ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
p     a|    b     c|    d     e     f|    g     h     j     k     l
```

Figure 12.2.3a Banner axis g-cards and p-cards with pipe characters.

In this example the pipe characters in the p-card will be printed on the accumulated table or each stub row that is defined for that table. Figure 12.2.3b displays the output of the banner defined above.

```
  Page 1
                                             Out Of This World
                                             Study Number s05-018

                                             ** FEMALES ONLY **
Age of Female Respondent
q.bb
                                        Gender          Age                Ethnicity
                                     ----------- ----------------- -----------------------------
                               Total  Male Fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
                               ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

Base: Total Female Respondents    50|    -    50|    2    30    18|    7     3    36     2     2
                                    |           |                 |
20-21 (20.5)                       2|    -     2|    2     -     -|    -     -     2     -     -
                                 4.0|    -   4.0|100.0     -     -|    -     -   5.6     -     -
                                    |           |                 |
22-28 (25.0)                      30|    -    30|    -    30     -|    6     2    19     1     2
                                60.0|    -  60.0|    - 100.0     -| 85.7  66.7  52.8  50.0 100.0
                                    |           |                 |
22-28 (25.0)                      30|    -    30|    -    30     -|    6     2    19     1     2
29-34 (31.5)                      18|    -    18|    -     -    18|    1     1    15     1     -
                                36.0|    -  36.0|    -     - 100.0| 14.3  33.3  41.7  50.0     -
                                    |           |                 |
Mean                           27.16| 0.00 27.16|20.50 25.00 31.50|25.93 27.17 27.46 28.25 25.00
Std. Dev.                       3.40| 0.00  3.40| 0.00  0.00  0.00| 2.46  3.75  3.61  4.60  0.00
Std. Err.                       0.48| 0.00  0.48| 0.00  0.00  0.00| 0.93  2.17  0.60  3.25  0.00
                                    |           |                 |
Total                             50|    -    50|    2    30    18|    7     3    36     2     2
                               100.0|    - 100.0|100.0 100.0 100.0|100.0 100.0 100.0 100.0 100.0
```

Figure 11.2.3b Table output using pipe characters in p-card definition.

11.2.4 Banners without g-cards and p-cards

It is also possible to create tables by using a standard axis as a banner. By adding several options to any given axis, it is possible to create a banner which can be just as readable as one created with g and p cards.

As mentioned previously, any given axis can be used as a banner. Simply use the axis name within the tab statement as the column axis, and **ccount** will format the table accordingly.

*Remember that the **colwid** keyword must be used on the a-line in order to define a specific column width. Otherwise, **ccount** will use the default formula for calculating column width.*

For instance, the axis shown in Figure 11.2.4a is a standard axis which defines region codes.

```
l qregion
ttlregion
ttl
n10base: total respondents;dsp
n01East;c=c(12,13).eq.1
n01MidWest;c=c(12,13).eq.2
n01South;c=c(12,13).eq.3
n01West;c=c(12,13).eq.4
n05Total;
```

Figure 11,2,4a Simple axis.

When used in the following tab statement as both the row axis and column axis,

```
tab qregion qregion
```

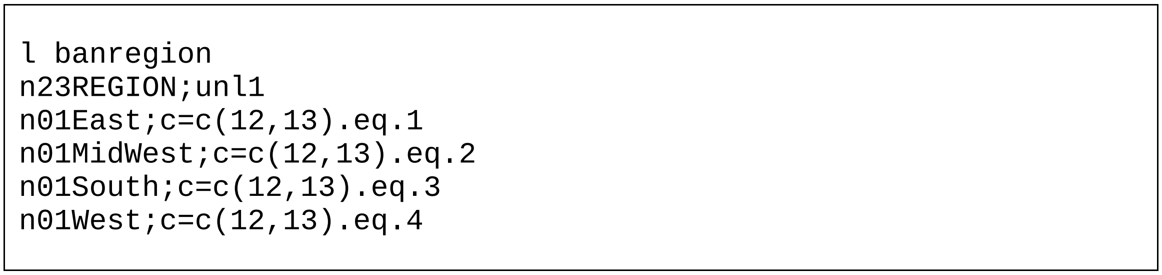
ccount will produce the table seen in Figure 11.2.4b.

```
  Page 1
                                             Out Of This World
                                             Study Number s05-018

Region



                                    Base:
                                    Total
                              Respondents          East    MidWest      South       West      Total

Base: Total Respondents               435            96        112        165         62        435

East                                   96            96          -          -          -         96
                                     22.1         100.0          -          -          -       22.1

MidWest                               112             -        112          -          -        112
                                     25.7             -      100.0          -          -       25.7

South                                 165             -          -        165          -        165
                                     37.9             -          -      100.0          -       37.9

West                                   62             -          -          -         62         62
                                     14.3             -          -          -      100.0       14.3

Total                                 435            96        112        165         62        435
                                    100.0         100.0      100.0      100.0      100.0      100.0
```

Figure 11.2.4b Cross tab of region axis by region axis.

While some may feel that this banner is not especially “pretty”, this table is, in fact, neatly formatted. This is a much faster option for tabbing than creating a custom banner (or banners) using g and p cards.

In order to achieve a slightly more “custom” effect, the axis “qregion” may be formatted as shown below in Figure 11.2.4c. This axis uses the **n23** subheading statement in order to achieve a similar effect to using g-cards. Note that the extraneous n10 baseline and the n05 total statements have been omitted. Since these items appear on the table as stubs, there is no reason to have them appear again in the banner. Omitting these items from the banner gives the banner a more “clean” appearance, which results in a more readable table overall.

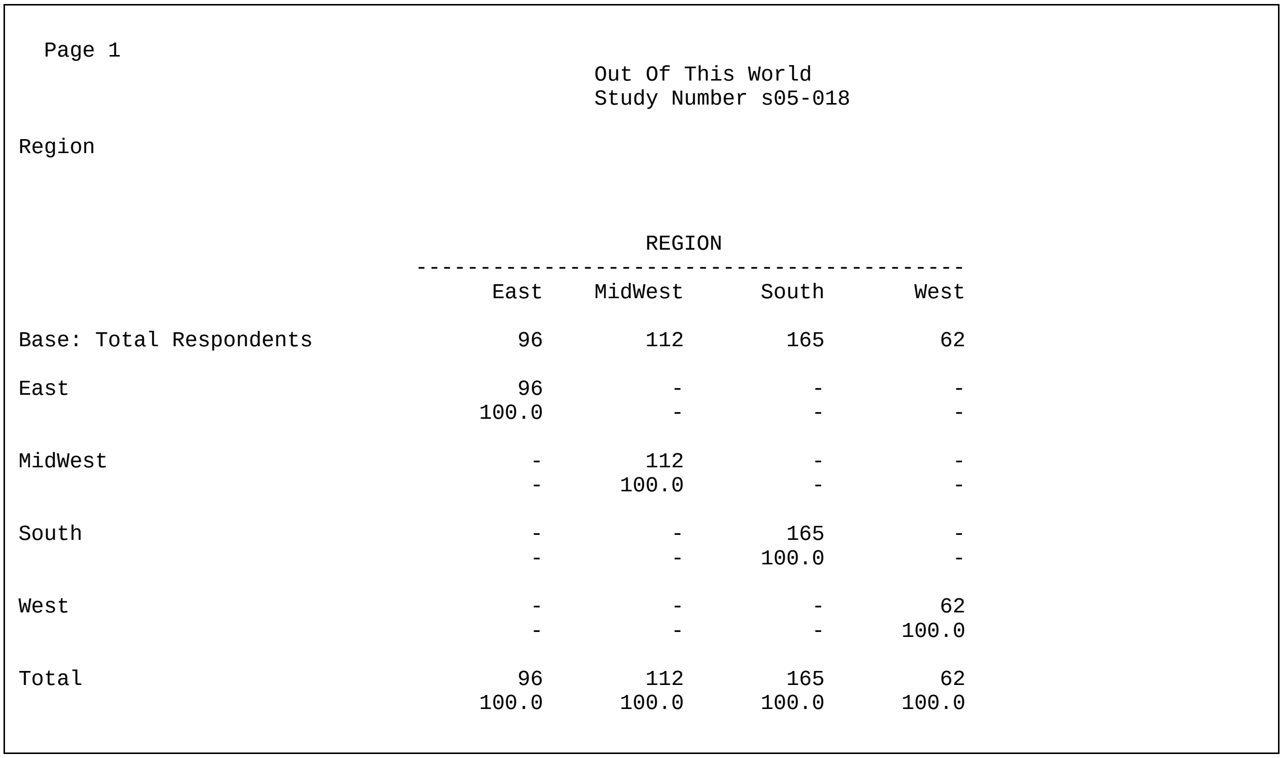


```
l banregion
n23REGION;unl1
n01East;c=c(12,13).eq.1
n01MidWest;c=c(12,13).eq.2
n01South;c=c(12,13).eq.3
n01West;c=c(12,13).eq.4
```

Figure 11,2,4c Revised axis for use as a banner axis.

The table shown in Figure 11.2.4d shows the result of the tab statement:

```
tab qregion banregion
```



```
  Page 1
                                          Out Of This World
                                          Study Number s05-018

Region



                                              REGION
                            -------------------------------------------
                                 East    MidWest      South        West

Base: Total Respondents            96        112        165          62

East                               96          -          -           -
                                100.0          -          -           -

MidWest                             -        112          -           -
                                    -      100.0          -           -

South                               -          -        165           -
                                    -          -      100.0           -

West                                -          -          -          62
                                    -          -          -       100.0

Total                              96        112        165          62
                                100.0      100.0      100.0       100.0
```

Figure 11.2.4d Revised region cross tab.

This table now begins to look slightly more like a table produced with a manually-defined banner. Again, the effort to produce this banner is significantly less than the effort needed to produce a fully manual banner. When many banners are needed, the time savings which can be realized from using this type of banner are great. However, most clients and users agree that a manual banner organizes the data in a more legible manner, and is worth the extra effort.

11.2.5 Using nested subheadings

Using nested subheadings is another, more specialized way to achieve the banner look without physically defining a banner. The keyword **hdlev** may be used in conjunction with standard n01 statements to produce a more clean appearance. The keyword must be associated with a level number which defines exactly where the subheading is to appear upon the page and which axis elements it is associated with. Figure 11.2.5a shows the generic construction.

n23text;hdlev=level_number;[options]

Figure 11.2.5a Using hdlev in conjunction with n23 statements

Consider the following axis definition. The usage of hdlev can be clearly seen. Defining the Gender subheading as hdlev=2 will force this subheading to appear under the hdlev=1 subheading, “All Regions”. Figure 11.2.5c shows an example of the output using this axis. Using the hdlev keyword is a practical and useful way to mimic banner results in standard axes.

```
l banhd1
n23All Regions;hdlev=1;unl1
n10Total
n23Gender;hdlev=2;unl1
n01Male;c=c39'1'
n01Female;c=c39'2'
```

Figure 11,2,5b Revised axis for use as a banner axis.

```
  Page 1
                                             Out Of This World
                                             Study Number s05-018

What age bracket most accurately defines you?
Q.27b)

                                                 All Regions
                               -----------------------------------------------
                                                           Gender
                                               -------------------------------
                                         Total            Male          Female

Base: Total Respondents                    245             128             117

18-29 years old                             89              36              53
                                          36.3            28.1            45.3

30-39 years old                             35              14              21
                                          14.3            10.9            17.9

40-49 years old                             18              18               -
                                           7.3            14.1               -

50 -59                                      58              15              43
                                          23.7            11.7            36.8

60 years or older                           45              45               -
                                          18.4            35.2               -

Total                                      245             128             117
                                         100.0           100.0           100.0
```

Figure 11.2.5c Revised cross tab using hdlev banner.

There is a limit of 9 nested subheadings which can be used with hdlev. That is, the maximum level number which can be specified is 9.

12. Other Table Options

There are a myriad of ways to “individualize” the table output in order to reflect a particular style or display additional information on any given page. Some of these options are explained in this chapter.

12.1 Footnotes

In order to display a footnote on a particular table, use the **foot** keyword in conjunction with table title statements. The **foot** keyword signals that the following table title statements are to be printed as a footnote into the table. The keyword **bot** also will print text on the table, but this text will be at the bottom of the page, rather than as a footnote at the bottom of a table. The generic usage of these keywords is shown in the figures below.

```
foot
tt[Y]footnote_text
[tt[Y]additional footnote_text]
...
```

Figure 12.1a Format for table footnote.

```
bot
tt[Y]page_bottom_text
[tt[Y]additional bottom_text]
...
```

Figure 12.1b Format for bot statement text.

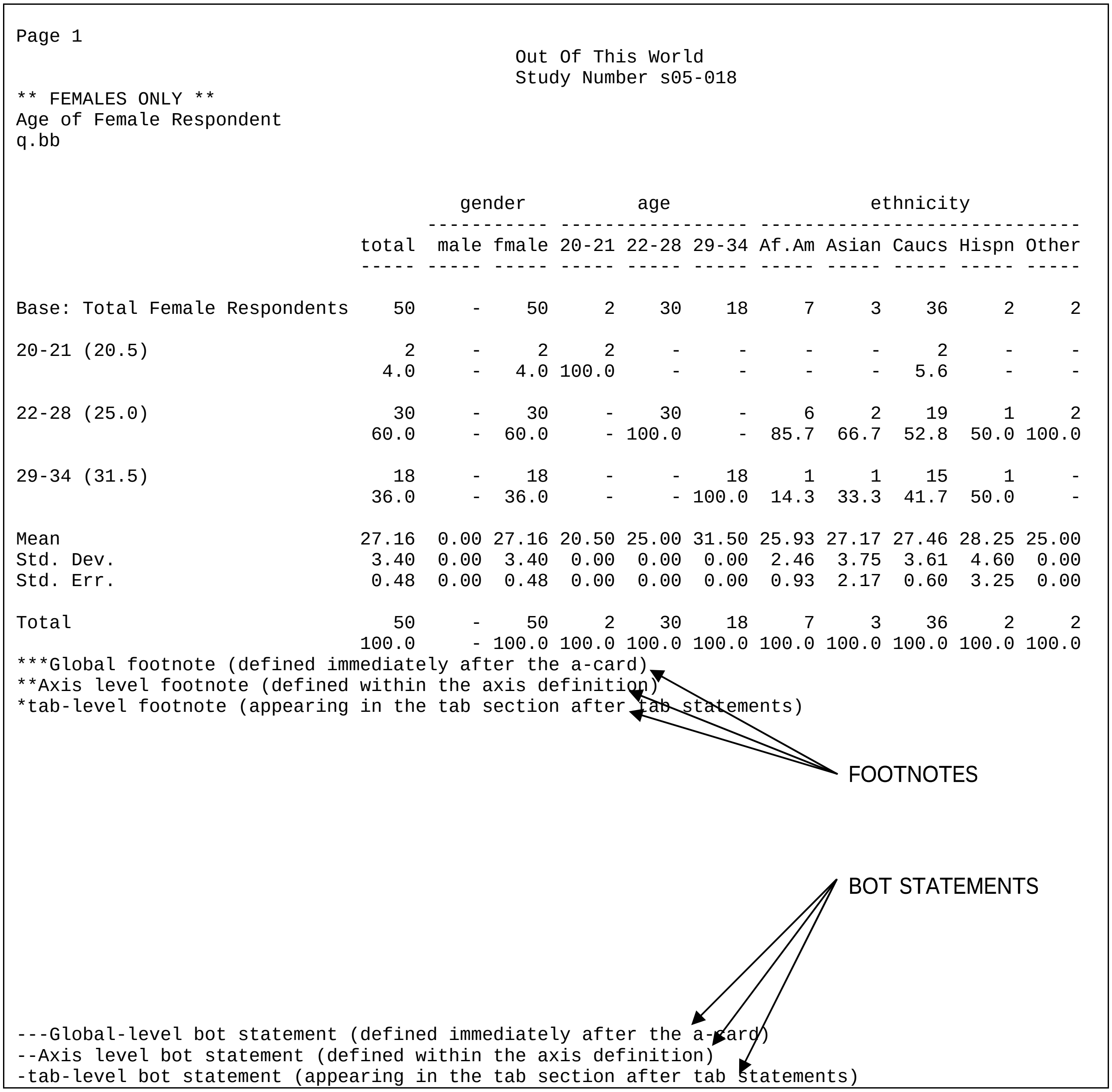
These footnotes can appear in several places throughout the run file. Both footnotes and bot statements may appear immediately after the a-card, within an axis definition or in the tab section between tab statements.

Footnotes and **bot** statements defined after the a-card will be treated as global options, and will appear on every table which is accumulated. Global footnotes are useful when a specific item of information is valid for all the tables in the run, and provide an effective way to display repetitive notes. The **foot/bot** statements which appear in the axis definitions themselves will only be applied to the axis (table) where they are defined. These statements are useful when an item of information pertains to one table (or axis) in the run. Lastly, the **foot** and **bot** statements which appear in the tab section of the run file will print on the table which precedes the statement definitions. Again, these are useful when statements apply specifically to the tabbed axes.

An example of footnotes and bot statements appears in figure 12.1c.

Global footnotes and bot statements take precedence over axis-level and tab-level foot/bot statements, and will appear first on the accumulated tables.

Axis-level foot and bot statements take precedence over any tab-level foot and bot statements.



```
Page 1
                                             Out Of This World
                                             Study Number s05-018
** FEMALES ONLY **
Age of Female Respondent
q.bb


                                        gender          age                 ethnicity
                                     ----------- ----------------- -----------------------------
                               total  male fmale 20-21 22-28 29-34 Af.Am Asian Caucs Hispn Other
                               ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

Base: Total Female Respondents    50     -    50     2    30    18     7     3    36     2     2

20-21 (20.5)                       2     -     2     2     -     -     -     -     2     -     -
                                 4.0     -   4.0 100.0     -     -     -     -   5.6     -     -

22-28 (25.0)                      30     -    30     -    30     -     6     2    19     1     2
                                60.0     -  60.0     - 100.0     -  85.7  66.7  52.8  50.0 100.0

29-34 (31.5)                      18     -    18     -     -    18     1     1    15     1     -
                                36.0     -  36.0     -     - 100.0  14.3  33.3  41.7  50.0     -

Mean                           27.16  0.00 27.16 20.50 25.00 31.50 25.93 27.17 27.46 28.25 25.00
Std. Dev.                       3.40  0.00  3.40  0.00  0.00  0.00  2.46  3.75  3.61  4.60  0.00
Std. Err.                       0.48  0.00  0.48  0.00  0.00  0.00  0.93  2.17  0.60  3.25  0.00

Total                             50     -    50     2    30    18     7     3    36     2     2
                               100.0     - 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0
***Global footnote (defined immediately after the a-card)
**Axis level footnote (defined within the axis definition)
*tab-level footnote (appearing in the tab section after tab statements)




---Global-level bot statement (defined immediately after the a-card)
--Axis level bot statement (defined within the axis definition)
-tab-level bot statement (appearing in the tab section after tab statements)
```

Figure 12.1c Example of standard footnotes and bot statements as displayed on a table.

12.2 Sorting responses [4]

In order to sort responses on a table in numerical order, simply add the keyword **sort** immediately after the l-card on an axis. This will order the n-statements below the table title in descending order. This keyword is useful when it is necessary to order responses by frequency of mention.

Use the **nosort** keyword on specific n-statements to exclude them from the sorting process. This is especially important on any elements which need to remain in a particular position within the axis. The **nosort** keyword is most commonly used on n10 statements, net statements, and any statistical calculation elements within the axis. Figure 12.2a gives an example of an axis which uses the **sort** keyword. Note that spaces are used in the **n01** statements for readability.

[4] This feature is currently under development.

```
l qc8;sort
ttlEthnicity
ttlq.c8
n10Base: Total Female Respondents;dsp;nosort
n01African American ;c=c(2121,2122).eq.1
n01Asian            ;c=c(2121,2122).eq.2
n01Caucasian        ;c=c(2121,2122).eq.3
n01Hispanic         ;c=c(2121,2122).eq.4
n01Other            ;c=c(2121,2122).eq.7;nosort
n01Prefer not to say;c=c(2121,2122).eq.8;nosort;nz
n05total;nodsp;nosort
```

Figure 12.2a Using the sort keyword within an axis definition.

12.3 Page numbering

ccount allows for the use of user-defined custom page numbers to be used. This is useful when a subset of tables needs to be reprocessed using a separate run file, when certain tables need to be replaced by revised versions. Rather than reprocessing all of the tables within the run, custom page numbering can be used so that the revised tables can fall in line numerically with the original tables.

12.3.1 Page Numbering

ccount always numbers the pages of a table, even when they are not specified. This standard numbering always begins at "Page 1", and continues until the last table has been processed.

To define custom page numbers, use the **pag** keyword, followed by the page number to be used. This keyword can be used anywhere in the tab section to define custom page numbers. Refer to the example below for the standard format of table page numbering.

```
pag X
```

Figure 12.3.1a Page numbering.

In this example, X is the number to be printed as the page number. Note that there must be a space after the **pag** keyword. Also note that **ccount** will continue to number pages consecutively after encountering this **pag** statement. Therefore, it is not necessary to specify each page number if consecutive pages are desired.

Any page numbers defined in the axes section will be ignored and no error will be thrown.

*Standard page numbers will be displayed using the "Page X" default format. In order to specify some other text to be used for page numbering, use the **pagtxt** keyword on the a-line.*

13. Advanced Editing Topics

This chapter will introduce some additional commands and features available in **ccount**. While some may still be under development, most are available and ready to be used with the latest current version of **ccount**.

13.1 Real variables in the c-array

As explained in chapter 4, **ccount** has the ability to assign real values to variables, and can even perform complex computations using these real variables. Storing real values in named variables are different than storing real variables into column locations.

When referencing real values, the `cx(x,y)` notation must be used as shown in Figure 4.2b. Assigning real values to columns, however, requires some additional specification. The number of decimal places to be used must be specified within the assignment statement in order for the real value to be properly assigned to the column locations. An example of this is shown in Figure 13.1a.

cx(col_start,col_end):decimal_places=real_variable

Figure 13.1a Assigning real values to column array.

The main difference in assigning real values to columns is the specification of the number of decimal places that **ccount** needs to reserve. This is accomplished by including the number of decimal places immediately after the column specification, separating the column specification and the number decimal places by a colon.

Always use a sufficient number of column locations when assigning real values. Most errors occur when a column is omitted for the allocation of the decimal place itself.

Consider the following example:

```
x5=123.45
cx(101,107):2=x5
```

In this statement, the real variable contained in x5 will be assigned to the columns `101` through `105`, with 2 decimal places. Be aware that specifying an insufficient number of columns for the assignment will result in a syntax error, but only specifying an incorrect number of decimal places will result in the value being rounded to the number of decimal places specified. For instance, if `x5` in the example above contained a value of `100.456`, and the assignment statement was left unchanged, the value that would be assigned to columns `101` through `107` would be `0100.46`. As can be seen, the value is rounded.

Specifying an incorrect number of decimal places in a real value assignment can lead to loss of data.

Please consider the following statements, which are similar to the previous example, but not identical:

```
cx(101,107):2=123.45
x9=cx(101,107)
```

When assigning real column values back to a real variable, the number of decimal places does not need to be specified, since the real variable assignment statement automatically handles the decimal.

13.2 Merging data

When data to be analyzed is split over a number of different files, it becomes necessary to “merge” external data into the current working data file in order to analyze the complete data. Rather than having to manually match and recode the data, **ccount** allows for external files, called “lookup” files, which can contain additional data information to be merged into the main data file for ease of analysis.

13.2.1 Call fetch

The “call fetch” statement is used to merge linked information from separate data files (data sets) into one long contiguous file. **ccount** accomplishes this through the use of the "fetch" statement, sometimes referred to as the "call fetch" statement. The fetch statement is essentially a subroutine (hence the "call") which merges data from an external (outside) file into the main data file which is being used in the current **ccount** run. This subroutine is automatically available to all users during any **ccount** run, and does not need to be defined within each run file which uses the call fetch statement/subroutine.

The call fetch statement uses a number of parameters, which are passed into the subroutine. The parameters include the external file name where the data will be merged from, the key field (from the main data file) starting location and the beginning column location where the merged data is to be placed within the main data file, also known as the destination cells. These columns must be referenced using standard column notation.

Note that the key field to be matched upon MUST be present in both files, and the data within the external file MUST be sorted on this key in ascending order. Otherwise, the statement will fail. Figure 13.2.1a gives the construction of the call fetch statement.

call fetch ($lookup_file_name$,key_start,destination_cell)

Figure 13.2.1a Construction of a call fetch statement.

The call fetch statement itself is not the only requirement for successful merging of data. The external data file must be formatted in a particular fashion as well. This external data file, sometimes referred to as the "fetch" file or the "lookup" file, contains the information that is to be merged/moved/transferred into the main data file. This lookup file must contain a key field that matches the key field in the main data file in order for **ccount** to make a correlation between the records and produce a match. If a match is made, the data from the lookup file will be transferred into the main data file into the specified destination cells.

In addition, the external file must contain 2 pieces of information on the first line of the file. This information must be in the form of 2 numbers. The first number gives the length (in characters) of the key field and the second is the total length of the line to be merged into the data file. All subsequent lines must begin with the key field, and must be sorted in ascending order.

It is suggested that the information contained in the first line of the external file begin in the second (or any subsequent) column. When the file is sorted in ascending order, this should guarantee that this line will always remain in the first line position, reducing the chance for error.

ccount uses the first line in the fetch file to determine the key length in the main data file for the key start position which was defined in the original call fetch statement. In Figure 13.2.1b, the key field length is defined as 7 characters wide. When a match is made between the key field in the lookup file, and the key field in the data file, the data in the lookup file which begins immediately AFTER the key field will be transferred into the main data file in the position which was originally defined in the call fetch statement. The length of the field to be transferred is given by the second number in the first line of the fetch file. If a match is NOT found anywhere in the lookup file, then the destination area remains blank, and no data is moved.

```
  7  20
051072809MOBILE
051153001TUPELO
051349802DALLAS
051393102ALBANY
051457301LAS VEGAS
051521604BURLINGTON
051569403TROY
051686003CINCINNATI
051761701ROCHESTER
051770702WORCESTER
051777601CARROLTON
052002201NEW HAVEN
052059102COHOES
```

Figure 13.2.1b Example of lookup file.

For example, consider the following call fetch statement:

```
call fetch ($mergedata.txt$,c1,c301)
```

Assume the lookup file mergedata.txt contains the information shown in FIGURE 13.2.1b. When executed within the edit section of the run file, this call fetch statement will reference the lookup file mergedata.txt, and will try to produce a match on the key field of the current record that is being processed. The key field is defined in the first line of the fetch file as being 7 columns. Therefore, the key field in the main data file must be in `c(1,7)` since the key field in the call fetch statement is defined as beginning in `c1`. If a match is made, then the information from the fetch file beginning in column 8 will be transferred to the main data file into column 301. All the columns from 8 through 20 of the fetch file will be transferred since the overall length of the fetch file is 20 as per the second number in the first line. The destination `c(301,320)` in the main data file will now contain the merged information from the fetch file.

The target columns within the main data file will be deleted if a match key is NOT found in the fetch file. If records are present in the main data file and not in the fetch file, the call fetch statement should be "skipped over" with a goto statement or a conditional filter in order to avoid these columns from being cleared.

Lookup files MUST be sorted in ascending order of the key field in order for the call fetch statement to be successful in matching and merging data.

When the run is complete, and all records have been read, the out2 file will contain the summary information for the matches. This information contains the name of the fetch file, the number of total records read from the main data file, the number of “calls”, or the number of times the call fetch statement was executed, the total number of "hits", or accurate matches made between the main data file the lookup file, and the number of "misses", or unmatched records from the main data file. Figure 13.2.1c shows an excerpt of the output from the out2 file following a run which included a call fetch statement.

```
Fetch file usage (1 file loaded):

  Records      Used    Unused     Calls      Hits    Misses   File
      358       358         0       358       297        61   mergedata.txt

ccount (C) V. Hoffmann  -  edit report
Page 2
           summary of exceptions

    358 cards read
    358 records
    358 (100%) records accepted
      0 (  0%) records rejected
```

Figure 13.2.1c out2 file showing summary of call fetch results.

There is no known limit to the number of fetch statements that can be used in a single run. However, there is a limit of 10 distinct (separate) fetch files that can be used in any given run.

The line length limit for any given fetch file is 32,000 columns.

13.3 Definelist

The definelist statement provides a convenient way to reference a series of values to be used without having to retype and modify this list many times. This series of values to be used can be a comma separated list of values, ranges, or any string. The definelist statement is generally used in conjunction with the **.in.** logic connector when a list is used many times within the same run file.

definelist name = (list)

Figure 13.3 Standard definelist statement.

The definelist is a convenient way to keep a static, consistent list of values throughout an entire **ccount** run. Using the definelist is also a much easier way to modify a series of static values. Rather than modifying the list of values everywhere they may occur within the run file, the user need only modify the definelist definition. See Figure 13.3b for examples of definelist statements used within a standard edit section.

```
definelist states=($NY$,$MA$,$TX$,$OH$,$KY$,$FL$)

r (c(101,102).in.states)

definelist validcodes=(01,05:13,17,99)

r (c(301,303).in.validcodes)
```

Figure 13.3b Examples of definelist.

Note that the syntax of a definelist is identical to that of the .in. logic connector.

13.4 Median[5]

Medians are used in statistical calculations, and is sometimes better suited than the mean to give accurate information about data. The median, put simply, is the middle value within a set of data, or distribution. Half of the given scores are below the median, while half are above the median. For highly skewed distributions, the median is a much better measure than the mean, since it is less sensitive to extreme scores.

In order to display a median when processing tables, a median statement must be used in conjunction with an n01 statement. This statement uses multipliers (inc=) in order to calculate the median, and is shown in Figure 13.4a below. This statement uses the column locations given in the inc= statement to determine how many values are in the given list, and what position the median will be in.

```
n01Median value; inc=increment;median;
```

Figure 13.4a median calculation.

There are several different ways to calculate the median score for a given distribution. All methods, however, rely on the "50% mark" – the theoretical line which denotes a value as being part of the series of values above or below the median. This 50% mark can be easily found by dividing the number of values plus one in the given list by 2. This number gives the position which is considered to be the median. Note that the data must be sorted in numerical order for the median to be valid. The generic version of the median calculation is given in Figure 13.4b.

```
median_position = ((number of values in list) + 1) / 2
```

Figure 13.4b median position formula.

If a given value falls directly on the "50% mark", it is considered the median. For instance, consider the values 2,3,5,6 and 8. Since there are 5 numbers in the list, divide (5+1) by 2. This gives a solution of 3. The median, then, is the 3rd score. Therefore, 5 is the median of this value list.

medint=		
0	=	Default formula. Interpolates between the value which exceeds the 50% mark and the previous value.
1	=	The exact value which exceeds the 50% mark is considered to be the median.
2	=	Interpolation is performed between the midpoints of the 50% mark value and the previous and next values.
3	=	Interpolate between the 50% mark value and the next value.

Figure 13.4b Median interpolation formulas.

If the median falls between 2 values (the median calculation produces a decimal), there are several ways to calculate the "true" median, some of which include interpolation of the values. While some methods interpolate between the median position and the previous value, others interpolate between the median position and the next value. Some methods ignore the interpolation, and just use the value which crosses over the calculated 50% mark.

[5] The median functionality is currently under development.

The specific method to use when interpolating the median score is defined on the **a-card** in the run file. This is specified using the **medint** keyword. The different types of interpolation are shown in Figure 13.4c.

13.5 Splitting data files

Occasionally, the need may arise to separate respondents from the input data file into two groups: those that have passed the requirements of the edit section, and those who have been flagged with the **rejected_** variable. This might be necessary in order to further investigate the rejected respondents' data. These "dirty" respondents would need to be separated from the "clean" respondents so that they could be more closely examined.

The terms "clean" and "dirty" are used often in reference to the data records which populate the files. Clean data files contain respondents who have passed all of the required conditions in the edit section. Thus, the data for these respondents seems to be valid, and they have cleanly passed through the edit section. Dirty data refers to those respondents whose data is invalid because of failing one or more of the requirements of the data within the edit section. These dirty respondents are flagged as **rejected_** in the edit section.

Note that respondents discarded with reject;return statements will not be displayed in the split files.

The split statement allows for two files to be created automatically during a **ccount** run. These two files, called dirty and clean, will be populated by **ccount** based on the status of the **rejected_** variable within the edit section. If a record has been flagged as rejected_ at the point when the **split** statement is encountered, then that record will be written to the dirty file. Otherwise, the record will be written to the clean data file. Figure 13.5a shows an example of using the **split** keyword.

```
/*15
r (c(182,184).in.(18:65))

/*q16
r sp c64'12'o

/*q17
r sp c65'1/5'o

split
```

Figure 13.5a Using the split keyword within the edit section.

While some may think that this is similar to using a standard **write** statement, it actually differs from it in many ways. First, the **split** statement causes 2 files to be created, a "clean" file and a "dirty" file. The write statement only writes data to the specified file. Second, the **split** statement does not require a preceding **filedef** statement to define the file being written to. Next, the split statement cannot be filtered with a conditional statement. Lastly, the split statement may occur multiple times in the edit section, although respondents will only be written to a particular file one time. Respondents will be written out to the particular file which they uniquely qualify for when the **split** statement is reached. A respondent will not be written to a secondary file, even if that respondents' disposition changes after a second **split** statement is encountered.

The split statement can appear multiple times within the edit section, but respondents will only be written out to any particular file once.

Records cannot be written to both the clean and dirty data files. Respondents will only be written to the file for which they uniquely qualify for at the point in which they encounter the split statement.

13.6 Right-justifying or Left-justifying column arrays

In order to right-justify or left-justify a series of columns within a column array, use the **rset** and **lset** commands, respectively. Figure 13.6a and 13.6b shows the syntax for these commands.

```
lset c(start_column,end_column)
```

Figure 13.6a lset command.

```
rset c(start_column,end_column)
```

Figure 13.7a rset command.

These commands will progress through the column array specified, and will shift any data found to the appropriate justified location. Note that only trailing and/or leading blanks are affected. Any blank columns which appear *between* non-blank columns within the specified column array will not be affected. Figure 13.7b below gives examples of how data is affected by the **lset** and **rset** commands.

```
data before lset and rset
                                                              Alphabetical sort
Total = 841
String                                                Frequency      Cumulative
------                                                ---------      ----------
   9 12                                              841  100.0%     841  100.0%

data after lset
                                                              Alphabetical sort
Total = 841
String                                                Frequency      Cumulative
------                                                ---------      ----------
9 12                                                 841  100.0%     841  100.0%

data after rset
                                                              Alphabetical sort
Total = 841
String                                                Frequency      Cumulative
------                                                ---------      ----------
        9 12                                         841  100.0%     841  100.0%
```

Figure 13.7b Effects of rset and lset commands in data.

14. Weighting

Weighting is the process of using an artificial multiplier to increase the tally any particular respondent will have on a table. Normally, each respondent counts as a single tally on any given table. Weighting gives the ability for each respondent to count as more (or less) than one on a table.

For complex survey designs, sample data can be weighted in order to obtain unbiased estimates for the total population being surveyed. Weighting causes the data to more closely resemble the actual representative population. Whether this population be the entire U.S. civilian population, or some subset of the households within in the United States who use or have used a particular item within the past 3 months. Weights within a survey can also reflect the probability of being sampled for the survey and will sometimes include adjustments for non-responsiveness, as well as post-stratification adjustments made to the data.

Weighting is useful when tables are required to reflect a particular sample, rather than just a particular set of respondents. Assuming a truly representative sample, 300 completed respondents on a table could be weighted to reflect the total US population who would normally qualify. Changing the tally amount so that 300 completes reflect 100,000 completes requires some calculation, which is sometimes referred to as the “art” of weighting.

Weighting can also be used to balance any sample biases that might occur during a survey. For example, if one group in the data set of 2 groups represents 65% of the total data, then weighting might be used in order to balance out the groups to a more even keel. The group which represents 65% of the total data would be weighted down so that it represents 50%, and the second group which only represents 35% would be weighted up so that it, too, would represent 50% of the total data. Weighting is the elixir which brings the data back into balance.

In the field of research, there is some controversy over weighting questionnaire data to correct a sample. Many researchers believe that weighting a sample reduces its reliability. Most believe that weighting techniques are permissible when “fine tuning” a sample, but should not be used to make major changes in the percentages of populations within the data file. Unfortunately, there is no consensus about what constitutes a “major” change. Circumstances and individual beliefs guide most researchers’ definition of acceptable weighting changes. If questionnaire weighting is used to inflate the sample sizes (for example, to project a sample to look like the relevant universe) any results from probability statistics (such as chi-squares, standard errors and t-tests) could be unreliable.

14.1 Types of weighting

Weighting is a powerful tool that must be used with respect. Since the data is being artificially incremented (or decremented) when tallied on the tables, results could be unreliable. Weights need to be applied to the data in a very careful manner to avoid disastrous results which could make the data tables virtually useless. There are several different methods of weighting, and while each method can achieve desired results for any given situation, each method is differently suited for a particular purpose of weighting.

14.1.1 Factor weighting

Factor weighting may be the most straightforward of all the weighting methods. Factor weighting assigns every record in the dataset a specific weight value. Each record (or groups of records) can be assigned a different weight value, based on filters.

For instance, consider an interview conducted over the Internet which results in a data file comprised of 300 total respondents. The original specification for the data was that the population be 50% male and 50% female. However, the actual population reflected in the data set is 35% male and 65% female. In order to balance this sample, factor weights can be used. By multiplying each record in the female subset by 0.7692307 (50/65), and multiplying each record in the male subset by 1.4285714 (50/35), the target percentages can be reached more closely. The weights for the female and male population within the data are 0.7692307 and 1.4285714, respectively.

In this example, the needed weight has been calculated by dividing the target percentage distribution by the actual percentage distribution. In general, to calculate a weight factor use the formula shown in Figure 14.1.1a.

weight= desired N / actual N

Figure 14.1.1a Generic weight calculation formula.

This method assures that each record in the defined group is multiplied by the weight defined by the user, which should in turn, result in the correct totals being printed and displayed upon the table or tables. Factor weighting is most useful when the actual weighted multipliers are known before the data is processed. Factor weighting also implies that the data being processed will never change once processing has begun. That is, the data remains static and will never change each time it is processed through the run file. If the populations within the data change, then the factors must be modified as well. If the factors are not modified, the weighted results will be unreliable, and the data tables may be completely useless.

14.1.2 Target Weighting

Target weighting allows for the user to define the targets, or totals, which are to appear in each cell of the weighted table. The weights specified in target weighting are actually the values which will appear on the table(s), and not the weights themselves. When processing these totals, **ccount** automatically calculates the resulting weight by dividing the target weight for any particular group by the actual number of records in that group.

Using the above example again, the user would specify that the targets for females and males are each 150. When a female data record is processed, **ccount** would automatically divide the target number of females by the total number of females in the data file to retrieve the correct weight to be applied to that particular record (150/195=0.7692307). When a male data record is processed, **ccount** would repeat this calculation using the numbers given for the male targets (150/105=1.4285714). These calculated weights should guarantee that the weighted targets are met. Once again, the generic formula for calculating weights has been applied. However, in this case the actual target populations have been specified rather than the target percentages.

Target weighting is slightly more forgiving than factor weighting since it does not assume that the data will be static each time it is processed. For instance, assume that the Internet interviews above are conducted again several months after the initial interviews have been completed. For the second wave of data, the actual percentages reflected in the data are different than the actual percentages in the first wave. For the second wave, males comprise 42% of the data, and the female population within the data file is only 58%. Rather than recalculating the factors needed to again balance this data file, the target weighting method will re-calculate the weights automatically based on the data which is being processed. Target weighting assures the user that regardless of any population shift within the data file, the targets will always be met.

Note that in target weighting, the total targets may add to more than the total records in the original data file. It is not necessary that the targets, or totals, specified add to the total number of records in the data file.

14.1.3 Input weighting

Input weighting is very similar to target weighting. With input weighting, the target percentages are specified, rather than the target totals. This method is useful when the totals are not known, but the percentage of total population for a given group is. By specifying the keyword **input**, **ccount** recognizes that the values given refer to weighted percentages rather than weighted totals. Weights would be calculated using these given percentages, and would be assigned to each respondent in each group or category.

Similar to target weighting, input weighting does not assume that the data file remain static for each run, only that the weighted percentages remain static.

14.1.4 Rim weighting

Rim Weighting is a statistical technique that allows the user to specify only the marginal values of each weighting axis. For instance, only the targets for each group need to be specified - **ccount** will work out the values of all of the combinations of the groups. For instance, consider the example where age groups need to be weighted by gender group. Only the "rims" of the weight matrix need to be defined, as **ccount** will interpolate the other values, and produce a weight matrix which aligns with the totals already specified.

Rim weighting is very powerful since each target is weighted at the same time. For samples that are fairly representative, the accuracy of rim weighting is uncanny. Another benefit of rim weighting is that target values are rescaled to the original base when calculated. This method ensures that each variable is distorted as little as possible while still trying to optimize the proportions to match those which were given in the statement.

Visual representation of rim weighting:

	TOTAL	*Male*	*Female*
TOTAL	545	270	275
Age1	55	**X**	**X**
Age2	100	**X**	**X**
Age3	125	**X**	**X**
Age4	265	**X**	**X**

Figure 14.1.4a Visual representation of rim weighting.

When specifying the weighted totals, only the "rims", or the outside totals, need to be defined. The values marked with "X" shown in Figure 14.1.4a will be automatically calculated by the program. Rim weighting can be used in conjunction with input weighting, in which case the rim targets can be specified as percentage values.

Rim weighting is one of the more simple weighting methods to implement and understand, since the intersection of the weight cells do not have to be interpolated by the user.

When used in conjunction with rim weighting, the keyword input specifies that the rim weights entered are target percentages, rather than target actuals.

14.2 Other types of weights

While standard weights are normally defined within the tab section of the run file, preweights and postweights can be defined within the edit section of the run file. This provides greater flexibility if it is paramount that weights be calculated “on the fly” during the run.

14.2.1 Preweights

Preweights are weights which are applied to each individual record before the target, factor or input weighting is applied. This allows for slight adjustments in the weighting to ensure balance. Preweights are often included in the data file, and these data locations are specified within the definition of the target, factor, or input weighting matrix. The preweights are read from the data file, and are applied to each record before any other weights are applied.

If they are to be used alone, then a dummy weight matrix must be defined. Many situations call for preweights to be used as the only weights to be applied. Preweights are defined using the standard weighting formula, and then a dummy weight matrix is defined which uses a weight of 1 for all respondents. In this way, the preweights act as the actual weights, and the actual weights have no additional effect on the data.

14.2.2 Postweights

Postweights, as the name implies, are weights that are assigned after all other weights have been applied. Postweights usually have no effect on the way targets are reached, and are only used when slight adjustments are needed for balancing out population percentages.

14.2.3 Zero weights

Applying a weight of zero for any respondent or group of respondents will effectively exclude this/these respondent(s) from the weighting calculations. That is, any respondent receiving a weight of zero will count on the accumulated tables as a null value, and will essentially be ignored.

While this is a valid application of weights, if a respondent is to be completely omitted from the tables, it is suggested that a standard `reject;return` statement be used. This is the preferred method to dispose of a respondent and exclude him/her from the accumulated tables. Assigning a weight of zero, while effective, creates additional overhead and still causes the respondent to be processed through the entire edit and tabulation section. Processing zero-weighted respondents through the tab section could lead to unexpected results. For instance, if the weighting is turned off for any reason, these unwanted respondents would still be included in the actuals for the table and would appear on the unweighted tables, even though they are essentially nonexistent in the weighted tables.

A respondent weighted to zero within the tab section is not identical to rejecting the record with a reject; return statement within the edit section.

14.3 Defining weights

There are two main ways to define weights within a **ccount** run. First, the weights may be declared within an ordinary axis which has been labeled as a weighting axis. The second method is by defining a weight matrix which specifies the weights and weight conditions. These two methods are not interchangeable and any weights defined within a run file must use one method or the other. Mingling the methods will result in a syntax error.

14.3.1 Weight matrices

Weighting matrices are defined in the tab section, immediately after the a-card by first assigning a unique number to the weight matrix. The axes then need to be defined for the particular weight matrix along with keyword options. Lastly, the weights (in target, factor, or input format) must be identified. The following figure gives the generic structure of a weight matrix.

wmN axis1 [axis2][axis3][...];[options];weights

Figure 14.3.1a Generic structure of a weight matrix.

The weight matrix can be identified with any unique single-digit number. Thus, wm1 and wm2 are valid weight matrix names. Weight matrix names do not have to be in order, but must only be numbered wm1 through wm9. The weight matrix name is followed by the axis names which are used to identify the weights. Options and weights follow the axis names. An abbreviated list of options appears in Figure 14.3.1b.

There is a limit of nine (9) weight matrices which can be defined in any given run file.

target	**=**	The target weighting methods is to be used.
factor	**=**	Specifies that factor weighting is to be used.
input	**=**	Input weighting, which specifies that the targets specified are percentages (sometimes referred to as proportions)
rim	**=**	Rim weighting.
total	**=**	Specifies that cell total must equal a particular value *(ex: total=100)*.
pre	**=**	Preweighting using the variable value given *(example: pre=x1) (where x1=0.7692307)*.
post	**=**	Postweight using the given value *(example: post=x3)*.
maxwt	**=**	Specifies the maximum weight to be used in the given matrix *(example: maxwt=10)*.
minwt	**=**	Specifies the minimum weight to be used in the given matrix *(example: minwt=0.25)*.

Figure 14.3.1b wm statement options.

As shown in the Figure, the options must specify which weighting method is to be applied. Only one method of weighting can be specified per weight matrix. Thus, the keywords **factor** and **rim** cannot appear on the same *wm* statement. In addition, the options can specify the preweighting and postweighting variables, as well as maximum and minimum weights which are to be applied to those records which fall above the maximum, or below the minimum, respectively.

The maximum number of axes per weight matrix is sixteen (16).

Applying a weight matrix of 0 (wm=0) to tables forces the tables to be run unweighted. This is not the same as applying a zero weight to respondents.

14.3.2 Weighting axes[6]

Another option for defining weights is to define the weights within a specifically designed axis, or axes. When using axes, the only weighting methods available are target, factor and input weighting. In order to specify that an axis be used as a weighting axis, add the keywords **wttarget** or **wtfactor** and the specific target or factor value immediately after the n01 statement which defines the weighting element. Figure 14.3.2a illustrates this.

```
l [axis_name];[keywords]
n01stub_text;[condition];[weighting_keyword]=[value]
```

Figure 14.3.2a Generic construction of weighting axis.

Keywords can also be used on the l-card to specify preweights, postweights and totals. By defining a weighting axis in this way, all elements which are to be weighted can be a**ccount**ed for. Once this axis has been defined, it must be specified immediately after the a-card as being the weight matrix for the tab section. In order to do this, simply assign the axis name to a weight matrix, as shown in Figure 14.3.2b.

```
wmN=axis_name
```

Figure 14.3.2b Defining an axis to be used as a weight matrix.

Any axis used as a weighting axis can also be used on a tab statement. When an axis is used in a tab statement, all of the weighting keywords are ignored, and the axis is treated as a generic axis.

Rim weighting cannot be used in conjunction with weighting axes.

14.4 Using weights

In order to apply weights to a standard **ccount** run, simply specify the weight matrix (wm) to be used on the a-card, l-card, any tab statement or on n01, n10, n15, n11, and net statements within the tab section of the run file.

In order to activate weight matrix 1 (wm1), for instance, simply place the statement *wm=1* at the end of the a-card. In order to activate weight matrix 4, place the statement wm=4 at the end of the a-card. This will activate the specified weight matrix, and it will be used globally when the tables are accumulated. Figure 14.4a shows the generic construction of this type of statement, while Figure 14.4b displays the use of this statement on the a-card within a standard run file.

[6] This feature is not currently supported.

```
wm=weight_matrix_number
```

Figure 14.4a Activation of a weight matrix.

```
/*define global table output options
a;dsp;flush;nopc;notype;op=12;side=30;printz;wm=6
```

Figure 14.4b Activating a global weight matrix on the a-card.

If a weight matrix (wm1, for instance) is only to be applied on a particular stub of an axis, use the *wm=1* statement on the particular n01 statement which should be weighted. Note that a *wm=0* statement can be used to specify that no weight matrix should be applied. Figure 14.4c illustrates this example.

```
/* display weighted and unweighted totals
l qwght
n01unweighted total;wm=0
n01weighted  total;wm=1
```

Figure 14.4c Example of using weighting on specific stubs within an axis.

Using a wm=0 statement guarantees that a particular statement or series of statements will be produced using unweighted totals.

Similarly, the *wm=1* statement can be used on a tab statement to weight the entire table. Again, the *wm=0* statement is used to specify that the second tab statement is processed with weighting turned off. An example of this can be seen in Figure 14.4d.

```
/* tab section
ttc**WEIGHTED**
tab age ban1;wm=1

ttc**UNWEIGHTED**
tab age ban1;wm=0
```

Figure 14.4d Example of using weighting on tab statements.

Note that when weighting a data set, there must be at least one case of all elements of each category represented in the input data file, as well as in the weighting spec. All cases must be accounted for – both in the weighting spec and the data file.

Weights assigned within any given run must cover all cases of the data, and all cases of the data must be weighted according to the spec. Failing to assign a weight to a data record for any reason will lead to run-time errors.

14.4.1 Weight matrices within a run file

Figure 14.4.1a shows an example of defining a weight matrix within a run file. As shown, the weight matrix is defined within the tab section after the a-card. This particular weight matrix uses rim weighting. The weighting used is the same scheme that is displayed in Figure 14.1.4a.

The weight matrix 1 is specified to be used, followed by the axes which are to be used as the weights. Here, both age and gender will be weighted. This is followed by the keyword **rim** to indicate that rim weighting will be used. Lastly, the rim targets are specified. These are separated by weight grouping in order to fully illustrate the rims used.

While this particular weighting schema only uses age and gender as the weighting rims, many more axes can be used within the definition of a weight matrix. In fact, up to sixteen (16) axes can be used when defining a weight matrix. This allows for great precision when specifying targets and/or percentages, but it can also lead to over-manipulation of the data file. Keep in mind that incorrect manipulation of the data could render the data meaningless.

```
/* struct statement
struct;read=0;ser=c(1,4);reclen=310
ed

end

/*define global table output options
a;dsp;flush;nopc;notype;op=12;side=30;printz

wm1 gender age; rim; 270; 275;
+55; 100; 125; 265

/* check weights with tabs
tab gender wban1
tab age wban1

/* define axes
l gender
ttlGender
n10Base: Total Respondents;dsp
n01Male;c=c151'1'
n01Female;c=c151'2'
n05total;nodsp

l age
ttlAge
n10Base: Total Respondents;dsp
n01 14-29 yrs.;c=c(152,153).in.(14:29)
n01 30-39 yrs.;c=c(152,153).in.(30:39)
n01 40-49 yrs.;c=c(152,153).in.(40:49)
n01 50+ yrs.;c=c(152,153).in.(50:99)
n05total;nodsp

/* banner
l wban1
g UnWgt  Wghtd
g Total  Total
g -----  -----
p     a      b
n01unweighted;wm=0
n01weighted wm1;wm=1
```

Figure 14.4.1a Example tab section using basic rim weighting.

Again referencing the run file shown in Figure 14.3.1c, the output tables for this run file will show the weighted and unweighted totals for comparison, as defined by the banner. The effects of weighting can be immediately seen. Totals have been preserved, but the actuals shown on the table have been artificially manipulated to meet the needs of this particular weighting scheme. Tables for this run are shown in Figure 14.4.1b.

```
Page 1
Gender
                               UnWgt  Wghtd
                               Total  Total
                               -----  -----

Base: Total Respondents          545    545

Male                              98    270
                                18.0   49.5

Female                           447    275
                                82.0   50.5

total                            545    545
                               100.0  100.0

Page 2
Age
                               UnWgt  Wghtd
                               Total  Total
                               -----  -----

Base: Total Respondents          545    545

 14-29 yrs.                       36     55
                                 6.6   10.1

 30-39 yrs.                       76    100
                                13.9   18.3

 40-49 yrs.                      143    125
                                26.2   22.9

 50+ yrs.                        290    265
                                53.2   48.6

total                            545    545
                               100.0  100.0
```

Figure 14.4.1b Output example of weighted table run.

As can be seen, the actuals accumulated on the tables have been manipulated compared to the original values, but the base totals remain intact. It cannot be stressed enough that artificially inflating or deflating the numbers on tables can lead to misleading results. Any weighting which is implemented should be carefully calculated and verified.

This weighting scheme can be duplicated using target weighting. However, all of the intermediate target points would have to be calculated instead of just specifying the rim totals. Thus, instead of just specifying the original 6 totals (2 age totals and 4 gender totals), 8 targets (2 age totals within each of the 4 age totals) would be required. Figure 14.4.1c illustrates the weight matrix definition statement which must be used to duplicate the tabular results.

```
/**revised weight matrix using target weighting
wm1 gender age ; target; 29; 26; 41; 59; 63; 62; 137; 128
```

Figure 14.4.1c Revised wm statement using target weighting schema.

The new targets used for this revised weighting scheme were calculated using the weighted results from the previous table, and extrapolating them in order to calculate the intermediate values for the intersection of gender and age.

```
/* define record structure of input file
struct;read=0;ser=c(1,4);reclen=310
ed

end

/*define global table output options
a;dsp;flush;nopc;notype;op=12;side=30;printz

wm1 gender age; rim; 270; 275; 55; 100; 125; 265
wm2 gender age ; target; 29;26;41;59;63;62;137;128

/* check weights with tabs
tab gender wban1
tab age wban1

/* define axes
l gender
ttlGender
n10Base: Total Respondents;dsp
n01Male;c=c151'1'
n01Female;c=c151'2'
n05total;nodsp

l age
ttlAge
n10Base: Total Respondents;dsp
n01 14-29 yrs.;c=c(152,153).in.(14:29)
n01 30-39 yrs.;c=c(152,153).in.(30:39)
n01 40-49 yrs.;c=c(152,153).in.(40:49)
n01 50+ yrs.;c=c(152,153).in.(50:99)
n05total;nodsp

/* banner
l wban1
g          RIM   TARGT
g UnWgt  Wghtd  Wghtd
g Total  Total  Total
g -----  -----  -----
p     a      b      c
n01unweighted;wm=0
n01   rim weighted wm1;wm=1
n01target weighted wm2;wm=2
```

Figure 14.4.1d Revised run file using rim and target weighting.

To compare the results, both weight matrices are implemented at the same time. Note the addition of a second weight matrix (wm2), and an additional banner point which uses this weight.

The results of this updated run file can be seen in the following figure. Note that the results of the target weighting are in fact identical to the rim weighting. While target weighting can sometimes be more precise, rim weighting has many advantages if only the total targets are known.

```
Page 1
Gender
                                       RIM   TARGT
                               UnWgt  Wghtd  Wghtd
                               Total  Total  Total
                               -----  -----  -----

Base: Total Respondents          545    545    545

Male                              98    270    270
                                18.0   49.5   49.5

Female                           447    275    275
                                82.0   50.5   50.5

total                            545    545    545
                               100.0  100.0  100.0

Page 2
Age
                                       RIM   TARGT
                               UnWgt  Wghtd  Wghtd
                               Total  Total  Total
                               -----  -----  -----

Base: Total Respondents          545    545    545

 14-29 yrs.                       36     55     55
                                 6.6   10.1   10.1

 30-39 yrs.                       76    100    100
                                13.9   18.3   18.3

 40-49 yrs.                      143    125    125
                                26.2   22.9   22.9

 50+ yrs.                        290    265    265
                                53.2   48.6   48.6

total                            545    545    545
                               100.0  100.0  100.0
```

Figure 14.4.1e Revised tabular output.

Input weighting can also be used to closely mimic these results. With input weighting, the population percentages need to be specified and can be used in conjunction with other types of weighting or the keyword **input** can be used on its own similar to target weighting. Both of these examples are shown in Figure 14.4.1e.

Input weighting can be used in conjunction with target weighting, but the code implication is redundant to using input weighting alone.

```
/**additional weight matrices using input weighting
wm3 gender age; rim; input; 49.5; 50.5; 10.1; 18.3; 22.9; 48.6

wm4 age gender; input; 10.7; 9.5; 15.2; 21.5; 23.3; 22.5; 50.7; 46.5
```

Figure 14.4.1f Additional wm statements using input weighting schema.

The percentages to be used for these statements were taken directly from the weighted tables, but can also be calculated using the standard weighting formula.

```
/* define record structure of input file
struct;read=0;ser=c(1,4);reclen=310
ed

end

/*define global table output options
a;dsp;flush;nopc;notype;op=12;side=30;printz

wm1 gender age; rim; 270; 275; 55; 100; 125; 265
wm2 age gender; target; 29; 26; 41; 59; 63; 62; 137; 128
wm3 gender age; rim; input; 49.5; 50.5; 10.1; 18.3; 22.9; 48.6
wm4 age gender; input; 10.7; 9.5; 15.2; 21.5; 23.3; 22.5; 50.7; 46.5

/* check weights with tabs
tab gender wban1
tab age wban1

/* define axes
l gender
ttlGender
n10Base: Total Respondents;dsp
n01Male;c=c151'1'
n01Female;c=c151'2'
n05total;nodsp

l age
ttlAge
n10Base: Total Respondents;dsp
n01 14-29 yrs.;c=c(152,153).in.(14:29)
n01 30-39 yrs.;c=c(152,153).in.(30:39)
n01 40-49 yrs.;c=c(152,153).in.(40:49)
n01 50+ yrs.;c=c(152,153).in.(50:99)
n05total;nodsp

/* banner
l wban1
g         RIM   TARGT RIM % INPUT
g UnWgt  Wghtd  Wghtd Wghtd Wghtd
g Total  Total  Total Total Total
g -----  -----  ----- ----- -----
p     a      b      c     d     e
n01unweighted;wm=0
n01weighted wm1;wm=1
n01weighted wm2;wm=2
n01weighted wm3;wm=3
n01weighted wm4;wm=4
```

Figure 14.4.1g Revised run file using rim, target and input weighting.

To mimic the results of the previous tables, two weight matrices are added to the run file, and new points which implement these weights are added to the banner. Figure 14.4.1g highlights these differences in the revised run file.

Although the weighting schemes for all matrices are calculated differently, the results are identical. Figure 14.4.1h proves that identical weighting results can be achieved using any of the methods described above. It is ultimately up to the user to decide which method fully meets the needs of the data.

```
Page 1
Gender

                                      RIM    TARGT RIM % INPUT
                               UnWgt  Wghtd  Wghtd Wghtd Wghtd
                               Total  Total  Total Total Total
                               -----  -----  ----- ----- -----

Base: Total Respondents          545    545    545   545   545

Male                              98    270    270   270   272
                                18.0   49.5   49.5  49.5  50.0

Female                           447    275    275   275   273
                                82.0   50.5   50.5  50.5  50.0

total                            545    545    545   545   545
                               100.0  100.0  100.0 100.0 100.0

Page 2
Age
                                      RIM    TARGT RIM % INPUT
                               UnWgt  Wghtd  Wghtd Wghtd Wghtd
                               Total  Total  Total Total Total
                               -----  -----  ----- ----- -----

Base: Total Respondents          545    545    545   545   545

 14-29 yrs.                       36     55     55    55    55
                                 6.6   10.1   10.1  10.1  10.1

 30-39 yrs.                       76    100    100   100   100
                                13.9   18.3   18.3  18.3  18.4

 40-49 yrs.                      143    125    125   125   125
                                26.2   22.9   22.9  22.9  22.9

 50+ yrs.                        290    265    265   265   265
                                53.2   48.6   48.6  48.6  48.6

total                            545    545    545   545   545
                               100.0  100.0  100.0 100.0 100.0
```

Figure 14.4.1h Revised tabular output showing different types of weighting.

14.4.2 Weighting axes within a run file[7]

Rather than defining weights with matrices, it is possible to define weights directly on an axis. In order to do this, simply add the keywords **wtfactor** or **wttarget** on the n01 statement to which the weights apply. The axis must then be assigned to a weight matrix which must be defined in the tab section before the first tab statement. Figure 14.4.2a gives an example of a run file which uses this method of weighting.

Note that the axis gender is defined as the weight matrix in the statement wm1=gender, and that weight matrix 1 has been applied globally to the tables by adding the statement wm=1 to the a-card.

```
/* define record structure of input file
struct;read=0;ser=c(1,4);reclen=310
ed

end

/*define global table output options
a;dsp;flush;nopc;notype;op=12;side=30;printz;wm=1

wm1=gender

/* check weights with tabs
tab gender wban1

/* define axes
l gender
ttlGender
n10Base: Total Respondents;dsp
n01Male;c=c151'1'; wttarget=270
n01Female;c=c151'2'; wttarget=275
n05total;nodsp


/* banner
l wban1
g
g  Wgt
g Total
g -----
p     a
n01;
```

Figure 14.4.2a Example of run file using axis as weighting driver.

An axis used as a weighting axis can also be used in a tab statement, since the keywords **wtfactor** and **wttarget** will be ignored when used in the context of a tab statement.

14.4.3 Using preweights and postweights

In order to specify the usage pf preweights when defining a weight matrix, use the **pre=** keyword on a **wm** statement and specify the columns which contain the weight to be used as a preweight.

[7] This functionality is still under development.

As shown in Figure 14.4.2a, the weights contained in columns 201 through 206 will be applied before the rim weights are applied. Preweights may be calculated within the edit section of the run file, but appear only on the wm statements within the tab section. Note that the reference for the columns uses the CX column notation for decimal values (see Section 13.1 for additional information). Postweights may be applied in the same manner.

```
wm1 age sex; pre=cx(201,206); rim; 100; 100; 200; 100; 200; 300;
```

Figure 14.4.2a Using preweights within a weight matrix definition.

14.4.4 Weights from other sources

There may be cases where data weights may already be populated within the data file or are merged in from other files into the data. There may also be cases where weights must be calculated "on the fly" within the edit section of the run file. In both of these cases, the weights must be applied using preweights. Since weights are normally assumed to be read in from the wm statement, any weights which originate from other sources must be applied using preweights. The constraint with using preweights is that they must be used in conjunction with other weights. These cases call for the creation of "fake", or a "dummy" weight matrix to be created.

This dummy weight matrix is very simple to create. A dummy axis is nothing more than an l-card and axis name followed by a blank n01 statement. An example of this can be seen in Figure 14.4.4a.

```
/*define global table output options
a;dsp;flush;nopc;notype;op=12;side=30;printz

/*define weight matrix
wm1 qfakewt; pre=cx(201,206); factor; 1

/* check weights with tabs
tab gender wban1
tab age wban1

/* define axes
l qfakewt
n01
```

Figure 14.4.4a Using preweights as weights in conjunction with a dummy weight matrix.

The dummy weight matrix shown causes a weight of 1 (essentially no weight) to be applied as the factor weight. Since the factor weight has no effect on the data, the preweights become the actual weights which are applied to the data.

14.4.5 Weighted Holecounts and Lists

As discussed in sections 9.1 and 9.2, **ccount** gives the ability to apply weights to standard holecounts and lists. This feature is useful for verifying table output as well as for comparison and statistical calculation purposes. For detailed examples, please read section 9.1.1 and 9.2.1.

Weights must be transferred into the data using wttran before weighted holecounts or lists can be generated.

14.5 Transferring weights into data

There may be instances when the weights for every respondent must be transferred into the data file for future reference and/or use. In order to transfer the weight used for each respondent into that specific respondents' data, use the **wttran** keyword.

wttran [weight_matrix_number] [column_reference]:decimal_places

Figure 14.5a Transferring weights into a data file.

The **wttran** keyword will transfer weights from the specified weight matrix into the data columns specified. The number of decimal places to use when transferring the weights must also be specified, so that the weights are all copied into the data in a standard format. Weights are transferred into the data right-justified, which causes all of the decimal points to line up within the data file. This is convenient for comparison purposes. An example of a wttran statement can be seen in Figure 14.5b.

Omitting the weight matrix number from the wttran statement will cause the default weight matrix, wm1, to be transferred.

In the Figure, the weights from weight matrix 7 will be transferred into the data columns 141 through 149, using 4 decimal places. Weights will be copied into the data into these locations, and can be viewed using lista statements to verify that they have been transferred properly, and that they fall into the expected range of weights for the particular run.

```
wttran 7 c(141,149):4
```

Figure 14.5b Example of wttran statement.

When copying weights into data, the standard column notation must be used since the assumption is that all weight matrices will contain real values. Using real column references (cx) will result in syntax errors.

14.6 Handling weights

There are other utilities which **ccount** provides which can help to handle and troubleshoot any weighting issues. These keywords and utilities are explained in detail in the following sections.

14.6.1 The weightrp file

The weightrp file provided by **ccount** is automatically generated any time a weighted run is processed. This file provides a summary report which includes specific weighting information about the run, including the type of weighting used, the weights for each category of weighting and the input and projected frequencies and percents. The calculated weights are also summarized in a standard, easy to read tabular format. An example of a weightrp file can be seen in Figure 14.6.1a.

The summary given in the weightrp file is an invaluable resource for comparison. Weighted values printed on the accumulated tables can be verified to make sure that the correct weights were applied. Table totals and bases can also be compared against the file, which can give a more thorough understanding of the effects of weighting.

```
CCWEIGHT - weighting program version 1.0 (Mar 04 2005 10:51:48)

Weighting matrix 1:



          Rim Weighting invoked for Weight Matrix 1

   normval: 545.000000,  thisval: 545.000000,  nrec: 545,  wmtotal:  0.000000


                      GIVEN
                    ---------
Dimension  1:       270.00000
                    275.00000
Dimension  2:        55.00000
                    100.00000
                    125.00000
                    265.00000


            INPUT        INPUT            PROJECTED     PROJECTED
          FREQUENCY     PERCENT           FREQUENCY      PERCENT
         -----------   -----------       -----------   -----------
             98.000       17.98             270.000        49.54
            447.000       82.02             275.000        50.46
         -----------   -----------       -----------   -----------
            545.000      100.00             545.000       100.00

             36.000        6.61              55.000        10.09
             76.000       13.94             100.000        18.35
            143.000       26.24             125.000        22.94
            290.000       53.21             265.000        48.62
         -----------   -----------       -----------   -----------
            545.000      100.00             545.000       100.00

Convergence occured on Iteration 3 with rms 0.000848 vs. limit 2.725000

CCWEIGHT - weighting program version 1.0 (Mar 04 2005 10:51:48)

Weighting matrix 1:

             RIM                   OUTPUT         OUTPUT
            WEIGHT               FREQUENCY      PERCENT
         -----------            -----------   -----------
           2.786736  *             269.979        49.54
           0.608430                275.021        50.46
                                -----------   -----------
                                    545.000      100.00

           1.480419  *              55.000        10.09
           1.470074  *              99.999        18.35
           0.870222                125.000        22.94
           0.894510                265.001        48.62
                                -----------   -----------
                                    545.000      100.00

          * indicates Rim Weight is outside the Range of 0.6 - 1.4


       Rim Weighting Efficiency  56.7 %

       Maximum Respondent Rim Weight : 4.125538
       Minimum Respondent Rim Weight : 0.529470

       Final maximum with pre/post weighting : 8.457353
       Final minimum with pre/post weighting : 1.853144

CCWEIGHT - weighting program version 1.0 (Mar 04 2005 10:51:48)

Weighting matrix 2:

             Summary of weighting
                 Matrix 2

  1 Given       29.00    26.00
   Counts        7.0     29.0
   Weights     4.143    0.897
  2 Given       41.00    59.00
   Counts       10.0     66.0
   Weights     4.100    0.894

  3 Given       63.00    62.00
   Counts       26.0    117.0
   Weights     2.423    0.530

  4 Given      137.00   128.00
   Counts       55.0    235.0
   Weights     2.491    0.545

Lowest weight: 0.530 Highest weight: 4.143
```

Figure 14.6.1a Contents of a standard weightrp (weight report) file.

14.6.2 Weighting error handling

To employ the error handling for weighting, use the default **wmerrors** keyword on the a-card or on any wm statement. In order to suppress weighting errors, use the **nowmerrors** keyword on the a-card or wm statement.

The **wmerrors** keyword is the default specification, and does not necessarily need to be re-specified on the a-card or wm statements since it is automatically activated for all **ccount** runs. This default means that processing will be stopped when any errors or warnings are encountered. Critical errors as well as any warnings will cause the compiler to stop. If **nowmerrors** is activated, then processing will only be halted for critical errors within the weighting process.

Appendices and Glossary

This section covers additional information related to **ccount**. Important key terms and summaries of commands can be found within this section. Quick reference guides will also be available within this section in order to further clarify key points made within the manual.

A.1 summary of struct (data structure) options

struct = defines the start of a struct line

read = defines the type of data file being tabbed

ser = location of serial (respondent) number

crd = location of card number (used for card data only)

reclen = length of records in data file (used on flat data only)

max = maximum number of cards (used on card data only)

req = card numbers that are required to exist in the data file (card data only)

rep = used for trailer card processing

example struct statement for card data:

```
      struct;read=2;ser=c(1,5);crd=c(6,7);max=10;req=1
```

example struct statement for flat data:

```
      struct;read=0;ser=c(1,5);reclen=2000
```

A.2 summary of edit/cleaning options

r (require) = basic require statement, used to test the validity of columns/logical expressions

sp = single-punch

b = blank

nb = not blank

spb = single-punch or blank

o = (only) used to further specify valid punches

emit = adds the specified punch into the specified column without overwriting existing punches

delete = deletes specifies punches without affecting other punches present in the specified column

A.3 summary of a-line (global tabulation) options

a	**=**	defines the a-line (a-card), and opens the tab section
dsp	**=**	double-spaces all rows
nodsp	**=**	suppresses the printing of double-spaced rows
op (op=)	**=**	declares the type of output to display on tables. The 3 most commonly used output options are: **0** = row percentages **1** = absolute figures (default) **2** = column percentages **5** = prints the text 100% under each base element **&** = total percentages (to the upper-left baseline number)
dec	**=**	number of decimals for absolute figures (*maximum:* dec=7)
decp	**=**	number of decimal places for percentages (*maximum:* decp=7)
flush	**=**	right aligns both absolutes and percentages in cells on tables
indent	**=**	auto-indent wrapped axis element text (*example:* **indent=3**)
spechar	**=**	prints specified characters instead of zeros in cells containing (weighted or unweighted) zeros (*most common:* **spechar=-***)
printz	**=**	prints blank tables (default is to suppress printing of blank tables)
noprintz	**=**	suppresses the printz keyword
type	**=**	prints output type in top right hand corner of the table
notype	**=**	suppresses the printing of the output type
side	**=**	can be used to alter the text width (default=24, maximum=120) (*example:* **side=30**)
paglen	**=**	defines the page length(in number of lines) (*example:* **paglen=60**)
pagwid	**=**	defines the character width of each page (*example:* **pagwid=140**)
nz	**=**	drops blank lines (stubs) in a table (default is to print all elements in a table)
nonz	**=**	suppresses the nz keyword
page	**=**	activates automatic page numbering (default)
nopage	**=**	suppresses page numbering
pc	**=**	prints percent signs after percentage figures on tables (default)
nopc	**=**	suppresses the printing of percent signs on tables
colwid	**=**	column width to use when using banners without g-cards and p-cards
pagtxt	**=**	Replaces standard page text (Page nnn) with specified text for international page numbering (*example*: pagtxt=Seite)

Figure A.3.1 Summary of a-card options

A.4 summary of table title and footnote options

ttl = left-justified title

ttc = centered title

ttr = right-justified title

foot = footnote (followed by a tt statement)

bot = print title on bottom of table (followed by a tt statement)

A.5 summary of n statement options

n00 = filters within an axes

n01 = basic counts

n03 = text only

n05 = subtotal

n07 = average

n09 = page break

n10 = base element

n11 = non-printing base element

n12 = mean element

n13 = sum of factors

n15 = non-printing basic counts

n17 = standard deviation

n19 = standard error of the mean

n23 = subheading

n25 = component value for mean and other statistical calculations

n33 = text continuation

A.6 summary of output files for a standard ccount run

out1 = expanded program listing compiled from all spec files
includes error messages relating to syntax errors found in program files

out2 = summary of data reads
includes respondent data of respondents that were rejected or failed any data checks

out3 = output summary listing
includes information about statistical tests

tab_ = tables file

hct_ = holecounts (marginals)
raw or manipulated counts of the current c array

lst_ = frequency distributions with mean calculations (lists/arrays)
raw or manipulated frequencies of the current c array

sum_ = ranked summary of data error messages
includes defined variable information

weightrp = report of the weights generated for each cell in each defined weight matrix

A.7 summary of cclean (ccount cleaning) options

temporary files are removed during a standard **ccount** run.

in order to remove unwanted files, one must use ***cclean***, or an equivalent **ccount**-cleaning program

cclean removes all intermediate and output files from the directory

options:

–a removes all intermediate files.

–y silently deletes files which normally prompt for permission to delete
may not be used with –n

–n silently keeps files which normally prompt for permission to delete
may not be used with –y

A.8 call fetch summary

what is call fetch?

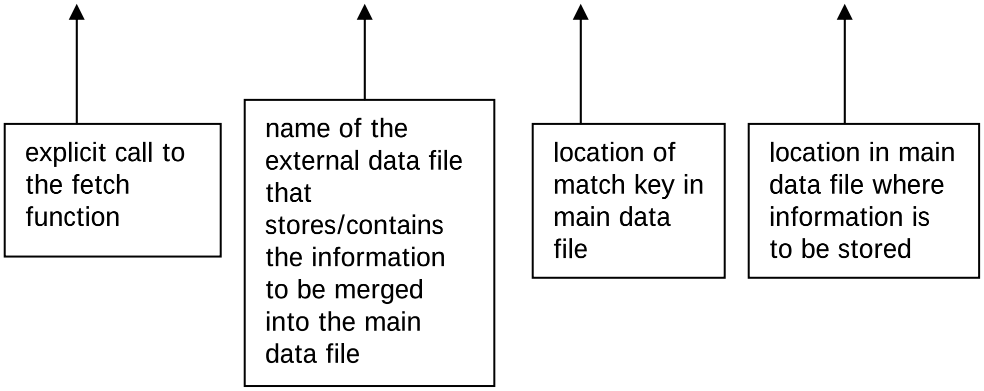
call fetch is a statement that enables the user to "fetch" existing data from an outside file and merge it into the main data file. This merged data can be used for cleaning existing data, for tabbing purposes, or to link "pre-wave" and "post-wave" survey data.

how does it work?

by matching key fields together from both files, it assumes a match and copies the data from the external file into the main data file to a specified location.

what is the syntax?

```
call fetch ($file_name$,key_start,destination_cell)
```



explicit call to the fetch function

name of the external data file that stores/contains the information to be merged into the main data file

location of match key in main data file

location in main data file where information is to be stored

what else is needed?

The external data file (sometimes referred to as the "fetch" file or the "lookup" file) must contain 2 pieces of information (integer numbers) on the first line of the file. The first number gives the length (in characters) of the key field and the second is the total length of the line to be merged into the data file. All subsequent lines must begin with the key field, and must be sorted in ascending order.

important notes:

- The match key can be any UNIQUE identifier (respondent number, phone number etc.).
- The fetch file **MUST** be sorted from lowest to highest based on the match key being used.
- ccount assumes the external file to be a single-record data file. The line length limit for fetch files is 99,999 columns. The maximum number of lines per fetch file is 65,000.
- The main data file can be either read=0 or read=2.

While there is theoretically no limit to the number of fetch statements a run file may have, there is a limit of 15 unique fetch files that can be used within any given run file. That is, a single fetch file may be called multiple times within the run file, but there exists a maximum of 15 different fetch files per run.

A.9 summary of maximum limits within ccount

The table below lists the practical limits within ccount for various options and output files.

Item	Limit
Max. number of filedefs including out2	51
max. number of weight matrices	9
max. number of axis per weight matrix	16
max. number of different table names (tab axis1 break, tab axis2 break = three different names) per run including wm axis	500
max. record length of data file	99999
max. number of variables (int, real, data) including default variables	1000
max. length of variable name	13 characters
max. number of data cards in run with reclen<=100	999
max. number of data cards in run with reclen>100 (up to max. 1000)	99
max number of axis names (l axis1 = one name) per run	600
max. number of tables (tab axis1 break, tab axis2 break = two tables) per run	1500
max. nested loops	10
max. length for report files	unlimited in practice
max. number of labels (goto 1234, do 1234 t1=..., 1234 continue)	500
max. lines per fetch file	65000
max. number of different fetch files	15
max. line length of one single line in fetch file	99999
max. pagwid in tables	10000
max. paglen in tables	10000
max. side space (row text width) in tables	120
max. colwid (column width) in tables	64
lowest whole number (integer)	-2,147,483,647
highest whole number (integer)	2,147,483,647
accuracy of real numbers	up to sixteen (16) significant figures
max. number of decimals in tab section (dec=x, decp=y)	7
max. length of internal line command buffer (eg. buffer which is available for generating C-code from Ccount syntax)	32768
max. length of path+filenames (eg. *include /home/user/includes/mynew.inc)	256

Figure A.9.1 Practical limits within ccount.

A.10 summary of statement options

Following is a table of the most commonly used statement options in ccount. The table explicitly defines which options can be used on each particular statement.

Option	Valid Statements for use with Option
dsp	a,tab,flt,l,nXX
nodsp	a,tab,flt,l,nXX
op (op=)	a,tab,flt,l,nXX
dec	a,tab,flt,l,nXX
decp	a,tab,flt,l,nXX
flush	a,tab,flt
spechar	a,tab,flt
printz	a,tab,flt,l,nXX
noprintz	a,tab,flt
type	a,tab,flt
notype	a,tab,flt
side	a,tab,flt
paglen	a,tab,flt
pagwid	a,tab,flt
nz	a,tab,flt,l,nXX
nonz	a,tab,flt,l,nXX
page	a,tab,flt
nopage	a,tab,flt
pc	a,tab,flt
nopc	a,tab,flt
colwid	a,tab,flt
pagtxt	a
indent	a,tab,flt,l,nXX
col	nXX (this axis element is a column element, default)
nocol	nXX (this axis element is not a column element and ignored if axis is used as column axis)
row	nXX (this axis element is a row element, default)
norow	nXX (this axis element is not a row element and ignored if axis is used as row element)
base	nXX (this axis element is a base element)

Figure A.10.1 Summary of valid statement options.